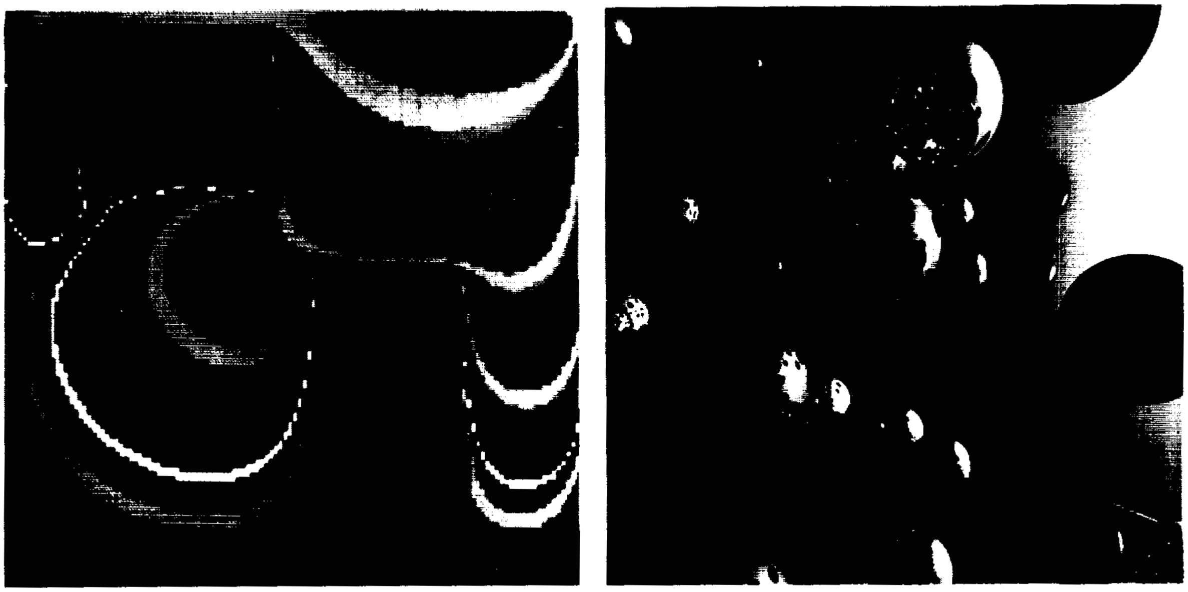
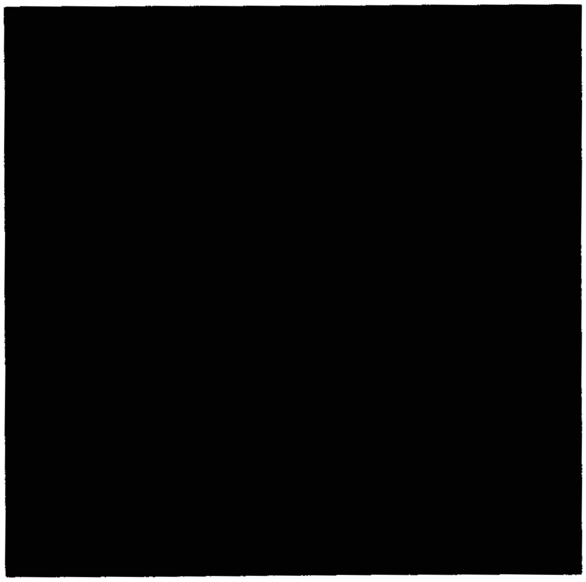
Proceedings

**FRONTIERS '88**

*The 2nd Symposium on the*

# *Frontiers of Massively Parallel Computation*





October 10—12, 1988
George Mason University
Fairfax, Virginia

IEEE Computer Society Order Number 892
Library of Congress Number 88-82088
IEEE Catalog Number 88CH2649-2
ISBN 0-8186-0892-7
SAN 264-620X

IEEE COMPUTER SOCIETY

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.
IEEE

IEEE COMPUTER SOCIETY PRESS

IEEE-NATIONAL CAPITAL AREA COUNCIL
IEEE

George Mason University

NASA
National Aeronautics and Space Administration
Goddard Space Flight Center

## COVER PHOTOS:

TOP LEFT: **Fluid Flow.** This picture shows the flow of water around rectangular objects in an open tank. Flow patterns are calculated using the Navier-Stokes equations on a 1,024-processor DAP 510.

TOP RIGHT: **Graphics.** "Mercury$^2$" by H. C. Delany, M.I.T. Media Laboratory. This picture shows reflective spheres over an alien sunset. It was computed at 1024 x 1024 pixel resolution with 5 orders of reflection in 6 minutes using a 16,384-processor Connection Machine–2.

BOTTOM RIGHT: **Stereo Image Matching** by J. P. Strong and J. E. Dorband, NASA/ Goddard Space Flight Center. A three-dimensional perspective view of terrain in northeast India created by automatically fusing a pair of 512 x 512 stereo images in 50 seconds using NASA's 16,384-processor Massively Parallel Processor.

Cover design by ST Systems Corporation (STX) Graphics—Joan Sargies, Jane Foltz, and Barbara Gogan–Marinaro.

Proceedings

The 2nd Symposium on the

# Frontiers of Massively Parallel Computations

The 2nd Symposium on the

# Frontiers of Massively Parallel Computation

Ronnie Mills, *Editor*
*ST Systems Corporation*
*Lanham, Maryland*

Proceedings of the second symposium sponsored by the
Computer Society of the IEEE,
NASA/Goddard Space Fight Center,
IEEE National Capital Area Council, and
George Mason University, and held at
George Mason University
Fairfax, Virginia
October 10–12, 1988

Supported by grants from
Active Memory Technology,
Digital Equipment Corporation,
LORAL Defense Systems–Akron,
Martin Marietta Aerospace,
Science Applications Research,
ST Systems Corporation, and
Thinking Machines Corporation

Proceedings

The 2nd Symposium on the

# Frontiers of Massively Parallel Computations

IEEE Computer Society Press

Washington ● Los Alamitos ● Brussels ● Tokyo

IEEE Computer Society The Institute of Electrical and Electronics Engineers, Inc.

The papers in this book comprise the proceedings of the meeting mentioned on the cover and title page. They reflect the authors' opinions and are published as presented and without change, in the interests of timely dissemination. Their inclusion in this publication does not necessarily constitute endorsement by the editors, the IEEE Computer Society Press, or The Institute of Electrical and Electronics Engineers, Inc.

Published by

IEEE Computer Society Press
1730 Massachusetts Avenue, N.W.
Washington, D.C. 20036-1903

IEEE Computer Society Order Number 892
Library of Congress Number 88-82088
IEEE Catalog Number 88CH2649-2
ISBN 0-8186-0892-7 (paper)
ISBN 0-8186-5892-4 (microfiche)
ISBN 0-8186-8892-0 (case)
SAN 264-620X

Additional copies may be ordered from:

IEEE Computer Society
Order Department
10662 Los Vaqueros Circle
Los Alamitos, CA 90728-2578

IEEE Service Center
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331

IEEE Computer Society
13, Avenue de l'Aquilon
B-1200 Brussels
BELGIUM

IEEE Computer Society
Ooshima Building
2-19-1 Minami Aoyama
Minato-ku, Tokyo 107, JAPAN

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

IEEE

# PREFACE

Individuals interested in developing, using, and selling massively parallel computers converged on George Mason University in Fairfax, Virginia, on October 10 for *Frontiers '88: The Second Symposium on the Frontiers of Massively Parallel Computation.* Four-hundred-forty-five registrants (including five from foreign countries) attended the 3-day conference. Seventy-six universities, 81 corporations, and 37 Government organizations were represented.

There were 52 oral presentations and more than 60 poster presentations. There were also exhibits, a tutorial (attended by 129 registrants), and a panel discussion. The wide attendance and scope of the papers indicates that the field of massively parallel computing is attaining ever greater importance.

The sponsors and grantors look forward to continuing this series of symposia every other year at university sites in the greater Washington, DC, area. We hope these *Proceedings* give you some feel of the excitement that Frontiers '88 generated.

| | |
|---|---|
| James R. Fischer | David H. Schaefer |
| NASA/Goddard Space Flight Center | George Mason University |
| Conference Chairman | Program Chairman |

***A note about the exhibits:***

Nine organizations mounted exhibits at the symposium. Four of these exhibits featured operational massively parallel systems:

> Active Memory Technology had both their 1024 processor DAP 510 and their 4096 processor DAP 610 performing demonstrations.
>
> George Mason University demonstrated the student-built GAM 2 pyramid structure, containing 1365 processing elements on six levels, in their Advanced Computer Architecture Laboratory.
>
> Martin Marietta Aerospace had a 40,000 processor Geometric Arithmetic Parallel Processor (GAPP) system in operation.
>
> Thinking Machines Corporation had a 32,768 processor Connection Machine–2 running and demonstrating a variety of applications.

Other exhibitors were the Space Data and Computing Division at NASA/Goddard Space Flight Center with an exhibit of Massively Parallel Processor (MPP) applications, LORAL Defense Systems–Akron with an exhibit on the MPP and the Associative Parallel Processor (ASPRO) computer, and the Cornell Theory Center with an exhibit of their Trollius Operating System. Two publisher exhibitors were John Wiley & Sons, Ltd., who advertised their new journal *Concurrency: Practice and Experience,* and the publishers of *Supercomputing Review.*

# FOREWORD

*This section was transcribed from welcoming remarks presented at the symposium.*

**Dr. George W. Johnson,**
**President of George Mason University**

George Mason University, in order to move quickly, has had to make certain bets. Two of those bets are represented by this conference. One bet was that we had always to anticipate what the cutting edge of developments in the fields that we chose to embark on would be. We decided that we would have to build a new engineering school, but that it would not be one built on the base of the physical sciences, but on the base of the information sciences. That was the first bet that we made.

The 2nd bet was that as a new university, we could not afford to be aloof and remote from our community, that we had to be what we call "interactive." We had to break down the walls between town and gown, the walls between marketplace and academy, and we have embarked aggressively on doing exactly that. You can see that the two come together in our School of Information Technology and Engineering, where the demarcation between business, government, and academy really is successfully blurred. And so this conference coming here at this particular time is really a signal event for George Mason University, and for that reason, among many others, you're very very welcome to the institution, and I'm glad to see you here. Thank you.

**Dr. Lee Holcomb**
**Director, Information Sciences and Human Factors Division**
**NASA Headquarters**

On behalf of NASA, I would like to welcome you all to this symposium—*The 2nd Symposium on the Frontiers of Massively Parallel Computation.* The NASA Office of Aeronautics and Space Technology has had a long history in parallel processing technology. Beginning in the early 1970's with the evaluation of the ILLIAC IV, we gained valuable knowledge on how to apply parallel processing power to aerospace computational requirements. In the late 1970's NASA was at the forefront of this important computing technology by funding the development of the world's first massively parallel processor, the MPP, which was originally intended for image processing applications. The MPP was designed, fabricated, and delivered to NASA in 1983, and an Applications Working Group was formed to encourage researchers from widely varied disciplines to conduct research on the MPP. That early investment is now bearing fruit, and is certainly evident, in part, by this symposium.

In the 1982 and 1983 time frame, the White House Office of Science and Technology Policy (OSTP) Federal Coordinating Committee on Science Engineering and Technology (FCCSET) established a subcommittee on supercomputing to coordinate the activities of the various federal government agencies. The early FCCSET committee focused on the concept of the federal government as a friendly buyer of supercomputers and on the concept of providing researchers access to supercomputers. Some of the programs that came out of this early activity were the Numerical Aerodynamic Simulation program at NASA/Ames, the NSF supercomputer centers that have been established throughout this country, and the Supercomputing Research Center in Maryland.

More recently, the FCCSET committee has focused on the government being more than a friendly buyer. A recent OSTP report titled "A Research and Development Strategy for High Performance Computing," which was issued in November 1987, calls for government sponsorship of research in high performance computers, software technology and algorithms, networking, and basic research, and resources. Each government agency that is participating in FCCSET is preparing a response to this report. NASA has developed a high-performance computing initiative which includes 3 parallel computing testbeds, one for computational aerosciences, one for Earth & space sciences, and one for spaceborne applications. We intend to support algorithm and architecture research and advances in software for massively parallel systems, both operating systems and languages.

To date, this initiative has been worked primarily within the government through planning activities of the FCCSET committee. NASA now wants to solicit stronger involvement from industry and universities in this program. Dr. Paul Smith of my staff, who is here today, will be formulating an approach for forming a government, industry, and university planning team to advance this country's massively parallel computational capability. Dr. Smith will be present throughout this conference, and we solicit your views on how best to form this partnership. At the hearings on the National Supercomputer Network in September 1988, Senator Albert Gore, challenged the government to strengthen its ties with academia and industry. This conference provides one forum to begin forming a partnership of federal, academic and industrial researchers in this important frontier. I'm happy to welcome you to this symposium and hope that you will find it stimulating and productive.

# ACKNOWLEDGEMENTS

*Frontiers '88 could not have taken place without the contributions of many people including:*

## Technical Meeting Committee

James Fischer *(Conference Chair)*
NASA/Goddard Space Flight Center

James Tilton *(Finance Chair)*
NASA/Goddard Space Flight Center

David Schaefer *(Program Chair)*
George Mason University

Ronnie Mills *(Publications Chair)*
S.T. Systems Corporation

Pearl Wang *(Tutorials Chair)*
George Mason University

Mark Stevens *(Publicity Chair)*
S.T. Systems Corporation

Samuel Earp *(Exhibition Chair)*
George Mason University

Barbara Framer *(Local Arrangements Chair)*
George Mason University

## Representatives of Cosponsors and Grantors

Ai C. Fang
NASA Headquarters

Anne Marie Kelly
Magdelene Johnson
Denise Felix
IEEE Computer Society

Thomas Doeppner
IEEE National Capital Area Council

Lydia Walls
Nancy Joyner
George Mason University

Bruce Alper
Kevin Linck
Active Memory Technology, Inc.

Jonathan Harris
Digital Equipment Corporation

Alan Hinkle
LORAL Defense Systems—Akron

Wade Pemberton
Eugene Cloud
Ron Sartain
Martin Marietta Aerospace

Ron Estes
S.T. Systems Corporation

Marvin Denicoff
Carol Bee-Latty
Thinking Machines Corporation

## Working Program Committee

Prof. David Schaefer
George Mason University

Prof. Marvin Denicoff
Thinking Machines Corporation

Dr. John Dorband
NASA/Goddard Space Flight Center

Prof. Michael Rice
George Mason University

Dr. James Strong
NASA/Goddard Space Flight Center

Dr. James Tilton
NASA/Goddard Space Flight Center

Prof. Pearl Wang
George Mason University

## Steering Committee

Carol Bee-Latty
Thinking Machines Corporation

James Fischer
NASA/Goddard Space Flight
Center

Chester Grosch
Old Dominion University

R. Michael Hord
MRJ, Inc.

Michael Rice
George Mason University

## Program Committee

Prof. David Schaefer
George Mason University

Dr. Ray Arnold
NASA Headquarters

Dr. Ken Batcher
LORAL Defense Systems—Akron

Dr. Jack Dongarra
Argonne National Laboratory

Prof. Michael Duff
University College, London

Dr. Milton Halem
NASA/Goddard Space Flight Center

Dr. James Hardy
Whitney/Demos Productions

Prof. Dennis Parkinson
Active Memory Technology

Prof. Tomaso Poggio
Massachusetts Institute of Technology

Prof. John Reif
Duke University

Prof. Anthony Reeves
University of Illinois

Prof. Azriel Rosenfeld
University of Maryland

Dr. Paul Schneck
Supercomputing Research Center

Dr. Steven Squires
DARPA

Dr. Guy Steele
Thinking Machines Corporation

Prof. Leonard Uhr
University of Wisconsin

## Tutorial Lecturers

Prof. Dennis Parkinson
Active Memory Technology, Inc.
Queen Mary College

Prof. David Schaefer
George Mason University

Prof. Pearl Wang
George Mason University

Dr. John Dorband
NASA/Goddard Space Flight Center

## Invited Speakers

Dr. Kenneth Wilson
Ohio State University

Gary Demos
Whitney/Demos Productions

Prof. Azriel Rosenfeld
University of Maryland

Dr. Guy Steele
Thinking Machines Corporation

## Session Chairs

R. Michael Hord
MRJ, Inc.

Dr. James Tilton
NASA/Goddard Space Flight Center

Dr. James Strong
NASA/Goddard Space Flight Center

Dr. Samuel Earp
George Mason University

Dr. Michael Rice
George Mason University

Jonathan Harris
Digital Equipment Corporation

Prof. Pearl Wang
George Mason University

Prof. Joseph 'Ja 'Ja
University of Maryland

Prof. H.J. Siegel
Purdue University

Prof. Chester Grosch
Old Dominion University

Dr. John Dorband
NASA/Goddard Space Flight Center

Prof. Marvin Denicoff
Thinking Machines Corporation

Special thanks to Maxine Schaefer for developing the database of abstract submissions, Millie Brice and Towanda Plater at NASA/Goddard for handling the numerous large mailings, Michele Braithwaite at George Mason University and Jim Elliott at NASA/Goddard for supporting publicity efforts, and the George Mason University student chapter of the IEEE for supporting the exhibition and audio visual needs of the conference.

# CONTENTS

## INVITED PRESENTATIONS

## SECTION I: ALGORITHMS
## Part 1: Oral Presentations

## SECTION I: ALGORITHMS
## Part 2: Poster Presentations

## SECTION II: APPLICATIONS
## Part 1: Oral Presentations

## SECTION II: APPLICATIONS
## Part 2: Poster Presentations

## SECTION III: ARCHITECTURES
## Part 1: Oral Presentations

## SECTION III: ARCHITECTURES
## Part 2: Poster Presentations

## SECTION IV: LANGUAGES
## Part 1: Oral Presentations

## SECTION IV: LANGUAGES
## Part 2: Poster Presentations

## ADDITIONAL PAPERS

# THE PANEL DISCUSSION

On Wednesday afternoon, October 12, a panel convened to discuss the following topics:

- What is the future of MIMD in massively parallel systems?
- Will massively parallel computing environments ever be comparable to those of vector processors?
- Can progress in the use of massively parallel computing take place in a world dominated by dusty FORTRAN decks?

Professor David Schaefer of George Mason University led the discussion. The panel members were:

Dr. Milton Halem - Goddard Space Flight Center
Professor Kai Hwang - University of Southern California
Professor Dennis Parkinson - Active Memory Technology and
Queen Mary College
Professor John Reif - Duke University
Professor Anthony Reeves - Cornell University
Dr. Paul Schneck - Supercomputing Research Center
Dr. Guy Steele - Thinking Machines Corporation
Professor Leonard Uhr - University of Wisconsin.

The following is an edited version of this session. It was prepared by David Schaefer from a tape recording of the discussion. He gratefully acknowledges help from Jim Fischer of Goddard, from notes taken during the session by David Middleton of ICASE, and from the panel members, all of whom reviewed the first draft and made suggestions.

The session started with the reading of a communication.

*David Schaefer*

Thinking Machines Corporation has given me a memo concerning their feelings about SIMD and MIMD. The memo, a little edited, is as follows:

> People have an emotional attachment to MIMD. They keep wanting it to be the answer, but they keep seeing SIMD come up as the answer.
>
> Appeal of MIMD: You only understand in terms of what you understand, and people know single processors.

Achievements of SIMD:

- high processor count
- applications fit (God is SIMD)
- very basic circuitry
- price performance ratio is good

The ultimate issue: Software is too expensive to be run on just one processor. You need to write as little code as possible and run each program on as many processors as possible.

MIMD: "If you are willing to accept a lot of complexity, you can do amazing things."

SIMD: "If you are willing to accept a lot of simplicity, you can do amazing things."

That is the end of the Thinking Machines communique. Milt, do you want to take the ball and give your answer to the question, "What is the future of MIMD in massively parallel systems?"

*Milt Halem*

With the proliferation and increase in power of our workstations and our growing powerful PC's, the impetus on industry will be to tie that capability together to develop a more powerful resource. It is fairly obvious that there is a place for MIMD in massively parallel systems and there is a place for SIMD in massively parallel systems. I think we will see them both continue to evolve and merge in the next dozen years.

*Schaefer*

Do you think the panel should agree on what we mean by "massively parallel?"

*Kai Hwang*

The definition of massive parallelism is a time function—it varies. If you said a thousand now, five years from now you would be embarrassed to say a thousand.

What I feel about massive parallelism in MIMD systems relates to computations in the neural computing area. I feel that is where massively parallel MIMD operations are needed. Neurons are not synchronized in the biological case. To model a large neural mass, we need a simulation which is asynchronous.

The real bottom line concerning MIMD massive parallelism is not the hardware—it's really the software. We don't know how to partition programs so that multiple instruction streams can handle them. So software is one of the major research areas where we should push in order to see really large scale MIMD machines.

I feel there are real massively parallel MIMD candidates (with even a million PE's) in the neural computing area.

*Dennis Parkinson*

Whenever you get two groups of human beings fighting very hard about any subject such as BASIC versus FORTRAN, you know it is really an irrelevant argument. This is another of these irrelevant arguments which keeps academics happy most of the time. The future is not going to be MIMD, the future is not going to be SIMD, it is going to be some mucky mixture of the two.

When I look at the system I use, it has a component, one of its many components, which is a SIMD processor. It also has a Sun host which has a few other independent units. I use a collection of processors and I'm using them for their different abilities to do different parts of my total task, and that really is what is going to happen.

I think the massively parallel components are probably going to be very much of a SIMD nature. But they are going to be connected into an environment which we would consider to be MIMD in principle.

*Schaefer*

I have a quote here, that maybe you recognize. It says "the optimal algorithm for a MIMD system is a SIMD algorithm, therefore there is no point in building MIMD systems." Does that sound at all familiar?

*Parkinson*

I think I have heard it before. I actually used it many times, but it was, in fact, first said to me by Enrico Clemente, who put together MIMD systems from array processors at the IBM labs.

MIMD has an apparent advantage for the dusty LISP deck people, and that smaller community which uses dusty FORTRAN decks. If you start from how we used to program a serial machine, then the MIMD model is the easiest one to play with. Naturally, most people are starting from there.

If you start from old-fashioned languages like LISP, then you really are stuck in a computation model which wants to do one thing at a time. The easiest way to do that is to break your problem up into a few large independent tasks.

If you want to go into thousands of processors or tens of thousands of processors or millions of processors (I aspire, for Christmas every year, of 16 million, by the way), you will find that you can optimally keep everything synchronized by using SIMD algorithms. That's the best algorithm for many MIMD architectures. That's why the quote is there.

*Tony Reeves*

The thing I see with the SIMD system is that there is a remarkable number of applications that map onto such an architecture. When you get a good mapping from an algorithm to the architecture, then there is very little that you can do to beat that in terms of performance. The additional cost of an MIMD system is never going to be justified in that context.

I guess what concerns me is that class, even though an increasing class, is a restrictive class. I have been moving some algorithms that don't fit well onto SIMD systems over to MIMD systems. And what do you know? They turn out to be a lot faster and a lot more efficient.

We say SIMD is easy to program. I think software is the key. We don't really know how to program an MIMD system and it is going to be a while, a number of years, before we have the proper formalized techniques for doing so. We don't know how to express problems for MIMD systems. This harks back to the type of mathematics we use. There is a tremendous amount of work to be done in algorithms and concepts.

Consider simulation, for example. I think the mathematical approach we take to it right now is very clumsy. We specify a vast number of operations that are not necessary. It is going to take a more flexible computing structure to take advantage of any techniques we devise to not bother with some of the redundant calculation operations.

The answer is that there has got to be a place for both kinds of systems. MIMD architectures are going to assume a more important role as we are able to make the hardware cheaper as more effective software systems and environments are developed.

*Schaefer*

Tony, do you see a million-processor, MIMD system?

*Reeves*

Why not?

*John Reif*

I guess there are perhaps a number of answers to your question. I am a partly theoretical computer scientist, and there are some interesting ideas on the theory side of this issue. You've probably all heard of P versus NP. Can you take any arbitrary sequential problem that is running on a conventional machine, and then can you take it and put it on a parallel machine so it runs in, say, polylog time? This is a deep theoretical issue, but the conventional wisdom from the theory community is that the answer is "no."

So what this means in practice is that there will remain out there, in spite of very brilliant algorithm people and brilliant architectures like the Connection Machine, and other machines like that, a vast collection of non-parallelizable applications areas. That's the bad news.

The good news is that, as we have seen in demos here and from many of the talks, there remains a very large class of scientific problems that we can parallelize. The great advantage of these specialized SIMD architectures is they are more cost effective than a Cray.

Cray type architectures, incidentally, include SIMD attributes in their vector operations, and they have very fast MIMD capability. That is crucial if you want a general purpose machine that can handle anything. As I mentioned, there are significant classes of application areas which you cannot parallelize.

SIMD architectures are not necessarily purely SIMD any more. They once were, say five years ago, but as an example there is a MIMD indirect address feature which was first added by the DEC people. It was also added in our project where we have 128 processors on a chip, each having MIMD addressing. The Connection Machine now has that capability as well. Probably many of the future SIMD systems will involve MIMD aspects.

What they probably won't have is full independent control. But they are evolving in that direction. MIMD systems will evolve very successfully and there will eventually be a capability of putting more than one MIMD processor on a chip, perhaps even dozens to hundreds. There will be million-processor MIMD systems if we believe in the continual evolution of VLSI, just as we have seen in SIMD applications.

What will not happen will be that MIMD will take over because for specialized scientific computing problems, such as matrix problems and many fluid-flow problems and so forth, it is more cost effective to have massively parallel SIMD. So it is really a coexistence of two types of intellectual cultures that I think will become somewhat fused. They will remain and have various costs and computation power tradeoffs even through the next century.

*Schaefer*

As soon as you have a mask register, you don't have a completely SIMD system anymore. Therefore, hardly any pure SIMD systems are around anyway.

*Reif*

Well, indirect addressing is a significant jump past conditional control at the processor level.

*Paul Schneck*

Professor Parkinson pointed out that this is a religious issue, so let me state immediately that I'm a polytheist.

Some of you will recall the IBM compatible systems. That first operating system made a 360/75 completely compatible to a 360/30. They all ran at the same speed.

Allen Turing pointed out in a little informal proof that all computing systems, in the sense that we know of them, are equivalent. A problem which is soluble on one is soluble on another. The only difference is speed. Of course it is that difference which is the essence of why we are all here and why some computers are supercomputers and others are not supercomputers.

When I look at a computer system, I don't look directly at the computer. (Remember Dave introduced me as a compiler person.) I see it through a programming language, actually through an algorithm and then through a programming language, then through a compiler. There is an operating system that gets in the way before I ever get on the machine, and there is a loader and a library. There are a lot of intervening layers.

I would challenge almost anyone in this audience (this is an analog of the Turing test) to figure out not whether the thing on the other end of the teletype line is a person or a computer, but whether it's a SIMD machine, a MIMD machine, or a sequential machine.

You've got a twisted pair going out of the room (if you want to go modern, a fiber cable) and all that you have in front of you is a keyboard and a screen. You get to write a program and the important point is capturing the ideas of the program. Then you leave it to the compiler writing community to worry about the issue of parallelization.

Not that those are trivial issues, but I think all of the harder issues of SIMD and MIMD and sequential pale by comparison. In fact, right now there are 80 to 100 university projects building various types of parallel machines. Very few of those will see the light of day as commercial machines. Very few of those have compilers or languages associated with them.

It is largely a software compiler and language expressivity issue. Right now, probably among the couple of hundred people in this room, there are a couple of hundred ideas as to machines. I dare say there are far fewer language ideas, and that is where the real difference will be.

*Schaefer*

So Paul, you do see massively parallel MIMD systems?

*Schneck*

Sure I do, not to the exclusion of anything else, however.

*Guy Steele*

I don't believe in driving nails with a wrench—unless that's all I've got; or in driving carpet tacks with a sledge hammer or spikes with a tack hammer—unless that is all I've got. And by the way, these are not hypothetical examples. I've been in all these situations both literally and metaphorically. So my position is that let a thousand flowers bloom. Some of them will prove to be perennial and the rest won't.

*Len Uhr*

Yes, I agree. There are many intriguing possibilities to be explored. MIMD systems certainly will develop in many fruitful directions, but probably very gradually in relatively simple small steps.

It might make sense to look at our technology curve and pack as much into as good a system as we can build economically. Indeed, we developed more powerful individual processors that way, and the Cray is our standard leading example. We have much more powerful processors than the one-bit processing elements that today we associate with SIMD, and remember there is no reason to associate bit serial PE's with SIMD. That just happens to be the case today.

So my point is that the MIMD people have a very good argument. Let's build a good cheap powerful processor. Let's put a lot of them together. Each one has its own controller and that is what MIMD means, not the power of each processor.

But now they have the problem that they have not been able to solve. How on earth do these independent computers talk to each other? The best I have heard now is 5,000, or maybe 500, instructions to get a message from me to you. So they have major problems, but they will probably make progress toward solving them. Almost certainly what we want is to combine the virtues of SIMD and MIMD.

SIMD also has its problems when synchronous operation and a fixed topology are not appropriate.

We do have, fortunately, many cases ofbeautiful mappings of algorithms, usually onto an array, but sometimes onto a pyramid. Without a good mapping, we really need something to augment the array or pyramid. At the moment I'd suggest we see whether we can't design powerful SIMD-MIMD systems. I'm hoping that we will be able to solve the major problems on both sides, and get many successful new architectures.

*Parkinson*

Your references to Turing reminded me of when I had a few less gray hairs than I have now and I went to the office of Tony Hoare (who had a number of things that he was responsible for, including things like Quick Sort and MIMD languages) to describe the DAP architecture. I gave my talk describing the lock step, one thousand single-bit processor system.

At the end he threw me completely by saying "It's all a lie. You have not described a multiple processor at all to me. What you have described is a system with a 1,000-bit word which is arranged as a 32 by 32 matrix and has a rather strange instruction set which does operations between these words."

He would say the same to Guy Steele about his sixty-four thousand processor hypercube. He would say it was a one-word machine, that every instruction just has a hypercube shaped word. It's a totally valid approach to SIMD architectures.

Even if you had a million processors with 16 bit words, Tony Hoare would sort of say that you have got this three-dimensional word which is xyz, and it is a totally valid view that these SIMD architectures are just single-processor machines. It all depends what kind of software you care to put on these machines and how you care to interpret the results. And as I say, the argument is religious.

*Hwang*

I believe this audience, most of you, are probably more experienced with SIMD machines because signal processing and image processing need fine-grain processors. It is really the state of the technology and the simplicity of the control of the SIMD machines that leads to the massive parallelism of today.

The control complexity of a MIMD machine is tremendous. So that is why we didn't see large scale MIMD machines. The fundamental difference is that, on the one hand, SIMD machines have higher efficiency. On the other hand, MIMD machines have greater application flexibility. So you are talking about a trade-off between efficiency and flexibility.

Sometimes I use this comparison, communism versus capitalism. O.K.? In terms of production, you want to use SIMD machines; in terms of consumption, you want MIMD machines. Right! We need a hybrid architecture. When we want to engage in extensive production operations, we use the SIMD portion. When we are involved in the area of applications, we need flexibility, we need interactions. The MIMD machines then play more of a role.

*Reif*

I would concur, but we need the resources that are available for Crays. The better Crays cost $15 million. One could have the hypothetical massively parallel machine with significant MIMD capabilities and with state-of-the-art SIMD capabilities as well. That would be very very exciting.

What we have currently is the first stage toward that. It is not anywhere near what could be, given significant monetary resources. Imagine a $15 million Connection Machine-type of computer with MIMD capability and incredible I/O. It would be very interesting then to compare its ability with the Cray. We are really not playing a fair game. The Cray, the full expensive version of the Cray, is far more expensive than existing SIMD machines.

*Schneck*

I have two comments, the first being, I would like to argue with John about economics. We are talking about the price of the Cray, not its cost. When you own a big piece of the market you price it at what you can get for it Cost, I think, changes the equation rather dramatically.

More importantly, Ken Iobst, who is sitting in this audience, his chin just dropped, has an interesting test for these machines. He simulates one ilk of machine on another. You simulate a connection machine or the MPP on a Cray. You can try simulating a Cray on a Connection Machine, or if you have an SIMD machine, you can try simulating a Cray on an Intel V-6. I wouldn't try that too long, but you get the essence of the experiment.

If machine A can simulate machine B and machine C and do it well, but machine B cannot simulate machine A, then maybe there is a hierarchy that one sees coming out of this. I think the hierarchy is that for simulation purposes, SIMD machines seem to do a good job of simulating even MIMD machines. Part of that may be because so much of the effort is spent in instruction access, instruction decode and control and so little of the hardware, relatively speaking, is spent on the disparate aims and disparate actions of the program, which is of course the only thing we really care about. So I would pose this simulation metaphor as something to think about.

*Uhr*

It may well be that if we can handle a 64 computer MIMD system (one big enough so we can't stick it on a bus) then we can go up to any number.

Assume we have 64 anarchists, the MIT "actors," each one an independent agent that does all sorts of AI style "reasoning." It's as if each one of us on this panel is thinking, and then we decide we want to interrupt each other. If we were all talking then you, the audience, wouldn't hear much of the panel, and none of us would hear much of anything, or do much of anything.

The point is there is a problem in 5 or 10 people coordinating effectively. We might examine the sociology of small group behavior to try to develop the kind of message passing that MIMD systems need.

On a related issue, I used to think that we were really being cost efficient in SIMD systems in that everybody was working all the time. Which indeed they are, because they have to be working all the time. But are they working on anything useful?

For computer vision, it is pretty clear they initially are doing useful work because when a general vision program starts out, it doesn't know what is going to be where and, therefore, it has to look everywhere for everything. But as soon as it gathers some information, then clearly everything is not going to be everywhere, but all the processing elements continue to cycle in synchrony.

As processing continues, I get more and more unhappy. All the processors are working all the time, but more and more of them are just doing nothing.

Now the MIMD system is able to reconfigure its structure and handle that kind of thing with much more efficiency. If it were only capable of passing messages, which today, of course, it is not.

*Schaefer*

Well, we have whipped through the first question in record time of 40 minutes. So, Milt, I am going to ask you this question, "Will massively parallel computing environments ever be comparable to those of vector processors?"

*Halem*

Well, if I should take this question literally, I think it is safe to say that people are not going to give up an operation that already has a long heritage in mathematics, such as vector and scalar operations. The number of applications that currently are running on vector processing machines is almost exclusive with the exceptions of a few disciplines, perhaps computer vision or neural networking, and even those have been put on vector processors.

I think it is pretty obvious that vector processors will continue to dominate the computing market. But that doesn't mean that it will be exclusively a vector processing or serial market. There certainly is a place for massively parallel processing because we have a class of problems that will continue to require an architecture which is more suited to those problems.

So I think the problem domain will guarantee a place for massively parallel processing, but that vector processors will still continue to develop and expand and dominate the markets.

Let me mention a very specific problem domain area—problems related to the space program. We will be acquiring images in space with arrays of sensitive detectors. The use of massively parallel processors will be required to reduce the data and to analyze this high volume of data.

But that is not the public market, that is not the mass market. In the mass market, we will continue to have growing vector processing capabilities, especially as vector processors shrink down into the work station domain.

*Hwang*

To me, "environments" represents application environments, that is the environment a user will see where he or she will be located. The environment includes user interface, language support, software support, run time support, debugging systems, etc.

I would say that the concurrent processing growth in environment build-up will be much slower because it is a lot more difficult. Vectorizing compilers vectorize the inner loops of do-loops. The inner loop operation is an identical operation being vectorized. So that is easy to detect in a program construct.

You see a lot of vectorizing compilers around. You don't see too many concurrentizing or parallelizing compilers. I have seen only two. One is the Alliant, which has a FORTRAN compiler and that can do a "do across." That is a do loop spread into several processors. The other is the CMU Warp machine compiler that can partition a job and spread it around.

Until we have very intelligent parallelizing compilers, I really don't see that a massively parallel environment is there yet. It will grow, but it will be very slow.

*Parkinson*

We have a large community of people now who have by heroic efforts modified their sequential FORTRAN to run on vector FORTRAN machines, even though they were promised that that wasn't going to be necessary. They are not going to lose that investment. Their problem is not to solve a particular problem in physics. It is to run an existing FORTRAN code.

The commercial realities of that means that even after fifty years we will still have things which accept that kind of code. There will be manufacturers who will make a lot of money from the fact that it is human nature to not want to learn anything new.

I don't know when Newton invented the calculus, but when I was a schoolboy you still were able to take courses in physics which didn't use calculus. People don't want to learn anything new, and these things are. People keep talking about algorithms. I wish people would talk about problems.

Everybody says "how do you do xyz algorithm on SIMD or MIMD machines?" Nobody comes to me and says how do you solve such and such a problem on the machine. They always talk about method and that's the difficulty. If we could only talk about our problems, then half of this argument would go away.

When John Reif was talking he said he knows algorithms which are not suitable for SIMD machines. I don't know any problems which are not suitable for SIMD machines.

*Reif*

It is important for the audience to know that as far as the theory community seems to know, there are many problems out there that cannot be parallelized. It was not algorithms I was talking about, it was problems. That means that for a fair amount of code out there, you cannot expect that it can be parallelized.

It is not the algorithms; in fact, we are relatively creative about new parallel algorithms. That is what I do a lot, and I should be more optimistic. But the absolute truth, as far as people who have thought long and deeply about this, is that we will not be able to parallelize the world away. There are fundamental problems with that. Even if you have gigantic massively parallel machines, they will never reach their full potential for every problem.

*Person in audience*

Give us one example.

*Reif*

Sure, there is the example of the Boolean circuit evaluation problem. Given a Boolean circuit, the problem is to evaluate it. Suppose the circuit has n inputs, size polynomial in n, and depth d. No one knows how to take an arbitrary circuit and evaluate it at a depth significantly less than d. Therefore an arbitrary sequential computation cannot be parallelized as far as anyone knows. There have been very very bright individuals that have looked at that problem.

On the other hand, that does not mean that we are out of our jobs. It means that instead we should look carefully at scientific computing problems that do, in fact, have good parallel solutions. Many people in the audience have had major application areas where they have found beautiful parallel algorithms. It is both a positive and negative thing.

You shouldn't take the religion of parallelism too far. The realistic answer is, it does work in many, many cases, but by no means is every problem parallelizable.

*Uhr*

That is obviously true. It is a complexity argument in terms of the worst case.

On the other hand, surely anything to do with intelligence, anything that the brain does is massively phenomenalogically parallelized. I would imagine that most data base problems and most scientific problems are highly parallelizable. So in terms of the real world and the real kinds of problems that do interest us, that argument should be sort of tempered.

*Reeves*

Will parallel computing environments ever be comparable to those of vector processors? I hope they never sink so low!

As Guy pointed out, there aren't many vectorizing compilers that really look very neat even today, and they have had many years of development to get into the act. I hope we make vast improvements in software over the next 20 or 30 years.

One should imagine that there will be tremendous improvements in both vectorizing compilers and parallelizing compilers, and in ways of expressing problems. This will far transcend what we currently do today in FORTRAN, although of course I don't expect FORTRAN to die in any sense.

I am sure that in the future there will be much better environments. I see these environments more tailored to specific application areas. Today we don't program in raw FORTRAN, we program with packages, which is a very limited higher level approach.

I see much higher level approaches as specific application areas evolve. Then there may be systems and environments tailored specifically to those application areas. These environments may be ported to more than one type of machine architecture, even to heterogeneous collections of different computing resources.

So I think we should look to much better paradigms than those we are currently using. Ultimately we will get better environments, more of the kind of environments that we deserve for both massively parallel and vector machines, or whatever our computing resources may be.

*Reif*

I don't think there is anything natural about the computing environments around vector processors. It is true that it is fairly easy to port certain types of scientific computing code into these vector processors. In part, that is due to work by Kuck at Illinois and his group.

In that type of technology, Kuck takes recurrence equations, the FORTRAN code, unwraps it and develops a recurrence equation model for certain inner loops, and vectorizes those. In fact that can be done for massively parallel SIMD architectures. Within five years, there will be systems that can do that for SIMD computing components.

I would also comment that again, I don't think the vector environments, which are FORTRAN based, generally speaking, are anything magical. I think they will persist, but I think that in the software world there are other environments, C-based environments, Connection Machine C*, things like that, that look to me like they would be more effective in the future.

One example, this is actually a star LISP example. The fellow who gave the talk last night about computer graphics wrote his original code in half a million lines of vectorized code. He ported that onto the Connection Machine and, this is second hand, the new code was approximately one to two hundred thousand lines of code instead of 500 thousand lines of coding. So already, there exists parallel computing environments about which I think the community can be very proud.

So I am fairly optimistic about evolving environments for massively parallel computing. There is also a lot of interest in DARPA. They are really pushing in this direction. It is possible to have very flexible, very powerful primitives for massively parallel computing. What is not possible again, is just that you can effectively use all that parallel computing power in all possible instances.

*Schneck*

Dave Schaefer asked me to join the panel because I am an argumentative cuss. Since I don't want to disappoint him, this is my opportunity.

It seems to me there are two issues here that are getting confused. One is languages and the other is environments. Although they are strongly related, they are not the same.

The language issue deals with expressivity and the way in which we write an algorithm or a problem. Environments have to do with what I like to call the psychological issues. How we deal with the programming process. I think the psychological issues belong in a different conference; I think that environments belong in a different conference.

I think we are much more concerned with technical issues of parallelism, issues of how programs work, how algorithms work, how we express parallelism in a concise and direct way. Not whether we get ten lines of code or whether when we change one package or subroutine, we need to change another one. Not whether we need to do a recompile or a make. I think those are all things that are second-order effects.

In the high speed computing community, much more time and effort is given to the running of the programs than to the writing of programs. Down at the PC level, much more time and effort goes into the writing of the programs. I think our attention ought to be carefully focused on the running of programs and the speed at which programs run, not on, from my point of view, the mundane issues of how to write programs and how to do it effectively and efficiently with the smallest number of people.

*Steele*

I suppose I am going to take a very pragmatic position, not that the previous arguments have not been pragmatic. I just want to tell you that I am going to take an extreme on that. I think that the issue here is purely an economic one. A lot of investment has gone into program environments and compilers and other tools for the developing of programs for vector processors.

If the approach of massive parallelism proves to be effective, then over the long run at least as much effort is going to be invested in producing good environments there. The question is "how can we go about improving those environments?", and the other question is "will it be worth our while to do so?".

I think reasonable people may have differences of opinion over what fraction of the market, measured in dollars, or problems to be solved, or whatever, is going to be tackled through massive parallelism in preference to vector or other approaches. If that fraction turns out to be large, the investment will be made.

What needs to be done to improve those environments is to look at the necessary abstraction and metaphors that will make things easier. It is necessary to capture the standard pattern of doing things and because the approach of massively parallelism is still relatively young, we haven't figured out what all the standard patterns and idioms are going to be.

To connect that with the previous remark, Tony Hoare might choose to characterize a SIMD machine as being a serial processor with very wide words. I agree that is a perfectly valid description for some programming purposes, a very useful description. The question is,"is that the most useful description for all ways in which one might use the machine?" I think that the answer to that is "no."

Sometimes you want to think of the metaphor as programming many individual processors. Sometimes you want to think about wide words. Sometimes you want to think about organizing your view of the machines in other ways. We need to figure out what these patterns are and then capture them in an integrated way with the other tools in the programming environment, so we can deal with them easily.

*Uhr*

It has never been clear to me, and I guess to a lot of people, why two-dimensional array processors shouldn't replace vector processors. About '78 or '79, the DAP group tried to sell a lot of DAP's and didn't get very far, I think for quite irrelevant commercial reasons.

Probably there are also relatively small but crucial things that still need to be ironed out. The lack of fast floating point operations, that you finally have in the Connection Machine, is one of the things that sort of put massively parallel systems behind. Others are the rigidity of a fixed size array and the fact that the array is not an integral part of a larger system.

We have a very comparable situation with image processing where we have pipelining vector processors and we have array processors. They are both sort of flourishing equally now, although the vector processors are much easier to handle and more popular. I personally think that the array processors will end up ahead.

I am suggesting that image processing is one of the better shots for massively parallel systems. In image processing, we don't have what seems to be a problem for numerical processing, where a short pipeline is better than a long one. I believe it is Fujitsu that has a long pipeline which may not be too usable. The short pipeline gives more flexibility but, of course, it gives slower processing.

In a similar manner, a fixed size array, like the DAP 64 by 64, is too rigid for applications using arrays of different sizes.

*Schaefer*

We are running out of time. Let's see if we can get this panel to give one word answers to the question "Can progress in the use of massively parallel computing take place in a world dominated by dusty FORTRAN decks?" "Yes" or "No", right!

*Halem*

I think the answer is pretty obvious. Over the last two years, we've seen the size of this conference double. I believe there is every reason to expect it will double again in another two years based on what we see coming in the market and what the agencies are putting in. We live in the world of dusty decks, and we are still making tremendous exponential growth.

*Schaefer*

Thank you for your one word answer.

*Hwang*

My answer is "yes", but I want to give you an explanation. We were talking about parallel programing environments. There is a special issue on language compilers and the environments for parallel programming which appeared last month in the *Journal of Parallel and Distributed Computing*. If you are interested, David Kuck edited a special issue of eight papers. One describes how IBM is developing a "PTRAN" environment. "PTRAN" stands for "Parallel FORTRAN". Just for your information.

*Schaefer*

We are having massively parallel one-word answers.

*Parkinson*

Yes, the FORTRAN programmers are going to die out of old age.

*Reeves*

Yes, of course. FORTRAN programmers of today won't die so quickly, but they will die.

*Reif*

I think the dusty FORTRAN deck will stay around until we are quite old, but I think that massively parallel computing will be real exciting in the next few years.

*Schneck*

John has said it, but let me just remind you since no one has said it before. We don't know what the next language will look like, but it certainly will be called "FORTRAN".

*Steele*

Yes.

*Uhr*

No, in the sense that we will be developing into a new world with new kinds of computers to handle new kinds of problems, and the old world will continue to exist.

# INVITED PRESENTATIONS

# Languages for Massively Parallel Computers

*Guy L. Steele, Jr.*
*Thinking Machines Corporation*

I'm here to talk about languages for massively parallel computers. Programming languages is a very wide field and there isn't any way I can possibly cover everything that's been done. I'm just going to talk about the ones I know about, which is always a good policy. This is an outline for what I'm going to discuss.

First off, we'll discuss some common themes I've seen over and over again in many programming languages that have been designed for parallel computation. These are common themes about the way you organize control structures of programs, the way you organize data, and about communication patterns. As we will see through the course of this talk, these themes tend to pop up in different ways in different language design. They are common threads and serve as a taxonomy I use to categorize the different aspects of parallel programming languages. Once I've shown you these common themes, I'm going to give you thumbnail sketches of three different parallel programming languages. They happen to be ones with which I am particularly familiar because they are running on, or are in the process of being designed for, the CM–2 computer system with which I work at Thinking Machines. After I've sketched out these three languages, I'm going to try to pull them back together and do a thematic comparison of them so we can see how the themes fit in and how various aspects of these languages can be ccmpared according to these themes.

First, I'd like to present some principles of language design that are merely guidelines that I've found useful in trying to design parallel programming languages. The first rule is don't start from scratch. It is very important to build on existing knowledge and traditions partly to avoid reinventing the wheel. And it is important to take advantage of programmers' existing familiarity about ways one can do things in programming languages and take advantage of certain traditions. For example, the standard scientific notation of floating point numbers is a tradition and you just do it that way unless you've got a really good reason to do it some other way in your language. A particularly important kind of tradition to build on in designing parallel languages is a set of expectations we build up about how serial programming languages are designed. Even when running parallel programs, there are still large parts of them that will be sequential, or that can be regarded as sequential programs operating on each of many data items at once. So, it is important to build on that set of traditions. A consequence of that first principle, and one approach, is to start with existing serial languages that are well understood, and then judiciously make extensions for parallelism.

On the other hand, which brings us to the second principle, don't settle for a superficial patch. It doesn't produce a really satisfactory parallel programming language simply to take Fortran or C and add two message-passing primitives, or add semaphores or something like that, and call that a parallel programming language. Yes, it is possible to program a parallel computer that way, but it tends to feel very low level because you simply slap the two primitives on top rather than

CH2649-2/89/0000/0003$01.00 © 1988 IEEE

integrating them into the structure of the language. You ought to arrange it so that the features you add for parallelism fit in well with the personality of the language and are integrated with the various other features of the language so as to interact smoothly. The other important thing is you want to be sure that the features you add support important paradigms of parallel programming, and those paradigms are the subject of my list of themes.

Given those principles, a third possible decision one might make is to concentrate on the data rather than on the control structures, and this is particularly appropriate for massively parallel computations. There are different styles of parallelism: there are MIMD and SIMD; there is focusing on large amounts of data, there is focusing on large numbers of processors, there is focusing on small numbers of processors in the 10–20 range. These call for different styles of programming language. My interest happens to lie in the massively parallel arena, so most of what I have to say will address languages and topics in that area.

If one chooses to concentrate on the data, then a strategy one can adopt is to take an additional sequential programming language, add a parallel data type—whatever that means—and that may mean different things in different languages, but having added a data type, try as hard as possible to use the existing language operators rather than inventing a whole slew of new ones. In this way, one can take advantage of understanding on the part of programmers of the semantics of the existing operators. On the other hand, you have to ensure that those semantics extend smoothly and in an intuitive way for the new data type.

Now, let's look at these common themes: elementwise parallelism, replication, reduction, permutation, and conditionals. There is nothing terribly profound about them, this is just a list of useful things. However, I would exhort you to examine each item on the checklist when programming in a parallel language.

One obvious theme is elementwise parallelism. Typically, if you've got two arrays, you want to add the elements of the arrays, componentwise, for example. This is a case where you're doing computation on corresponding elements of arrays. But, the elements in correspondence aren't interacting much. That's one kind of thing you want to see. That happens so often, it is so pervasive in parallel programming, that it seems commonplace to mention it. Yet, you want to ensure that, because it is so commonplace, that it is easy to say in the language. As we'll see when we get to the thumbnail sketches, sometimes, it's not so easy to design languages that make that smooth.

Another example is replication: taking a small amount of data and making more of it. This breaks down into several cases: there is the one-to-many case which you might call broadcasting; there is the few-to-many case, which can be in either a regular or an irregular pattern. The regular case is sometimes called spreading (in Fortran, for example). The converse of replication is reduction, where you're taking many data items and reducing them to a few items or one item. Then, there is the question of how you do that reduction. You might take many data items, reduce them to one data item by choosing one and discarding the rest. Or, you might take the sum over many values or the product, or the maximum or minimum, or the greatest common divisor. There are any number of interesting reduction operators, some of which are more useful than others.

Then, there is the case of permutation, where the amount of data doesn't change, but you are rearranging it in some interesting way. That breaks down into dozens of subcategories, such as shifting of the Cartesian grid or arbitrary permutations controlled by a permutation vector.

By the time we get to substantive conditionals, it is a control structure issue rather than a data stucture issue. In fact, one can try to categorize other kinds of control structures as well, except I don't see a theme that pops up in any regular way in all the programming languages except for conditionals. It is very frequently the case that, based on the value of some piece of data, you want to make a choice of doing this or that computation. Sometimes, this is the most difficult theme to embed in a parallel programming language in a true way.

So, that is my list of themes. I'm now going to give you thumbnail sketches of three programming languages. The first is the particular dialect of Fortran running on the CM–2, which is not particularly specific for the CM–2. It is precisely Fortran 77 with the proposed Fortran 8X array features added in. But no other features of Fortran 8X. There was no particular reason to leave out the other ones, it was just an implementation decision made for the purpose of this project. When we set out to do Fortran at Thinking Machines, we were prepared for the possibility of having to do it our own way and invent a new language. That is something we are loathe to do when we can stand on the shoulders of some other giants. We were delighted to find, in fact, that the Fortran comittee, X3J3, has come up with a very reasonable and plausible design for a data parallel Fortran in the proposed array features.

This is going to be a very quick sketch of a parallel Fortran since there is no way that I can do justice to any of these languages in 10–15 minutes, or even in an hour. I'm simply going to show you some highlights of the language that will hook into the developments I want to show you later.

First of all, the Fortran 8X array extensions provide for elementwise expressions. If you declare three arrays (A, B, and C), each with 1,000 elements, they are said to conform since they are the same length. Because they conform, I can mix them in expressions. For example, I can write A = B. That means copy the entire process of b into a elementwise. So, that B1 gets aassigned to A1, and B2, to A2. Similarly, I can do elementwise addition. In A = B + C, the corresponding elements of B and C are added together and were both assigned to A. Or, I can do multiplication. like
A = B x C. This is elementwise multiplication. I can also use the built-in library intrinsics from the built-in subroutine library. For example, in
A = SQRT(B)/SIN(C), I can take the square root of every element of B, and a sine of every element of C, do an elementwise division, and assign the result to A.

Also, there is no reason why arrays have to be constrained to be one-dimensional. I could have taken matrices for five-dimensional objects, and as long as their corresponding dimensions matched, I could use them in an elementwise fashion. What this technique of language design manages to do, is to take operations on arrays and make them look syntactically like operations on single items. In effect, I have overloaded addition and multiplication and extended it to the array case.

Another important rule in language is scalar extension. Again, this is something that is so natural and so embedded into mathematical and programming notations, that it hardly seems worth remarking on. With scalar extension, you can take a scalar and mix it with an array within an expression. The rule is that the scalar is automatically replicated to match the array. For example, if A = 0, the effect is to replicate the zero so that there are enough zeroes to match all of the components in A. So that every element in A gets cleared.

Similarly, I can write a complex expression such as A = B/2 = C/(I+4), and it looks perfectly natural. I'd take every element of B, divide by 2, take every element of C, and divide by I + 4 (which is scalar), add them, and assign to the corresponding elements of A. As you look at it closer, you'll see that parts of this expression are scalar and parts are arrays. So, for example, the constant 2 is scalar. The addition operation (I+4) has scalar operands, so it will be executed in scalar mode. First, this scalar subexpression is computed. Then, the result is replicated, and then divided into C.

The Fortran 8X proposal provides for reduction by providing a series of new intrinsics such as SUM, MAXVAL, PRODUCT, MINVAL, COUNT, ANY, ALL, etc. In the simplest case, you can just give it an array as an argument and get a scalar value back. A more complex variant involves specifying an optional mask argument, the value of which is an array of logicals that matches the first array argument and specifies which ones are to participate in reduction. For example, in the case SUM9A,MASK=A.GT.00, it adds up only those elements of A that are greater than zero and ignores the elements of A that are less than or equal to zero. Also, it is possible to do summation over an entire array, along rows or columns of a matrix. For example, REAL M(100,100) provides a matrix of 100 x 100 elements. SUM(M,DIM=2) sums up rows. So, the result of either of these would be an array of length 100. Given a square, it produces a result that is either stored along the top or down the edge.

The converse of the reduction intrinsics is an intrinsic called "spread." Spread can take an array of some smaller dimension and replicate it along a new axis so as to make an array that's of a rank one higher. For example, in REAL A (100), M(100,100), I have a vector A of length 100 and a matrix, M. Then, M = SPREAD(A, DIM=1,NCOPIES=100) says: take A and make 100 copies of it. The new dimension is to be dimension 1 and should be assigned to M. Because the new dimension is dimension 1, the vertical direction, A, gets replicated and is used to fill in each row of M. On the other hand, if I had said DIM = 2, then A would be used to fill in the columns of M instead.

A very interesting feature of the proposal, which is in the category of removed extensions, is vector-valued subscripts. This is a feature that the committee has debated and both inserted and deleted a few times. It is one of those things that looks really important, but that might be hard to implement on some machines. We decided to include it in Fortran for the CM–2, because it seems to be a very valuable thing, and we have a reasonable implementation for it. Vector-valued subscripts can be viewed as simply allowing the subscripting operation to be componentwise in the same way that addition and multiplication are. For example, in

```
REAL A(100), B(593)
INTEGER V(100).
```

I have a vector A of length 100, a vector V of length 100, and a vector B of length 593. The assignment A=B(V) means that for every element of V, use its value to subscript. This produces a vector of results of the subscripting operation which then gets assigned to A. So, you can see why the length of V has to match the length of A because there will be one result in the subscripting operation for every element of V. The length of B is not relevant to the conformality property. Rather, the rule is that the value stored in vector V would have to be suitable indices for B. By using the subscript vector this way, you can encode fairly arbitrary patterns of data rearrangement. For example, V might be a permutation vector, in which case, you'd want B to be the same length of A. You can also arrange for few-to-many replications, since it might be that V has duplicate values. In that case, many elements of A might receive copies of the same element of B.

You can also use a vector-valued subscript on the left-hand side as in B(V)=A. In which case, elements of A will be assigned to places in B that are dictated by V. In this case, the language designers imposed the rule that no collisions are allowed. That is, it is forbidden for V to contain duplicate values. This avoids the problem of what happens when two values try to get assgned to the same location. That problem was solved in Fortran by fiat. We'll see later that in C*, it is solved in a different way.

Another of the very powerful removed extensions is the FORALL statement. This is another feature that has come in and out of the standard, and I'm not sure what its status is as of this week. It is a very powerful statement. You can say all kinds of marvelous things with it. It is effectively like a parallel DO loop (or at least a DO loop that is easier to parallelize than the standard DO loop, which has sequential semantics). For example,

```
FORALL (I=1:100)
  A(I)=B(I)*I
```

The body of the FORALL statement must be a single array assignment statement. That is an important restriction that is imposed because it eliminates the problem that can happen if you have multiple assignment statements in the regular semantics for "execute all assignment statements where I=1, then execute all of the assignments for I=2." Or, whether you execute the first assignment for all values of I and then the second for all values of I, or some other scrambled order. You avoid that set of problems by saying that there will only be one assignment statement within a FORALL.

FORALL allows you to solve a problem with the vector-valued subscripts where you have duplicate indices on the left-hand side. If you write a sufficiently elaborate statement, you can specify how to resolve collisions. For example,

```
FORALL(I=1:593)
  B(I)=SUM(A,MASK=V.EQ.I)
```

In this case, I tried to represent B(V)=A, but, where V may have duplicate values. If there are duplicate values, then I want the corresponding elements of A to be summed and have the sum of all the values acquired there put into B. What I am saying is: for I runs from 1 to 593 (which is the length of B, not A), B(I) gets the sum over A in the positions where V equals I. So, in effect, instead of using subscripting, I have used this calculated logical mask instead. Now, whether the implementation of that will be as efficient as what you would expect from B(V)=A is a question that can only be decided by looking at the implementation and the architecture. Some architectures will do a much better job of this kind of thing than others. So, it is possible to say it in the language, but whether or not it is an effective statement of what you want, one that will be efficiently executable, depends on your implementation. This points out the difference between expressiveness and effectiveness.

There is a parallel condition statement that is very much like a parallel IF. The syntax is as follows:

```
WHERE (A.NE.0)
  B=B/A
  C=C/A
ELSEWHERE
  B=0
  C=C*3
END WHERE
```

In this example, I am saying, in positions where A does not equal zero, you can divide both B and C by A. In all other places, you execute the other statements in the corresponding positions of B and C. Again, there is a restriction. The statements that are controlled by the WHERE statements are only permitted to be array assignment statements, and all of the arrays assigned to it must conform to the array logical expression that appears as the predicate of the WHERE. So, it better be the case that B and C conform with A.

A very important observation of this language design is that very few of these features matter. If you don't use these array features, if you don't use the array expressions and the reduction transitions for work, then the language looks like plain old sequential Fortran—it's completely upward compatible. So, any Fortran 77 code is, in fact, CM Fortran code, and also full Fortran 8X code. That is a nice property of the design.

Okay, now let's take a look at C*. C* is a parallel dialect of C that was developed at Thinking Machines Corporation, but which seems also to be in use elsewhere in the world now. In fact, I think there is a project at the University of New Hampshire that Michael Quinn is involved with and he is implementing it on, I believe, the NCUBE machine. A language isn't real until it is running on more than one machine. So, C* is at least real in that sense, if not in other senses.

Okay, now, let's look at a particularly nasty language design problem. We would like to have scalar extension in C*, much the same way that we did in Fortran because it is so convenient. But, suppose that X is the name of an array. What does X+1 mean? Well, unfortunately, that already means something in C. Because in C, unlike Fortran, arrays are pointers, and to have an array with a bunch of things is the same as to have a pointer to the first element. In C, X+1 already means to do pointer arithmetic. It means take the address of X (which, if it is the name of an array, then it is a pointer), and increment that pointer by 1 to point to the next element. So, if you just say, "Well, we'll just let that mean add 1 to every element of X," it won't work.

How do you get around that? This is a problem that I worried over for several months and just couldn't see how to do it. Then finally, we came up with a solution that sort of goes in the back door. Instead of trying to wedge the parallel data type in using C arrays, which won't work because arrays coerce to pointers, we introduce the parallel data type in another way and then later add the necessary "arrayness." This was accomplished by adding two new storage classes to C that describe where the data reside. The keywords used are: "mono" for scalar data, and "poly" for parallel data.

Poly data are organized into domains. For example,

```
domain particle {
float x, y, z;
float mass;
float vx, vy, vz;
};
domain particle w[10000];
```

Each particle can have a bunch of components. Here, I have chosen position, velocity coordinates, and mass as a demonstration. Within a declaration of a domain, given that declarations are implicitly poly, I could have written the word poly explicitly in front of each of these declarations. In fact, the words mono and poly tend not to pop up too much in actual C* code simply because the defaults are arranged so you get what you want by the time that you write it. Since these are implicitly poly, these will represent parallel data. There is an essential rule about domain arrays. When you make an array of domains, the result is to get something that can be processed in parallel. In the above example, we have an array of 10,000 particles called W.

When you select a domain, you are activating parallel processing. It is as if for every instance of that domain that you have declared, there is a separate processor that can execute code. To select a domain, I would like to introduce a new statement type (square brackets around "domain particle" with the body enclosed by {} and a . in between). Within the selection statement, it can again be arranged so that parallel code looks like serial code. This is the nice property that Fortran also had, but it was achieved in C* by alternate means.

Once I have selected the domain of particles, all of the data that are declared to be around in that domain (such as x, y, z, mass, vx, vy, and vz) can be referred to as if they are scalar values. So, it is as if you wrote code within a selection statement that is to act on each particle independently. In computing this, I can declare a new variable which temporarily becomes part of the domain. In the example:

```
[domain particle].{
  float v2 = vx*vx+vy*vy+vz*vz;
  float k = mass*v2*0.5;
      .

      .
  if(k>1.0E9) blooey0;
```

a new automatic variable, v2, is allocated for each particle. For each particle, I calculate vx squared plus vy squared plus vz squared and store that into v2. Then, I can compute the kinetic energy by multiplying mass by v2 by 0.5. Notice the implicit use of scalar extension here. If I simply use a mono value here, then there is implicit replication. I can also use all kinds of sequential program constructs in here, such as IF statements. I can use, in fact, any C control structure within the parallel code. Now, how all that works out is really tricky, and I will address that in a few minutes. In the preceding example, the function, blooey, will get called only on behalf of particles whose energy k exceeds 1 billion.

So, within a domain, code is parallel. It looks just like serial code except that if you mention poly quantities, the effect is to get elementwise operation. Now, if you use mono data in parallel code, it results in broadcast. This is how you get X+1 to behave as expected, because in some sense, X is not the name of an array in this theory of data types. Rather, X is the name of a scalar that is, however, declared as a poly value within an array of particles. So, X is technically not an array—it is a scalar value with some "arrayness" hiding outside having to do with the domain data type. So, this is how we do the end run around the array problem.

An interesting consequence of all this is that we find the reduction operators are already there in the language in the guise of compound assignment, which is a set of operators that Fortran does not have. If you use a scalar value on the left side of an assignment and a parallel value on the right side, you can get lots of assignments to happen at once to the same place. In this example:

```
mono float total-mass=0;
[domain particle].{
  total_mass +=mass;
}
```

I have declared a mono variable called total_mass and initialized it to zero. Then, I activate all particles and ask each particle to add its mass to the total_mass. This requires a new semantic rule which is that in cases where you have side effects like this that collide in single locations, you have to have a rule that says it is okay, and defines what the result is.

We have chosen to state the rule as simply: the assignments happen as if in some serial order. That is one way of resolving the conflict. Of course, you don't implement it that way, you implement it by making a binary tree, for example, and doing the summation in logarithmic time (or some other technique). The net effect from the language point of view is that all of the masses get added into the total_mass. Since C provides these built-in compound assignments for all kinds of operators, there is multiplication assignment, OR assignment, AND assignment, etc., and this gives you pretty much all of the standard compound operators. C does not have MAS and MIN operators built into it, so we added those into C* purely so that we could get compound MAX and MIN assignments so that we could get this reduction effect because it is so useful.

The fact that arrays were pointers was a decided disadvantage in trying to get array features into the language. However, there is another part of the language where you can, in fact, turn pointers to great advantage. Pointers in C are perfectly general. For almost any data object, you can make a pointer to it, and then pass that pointer and assign through that pointer or reference through that pointer. C* simply carries that over directly. If you think of each particle as residing in a different processor, then a consequence is in effect that one processor can have pointers to the memory of another processor. Every time you do a pointer indirection, that is potentially interprocessor communication.

In the following example, let's suppose that every particle has an additional component that is a pointer to some other particle called nearest (assuming it will have a pointer to the particle that is nearest it). We can then do such things as having every particle compute the distance (dx) between it (in the x direction) and its nearest neighbor.

```
domain particle {
  domain particle *nearest;
};

[domain particle].{
  float dx=x-nearest->x;
          .
          .
}
```

You basically say: assign to dx my x and subtract from that value obtained by taking the pointer to my nearest neighbor, indirecting through it, and selecting its x component. You do not necessarily have to have pointers to nearest neighbors, you can have any kinds of pointers you like. This is one of those cases where identical-looking statements in the language, depending on the values of those pointers, can have radically different implementation consequences. If you have an architecture that supports nearest-neighbor communications particularly well, but supports general communications fairly poorly, then it is not surprising that nearest-neighbor communication will be faster than the general case. However, that is from the point of view of the implementation. From the point of view of the language, all pointers are alike.

Also note that if one combines the idea of using compound assignments to do reduction, this allows the possibility of doing many-to-one or many-to-few reductions.

```
[domain particle].{
  float nearmass=0;
  nearest->nearmass+=mass;
          .
          .
}
```

If you have every particle create a new particle nearmass and initialize to zero, and then have every particle add its mass into the nearmass of its nearest neighbors, then you will end up with a complicated pattern. Depending on how you precompute the pointers, you will often end up with complicated patterns of many-to-few reductions.

In C*, all control statements may be used in parallel code. The simplest statement is the IF statement where E is a poly value.

```
    if (E) S
    if (E) S1 else S2
```

In particles in positions where E is true, you execute S, in particles where it is not true, you don't execute S. And similarly for the if-then-else statement. So, these work as expected. Furthermore, the generalized semantic theory gives "while" statements a reasonable meaning, and even "break," "continue," and "goto." It is beyond the scope of this talk to describe how all of that works, but I can refer you to a Thinking Machines Technical Report that describes it.

A final observation is that if you don't use domains at all, then C* code looks like plain old serial C code. You can take any C code and run it through the C* compiler, and it is a valid program. It executes serially and behaves like any ordinary C code.

Now, I am going to take a look at CM-Lisp. This is the highest level and most abstract language. It is an attempt to take the symbolic programming language, Lisp, and cast it into a parallel framework. It has always been the cultural attitude of Lisp that it tries to abstract a good deal away fronm the details of the machine hardware and to provide not just numerical objects to compute on but, in fact, abstract symbolic constructs. The first-order strategy is very similar to what was done in Fortran and C, which is to introduce one new data type. This then gets operated on in parallel. However, the data type has a very different nature. In keeping with the perhaps puckish sense of humor on the part of the Lisp hackers, this data structure is given a strange name. It is called a "xapping."

A xapping is an unordered set of ordered pairs. Each of the two elements in the pair may be any Lisp object. Lisp objects may look like numbers, atomic symbols (look like identifiers), and lists (ordered sequences of Lisp objects). In the case of xappings, we take a sequence of pairs and write them between braces:

```
{sky->blue apple->red grass->green}
```

A pair is written as the index object, then a right arrow, then the value object. The preceding is a xapping of three pairs that map sky to blue, apple to red, and grass to green. The indices may be any Lisp objects, but they must be distinct, and the values may be anything.

[Editors Note: the details on CM-Lisp have been removed. If you would like further information on CM-Lisp, refer to Thinking Machines Technical Report PL87-6.]

Now, let's do a thematic comparison of the languages: Fortran, C*, and CM-Lisp. First, let's look at elementwise parallelism. That is achieved in Fortran simply by making two arrays be an operand to, for example, the + operator (A + B). That does not work in C* because addition on arrays already means something else. Instead, you have to write A + B in the context of parallel code: within a selection statement using poly data. In the case of CM-Lisp, A and B are xappings, so instead of using +, you have to use alpha+. You have to explicitly say that you want the + distributed over it. The reason for that difference is that CM-Lisp, unlike the other two, allows nesting of symbolic data.

For broadcasting, which is one-to-many replication, Fortran and C do it by simply having a built-in rule about scalar extension. In the case of Fortran, it is when scalar data meet an array. In the case of C*, it is when mono data meet poly data, the mono data get replicated. CM-Lisp has an operator for that. It is the job of alpha to do that replication. In the case of many-to-many replication, there are widely different mechanisms. Fortran can do irregular communication by using vector-valued subscripts. A particularly interesting regular case is accomplished by the SPREAD intrinsic. It turns out that both these cases are accomplished in a single syntactic way in C*, which is that you compute a pattern of pointers and then do pointer arithmetic (effectively vector-valued subscripting). In the case of CM-Lisp, regular communication can be accomplished with alpha because you can apply alpha to a xapping to make a xapping of xappings. Again, the concept of a nested data stucture actually simplifies things here. You get irregular patterns by doing alpha aref where aref is the Lisp subscripting operator.

In the case of reduction, I have broken it down into many-to-one and many-to-few reductions and in both the regular and irregular cases, because there is an interpreting pattern here. Many-to-one cases and regular many-to-few cases are both handled in Fortran by different cases of the SUM intrinsic (depending on what arguments you give it). To get irregular many-to-few reductions, you have to use some complicated form of the FORALL statement. In C*, all three of them are accomplished in much the same way by using compound assignments. The difference between them depends simply on what you use on the left side. If you use a mono 1 value on the left side, you get a many-to-one reduction (m+=x). If you use a poly value on the left side particularly involving pointers, you get many-to-few reductions and regular and irregular computations look the same (*p+=x). In the case of CM-Lisp, the data operator handles all three cases. If you want many-to-one reduction, you only have to supply one argument. If you want complicated values, you supply two arguments and you get the xipping effect. A peculiar thing here is the strange pattern where each of the languages has two cases the same, but Fortran does not have the same two the same as the other languages. This points out a difference in the styles of language design.

Now, let's look at permutations. In the case of Fortran, there are some built-in intrinsics for doing regular permutations. In the case where you have rectangular arrays (which is what Fortran is good at), there are intrinsics to do circular shifts along any axis, end-off shifts that shift in zeroes and discard shifted-out data, and a matrix transpose operator. In the case of C*, you just do it with general mechanisms again by just computing the relevant pointer pattern and hoping that the compiler will recognize a special case pattern or that the underlying hardware/software will deal with that case as well. You can take these separate cases and bury them in macros to make them linguistically convenient. In the case of CM-Lisp, one can also use calculated indices, such as in C*, but there are also intrinsics that we built into the language to handle interesting special cases.

As for conditionals, I am going to contrast the scalar and parallel case. Fortran uses the IF statement to do scalar conditionals, but it uses a separate kind of WHERE statement that is otherwise syntactically very similar except that it has restrictions as to what you can put in the body. I think that is why they chose a different key word, namely the WHERE statement. Certain kinds of conditionalization are accomplished with a special mask=argument in some of the intrinsics. In the case of C*, the same if statement or the same conditional expression serves in both the scalar and parallel cases. It is merely a matter of whether you use mono or poly data. In the case of CM-Lisp, again, the if construct serves for both cases. However, you have to distinguish whether you want the scalar or parallel case by the explicit use of alpha. That has to do with the nesting of data structures.

Finally, let's compare the languages by looking at the data structures. Fortran provides arrays and the elements are scalar. At least in the dialect I described, which is Fortran 77 with the 8X array extensions. Fortran 8X also provides many other kinds of data structure constructs including record structures. So, for full 8X, the following restriction does not apply. For the dialect that I discussed, there are only scalar elements and the indices are strictly integers. C* gets its parallelism by introducing domains and then there is a funny thing that arrays of domains are special. You can select and activate them. Or rather, you can activate their domains, which causes the array elements to become active. Again, the indices are restricted to being integers. In the case of CM-Lisp, we introduce xappings, which you can think of as being very much like arrays except the indices and elements may be any Lisp objects. This makes it fit in better with the symbolic computation nature of Lisp.

So, to wrap up, each of these common themes having to do with elementwise operation and replication or reduction showed up in all of the languages. However, they got realized in very different ways because they had to be embedded in such a way as to suit the personality of the language and the standard cultural way that things are done in that language. I think it is important to take that into account when doing a language design.

I would like to point out some patterns that I think will be important in the future. Themes that I think will become increasingly common in the future include the parallel prefix operators. These are what APL calls a "scan operator." Not only just the simple case of taking an entire array and doing parallel prefix, but also taking an array and breaking it into pieces and doing parallel prefix. Some work has been done in this area by an MIT student, Guy Blelloch, on models of computation that are based on doing segmented scans. There are some fairly powerful ideas on how to build up algorithms that way. I think that sorting is going to become increasingly important and we have to understand better how to use that in a language context. The ideas of convolutions are also important, including simple ideas like taking an array, looking up/down/left/right, adding them up, and dividing by four, which happens all the time, as well as more complicated patterns of convolution. Here, we not only need to understand these patterns and provide them as facilities, we need to do more than just provide subroutines in a library, we need to understand how to integrate these patterns into the language designs. They need to fit in smoothly with everything else that is going on.

I think I've raised more questions than answers in giving this talk, but these are what I see as important topics for the future of parallel language design.

# Generating Movie-Quality Animated Graphics with Massively Parallel Computers

*Gary Demos*
*Whitney-Demos Productions*

I'll start by posing some challenges to the audience. I'd like to challenge you to push the state of the art in scientific computing. Graphics, as a discipline, or, as a software problem, really acts very much like many scientific problems, and tends to stress a lot of the directions of the general use of the machine, and the general performance capabilities.

We found with the CRAY X/MP that the vector scalar balance was pretty good for doing graphics. And, scientists, I think, have found, in general, that it was a fairly good balance for their codes.

We're beginning to see a lot of scientific and engineering applications running on massively parallel systems, and, it's a challenge to see just how far we can take that. I think the potential for massively parallel machines to go way beyond their present power level is quite large.

Let's talk a little about the topology of massively parallel machines. It's obvious that two-dimensional grids are quite useful for some problems. However, I'm sure a lot of people here have run into those problems where you have to do funny things to get the grid to solve the problem. In many ways, scientific problems are modeling three-dimensional space. At least, most of them are. And, most of them are beginning to model timeframe problems, as well.

For this you want a 3-D, maybe 4-D—typically not more than 5-D—kind of grid. Now, obviously, having eight directions is more helpful than four in the sense of a 3-D grid. If you have a hypercube that's got 4,000 nodes, you essentially have 12 wires at each cell of the hypercube.

A 12-wire hypercube also has, by its attributes, 12 nearest neighbors. So, if you think of this as the dimension, it's really something like 3.5, Now, $2^{3.5}$ is 12. I'm not sure whether this is three or four dimensions.

If you think of the cell dividing space as being a cube, it really has six faces. It doesn't have four. It doesn't have eight. Many three-dimensional scientific codes are based around cell subdivisions.

This brings up the whole issue of how to subdivide space and map it onto massively parallel topologies. If you're thinking of a 3-, 4-, or 5-D kind of a space-filling topology, what you really need is spaces that can be close packed. And, of the regular polyhedron, in which all the faces are the same, and all the angles are the same, there aren't very many such things. And, those that there are can be kind of strange. Cubes tend to be good ones.

Three dimensional shapes that can be close packed are tetrahedrons. But, I don't know really how you solve a tetrahedral composition of 3-D space. I'm sure there are other irregular polyhedrons that are made up of two classes of faces, e.g., a square and a triangle or, maybe, some asymmetrical shapes. But, these are very difficult to construct scientific codes around.

Anyway, I think this is part of the challenge. Obviously, a general purpose routing capability is one way to solve the issue.

CH2649-2/89/0000/0015$01.00 © 1988 IEEE

But, there are natural topologies in three dimensions that deal with some of these issues.

Another problem is local subdivision: if you have a grid around an aircraft, and you want to have finer resolution around the fronts of the wings, it's nice if the grid can subdivide in those regions without leaving gaps where pieces are not connected.

One of the big topics for several years has been SIMD versus MIMD. And, it might even be possible to think of some hybrids. I heard some discussed today where each processing node has some limited computing capability, but, in general, the whole thing can turn over as a unit. These things remain to be investigated.

For graphics problems, the main issue really is numeric power. And, it's easiest to deal with that numeric power with floating point operations. Let's just think in terms of this scientific problem and being a powerful machine and just in terms of floating point power. I realize there are interconnectivity issues, and other issues.

But, let's just think of it in terms of floating point for a second. Let's say that we have a goal of obtaining a machine that was 1,000 times more powerful than an XMP processor. Now, obviously, that's something that a lot of people would be excited about. Let's just figure out what that might mean.

If I have 4,000 processors, that means that each processor has to be 25 megaflops. Well, that's a lot of megaflops. This is, by the way, taking the basepoint that an XMP is about 100 megaflops. And, I realize it can vary anywhere from 10 megaflops to about 200, depending on how your code's written.

But, let's just say it's a 100. So, I'm giving it a pretty good peak rate there. So, this is—1,000 times that is about 100 gigaflops. So, 4,000 processors with each processor at 25 megaflops—you could do it with 16,000 processors where each processor had 6 megaflops. That might be a little more possible.

You could do it with 1,000 processors where each had 100 megaflops. Now, that sounds hard. You could do it with 64,000 processors where each one had a megaflop and a half. I think it's somewhere in this space. Now, I don't know how to optimize this in trying to find the peak point, where you'd get the most bang for your buck, if you will. I don't think scientific codes care that most of them have million-node meshes. Certainly, graphics doesn't care.

If you have a 1,000-x-1,000 screen, that's a million picture elements. If you want to put a million polygons on it, that's another handy million. You get 1,000 frames per minute, roughly, within an order of magnitude, so, you could do 1 billion things in parallel and just compute 1 minute's worth. So, you can go pretty parallel in that sense.

One of the issues facing everybody, of course, is 32 versus 64 bits. Some of the scientific community absolutely insists on 64 bit. Other people seem to be willing to work with 32. This tends to create shop cultures. Certain shops are 64-bit only. Other shops are hybrids, or 32 bit, or whatever.

Graphics can be done on either, depending on the algorithm. Most graphics algorithms can be done pretty well with 32 bits with an occasional 64-bit computation in a couple of isolated spots. But, I think a scientific machine really has to have high performance in the 64-bit range to satisfy the broad number of users. Because, clearly, there are a number of machines out that are largely parallel. And, each of them is serving certain niches.

But, if you want a general machine serving most problems, I think you'd want to think in terms of 64. That's just my personal bias. In the work that we did, and, I think, also in scientific work, the 1-bit granules are not used. In some sense, they're good for other kinds of simulations than scientific or graphic type work. So, in some sense, I thought of that machine as being a smaller number of floating point chips as opposed to a larger number of 1 bit processors.

Of course, some of the problems you encounter in programming massively parallel machines is how to track the state, measure its performance, and evaluate the available tools. Some presentations on these issues were made earlier today. And, something, I think, is pretty interesting — I think if you use graphics to display parallel variables where you have a large number of them and you can put them in some coded way on the screen where you can watch the machine and similarly use it to display utilization, router activity, grid activity — those kinds of things that it can be used for.

At the other end of the spectrum—not the high end, but the lower end—when we go home and play with our personal computers, or we get something on our desktop for our office — there's a trend toward rapidly increasing power. There's really two reasons for this. One is the RISC technology seems to have pushed the performance of small machines, and these machines are starting to become something that's moving towards our home and our desktop. But, also floating point power becomes much cheaper, particularly, pipeline floating point.

And, it's even possible to get parallelism in a Macintosh II or a PC. You can put boards in there that will accelerate you up to significant speeds. And, it won't be very long before you have the power of a CRAY 1, which is about 50 megaflops, on your desktop for about $10-15,000.

There are some interesting things going on in our field—revolutionary things. I'm going to digress a little bit and talk about our experiences with the CRAY X/MP. We wrote an algorithm for graphics that ended up being about 500,000 lines of code, which, by my measures, is a large code.

The algorithm averaged about 500,000 polygons in a frame. That's a lot of complexity. A polygon is a unit of surface. It's a piecewise linear spatial way to approximate 3-D images. We typically approximate them by their surface because one typically doesn't see inside of things unless one is refracting light through crystals or something. We typically made frames at 2,000-x-2,500 pixels with 36 bits per pixel—that's 24 megabytes. So, we pushed a lot of data in each frame.

We made some frames that are 5,000-x-4,000, or 75 megabytes. We made use of the 100-megabyte-per-second channels. People are beginning to explore this. Doing graphics has a lot of bandwidth requirements. Feeding the tube requires a lot of bandwidth.

Just to give you a sense of how much computing is involved, in *The Last Starfighter*, we did about 30 minutes of film, and we had to compute on a CRAY–1 for about 6-9 months, and we were backed up with about 2 years' worth of computing.

We brought in the X/MP and then computed on it for about 6 months. If you just figured in two processor X/MP time, it was almost 9 months of solid computing—that's a lot of cycles. And, we were producing about 1.5

hours of imagery per year for an average frame time of about 2 minutes/frame.

We found that there is a perceptual threshold for interacting with complex pictures of 5 minutes. If you can make your frames in less than 5 minutes, everyone's productive. If you take longer than 5 minutes, nobody makes any progress and the production grinds to a halt.

We also find in computer graphics that an issue such as volume visualization, where you have a big field of 3-D images and you want to see what's going on, that the data can be very complex. You may be looking through many layers. And, it requires the same high levels of computing such as the kinds of graphics that we've been working on for entertainment.

We've declared parallel variables on the CM–2 by creating an operator,which is really a 1-character prefix on our parallel variables. We used exclamation points with the data we processed on the CM. So, every parallel variable (P-VAR) was "bang" something or other. We called the renderer the "bang render," the thing that makes the pictures. But, anyway, I think it's my vote that is counter to everything that everybody's doing to make the parallel variable jump out at you by having them look different.

I think it makes the code easier. I think the whole general trend toward having languages hide the machine's activities is wrong. I think it's much better to just bring it up and let you control this machine. You've got to think in terms of these massively parallel machines, anyway, or you're not going to get to square l. Nobody's going to throw a normal FORTRAN program on these machines. Or, if they do, they're not going to like it.

We made use of *LISP, which is the parallel LISP dialect for the CM and used the symbolics front end. For me, who had been programming in FORTRAN for many years, to change from normal, vectorized FORTRAN, to parallel LISP was a pretty radical culture shock, but, I'm alive to tell you about it.

However, not very many people are willing to go through this kind of psychological lobotomy, or whatever it takes to redesign your brain patterns. Programming the CM is like solving 3-D puzzles. There are people who are good at it, but, I'm not one of them. My work cycle on the CRAY was typically as follows: I would think about something to code up, e.g., a couple of pages. I would think about it for a day. I would code it for a day. And, I would go debug it for a day. On the CM, I'd think about it for about 3-4 days, code it in about 2 hours, and debug it anywhere from half a day to a week. Some of the problems were very hard to debug because it was difficult to see the state of the machine—what it was doing.

You had to build á lot of test cases to find out what the processors were sending or getting or whatever it may be. Anyway, there are different coding styles. It's a different lifestyle, if you will. It's still in the middle of the night for most of us. But, other than that, it's a different function.

On the CRAY, we used precision film-recording scanning systems that use cathode ray tubes. With the CM, we were targeting mostly commercial production, and we used "digital video." There's a broadcast industry standard now called "422 Component Digital Video," for short, it's called CCIR 601. This particular standard is a way of sending numbers to represent the picture. It's a vast improvement over analog signals, where they're always drifting around on you. And, there are media that are kind of interesting peripherals that support this digital medium. One is a disk system that holds 100 seconds, and you can random access it. It's made by a company called Abekas.

There's also a tape machine that will store this visual video made by Sony. It's called a DVR 1000; the format on the tape is called Dl; but, the data are transmitted at 20 megabytes per second—a healthy data rate. It's over an eight-bit-wide differential ECL signal level cable. And, this 20-megabyte-per-second stuff can be laid down on a tape. Well, that's a hell of a tape machine.

Each tape is a little cassette that will hold either 35 minutes; a slightly larger one will hold 75 minutes. That's 50 gigabytes! A little tape of 50 gigabytes transfers at 20 megabytes per second, and has a disk that goes with it. That was pretty exciting stuff. Our connection to it was over the Ethernet. Oh, well, at least we've preconditioned all the data. So, it ran 100 Kilobytes per second, or whatever the Ethernet would go. We were going to hook up a VME, but didn't get around to it. We also made some use of the HBTV standard, which is the high definition video spec. There are people who have produced systems for it including Phillips in Europe and Sony and NHK in Japan. Anyway, it's 1,125 lines. They account for the lines that scan back that you don't see. It's really a 1,040-lines visible system.

But, it's not a square screen or even a 5-to-4 type screen. It's a 1.6-to-1 aspect ratio which is the old European movie format. So, it's 1,040 lines-x-1,660 pixels. We made some use of this. We actually used nonsquare pixels and used 1,280 lines and stretched them.

We were not at the optimum performance on the CM that we would be if we just kept pounding on our performance and our software. On the CRAY, we figured that we'd put 8 man-years into the algorithm and 16 man-years into optimizing it. We had put the 5, or whatever, man-years in the CM; but, we hadn't put the subsequent 10 in the optimization, so, we had a long way to go. But, among the things that were—that we could see to do to improve ourselves—is that we had about 35-percent efficiency based on the front-end issuing instructions. It wasn't keeping the CM busy. It wasn't issuing instructions fast enough. We would have benefitted a lot from the later CM operating system release where they have something they call "variable virtual processor ratios."

The way that that machine uses its floating point chips and pipelines, it does not get very good floating point efficiency in terms of its peak unless you use high virtual processor ratios. We had to use a virtual processor ratio on our code of 1, meaning we just barely fit with the number of temporary variables. We have a large number of temporary variables. But, we could've had a few variables that would've fit in there and we could've used large virtual processor ratios for that and gotten significant improvement—probably a factor of 2 on our code.

We also found that we used mostly many-collision routing, that is to say, lots of people wanted to get from or send to the same place. Let's say a processor was a pixel. And, at some point, every processor's a polygon. And, it says, okay, I'm going to see what pixels I touch. And, it's going to start to scan itself out. As it does, it's going to send to some other processor that's a pixel. So, it does that. And, now every processor puts on a different hat and pretends to be a pixel. And, it sees who all the polygons are that touched it.

So, we made almost no use of our graphics hardware on the news grid. It was almost all general routes with lots of collisions. So, for any of you who are designing machines and are interested in complex graphics algorithms, that's our experience, at least for the kind of algorithms I design.

Others may design other kinds of algorithms. We also used indirect addressing. That's how you build things on the list that are different length, where every processor has a different length list. It was fundamental through our design. We also used some of the more exotic operations that the CM provides in *LISP—scans, ranks, and enumerates copy scans and segmented copy scans, etc.

We made lots of use of them. Everytime they would say "here's something funny and it does something funny. Is it useful?" we'd say "yeah," and we'd stick it over here. We made lots of use of all that and, it became important for performance.

Another thing we did was adopt a concept that has been bouncing around known as data flow in the industry where you group pieces of work to be done together. If you're going to turn off 90 percent of your processors and only have 10 percent active, that's not very good use. But, if a whole bunch of different things need the same—let's say, a whole bunch of users need a multiplier, but each one would only be 10 percent active if you did them separately.

So, you'd collect them all together. You'd do the multiplying and then, send them all back. Now, multiply isn't a big enough unit of work to justify all that sending and collecting, but, we had units of work that were similarly small which meant entirely different things because they were from completely different parts of the code. But, we would collect them up, do them all in parallel and send them back. And, that turned out to gain significant speed.

# The Impact of Massively Parallel Computers on Image Processing

*Azriel Rosenfeld*
*University of Maryland*

The use of massively parallel computers for image analysis was first suggested in a paper entitled "A Computer Oriented Approach Toward Spatial Problems," which appeared in the proceedings of the IRE, if I remember correctly after 30+ years. It was suggested that if you took an image and loaded it into a bit and then connected an array of processors—one pixel per processor, perhaps—you could do an awful lot in a small number of computational steps using pixel parallelism. I want to start off from that baseline and say something about what kind of things we want to do with images once we get them loaded into a massively parallel system, and how hard some of those things are going to be. We know how to massively parallelize some operations; but, we don't necessarily know where the bottlenecks are. I have my own prejudices, and I'll comment on them as I go along.

Let's look at real-time vision. You point a camera at a scene. You get a video image, grab a frame, digitize it, and now you have this massive data set, but the frames keep coming at you. You're dealing with perhaps many megabytes per second, depending on frame rate and size. One of the serious limitations in our business is that you often have to do things in real time. These limitations have no meaning, however, if you get an image every day from somewhere out in space, and you are willing to take 24 hours to make some decision about it. For most systems, we don't have that kind of luxury. In most real-world applications, for example, if you are trying to create eyes for a robot that actually has to move around and manipulate things in real time, then implementing what the system has to do in real time at low cost is a challenge.

Let's go into the factory and look at the industrial machine vision systems that are being sold. They don't work with massive parallelism, because massive parallelism is not something you can put on a chip for a few thousand dollars so that you can install them economically. So, they work with the sort of simple operations that you can perform sequentially more or less at frame rates, which tremendously limits what they can do. The things that computer vision researchers were inventing, exploring, and developing 20–30 years ago, you can now do in real time on a single-processor system.

Massive parallelism offers the tantalizing promise of being able to do less trivial things in real time once the cost of massively parallel systems comes down. The axiom is that computer power is getting cheaper, and this trend will continue. We are far from being up against a stone wall.

There are experts on hardware here who will tell you how soon you'll be able to buy a 1,000-x-1,000 mesh-connected system for $1,000. It'll be awhile, no doubt. But, unless the economic pressures for doing it go away, it would surprise me if, by the turn of the century (plus or minus a few years), we weren't within shooting distance of that target, if not already there. This implies that although we keep coming up with ever more diabolical ideas about how to torture

CH2649-2/89/0000/0021$01.00 © 1988 IEEE

data and squeeze drops of wisdom out of it, anything we can do today will eventually be done in real time at low cost. This is my underlying hypothesis.

What types of data do we need to process when we try to do computer vision? What stages do we go through from the time we get the image into our retina (so to speak) until the time we are able to do things? What types of operations do we need to perform on these types of data, and in what way can we speed up those operations using parallelism?

It's fashionable to say nowadays that computer vision has two major goals: constructing a map of the environment, and recognizing the objects.

The catch for goal 1 is that the environment is three-dimensional (3-D), while the camera image is two-dimensional (2-D). To map the environment, you need surface topography. If I pointed a camera at this room, there's a lot of depth here, lots of objects occluding one another, and they are very complex objects.

With regard to goal 2, recognizing the objects, lets look at the people in this room. Can I recognize and count them? Can I tell the men from the women? Can I distinguish which ones are wearing eyeglasses or beards? Such recognition tasks are beyond the state of the art. You might come up with a technique that recognizes 80—90% of them, but, that doesn't mean you're doing it right—and when you blow it, you really blow it. Maybe the slide projector down the aisle will show up as a guy wearing a funny hat because your technique doesn't know about slide projectors and projector stands.

Recognition is a very hard and open-ended problem. Naturally, there are simple domains in which we can do topographic recovery; but, very little of the topographic recovery stuff has successfully been done robustly on real data. Much of it gets demonstrated on synthetic examples. Vision problems are not easy. We're starting with an image and we want to end up with certain products, or outputs, one of which is a depth map. Many people are using range sensors nowadays instead of TV cameras, although they're very slow, because they give you a depth map. Recognition is still a mess, though, because objects are 3-D. You only see one side of them. They hide one another. Objects may not even be precisely defined. (Give me the precise definition of the human head or, for that matter, a precise definition of a beard. Heads and beards come in many varieties.)

I want to stress the data types involved in vision. If you are trying to get from the pixel array to the depth map, which is also a pixel array (the pixel means something else), the data types involved are primarily pixel arrays, and you might actually get away with pixel parallelism, with what we might call "retinoptic" processing, involving processes that look at local patches of the data, chew them up, and spit them out again in array form in a way that is now more meaningful: it's now a depth map instead of an image.

In this situation, you are basically processing arrays of data; the basic data type is the array. The kinds of operations you're performing are primarily local operations where you look at little pieces of the array and infer the logical topography by massaging them. If this was all we wanted to do, it would substantiate the contention that the big bottleneck in doing vision quickly is the massive local processing of all those pixels in parallel. Mesh-connected machines like the Massively Parallel Processor (MPP) might, in fact, be the basic solution. And, even though these machines, in their present states, still have some limitations, they're developing and improving all the time.

When you come to recognition tasks, however, it's fairly clear that you need other types of data representations. What's a human face? It's got eyes, eyebrows, nostrils, lips, and so on. In order for it to be a face, however, the parts have to be in the right places. So, now we are talking not just about images as pixel arrays, but about image parts. We're talking about eyes being almond-shaped, with pointy ends. We're talking about noses being aquiline—that even sounds 3-D. We're talking about mouths being pursed or smiling. We're talking about image parts—about geometric properties of those parts, and relations among the parts.

So, when we want to do recognition, we're not just talking about pixel arrays; We're talking about other kinds of data and data types—other kinds of information about these data types, and other kinds of processing of these data types.

A computer vision system may be confronted with a variety of data types. It certainly starts with array data and, at the very beginning, a particular numeric array—that's the pixel array. It may go from there to do all kinds of derived arrays. Some of them may no longer be numeric; some of them may be symbolic and look like overlays. Even in the domain of numeric arrays, which may not even be scalar valued. These arrays might represent surface orientation. They might represent textural information, which is painfully gathered on a local basis in the neighborhood of each pixel. In short, there are many array-like representations.

Above and beyond that, we must eventually start extracting geometric entities in two and three dimensions from the pixel array. Now, we are confronted with how to represent geometric entities—patches of the image, patches of surface, or pieces of solid. These are entities in two dimensions, two-and-a-half dimensions (surface patches), and pieces of solid, not all of which you can see. A vision system must deal with the representation of that kind of information and its processing.

Going up to a still higher level of abstraction, how do you represent this data collection about pieces of the image and their properties and relations? The old standby is that you create some kind of labelled graph in which you represent the image pieces as nodes. The graph then tells you how they are related. That's how you get from the array to some sort of abstract structure. The reason for doing this is object recognition. The description of the object is in terms of parts and their relations. So, somebody has to get that kind of information out of the image and check it against the models—the descriptions of what a generic thing is going to look like. Modeling is hard. Description is hard. Making them meet halfway so that you can check one against the other is hard.

There's an even more abstract data type that we might call "knowledge." I won't even try to speculate how easy or hard it is to do vision in an AI-ish [artificial intelligence-ish] context in which you can reason about what you are doing. The processes of extracting parts from an image are not very AI-ish. The processes of setting up the data structures are not very AI-ish. People are attempting to make use of AI-ish control structures in doing some of the higher level massaging of the more abstract data types. Parenthetically, I would contend that if the AI-ish approach is going to do the vision community any good, they ought to start using it, even down at the pixel level. But, since I'm not prepared to prove that speculation, let's pass it by.

The real thing I want to call your attention to is to not assume that the bottleneck is only at the pixel level. Yes, the data you begin with in your vision system are pixel arrays of various sorts, whether the original one or all sorts of derived ones. Yes, those arrays involve fairly massive amounts of data. For example, 1,000-x-1,000 pixel images give rise to arrays of 1 million pixels, and if we have 1 or a few bytes per pixel, we're talking 1 or a few million bytes of data. Yes, that's a lot of data. But massive parallelism is approaching the million level, even if it hasn't quite gotten there yet. (Maybe you'll hear product announcements at this meeting.) We already know how to break the bottleneck at the pixel level for certain types of operations through massive parallelism, because, as massive as the parallelism may be, we're on our way toward it. And, once we achieve that level, when the parallelism of the machine is equal to that of the problem, then we can process every pixel in the image simultaneously. If all we're trying to do is some kind of local processing, maybe repeated local processing (local means not very large neighborhood sizes), of the pixel arrays, massive parallelism of the conventional kind—mesh-connected machines—would allow us to do it very fast. Thus, if the bottleneck occurred at the pixel level, conventional massive parallelism would break it.

The question is, what about the possible bottlenecks at the more abstract levels? Here, an optimist might say that, at these levels, we're working with fragments of the image; and how many of them are there? Perhaps just a few hundred. But how many bytes does it take to tell us everything we want to know about one of these fragments? Not a vast number. Then, why do I insist that there may be problems up ahead?

Because we may run into a combinatorial explosion. True, I said that from your million-pixel image, all you need do is extract, say, 1,000 image fragments. What gets you into trouble is that you pull out these 1,000 fragments in 100 different ways. Anybody who thinks you can run one canonical segmentation technique on an image and get the definitive thousand atomic image fragments is wrong. You need to extract the fragments in many different ways. So, in fact, they represent possibly overlapping inconsistent interpretations of pieces of the image. Then, you need to put those fragments together in combinations. True, you're almost certainly not considering arbitrary combinations of the thousand-image fragments (2-to-the-thousandth-power combinations); you're probably looking only at certain connected combinations, and although I'm not prepared to count them, it's certainly not a fully combinatorial problem. But, there are still many combinations, and that's where the true bottleneck may lie.

What kinds of operations do we want to perform on these various data types? There is a taxonomy; it's a kind of textbook of basic image processing and analysis techniques organized by type of operation.

If I'm given a pixel array, I might want to work on it one pixel at a time. I might want to do a stretch of the gray scale or a thresholding. I might want to do a huge variety of local operations. These are a generalization of point operations, where we're not just operating on single pixels, but one pixel and a few other related pixels everywhere in the image. It's obvious how to do that kind of thing in a massively parallel way, but some kinds of things get a little less obvious.

Suppose I want to do statistics on the image, perhaps to analyze its texture. How do you get global statistics on a 1,000 x 1,000 image? Not by local operations. Somehow, you've got to get all the information together in one place, so you can count noses, so to speak. (How many occurrences of some particular local property are there in the entire image?) The mesh doesn't support that too well. A 1,000 x 1,000 mesh looks like its's giving you a millionfold speedup factor in processing, but that's for local or point operations. It's giving you only a 1,000-fold speedup in statistical computation, because you still have a communication problem.

Other kinds of image transformations provide other problems. There are geometric image transformations that perform arbitrary warping of our image to correct distortions. There are other kinds of transformations in which the output is still an image, but it no longer has even a geometrically distorted point-by-point correspondence with the input image. Finally, there is the large class of segmentation operations that perform the segment extraction of the image parts. The input is an array, but the output no longer is.

Suppose we have managed to pull out of our million-pixel image 1,000 image fragments, or something on that order, and we have somehow represented them (without giving a lecture on representations), so we now have descriptions that are sufficient to reconstruct each of those fragments. In other words, we have a collection of geometric entities. What sorts of things do we want to do with them?

It starts right out as combinatorial in that we want to assemble them in various ways. I may need to take unions of collections of them. I may need to intersect some of them. I may need to derive other subsets from them. I don't think there's a general agreement on the taxonomy on what you may want to do with image parts. What are the geometric computations you need to perform? And how can they be efficiently performed? What are the good ways of representing the geometric entities?

Yes, you only have 1,000 entities; but, you may need to deal with a very large number of combinations of the entities. And every time you form a new combination, you may have to recompute everything, especially since the images are coming along at 30 per second. Whatever it is you do, you may have to do it again, especially if things changed rapidly. On the very next frame, you may have to extract and/or combine fragments, a different combination every time, and then compute derived structures, geometric properties, and decide on geometric relations of all sorts on the resulting mass of data.

We have accumulated a lot of ideas over the past decades as to the types of things we need to do. We have reasonably efficient algorithms for doing them. We can now look at this body of tasks and ask how can we speed them up? Is massive parallelism useful when you are trying to handle the combinatorics of search with the goal of combining image parts so that you can get to the next stage of description?

Similar remarks are true at the next level of abstraction, the graph level, where we have thrown away the geometric details. A geometric entity is now represented by a graph node at the location of the entity. But graphs are combinatorial objects, too. Even at the graph level, you get into the combinatorics of considering collections of nodes, and the complexity gets at least polynomial. I'm not a graph theorist. I'm not asking what the taxonomy of computations is that you want to do on labelled graphs. I'm only asking what a taxonomy of labeled graph computations is that a vision person might want to do.

When you look at vision benchmarks nowadays, you find that the creators of the DARPA vision benchmarks deliberately stuck their necks out and said "What about computational geometry and graph algorithms?" The next DARPA architectures workshop will be held next week, in Avon, CT, where people will report on a unified benchmark involving operations at all the different levels of abstraction.

To summarize: What's the vision problem? What are we trying to do? Topographic recovery? Object recognition? What sorts of data do we need to process? The pixel array? Derived arrays of all kinds? Geometric objects, represented in various ways? Still more abstractly, labeled graph structures? Beyond that, I don't even want to suggest anything.

What kind of operations do we want to perform on these data types? We have a long list for the array types, a shorter list for the geometric types, and a still shorter list for the graph types. I don't claim that the list is really shorter, but only that I've been too lazy to think harder and come up with a convincing, definitive taxonomy of what we may want to do. Now comes the question, "What about the speedup of these operations using various forms of parallelism?"

The pipeline idea yields operation parallelism. No sooner do you finish doing an operation on the first little piece of your image, then, treading right on its heels, comes another processor that starts the next operation on the partial output of the first operation. By doing this, you're overlaying operations; if you can do one operation at frame rates, then you can do K operations, not in K times the frame rate, not K times slower; by overlapping, you can do K operations practically at frame rates.

The MPP, and the other mesh-connected parallel systems, provide another approach, which allows an operation to be performed in parallel at every pixel. The tree-structured machines represent something that you might call orthogonal to the mesh. It's a different interconnection structure, but tremendously powerful for certain purposes. If you want to do statistics operations—for example, if you want to do histogramming, a tree is great.

A pyramid is basically the cross-product of the mesh and the tree. It has the advantages of both. I'm not going to give you a pep talk on pyramids, but, it should be clear from the program of this meeting, and almost any meeting these days, that pyramids are currently undergoing a wave of popularity, which is nice.

Finally, there's a wave of commercial machines using hypercube achitectures. Hypercubes—if they're sufficiently massively parallel—are very advantageous. In terms of communication flexibility, they can simulate pyramids very handily.

These architectural ideas have been around for some decades. The mesh has been around in a conceptual way since 1958, so it's having a 30th anniversary. Eventually, a succession was built. People have been talking about hypercubes for 10–20 years. People have been talking about pyramids for 10–15 years. Using all these kinds of parallelism—a lot is known about that. But the vision problem is still a challenge.

Suppose I was showing slides instead of these dull black-and-white, alphanumeric transparencies. Suppose I hit the slide changer button and up on the screen appeared a slide of an octopus. How long would it take you to recognize it? A fraction of a second. Suppose I hit the button, and the next slide is the Eiffel Tower. Again, in 1/10th of a second, you think "Eiffel Tower." They are familiar objects. A typewriter is familiar. My Doberman Pinscher (which I don't have) is familiar. And so on. You never saw that particular dog before, and maybe you don't know breeds that well; but, in 1/10th of a second, you recognize it as a dog. It's a familiar object. You didn't expect it, but you recognize it in a fraction of a second. You have a long-term memory in your head. As a child, you learned at an incredible rate. It's been estimated that a child learns to recognize 5,000 objects by the age of 10. (This is based on counting the entries in a picture dictionary.) You can name more than 5,000 objects reliably. You have all this information stored in your head. And you can instantly access it in a fraction of a second. How much processing could have gone on in your head from the time the light hit your retina to the time the word "octopus" came to the surface?

Neurons are slow; they take milliseconds. There is controversy as to exactly how neurons do their computations (computation is the best metaphor we have these days), but, whatever it is they do, and however they encode it, and however it is represented computationally—somehow, there is something going on in your head that goes from the light hitting your eye with an octopus pattern to the word "octopus" coming up in your short-term auditory memory, and coming out as a word.

When we do that in a few tenths of a second, uncoached, unprompted, and unexpected, how many neuron firings were there in that time? It's on the order of hundreds. How do you vision in the order of 100 cycles? What kind of architectures can achieve such performance? How do we program them? That's the next lecture.

# SECTION I: ALGORITHMS

## Part 1: Oral Presentations

# HOW TO CLUSTER IN PARALLEL WITH NEURAL NETWORKS

Behzad Kamgar-Parsi
Center for Automation Research
University of Maryland
College Park, MD 20742

J. A. Gualtieri
Code 635
NASA GSFC
Greenbelt, MD 20771

Judy E. Devaney
Science Applications Research
4400 Forbes Blvd.
Lanham, MD 20706

Behrooz Kamgar-Parsi
Dept. of Computer Science
George Mason University
Fairfax, VA 22030

## ABSTRACT

Partitioning a set of $N$ patterns in a $d$-dimensional metric space into $K$ clusters – in a way that those in a given cluster are more similar to each other than the rest – is a problem of interest in astrophysics, image analysis and other fields. As there are approximately $\frac{K^N}{K!}$ possible ways of partitioning the patterns among $K$ clusters, finding the best solution is beyond exhaustive search when $N$ is large. We show that this problem in spite of its exponential complexity can be formulated as an optimization problem for which very good, but not necessarily optimal, solutions can be found by using a neural network. To do this the network must start from many randomly selected initial states. The network is simulated on the MPP (a 128×128 SIMD array machine), where we use the massive parallelism not only in solving the differential equations that govern the evolution of the network, but also by starting the network from many initial states at once thus obtaining many solutions in one run. We obtain speedups of two to three orders of magnitude over serial implementations and the promise through Analog VLSI implementations of speedups comensurate with human perceptual abilities.

Keywords: Combinatorial Optimization, Synchronous Analog Network, Parallel Simulation, SIMD.

## INTRODUCTION

Problems that involve data analysis are becoming increasingly severe in that data sets are becoming very large and their rate of acquisition is growing rapidly. It is clear that humans possess immense computational power for solving certain problems through visualization and that what is needed is the development of algorithms that have some of these capabilities.

The value of neural networks – whose development has been motivated by human beings' computational capabilities – as a computational device is yet to be explored. In fact, little is known about the reliability and complexity of these algorithms, and how they scale with the size of the problem. The work we present in this paper is an attempt to answer some of these questions. For this, we will concentrate on the problem of data clustering – a problem of interest in astrophysics, image analysis and other fields. The conjecture is that because of the many connections among neurons, neural networks should be particularly useful for the class of problems that involve collective decision making, of which one example is unsupervised clustering. Here the patterns must decide together how to partition themselves into subsets according to a given criterion. The problem considered here, as in all partitioning problems, is a *discrete* optimization with a goodness-of-fit criterion. By embedding this discrete problem in the continuous space of an analog network one can perform a downhill search on the energy surface which is more purposeful and effective than the search in the discrete space. Until hardware implementation of analog neural networks in VLSI become available – which is expected in the next few years [1] – simulation is going to be an indispensible tool in the study and design of these systems. Analog networks are intrinsically synchronous and hence well suited for simulation on massively parallel SIMD machines.

In this paper, we simulate the neural net we propose for solving the clustering problem on the MPP [a 128×128 SIMD array machine with 1024 bits of local memory per processor]. The issue of performance of neural net algorithms on parallel machines is also addressed. Before we proceed, however, we will discuss the clustering problem in some detail.

## THE CLUSTERING PROBLEM

By clustering we mean partitioning a set of $N$ patterns (the patterns are represented as points in a $d$-dimensional metric space) into $K$ clusters in a way that those in a given cluster are more similar to each other than the rest. As there are approximately $\frac{K^N}{K!}$ possible ways of partitioning the patterns among $K$ clusters [2], the problem has exponential complexity and finding the best solution is beyond exhaustive search. As is often employed, we let our criterion for *best solution* be the minimum square-error. That is, representing the patterns by $d$-dimensional points $\{\vec{r}_i | i = 1, \ldots, N\}$, the best solution is the one minimizing $\chi^2 = \sum_{i=1}^{N} (\vec{r}_i^{(p)} - \vec{R}_p)^2$ with respect to $\{\vec{R}_p | p = 1, \ldots, K\}$. Here cluster $p$ contains the subset of the points, $\{\vec{r}_i^{(p)}\}$, and its centroid is given by $\vec{R}_p = \sum_{i=1}^{N_p} \vec{r}_i^{(p)}$, where $N_p$ is the number of points in the cluster. A partitioning based on such a criterion is also known as minimum variance partioning. Because of the complexity of the problem, finding the best solution may not be possible. This, however, is not a major concern, because in practice usually only *a good* solution is sufficient.

Due to the importance of this problem many methods have been proposed by various researchers. (See Jain and Dubes [3] for a survey of the literature.) Many of these approaches are based on iterative schemes and often the differences between the suggested algorithms are quite subtle. The number of clusters $K$ may or may not be fixed. For a given value of $K$, the essence of iterative algorithms is as follows.

After the initial partioning of the patterns into $K$ clusters, their centroids, i.e. seed points in the $d$-dimensional metric space of the patterns, are computed. Each pattern is then assigned to the cluster with the nearest seed point and new centroids are computed. The process is repeated until the partitioning ceases to change. However, the process of the computation of new centroids can be carried out in two ways: (i) Keep the centroids fixed until the distances of *all* patterns to the $K$ centroids are computed [4]; (ii) Update centroids as frequently as one pattern is found to be closer to the centroid of a cluster other than the one it is assigned to. In this case, the pattern is immediately reassigned and the centroids of the winning and the losing clusters are updated [5]. This method is sometimes referred to as $K$-means. Note that for a parallel machine, where the distances of the patterns from cluster centroids can be computed simultaneously, the first approach appears to be more efficient.

The neural net approach that we propose has many similarities with the iterative scheme described above. As will be explained later in more details, the major difference, however, is that the neural net allows a given pattern to belong to several clusters until the final iteration. That is, at least during the execution of the algorithm, a given pattern belongs to all clusters, though with different weights. The closest conventional method to this is the one proposed by Gordon and Henderson [6]. In their method, however, the sum of the weights for every pattern is restricted to one at any given iteration; thus, it dose not possess the full flexibility of neural networks.

As for the initial cluster centroids, one may take the first $K$ points of the input data, which is very simple and inexpensive; or if one suspects the input points are prearranged in some special way, one may choose at random any $K$ points of the input data [7]. More elaborate and expensive methods for choosing more promising initial centroids have been proposed in the literature (see Ref. [8] and [3]). Such methods, however, are not of interest to us.

## OPTIMIZATION WITH NEURAL NETS

It has been recognized in recent years that artificial neural networks have computational properties [9,10]. The Hopfield model of neural network, which we use in this work, is particularly suitable for solving certain optimization problems. A neuron is a simple nonlinear processor that is connected to many (possibly all) other neurons in the network; it adds up the signals it receives from other neurons and fires a signal accordingly. The state of the network, that is the firing rates or activities of the neurons, through interactions with each other, change with time but eventually the network settles into a steady state where the neuronal activities remain constant. The energy of the Hopfield network is Lyapunov (i.e. it does not increase with time) and its minima are the steady states of the network. It is this property of neural networks that is used in optimization. The approach is to cast the problem in terms of an energy function that is then minimized by the corresponding network as it evolves spontaneously from some randomly selected initial state to states of lower energy. The energy function has typically many minima that represent valid solutions to the problem; deeper minima correspond to good solutions and the deepest minimum to the best solution.

In this paper we use analog neural nets, because they outperform digital nets in solving optimization problems [9,11]. Many problems of interest, including the problem we address in this paper, can be cast in terms of an energy function, $E$, that is quadratic in the neuronal activities and has the form [9],

$$E = -\frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} T_{ij} V_i V_j - \sum_{i=1}^{n} I_i V_i + \frac{1}{\tau} \sum_{i=1}^{n} \int^{V_i} dx \, g^{-1}(x) . \tag{1}$$

Here $n$ is the number of neurons in the network, and

$V_i$ ($0 \le V_i \le 1$) is the activity or firing rate of neuron $i$. The first term in (1) is the interaction energy among neurons, and the elements of the connection matrix, $T_{ij} = T_{ji} = -\frac{\partial^2 E}{\partial V_i \partial V_j}$, are completely determined from $E$. In the second term $I_i$ is the bias or activity threshold of neuron $i$. The third term encourages the network to operate in the interior of the $n$ dimensional unit cube $\{0 \le V_i \le 1\}$ that forms the state space of the system. In this term $\tau$ is the self-decay time of the neurons, and $g(u)$, a sigmoid function, is the gain or transfer function of the neurons that relates the input $u_i$ to the output $V_i$. A standard form for $g$, which we will also use, is

$$V_i = g(u_i) = \frac{1}{2}(1 + \tanh\frac{u_i}{u_0}) = \frac{1}{1 + e^{-2u_i/u_0}}, \qquad (2)$$

where $u_0$ determines the steepness of gain. The neuronal activities, $V_i$, as well as the input signals, $u_i$, depend on time $t$. The evolution of the network is determined by the $n$ coupled ordinary differential equations, $du_i/dt = -\partial E/\partial V_i$, which are

$$\frac{du_i}{dt} = -\frac{u_i}{\tau} + \sum_{j=1}^{n} T_{ij}V_j + I_i. \qquad (3)$$

We will set $\tau = 1$, so that time is measured in units of $\tau$. Note that the *bias*-term can be eliminated from the energy and instead incorporated into the gain function if we define $V_i = g(u_i - \tau I_i)$.

To find a solution (i.e. a minimum), we start the network from a randomly selected state and let it evolve freely until it reaches a minimum of the function $E$ and stops. As is usual in dealing with computationally intractable problems, we find not just one but several solutions by starting the network from different initial states, and then take the best one as *the solution* which may or may not be the optimum. Since a neural network converges rapidly to a minimum we can afford to run it many times thus ensuring that we find at least a very good solution. Below, we discuss how to construct an appropriate network for solving this problem.

## CONSTRUCTION OF THE ENERGY FUNCTION

We want to partition a set of $N$ points in a 2-D plane into the *best* $K$ clusters (generalization to arbitrary dimensions is trivial) – best in the sense that sum of the squares of the distances of the points from their respective cluster centroids (i.e. sum of "within cluster variances") is minimized. We formulate the problem in a manner that can be solved by a neural network; that is we cast the problem in terms of an energy function that can be minimized by the network.

The energy function will consist of two parts: (i) constraint terms which make certain a point, at the end of the search, belongs to one and only one cluster; (ii) the cost term which is the sum of the residuals and is the function we actually wish to minimize. The formulation can best be illustrated through an example. Let us consider the case where we wish to partition $N = 10$ points into $K = 3$ clusters. A *possible* solution (not necessarily the best one) would be that, say, points 1, 2, 6 and 9 belong to cluster $A$, points 4 and 5 belong to cluster $B$, and points 3, 7, 8 and 10 belong to cluster $C$. This particular solution can be represented by the 3×10 rectangular array given in Table 1, where the rows are labeled by the clusters and the columns are labeled by the points. The elements of this matrix are 0 or 1 with the interpretation that "element A1=1" indicates that point 1 belongs to cluster $A$, "element B1=0" indicates that point 1 does not belong to cluster $B$, and so on.

Table 1: A possible solution for partitioning 10 points into 3 clusters.

| Cluster | Points | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| A | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| B | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| C | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |

If we think of the elements of this matrix as the activities of neurons ($n = K \times N$ neurons altogether), and denote them by $V_{pi}$, where $p$ and $i$ refer to the cluster and the point, respectively, then the constraint part of the energy function, $E$, can be expressed as

$$E = \frac{A}{2}\sum_{i=1}^{N}\sum_{p=1}^{K}\sum_{q\neq p}^{K} V_{pi}V_{qi} + \frac{B}{2}\sum_{i=1}^{N}(\sum_{p=1}^{K} V_{pi} - 1)^2, \qquad (4)$$

where the coefficients $A$ and $B$ are positive constants. The *A-term* has its minimum value (i.e. zero) if in each column (representing a point) at most one neuron is active and the rest are off. The *B-term* has its minimum value (also zero) if the sum of activities in each column equals 1. The two terms together enforce the *syntax* of the solution given in Table 1.

There is an additional constraint that we should, in principle, include in the energy function: that each cluster should contain at least one point. In terms of the solution matrix of Table 1 it means that in each row there should be at least one fully active neuron. Such a constraint can be imposed by $\sum_{p=1}^{K} \Theta(1 - \sum_{i=1}^{N} V_{pi})$, where $\Theta(x) = 0$ for $x \le 0$ and $\Theta(x) = 1$ for $x > 0$ is the step function. However, since this term is nonanalytic its inclusion in the energy function creates problems and a better strategy appears to be to leave out this term and rather reject those solutions that violate this constraint. In our simulations of neural networks (several thousand trials) the solutions never violated this constraint. Therefore, it appears that the absence of this constraint from the energy function is of little consequence.

To complete the energy function we must also formulate the *cost* term. We denote the square of the distance of point $i$ from the centroid of cluster $p$ (i.e. the residual) with $R_{pi}$ which is given by

$$R_{pi} = (x_i - X_p)^2 + (y_i - Y_p)^2, \qquad (5)$$

where $(x_i, y_i)$ are the coordinates of point $i$, and $(X_p, Y_p)$ are the coordinates of the centroid of cluster $p$. Here we have chosen the Euclidean distance as the metric; but one can define any metric one wants. Let us consider again the solution represented by Table 1. The sum of residuals or the cost for this solution is

$$(R_{A1} + R_{A2} + R_{A6} + R_{A9}) + (R_{B4} + R_{B5}) \\ +(R_{C3} + R_{C7} + R_{C8} + R_{C10}), \qquad (6)$$

which can be written as

$$\sum_{p=1}^{K}\sum_{i=1}^{N} R_{pi}V_{pi}^2. \qquad (7)$$

Hence the energy function $E$, including cost and constraint, for this problem can be expressed in the final form

$$E = \frac{A}{2}\sum_{i=1}^{N}\sum_{p=1}^{K}\sum_{q\neq p}^{K} V_{pi}V_{qi} + \frac{B}{2}\sum_{i=1}^{N}(\sum_{p=1}^{K} V_{pi} - 1)^2 \\ +\frac{C}{2}\sum_{p=1}^{K}\sum_{i=1}^{N} R_{pi}V_{pi}^2, \qquad (8)$$

where $C$ is also a positive constant. When the constraints (or the syntax) are satisfied the $A$-term and the $B$-term vanish and the energy function, $E$, reduces to just the cost term, therefore *deep minima* of $E$ correspond to *good* solutions, and the deepest minimum to the best solution.

The network dynamics, obtained from $-\partial E/\partial V_{pi}$, are

$$\frac{du_{pi}}{dt} = -u_{pi} - A\sum_{q\neq p}^{K} V_{qi} - B(\sum_{q=1}^{K} V_{qi} - 1) - CR_{pi}V_{pi} + I_{pi}. \qquad (9)$$

Note that (8) is only the quadratic part of the energy function corresponding to the first term in (1), and that the two terms $I_{pi}$ and $-u_{pi}$ in (9) come from the second and third terms in (1), respectively.

To find a solution we assign random values between 0 and 1 to all the $n = K \times N$ neuronal activities, $V_{pi}$. Thus the $N$ points are partitioned into $K$ clusters. Note that the partitioning is not done in the proper sense that a point belongs to a particular cluster and to no others; rather, point $i$ is partitioned among all the $K$ clusters with varying strengths that are the magnitudes of $V_{pi}$, that is, we interpret $V_{pi}$ as the strength of hypothesis that point $i$ belongs to cluster $p$. Hence the centroid of cluster $p$ is obtained from the weighted average

$$X_p = \sum_{i=1}^{N} x_iV_{pi}/\sum_{i=1}^{N} V_{pi}, \qquad Y_p = \sum_{i=1}^{N} y_iV_{pi}/\sum_{i=1}^{N} V_{pi}. \qquad (10)$$

As the state of the network changes with time the centroids, as well as the residuals $R_{pi}$, also change. Starting from this randomly selected initial state the network evolves toward states of lower energy according to the equations of motion (9), until it reaches a minimum energy state and stops. The downhill motion of the network on the energy surface is guided toward a proper solution (one that satisfies the constraints) by the $A$- and $B$-terms and toward solutions of good quality by the $C$-term. As the network is searching for a solution the constraints are most surely violated since most neurons are partially active. Only at the end of the search when a solution is found the clustering becomes unambiguous. Note that the energy $E$ also contains other minima that do not correspond to solutions (i.e. violate the syntax); such minima when found by the network are of course rejected as meaningless.

We remark that the cost term (7) can be written as a linear function of activities such as $R_{pi}V_{pi}$ which is *bias*-like rather than *interaction*-like. However, bias-like terms are not as effective in breaking the symmetry among the states that satisfy the syntax, and leave the energy landscape more flat. Hence it will not be as easy for the network to find valid solutions as it frequently becomes stuck in the middle of the $n$-dimensional unit cube. This is confirmed in our simulations, where the rate of success for finding valid solutions drops significantly when we use the linear form for the cost.

For simulations we have chosen the following values for the parameters of the energy function: $A = B = 1$, $C = 0.9/R_{avg}$, all $I_{pi} = 1$, and the gain function parameter $u_0 = 0.1$. Scaling parameter $C$ with the average residual $R_{avg}$ is necessary to ensure good solutions, because as the network evolves, the residuals become generally smaller and the cost term becomes less effective in driving the network toward good solutions; this rescaling of parameter $C$ keeps the cost term of the same order of magnitude as the syntax terms.

## PARALLEL IMPLEMENTATION

We have simulated the behavior of the neural net on the MPP. To do this we first generate a random initial state $\{V_{pi}(t = 0)\}$ and then solve the equations of motion (9) to find which of the minima (or solutions) it converges to. Solutions of ordinary differential equations, such as the equations of motion, lend themselves very nicely to a massively parallel computational approach. In addition, since we want to find several solutions starting from different initial states – as is usual in computationally intractable problems – we run several trials at once on the MPP. Thus the speedup comes from parallel solution of the differential equations as well as running several trials at the same time.

We use the Euler method [12] with a fixed time step $\delta t$ to solve the differential equations (9), i.e. we iterate the set of $n = K \times N$ equations,

$$u_{pi}(t+\delta t) = u_{pi}(t) + \delta t\{-u_{pi}(t) - A\sum_{q \neq p}^{K} V_{qi}(t) \\ -B[\sum_{q=1}^{K} V_{qi}(t) - 1] - CR_{pi}V_{pi}(t) + I_{pi}\}, \quad (11)$$

until the system converges to a stationary state. The only stopping criterion we use is when the changes in the firing rates become insignificant, i.e. when all $|V_{pi}(t+\delta t) - V_{pi}(t)| < \epsilon$, where $\epsilon \ll 1$. After the network converges to a solution, we must check if it is a valid solution that satisfies the syntax, i.e. for every point $i$ we must have one $V_{pi} = 1$ and all the rest $V_{qi} = 0$ for $q \neq p$. In analog networks the activity of a neuron can never become exactly 0 or 1 and can only reach close to the limits. Therefore, if $V_{pi} < \eta_0$ we take $V_{pi} = 0$, and if $V_{pi} > 1 - \eta_1$ we take $V_{pi} = 1$, where $\eta_0$ and $\eta_1$ are small positive numbers. In the simulations we have chosen the following parameter values: time step $\delta t = 10^{-3}$, convergence parameter $\epsilon = 10^{-4}$, and the syntax parameters $\eta_0 = \eta_1 = 0.2$.

Mapping onto a SIMD parallel processor was accomplished by assigning a unique processing element to each data point. With this requirement, all of the necessary operations reduce to simple array arithmetic, parallel sums, row and column broadcasts, and global boolean tests. All of these are the strong points of a massively parallel processor such as the MPP. Since the MPP has 16384 processors, fewer data points allow more separate trials to be run in parallel. Thus, for example, the 128 point case allowed for 128 trials with different starting conditions to be run at the same time. The overhead to the program to keep track of the different trials is trivial since the data movement required is straightforward and controlled by the programmer. The set of data points is replicated for each trial run in parallel.

Each processor has stored in its memory its coordinate values $x_i$ and $y_i$, the neuronal activities $V_{pi}$, input signals $u_{pi}$, residues $R_{pi}$ for $p = 1, \cdots, K$, convergence indicators for each neuron, and other ancillary information. The processing begins with the calculation of the centroids of each cluster according to (10). This involves a simple array multiplication of the $x_i$ and $y_i$ by $V_{pi}$ for each cluster $p = 1, \cdots, K$. This result is summed using the cascading sum technique [13] and divided by the sum of $V_{pi}$ for each cluster. These centroids are broadcast in parallel over the remainder of the array using the MPP micrcoded broadcast primitive. This primitive, designed by Rudi Feiss (described in [14]) is very fast using only 231 cycles to broadcast a row or column – 128 32 bit numbers – to the remainder of the rows or columns of the 128×128 array. Then we calculate the residues from (5) which involves more array arithmetic. The new input signals $u_{pi}(t+\delta t)$ are calculated from (11) and the new activities $V_{pi}(t+\delta t)$ are calculated from the sigmoid function (2). These are all array arithmetic operations. A boolean mask for each cluster is created in parallel to record where the new activities are different from the old activities by more than the convergence parameter $\epsilon$. A logical 'or' (implemented as the ANY function in MPP Pascal) on the masks determines whether the convergence criteria has been met for all activities. This logical 'or' directly translates into a hardware instruction on the MPP and thus allows simultaneous checking of conditions which on a serial processor would have to be done individually. Updating of all neurons for each trial was continued, regardless of whether a particular trial had converged, until all trials had converged. Thus unnecessary bookkeeping time is eliminated.

Thus the speed on the MPP is obtained from, (i) the mapping which allows most operations to be formulated in terms of array arithmetic, (ii) the movement of data among the processing elements which can be done with parallel algorithms, and (iii) the global boolean tests which are done by the machine hardware. For the case of 128 points to be clustered into 5 clusters, 128 trials were run simultaneously. This required 19 seconds per 500 iterations. The corresponding CPU time on a VAX 8800 was 2940 seconds (a speedup of over 150 times), and 21100 seconds on a VAX 11/780 (a speedup of about 1100 times).

## EXAMPLES

To study the performance of the neural net we have tested it on some examples. In the first data set, there are 128 points divided among 5 clusters with within-cluster Gaussian distributions (Fig. 1a). Here the 5 clusters are rather well defined and out of the 128 trials the neural net found the optimum clusters 128 times. The average number of iterations for convergence was 4263; since $\delta t = 10^{-3}\tau$, the average convergence time is about $4.3\tau$, where $\tau$ is the decay time of a neuron. In VLSI implementations of neural networks that are currently in progress [1], the decay time of neurons, $\tau$, is in the range $10^{-6} - 10^{-3}$ second, hence the convergence time of the network should be in the range of a few micro-seconds to a few milli-seconds. Note that from numerical solution of differential equations one can only obtain an estimate of the actual convergence time, because the number of iterations for convergence depends on the value of the convergence parameter as well as the time step. Obviously if the convergence parameter is made smaller it will take more iterations for the network to meet the convergence criterion, resulting in a higher estimate for the convergence time. On the other hand if the time step is made smaller by, say, a factor of 10, it will take fewer than 10 times the number of iterations to converge, thus resulting in a lower estimate for the convergence time. Fig. 2 shows in more detail the number of iterations for the convergence of all the 128 trials.

The conventional method of Forgy [4] in 128 trials found the best clusters only 46 times and various other solutions 82 times. The average number of iterations for convergence was 7. Clearly, in this example, the neural net outperforms the conventional method, in that it finds the best solution much more frequently. On the other hand, the conventional method takes far fewer iterations to converge than the neural net. But we should bear in mind that these are *simulations* of the neural net, and that the number of iterations needed for convergence is not the true measure of the processing time of the network. The convergence time of an actual analog VLSI network must be measured in $\tau$, the characteristic time of a neuron, which is in the micro to milli-second range.

To test the performance of the network in cases where clusters are fussy, we started from the data points of Fig. 1a, randomly selected 10% of the points and distributed them uniformly throughout the unit square (Fig. 1b). Thus we obtained 5 clusters with uniform background noise. The neural net in 128 trials found the best clusters 28 times. It failed to find valid solutions statisfying the syntax 46 times. This large number of failed solutions can be interpreted as an indication that the clusters are fussy, that there are outliers, and that perhaps the specified number of clusters, $K = 5$, is too few. However, even when the syntax is not satisfied we can extract a valid solution with the following scheme. For each point $i$ set the largest $V_{pi}$ to 1 and all the other $V_{qi}$ with $q \neq p$ to 0, and interpret this solution as the one favored by the network, thus we obtain 128 solutions. Conventional algorithms always find valid solutions and cannot give an objective indication of the fussiness of clusters.

Similarly to Fig. 1b, we generated other data sets by increasing the background noise to 25%, 50%, 75%, and 100% (i.e. no clusters). These data are shown in Fig. 1c-f. The results of partitioning the data among 5 clusters obtained, in 128 trials, with the neural net and with Forgy's method are listed in Table 2. The average estimated convergence times for the network are given in units of $\tau$. Two points of note in this table are: (i) As the 5 clusters become less discernible the network increasingly fails to satisfy the syntax indicating that clusters are fussy and that 5 clusters are not sufficient. The conventional method, on the other hand, always finds valid solutions, and although the variety of solutions that it finds increases (this is true in both methods) which may be taken as a clue to the fussiness of clusters it is not as objective an indicator as the failure to satisfy the syntax; (ii) When there are well defined clusters the neural net performs better than the conventional techniques which is reflected in the lower average $\chi^2$ ($\chi^2$ is the sum of within-cluster variances) for solutions found by the neural net. And as clusters become fussier the quality of solutions found by both methods become comparable.

Table 2: In this table the results obtained by Forgy's conventional algorithm are compared with those by the neural network. The Data refer to data points of Fig. 1a-f. These are based on 128 trials.

| Data | Conventional | | | |
|---|---|---|---|---|
| | Iter | Best Var | Best% | Avg Var |
| a | 7 | 0.62 | 36 | 1.23 |
| b | 8 | 1.06 | 34 | 1.57 |
| c | 8 | 1.95 | 12 | 2.27 |
| d | 10 | 2.94 | 2 | 3.14 |
| e | 10 | 3.88 | 10 | 4.11 |
| f | 10 | 4.13 | 2 | 4.64 |

| Data | Neural Net | | | | |
|---|---|---|---|---|---|
| | Time | Best Var | Best% | Avg Var | Synt% |
| a | 4 | 0.62 | 100 | 0.62 | 100 |
| b | 7 | 1.06 | 22 | 1.24 | 64 |
| c | 7 | 1.95 | 19 | 2.03 | 9 |
| d | 8 | 3.00 | 15 | 3.04 | 0 |
| e | 6 | 3.89 | 1 | 4.11 | 1 |
| f | 8 | 4.46 | 2 | 4.71 | 0 |

Iter: is the average number of iterations for convergence.
Best Var: is the variance of the best solution found.
Best%: is the percentage of trials that found the best solution.
Avg Var: is the average variance of the solutions found.
Time: is the average estimated time of convergence in units of $\tau$.
Synt%: is the percentage of trials that found solutions satisfying the syntax.

In Fig. 3, we have plotted the trajectories of the centroids of the 5 clusters as a function of time for all the 128 trials for the data of Fig. 1a. It can be seen that although the centroids start from different places in different trials, they all eventually converge to the same 5 points which are the true centroids of the 5 clusters. This clearly shows that the network succeeds, in every trial, in finding the structure in the data. In Fig. 4, we have plotted the centroid trajectories for the data of Fig. 1f. The spreading of trajetories (as contrasted to the contraction of trajectories in Fig. 3) of different trials, shows that where there is no underlying structure in the data, the network does not prefer any particular clustering and hence finds many different solutions.

## CONCLUDING REMARKS

Preliminary results for clustering with neural networks are promising. The neural net appears to outperform conventional iterative techniques, when there are well defined clusters since it finds better solutions more frequently. And when clusters are fuzzy, or when the number of clusters we specify is not compatible with the structure of data, the neural net indicates that it cannot find valid solutions easily, and that something may be wrong. This indicator is an objective measure and hence more reliable than the user supplied bounds and tolerances for conventional techniques. Work on larger data sets is in progress.

The clustering criterion we have used in this paper, that is minimum sum of within-cluster variances, results in convex compact clusters. Often clusters are not round or compact. By adding to the energy function, appropriate terms that favor closeness of a point to its neighbors (and not just to the cluster centroid), one can design a network that finds non-convex elongated clusters of various shapes.

Simulations of the neural net on the MPP for the clustering problem are two to three orders of magnitude faster than simulations on serial machines such as the VAX 8800 and VAX 11/780. The speedup is due to parallel solution of the differential equations that govern the behavior of the network, as well as running several trials at the same time. However, the real benefit of neural nets may lie in the future when they can be mapped on analog chips. There are forecasts that analog VLSI neural nets will become available in several years [1]. These devices will have processing times in the micro to milli-second range, making their performance comensurate with human perceptual abilities.

# References

[1] C. Mead, "Real-time analog computation in VLSI neural networks", in the First Annual International Neural Networks Society Meeting (Boston, 1988).

[2] W. Feller, *An Introduction to Probability Theory and Its Applications*, 2nd edition (John Wiley, 1959) Vol. 1, p. 58.

[3] A.K. Jain and R.C. Dubes, *Algorithms for Clustering Data* (Prentice Hall, 1988).

[4] E.W. Forgy, "Cluster analysis of multivariate data: efficiency versus interpretability of classifications", Biometric Soc. Meetings, Riverside, California. Abstract in *Biometrics*, **21**, 768 (1965).

[5] J.B. MacQueen, "Some Methods for Classification and Analysis of Multivariate Observations", Proceedings of Fifth Berkeley Symposium on Mathematical Statistics and Probability, Vol. 1, p. 281 (1967).

[6] A.D. Gordon and J.T. Henderson, "Algorithm for Euclidean sum of squares classification", *Biometrics*, **33**, 355 (1977).

[7] D.J. McRae, "MIKCA: A FORTRAN IV iterative k-means cluster analysis program", *Behavioral Science*, **16**, 423 (1971).

[8] M.R. Anderberg, *Cluster Analysis for Applications* (Academic Press, 1973).

[9] J.J. Hopfield and D.W. Tank, "Neural computation of decisions in optimization problems", *Biological Cybernetics*, **52**, 141 (1985).

[10] *Neural Networks for Computing*, edited by J.S. Denker (American Institute of Physics, 1986).

[11] B. Kamgar-Parsi and B. Kamgar-Parsi, "An efficient model of neural networks for optimization", in Proceedings of the IEEE First International Conference on Neural Networks, edited by M. Caudill and C. Butler, Vol.3, p. 785 (1987).

[12] C.W. Gear, *Numerical Initial Value Problems in Ordinary Differential Equations* (Prentice-Hall, 1971).

[13] H.S. Stone, "Problems of Parallel Computation", in *Complexity of Sequential and Parallel Numerical Algorithms*, edited by J.F. Traub (Academic Press, 1973).

[14] J.E. Devaney, "The MPP – a Totally Different Approach to Programming", presented at the IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Data Base Management (1985).

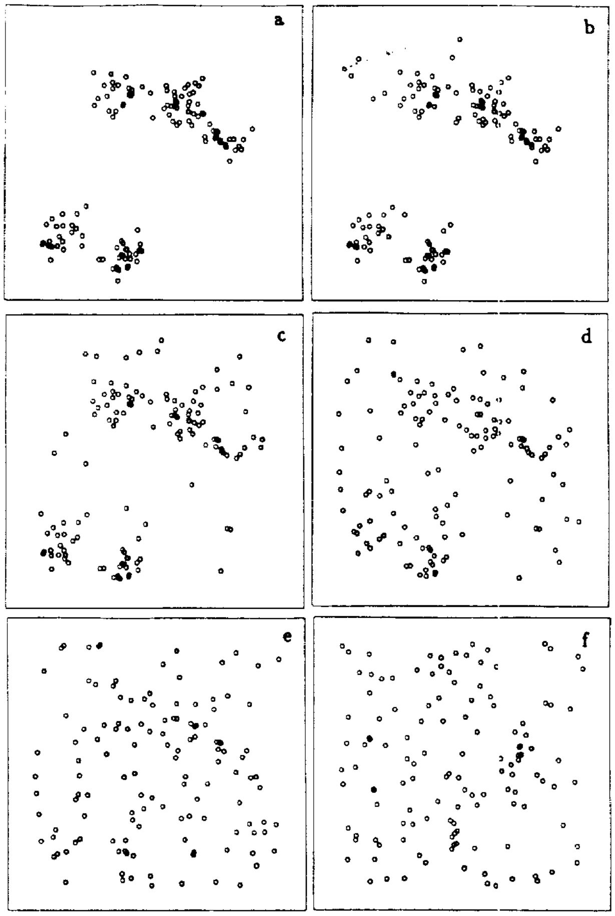


Fig. 1. 128 points divided among 5 clusters and respectively 0,10,25,50,75,100 % uniform background in a,b,c,d,e,f.

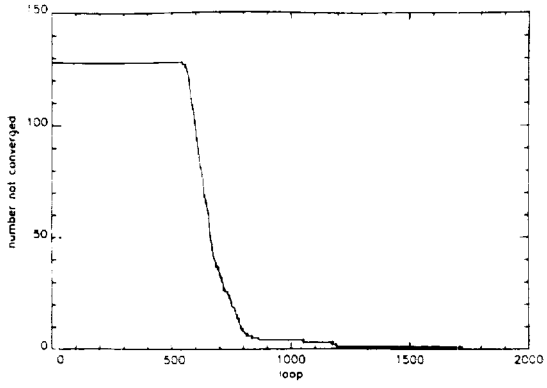


Fig. 2. Number of trials not converged versus iteration for the data in Fig. 1a. (0 % background) (loop is the iteration number).

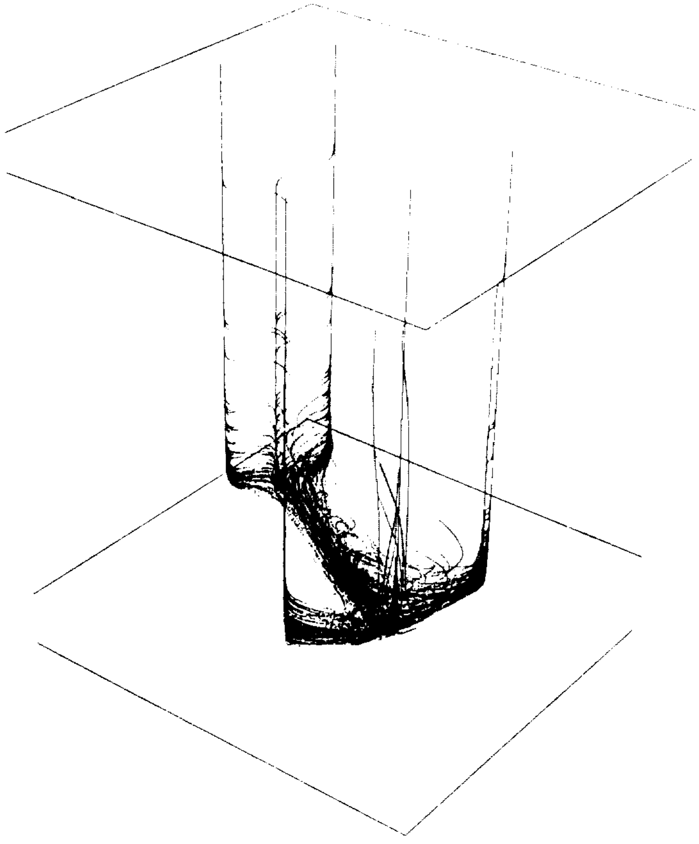


Fig. 3. Trajectories of the five cluster centroids for all 128 trials for the data in Fig. 1a. (0 % background). Lower left corner of Fig. 1a. corresponds to back top corner in this figure

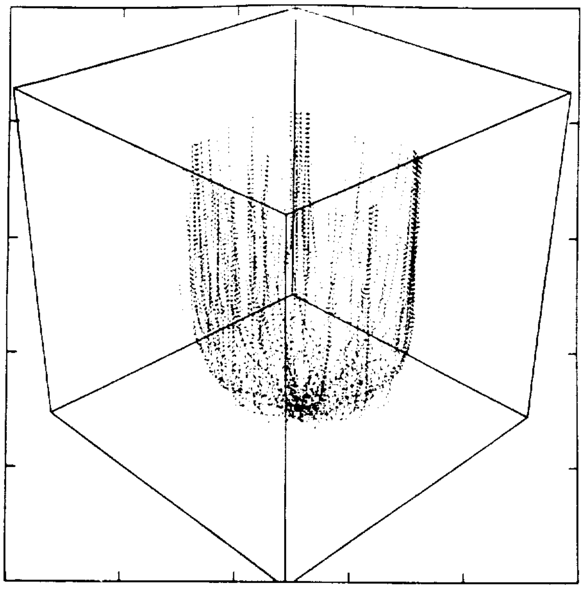


Fig. 4. Trajectories of the five cluster centroids for the data in Fig. 1f. (uniform distribution - 100% background).

# MODELING NEURAL NETWORKS ON THE MPP

Joe Hicklin Howard Demuth

Department of Electrical Engineering
The University of Idaho
Moscow, Idaho

## ABSTRACT

A network of fixed-connection-weight neuron-like elements has been simulated on the massively Parallel Processor (MPP) in two ways. First, the square connectivity matrix of a 128 neuron network was mapped onto the square MPP processor array. This allowed a highly parallel simulation in which 128 MPP processors were active at all times. Next, a 128 by 128 array of neurons was mapped onto the 16384 MPP processors. Here the MPP processor limits neuron connections somewhat but all MPP processors are active at all times and a large speedup is obtained. The first simulation, based on the mathematics (weight matrix), produced a significant speedup but tended to obscure the second faster simulation based on mapping the physics (entire physical description) of the neural network onto the MPP. The authors experience suggests that alternative mappings onto the MPP should be sought and examined carefully.

Keywords: massively parallel processor, neural network, neural network simulation.

## INTRODUCTION

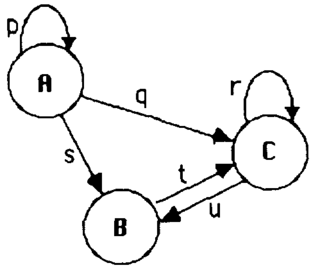
This paper describes two simulations of neural networks on the Massively Parallel Processor (MPP). The first simulation seemed to be a natural fit of the the mathematics involved to the MPP architecture. It gave a significant speedup but it was found that only a small percentage of the potential power of the MPP was being utilized. The second simulation mapped the physical process under study to the MPP and led to a much better utilization of the MPP. In the first implementation we fell into what we believe may be a common trap, that of simulating the mathematics as opposed to the process. We would like to make this trap clear so that others may recognize it and perhaps avoid it in the future.

This paper provides a brief description of the MPP and an outline of the neural model to be simulated. This is followed by a description and a comparison of the two simulations implemented on the MPP.

Logically, the MPP is a single instruction stream, multiple data stream (SIMD) machine that has a rectangular mesh of 128 by 128 one bit processors. Each processor has 1024 bits of local memory and can communicate directly with its four nearest neighbors, including opposite edge neighbors. This topology can be used in several ways to simulate the neural networks described below.

## NEURAL NETWORK MODEL

A simple example of the kind of neural network studied here is shown in Figure 1. The neuron bodies are labeled A, B, and C. Their connections are labeled p through u. The connection from A to C says that when A "fires", a signal of strength q is sent to neuron C. A neuron fires when the sum of its accumulated activation and its present inputs exceed its threshold. On firing, the accumulated activation of a neuron is set to zero.



| | | From | | |
|---|---|---|---|---|
| | | A | B | C |
| To | A | p | | |
| | B | s | | u |
| | C | q | t | r |

Figure 1, A Simple Neural Network

The operation of an N neuron network like that shown in Figure 1 can be described mathematically as follows:

$A_i(t_p) \triangleq$ activation of neuron i at time $t_p$ ( real number )

$W_{ij} \triangleq$ strength of connection from $j^{th}$ neuron to the $i^{th}$ neuron

( Also, the $i,j^{th}$ element of the weight matrix W. )

$T \triangleq$ threshold at which any neuron will fire

$$F[A_j(t)] \triangleq \begin{cases} 1 \text{ if } A_j(t) \geq T \\ 0 \text{ if } A_j(t) < T \end{cases}$$

$$A_i(t + 1) = (1 - F[A_i(t)]) * A_i(t) + \sum_{j=1}^{N} F[A_j(t)] w_{ij}$$

Here the first term says that if a neuron fired on the last time step its activation is not carried forward and its contribution to the current activation is zero; but if it did not fire its previous activation is carried forward to the new activation. The second (summation) term indicates the current effects of the firings of other neurons.

This neuron model has a variety of simplifications. First, time is discreet. Second, all neuron thresholds are identical. This constancy is not a limitation for the effect of different thresholds can be achieved through adjustment of the weight matrix elements. Third, the neuron connections are not adjusted once the simulation begins. Thus, no "learning" involving change in the elements of W is involved. This last approximation allows the behavioral investigation of relatively large non-learning networks.

The activation update phase of the simulation of the models requires the majority of the computation time. Here the activation vector for the next time step is generated from the current activation vector and the weight matrix. On each time step some elements of the activation vector are above threshold, and so the associated neurons fire. A binary firing vector, F, is generated with a one in the elements associated with neurons that fire on this time step and a zero in the others. Each neuron that fires on a given time step alters the activation of the other neurons by the amount in the element of the weight matrix corresponding to the connection from the firing neuron to the receiving neuron.

## FIRST SIMULATION

A first look at the problem reveals that the topology of the MPP is the same as that of a weight matrix. If each processor served the function of a single element of the weight matrix then the problem would be well mapped to the hardware. For this strategy a 128 neuron system, matching the column/row size of the MPP, would be simulated. Each processor has its particular weight stored in its local memory. Since the activation vector has 128 elements, it can be handled by a single row or column of the MPP. Activation update steps proceed as follows. The activation vector is held in the uppermost row of the MPP. A firing vector is generated in the top row as each processor in that row determines if the activation held there is above threshold. As shown in Figure 2, this firing vector is passed downward across the entire array so that each processor in the i'th column can know whether the i'th neuron is to fire on this time step.

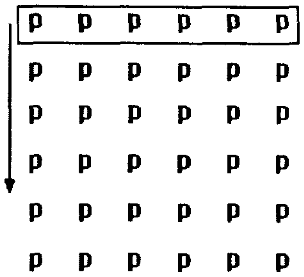


Figure 2, Firing Information Moves Downward

Next all processors in columns that fire pass their weights to the right while the other columns pass zeroes. As shown in Figure 3, these weights are all passed to the rightmost column where they are summed. This results in a vector in the rightmost column that represents the change in A due to the firing of the neurons.

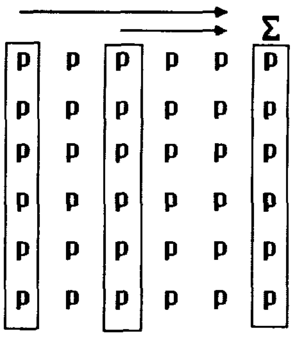


Figure 3, Weights move to the Left

This vector must be added to the old A vector in the top row to form the new A. (More correctly, it is added to the old A vector after the firing neurons in the old A vector have been reset.) This stage is time consuming since 256 shifts are needed to move all the data from column form to row form. The activations are shifted as shown in Figure 4.

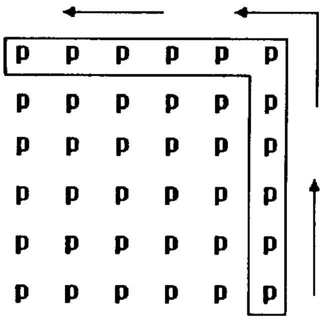


Figure 4, Activation Shifts from Left to Top

As described above, 128 processors work on the problem in parallel at every step except the final shifting stage. This gives a tremendous speedup over single processor simulations. Another way of looking at this performance however, is that at any given time, about 1% of the MPP is being used while the remaining 99% is idle. This second view prompted a search for a better method of simulation.

## SECOND SIMULATION

The second simulation can model a much larger neuron population and utilizes the MPP more fully, but it does this at the expense of altering the neural model slightly. Each processor now represents a single neuron and stores its own activation level, as well as the row of the weight matrix containing the weights from other neurons to itself, in its own local memory. Due to the 1024 bit memory limitation of the MPP, there is not room for all connections to all 16,384 neighbors, and only those connections to the 24 nearest neighbors are held. This is not as drastic a change in the model as it might seem, for in the brain, connections between nearby neurons are vastly more common that those between distant neurons. A second change in the model is that now the neurons have neighbors in two dimensions rather than only one as in the previous model. This change also leads to a more realistic model, as the cerebral cortex is essentially a two dimensional sheet of neurons.

The activation update procedure is shown in Figure 5. There each processor (neuron) determines if its activation is above threshold. Each processor then passes a one bit flag to the processor on its right, informing that processor as to whether the first processor is firing. Each processor stores this information and then passes it upward. The process repeats and each processor passes the flag to the left. In this manner, the flag spirals outward around the processor that generated it until it has reached all 24 of the nearest neighbors. If more then 24 connections are desired, the flag can spiral outward as far as necessary. Thus, every processor communicates its firing with its 24 neighbors using only 24 shifts. Notice that the entire bit plane is shifted so that all 16,384 processors pass their firing information together.

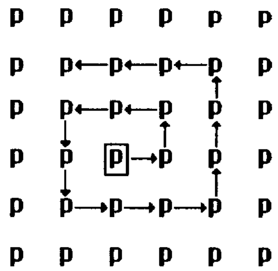


Figure 5, Data Flow in the Second Simulation

After this communication phase, each processor sums its weights associated with its neighbors that fired and updates its own activation, completing the activation update. Every processor is used in every step, achieving full utilization of the MPP. The size of the network has been increased by a factor of 128, the total number of connections has been increased by a factor of 24, and the similarity of the model to the real world has been improved.

## CONCLUSION

The simulation of neural networks on the MPP architecture may be done in at least two ways. The second simulation described above allows connections between a limited set of neural neighbors but all of the processors of the MPP are active at all times and a much larger network can be simulated. The update step in the second simulation runs in approximately one fifth the time of that of the first simulation and processes 24 times as many connections. This yields a factor of 120 in the number of connections processed per unit time. The experience of the authors suggests that the simulation of neural networks on massively parallel machines can be done in several ways not obvious at first glance and that alternatives should be examined carefully as some may yield higher computational rates than others. We first attempted to match the MPP to the matrix mathematics of the model. Then we tried to match the physics of the situation. It is interesting that the mathematical modeling got in the way here. In a sense, the hardware of the MPP more closely matches the brain that the mathematical model originally chosen.

# ARTIFICIAL NEURAL NETWORK ON A SIMD ARCHITECTURE

Joe R. Brown, Melissa M. Garber, and Steven F. Venable

*Martin Marietta Electronic Systems*
*MP 1304, P.O. Box 555837*
*Orlando, Florida 32855-5837*

## ABSTRACT

An implementation of a fully connected artificial neural network using the multi-layered perceptron model is described. The neural network is implemented on Martin Marietta's systolic array processor based on the Geometric Arithmetic Parallel Processor ($GAPP^{TM}$) chip. Arrays of GAPP chips make up a single instruction multiple data (SIMD) class machine which has fine-grained connections and is fully programmable. Previous application areas of the GAPP system are image/signal processing, computer vision, and knowledge-based processing. The neural network is a relatively new processing model for the GAPP, but one that readily maps onto the architecture of the overall array processor. The proof-of-concept neural network was a multi-layered perceptron model which used the back-propagation learning paradigm. This initial network had fewer than 100 nodes in three layers, and was trained to recognize letters of the alphabet. Work is progressing towards implementing a massive artificial network environment (more than 40,000 nodes and more than 10,000,000 connections) on the GAPP-based array processor. Alternate learning techniques are also being investigated.

**Keywords:** Neural Networks, Back Propogation, Parallel Processors, Single-Instruction Multiple-Data (SIMD), Character Recognition

## INTRODUCTION

The Image and Signal Processing Section of Martin Marietta Electronic Systems is developing artificial neural network implementations in the Geometric Arithmetic Parallel Processor ($GAPP^{TM}$). The GAPP is a Martin Marietta-developed systolic array processor, made up of one-bit processing elements, or cells, connected in a two-dimensional nearest-neighbor mesh. Adhering to the axiom, "the algorithm is the architecture," GAPP array size may be directly tailored to the size of the problem being solved, making possible many different configurations of GAPP systems. A single instruction multiple data (SIMD) class machine, previous applications of the GAPP system were image and signal processing, associative processing, and knowledge-based processing. The neural network is a relatively new processing model for the GAPP, but one that readily maps onto the architecture using techniques developed from the above mentioned application areas.

Artificial neural networks attempt to model the human cognitive process in a computer. They are massively parallel hierarchically interconnected networks of simple elements which interact with the real world similar to the way biological nervous systems do. Neural network models have the greatest potential in areas where many hypotheses are pursued in parallel and high computation rates are required, areas which take full advantage of the GAPP's parallel architecture. One such promising field of application is pattern recognition. Traditional artificial intelligence (AI) algorithms for pattern recognition are too specialized: they are designed to deal with information in a single, strict form. For example, in machine vision there are separate algorithms for boundary, disparity, curvature, shading, and spatial frequency information. Such algorithms typically use different computational schemes to analyze each type of information, so that fusing multiple types of information into a single general-purpose vision algorithm is difficult.

The neural network learning paradigm offers a unique solution to this problem. Requiring only a general algorithm for learning, neural networks automatically and dynamically determine what information is salient to a solution. The form that information must assume is therefore much less rigid than that required by traditional AI systems, allowing easier correlation of different levels of information. For example, this implementation uses pixel-level inputs to a network which recognizes letters.

Our first implementation in the GAPP system has been successfully trained on letters of the alphabet. After training, the implementation allows the user to place various inputs under a camera and in this way examine the extent to which the net has generalized the learning of the inputs. Via the camera, rotated and corrupted versions of the inputs can be provided to help in identifying some of the salient features determined by the net.

CH2649-2/89/0000/0043$01.00 © 1988 IEEE

## MODEL DESCRIPTION

The network model selected is that of a fully connected, three-layer feed-forward net containing a total of 91 nodes and 1728 weighted connections. For ease of initial development, the input domain of letters was broken into a 5-by-7 grid in which it is possible to pixelize all 26 letters of the alphabet. As these 35 cells were selected as the input to the net, the first layer, or input, layer contains 35 nodes, one per input pixel. Further, an input can only take on the value zero or one corresponding to the cell being off or on, respectively.

The third, or output, layer was selected to have 27 nodes, one for each letter of the alphabet and one to designate "other than letter" recognition. The second, or hidden, layer was initially chosen to be 25 nodes based on a rule of thumb stating that the hidden layer should be roughly 2/3 of the input or output layers. It was later increased to 27 nodes after some experimentation on a VAX-based network indicated that using these two additional nodes provided faster network settling.

While the input nodes can only take on values of zero or one due to input considerations, the values at both the hidden and output nodes are positive 10-bit fixed point numbers (10 bits to the right of the decimal point) and range from 0.0 to 0.99902. The decision to use 10 bits was somewhat arbitrary, although the decision was influenced by the amount of available GAPP memory. The output node with the largest value is selected as the total net output. If two or more nodes have the same value, they are all selected. Because the net is expected to output a value greater than zero and greater than the smallest non-zero value (0.000976), an additional constant-valued node was added to the output layer. The constant-valued node is selected if the value of all other nodes are less than or equal to it. This node is used to signal an internal network error condition and is not included in the count of 91 total nodes. The selected output nodes, including this special constant-valued node, are then highlighted on the display which shows each letter of the alphabet.

There are two other special nodes, one on the input layer and one on the hidden layer. Termed "bias nodes," these constant-valued nodes supply an offset or bias to the discriminant function at the node layer to which the node connects. These nodes have no incoming connections but do have weights and connections to the next node layer and for all practical purposes act as just another node on the layer (thus appearing as 36 input nodes and 28 hidden nodes). The value of these nodes is considered to be one (actually 1.0 on input, 0.99902 on hidden) so that the value of the weights on connections are the actual bias for the discriminant function.

The net model used is a fully connected net: each node on a layer is connected to all nodes (excluding the bias nodes) on the next layer (see Figure 1). Attached to each connection is a weight which is multiplied by the node value to provide the value arriving on the connection to the destination node. Weights in our implementation are assigned 15-bit fixed point numbers, ranging from $-16.0$ to $+15.9995$ (4 bits to the left of the decimal point, 11 bits to the right). Following the standard net training techniques, these weights are adjusted to provide a "trained network." Note that a weight of zero would map a non-connection.

## NETWORK EXECUTION

To exploit the power of the GAPP system, the model was laid out such that a unique processor cell is assigned to each nodal interconnect as well as assigning a processor for each node, thus utilizing a total of 1756 processors. The input node data is duplicated down rows of cells and then summed across the columns producing the hidden node values. These are in turn duplicated across columns and summed down the rows of cells, providing the final node values. This final node layer also uses one processor per node, utilizing a total of 1756 processors.

The network is executed by following the standard feed-forward operations. These processing steps are described below.

The value at node $k$ on any layer $j$ is defined as:

$$x_{kj} = f(\sum_{i=1}^{N} w_{ik} x_{ij-1} - \theta) \qquad (1)$$

where $f()$ is a limiting non-linear function (see Equation 2); $N$ is the number of nodes on the previous level $(j-1)$; $w_{ik}$ is the weight connecting nodes $i$ and $k$; $x_{ij-1}$ is the value of node $i$ on the previous level; and $\theta$ is the bias (or value of the weight on the connection from the bias node to node $k$).

1. **Place input in input nodes.** For training this is a fixed input with known correct output. For camera input this entails thresholding the camera's 8-bit data to values of zero and one, then pixelizing and shifting the data to the GAPP cells designated as input nodes.

2. **Spread the input across the weights and multiply.** As there is a distinct weight for the connection from each input node to each hidden node (i.e. 27 weights per input node), the input node values are spread across an array of 36 (35 inputs + 1 bias) by 27 GAPP cells. This distribution allows the process of multiplying the node values times the weight values to be done in parallel.

3. **Sum the values arriving at each hidden node and pass through the non-linearity function.** The products from step 2 are summed across all input nodes, including the bias. All hidden nodes are summed simultaneously in the GAPP system. These sums are passed through a pseudo-sigmoid function [$f(x)$ in Equation 2] to provide the hidden node value. The desired sigmoid function is $f(x) = 1/(1+e^{-x})$. But for easy implementation this function was coded as a group of conditionals

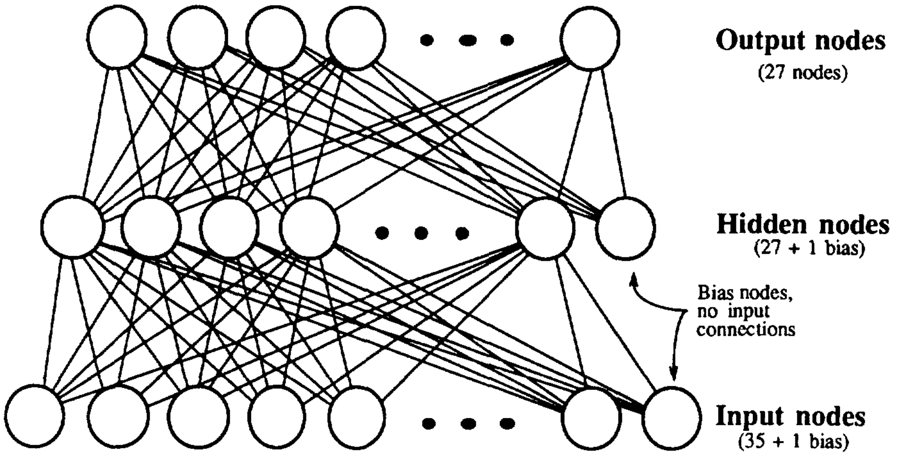


Figure 1: Network node connections

which simulate the sigmoid function as follows:

$$f(x) = \begin{cases} 0.99902, & \text{if } x \geq +5.0; \\ 0.000976, & \text{if } x \leq -5.0; \\ \frac{x+11}{16}, & \text{if } +1.0 \leq x < +5.0; \\ \frac{x+5}{16}, & \text{if } -5.0 < x \leq -1.0; \\ \frac{x+2}{4}, & \text{otherwise.} \end{cases} \tag{2}$$

4. **Spread the hidden node values across the weights and multiply.** Repeat step 2 using the values obtained in step 3 for the 28 hidden nodes and 27 output nodes.

5. **Sum and threshold.** Repeat step 3 using the values for the 27 output nodes.

6. **Determine the nodes with maximum value and display accordingly.** The value at each output node is compared to the others and those with the largest value are identified. The desired result is that only one node has the largest value. When this node (or nodes) is identified, a display is generated in which the letter corresponding to the node is highlighted.

## NETWORK TRAINING

The training algorithm selected is the back propagation technique which uses a gradient heuristic, enabling a network to self-organise for improved performance over time. Back propagation requires a specific training period in which the correct (or desired) output is known for each potential input that will be trained. All possible inputs do not need to be shown, but rather only a subset of the inputs. The actual sise and contents of this subset is not known and is expected to be domain dependent. Only those outputs that are trained can be expected to be correct.

Back propagation consists of adjusting the weights by a small amount based on the difference between desired node values and present node values after executing the net for the given input. The weight adjustment is based on the following equations:

$$w_{ij}(t+1) = w_{ij}(t) + \eta\delta_j x_i \tag{3}$$

where $w_{ij}$ is the weight on the connection from either input or hidden node $i$ to node $j$ on the next layer; $\eta$ is a gain term; $\delta_j$ is an error term given in Equation 4; and $x_i$ is the value of node $i$. For the hidden-to-output node level:

$$\delta_j = y_j(1-y_j)(d_j - y_j) \tag{4}$$

where $y_j$ is the value of output node $j$ and $d_j$ is the desired value for that output node. For the input-to-hidden node level:

$$\delta_j = x_j(1-x_j)\sum_k \delta_k w_{jk} \tag{5}$$

where $k$ is over all nodes in the layers above node $j$.

In our implementation, these steps are followed precisely with the gain term ($\eta$) variable at execution time. The best results have been achieved with a value of 0.0625 (1/16). While most literature references gains of 0.3 to 0.5, the gain used in our implementation is believed to be smaller due to the increased speed of training, errors in weight adjustment due to truncation, and lose of significant precision in the use of fixed point, rather than floating point operations.

The actual steps in training are:

1. **Execute the net.** The exact steps given under execution are performed with the correct output known for the input used. The only operations not performed are determining maximal output node and generating the display.

2. **Calculate the error delta ($\delta$) on the output layer.** Here the desired node value is considered to be 0.99902 if that is the node which corresponds to the letter being trained, with a value of 0.0 for all other nodes.
3. **Spread the delta and multiply.** This is the same operation as used to execute the net except the spread operation is in the reverse direction: the delta is spread from the output nodes over the previously spread hidden node values. These values are then multiplied by the node values and the gain ($\eta$).
4. **Alter hidden layer weights.** The product from step 3 is then added to the weights to produce new weights for the next execution iteration.
5. **Compute next layer error delta.** The previously computed delta (step 2) is multiplied by the weights and summed for all hidden nodes. This sum is then multiplied by the hidden node variance, per the back propagation algorithm.
6. **Spread the delta and multiply, alter input layer weights.** Repeat steps 3 and 4 using the weights on the input layer.

After these operations the network has had one iteration of training for the given input. Typically many iterations are performed for all possible outputs. In this example the training was performed on the four "perfect" (meaning no corruptions in the data) inputs of **T**, **A**, **X**, and *blank*. With some experimentation the primary features which the net extracted for three of the four inputs were found to be the center line for the **T**, left and right sides for the **A**, and most anything else for **X**. If no input cells were on, or if even any one at random was on, the *blank* was considered. Figure 2 shows the training input to the network and Figure 3 shows some test case inputs. Note that while the network was trained on only the inputs shown in Figure 2, the net can still recognize pieces of the inputs shown in Figure 3. After some analysis of the weights during training, it was found that all the desired output nodes were driven high while all other output nodes were low and remained so. All remaining iterations appeared to act only to differentiate between the inputs. Further experimentation and analysis must be done to study what the values at the hidden nodes represent.

## CONCLUSION

We plan to investigate at least two alternative learning strategies to provide more flexible training capabilities: Grossberg and Carpenter's Adaptive Resonance Theory (Ref. [3]) and Genetic Algorithms as described by Booker, Goldberg, and Holland (Ref. [1]). The Adaptive Resonance Theory self-organizes stable pattern recognition codes in response to an arbitrary input environment. This theory is based on multiple interacting memory systems to monitor and adaptively react to the novelty of events without an external teacher. Genetic Algorithms, which model gene pools, have been proposed mainly for pattern classification. They are based on replacing weak classifiers by recombining components from strong classifiers, similar to Darwin's theory of natural selection.

In summary, the artificial neural network is a promising processing paradigm which may be used to enhance existing AI techniques. As mentioned previously, future research activity in neural networks at Martin Marietta will examine alternate learning strategies and architectural topologies which are "GAPP-able". We will pursue applications specifically in the areas of image understanding and spatial reasoning, and, in general, expert systems which learn. With our past experience in pattern recognition and feature detection, coupled with our proprietary GAPP architecture, we believe that we can produce neural network systems that are well suited to real-world problems.

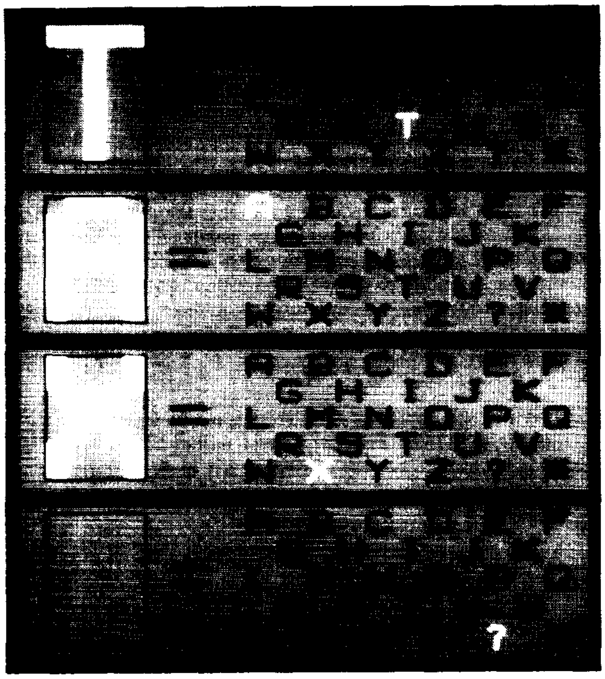


Figure 2: Training inputs

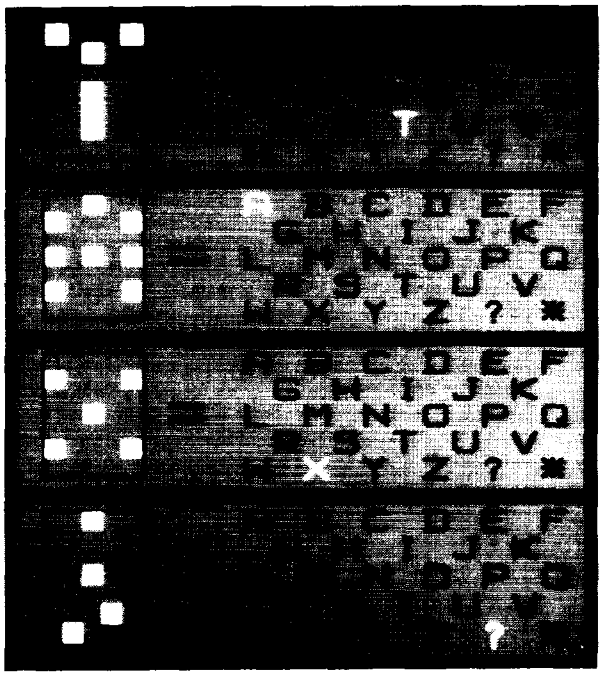


Figure 3: Test inputs

## References

[1] Booker, Lashon B., David E. Goldberg, and John H. Holland, *Classifier Systems and Genetic Algorithms*, The University of Michigan, Cognitive Science and Machine Intelligence Laboratory, Technical Report No. 8.

[2] Carpenter, Gail A. and Stephen Grossberg, *Associative Learning, Adaptive Pattern Recognition, and Cooperative-Competitive Decision Making by Neural Networks*, SPIE Vol. 634 Optical and Hybrid Computing 1986, pp. 218–247.

[3] Grossberg, Stephen, *Competitive Learning: From Interactive Activation to Adaptive Resonance*, Cognitive Science 11, 1987, pp. 23–63.

[4] Hopfield, John J., and David W. Tank, *Computing with Neural Circuits: A Model*, Science Vol 233, 8 August 1986, pp. 625–633.

[5] Jones, William P. and Josiah Hoskins, *Back-Propagation*, BYTE, October 1987, pp. 155–162.

[6] Kohonen, Teuvo, *Self-Organizing Feature Maps and Abstractions*, Proceedings of the Third International Conference on AI and Information-Control Systems of Robotics, 11–15 June 1984, Snolenice, Czk., pp. 39–45.

[7] Kohonen, Teuvo, *Clustering, Taxonomy, and Topological Maps of Patterns*, Proceedings of the Sixth International Conference on Pattern Recognition, Vol 1, 19–22 Oct 1982, Munich pp. 114–128.

[8] Lippmann, Richard P., *An Introduction to Computing with Neural Nets*, IEEE ASSP Magazine, April 1987, pp. 4–22.

[9] Tank, David W. and John J. Hopfield, *Collective Computation in Neuronlike Circuits*, Scientific American, pp. 104–114.

# CHARACTERIZING THE ERROR FUNCTION OF A NEURAL NETWORK

Barbara Moore*, Marcelo Fogaça*, and Alan Kramer†.

*MIT AI Lab, NE43-826, 545 Technology Square, Cambridge MA 02139
†Department of EECS, Cory Hall, UC Berkeley, Berkeley CA 94720

## Abstract

We consider several means of exploring the error function of a multi-layer, feedforward neural network. In particular, we look at hyperplane configurations over time, and the generalization of the network function to a region of the input space. Using this approach, we analyze the results of several experiments run on the massively parallel Connection Machine computer, as well as other reported results and observations.

*Keywords:* Neural Networks, Error Function, Layered Feedforward Network, Connection Machine, Hyperplanes, Geometric Analysis of Neural Networks.

## 1 Introduction

Feedforward neural networks can "learn" surprisingly complex input/output mappings using simple gradient-descent algorithms which minimize an error function whose variables are the network weights. These networks are an attractive computational paradigm not only for their adaptation capabilities, but also because they have natural parallel implementations.

However, even massively parallel implementations of neural networks may require extremely long training times. We are hoping to reduce training times by incorporating into the learning algorithm knowledge of both the structure of the problem to be learned and the learning network topology. To this end, we are investigating various means of characterizing the shape of the error function in weight space, for the case of a feedforward layered network and the sum-of-squared-differences error function. We have also found it useful to look at the movement in input space of hyperplanes corresponding to the weights on links to hidden units, over the course of training with the classical back-propagation algorithm. (This representation of network state is referred to below as the hyperplane configuration.) There is often structure inherent in a network's architecture and in a problem (or training set) which is reflected in the error function and in the hyperplane configurations over time. An understanding of this structural information can help to better determine parameters such as size, connectivity, and initial weight values of the learning network, and can contribute to the design of faster learning algorithms and input preprocessing stages. Our analysis also allows us to provide explanations for empirically observed phenomena, such as "flat spots" in the energy surface, and improvements in performance with extra hidden units.

In Section 2 we describe the implementation of the back-propagation network training algorithm on the Connection Machine. Section 3 presents results of a scaling experiment, in which we study the effect on learning time of the number of hidden units. We also present results of an experiment on adjusting the length of the initial random weight vectors. Section 4 discusses our explorations of the shape of the energy function in weight space, and developes the hyperplane configuration. In Section 5 we apply the hyperplane analysis to explain our experiments and other empirically observed phenomena. Section 6 suggests several methods for obtaining speedup in training times based on our analyses, and summarizes the paper.

## 2 Connection Machine Implementation of Back-Propagation

To run our experiments, we wrote an optimized Connection Machine version of the back-propagation learning algorithm for multilayer feedforward neural networks. (We do not describe that algorithm here; see reference [4]. The node function for the units is assumed to be the sigmoid function varying between 0 and 1.) The Connection Machine's massive parallelism is well suited for neural network implementations. For most of the problems we experimented with, the Connection Machine allowed us to completely parallelize the network, with one processor allocated to each node and each weight.

CH2649-2/89/0000/0049$01.00 © 1988 IEEE

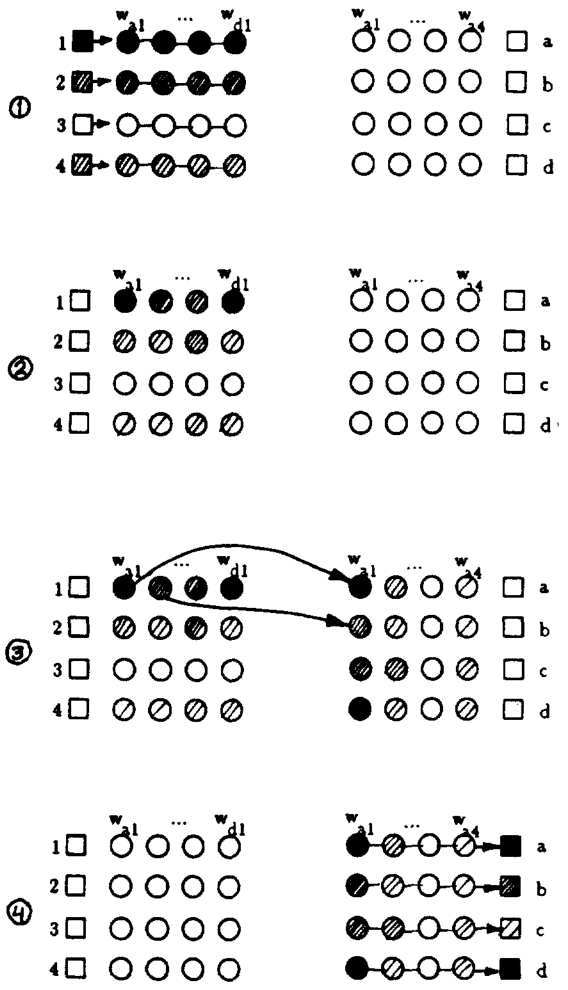


Figure 1. Implementation of neural network on the Connection Machine: snapshots during feed-forward of activity from one layer to the next. Squares at left (1,2,3,4) represent node processors in layer $i$. Circles to their right represent forward-link processors from layer $i$ to layer $i+1$. Squares at right (a,b,c,d) are node processors for layer $i+1$. Circles preceding squares are backward-link processors from layer $i$ to layer $i+1$. In the first snapshot, activities at the layer $i$ node processors are forward scanned into their forward-link processors. In the second snapshot, these activities are scaled by the weights in each forward-link processor. Third, the link values are routed from the forward-link to the backward-link processors. Fourth, a forward scan is used to sum link values into the layer $i+1$ nodes, where the sigmoid function is applied to obtain layer $i+1$ activities.

The Connection Machine is a powerful fine-grained parallel machine having between 4000 and 16000 processors. It is a single-instruction multiple-data (SIMD) machine. Each of the processors is a simple 1-bit processor with 4000 bits of memory (the CM-2 has more memory and floating point operations). There are two modes of communication among the processors. In the first, the processors are connected by a mesh of wires into a two-dimensional grid network (the NEWS network, so-called because the connections are in the four cardinal directions), allowing rapid direct communication between neighboring processors. The second communication mode is the *router*, which allows messages to be sent from any processor to any other processor in the machine.

The back-propagation algorithm that we implemented on the Connection Machine takes advantage of the fast parallel *scan* operation [1]. One processor is assigned to each node in the neural network. Each node processor is preceded by a linear string of "backward-link" processors, each of which stores information corresponding to a link from a node in the previous layer. Each node processor is also followed by a set of "forward-link" processors, each corresponding to a link to a node in the next layer. The purpose of this linear arrangement is to allow use of the fast scanning operation to pass information along the linear string of node and link processors.

Forward propagation starts with the inputs being loaded into the first layer node processors. Then a forward scan operation sends the activities from these nodes into their associated forward-link processors. Each forward-link processor multiplies this activity by its weight value, and the routing network of the Connection Machine is used to send this value to the corresponding backward-link processor in the next layer of processing nodes. The final step in the loop is another forward scan which adds the values of the backward-link processors into the node processor for the second layer. Then each node processor in this layer computes the sigmoid output function and we start again on the next layer with a forward scan (Figure 1).

Back-propagation of error works in a similar way, using the backward scan operation instead of the forward scan. Errors instead of activities are transmitted backward through the network, and weight values at the node processors are updated.

## 3 Some Experimental Results

The problems we are investigating include the parity function of $n$ inputs and classification of $n$-dimensional vectors over real-valued intervals: $R^n \rightarrow \{0,1\}$. The parity function is useful because it scales in an obvious way, and because it is the most complex Boolean mapping to learn. Many of our experiments used two-input XOR as the input; although occasionally maligned as being somehow unrepresentative of problems to which neural networks ought to be applied, we note that it is in fact very useful as the simplest binary problem which requires a hidden layer in a network. The problem of classifying real-valued vectors problem since there has been less work with real-valued inputs than with binary-valued inputs, although many potential applications involve analog data.

### 3.1 Learning Time as a Function of Hidden Layer Size

One critical question in the design of a neural network is the number of units in the hidden layers. There is often a minimum hidden layer size below which the network is not capable of representing certain input-output mappings. Even above this minimum size, however, it may still be difficult for the network to learn. By running experiments, we have confirmed reported results that adding hidden units reduces the learning time. The speed of our parallel implementation allowed us to run enough experiments to get reasonable statistics on this phenomenon.

In all of the experiments described in this paper we used a network with one hidden layer. The input layer had two units and the output layer had one unit, while the number of units in the hidden layer was varied. We used the XOR problem as the mapping to be learned by the network. The learning algorithm employed was back-propagation, with a learning rate of 1.0 and the momentum term set to 0.0. The initial weights of the connections in the network were randomly initialized to values in the range -1.0 to 1.0. In each run, the stopping criterion adopted was that the error must be less than or equal to 0.02. The learning time of a network is then defined as the minimum number of passes through the backpropagation algorithm that are necessary to bring the error of the network to 0.02 or less. (We make the assumption that results on learning times obtained for one set of parameters scale simply for other values of these parameters. Setting the momentum term to 0 was useful in that it allowed us to interpret our results as reflecting the true shape of the energy function in weight space.) In our experiments we varied the hidden layer size from 2 to 128 units.

Table 1 summarizes our results on the scalability of the hidden layer size. It confirmes reported results that adding hidden units reduces the learning time of a network.

The relationship between the learning time and the size of the hidden layer resembles a hyperbolic curve (Figure 2), and so we looked for a linear correspondence between the learning time and the inverse of the size of the hidden layer for a particular network. Figure 3 shows a plot of learning time as a function of the inverse of the size of the hidden layer. A linear regression was performed using the data for 2 to 48 hidden units in order to obtain an equation relating the number of hidden units and the learning time. (As discussed in section 5, this model breaks down for a large number of hidden units because of overshoot effects.) The equation obtained was:

| *hidden units* | *trials* | $\bar{t}$ |
|---|---|---|
| 2 | 60 | 968 ± 254 |
| 3 | 60 | 802 ± 144 |
| 4 | 60 | 719 ± 128 |
| 5 | 60 | 665 ± 114 |
| 6 | 60 | 618 ± 93 |
| 7 | 60 | 597 ± 85 |
| 8 | 60 | 580 ± 59 |
| 9 | 60 | 544 ± 55 |
| 10 | 60 | 545 ± 57 |
| 12 | 40 | 504 ± 56 |
| 16 | 40 | 484 ± 55 |
| 20 | 40 | 467 ± 44 |
| 24 | 40 | 453 ± 36 |
| 32 | 40 | 452 ± 34 |
| 48 | 40 | 432 ± 34 |
| 64 | 40 | 389 ± 24 |
| 96 | 40 | 318 ± 33 |
| 128 | 40 | 373 ± 61 |

*Table 1.* Learning time vs. number of hidden units.

$$t = \frac{A}{h} + B, \qquad \text{where}$$

$$A = 1140,$$

$$B = 420,$$

$$t = \text{learning time},$$

$$h = \text{number of units in hidden layer.}$$

The model described by the equation above was tested with our previous results, as described in the Table 2. The values for the learning time that were predicted from the model equation fell within 3.3 % of the mean value obtained for each case.

| *hidden units* | *predicted* $\bar{t}$ | *actual* $\bar{t}$ | *error* |
|---|---|---|---|
| 2 | 990 | 968 | + 2.3 % |
| 3 | 800 | 802 | - 0.2 % |
| 4 | 705 | 719 | - 1.9 % |
| 5 | 648 | 665 | - 2.6 % |
| 6 | 610 | 618 | - 1.3 % |
| 7 | 583 | 597 | - 2.3 % |
| 8 | 563 | 580 | - 2.9 % |
| 9 | 547 | 544 | - 0.6 % |
| 10 | 534 | 545 | - 2.0 % |
| 12 | 515 | 504 | + 2.2 % |
| 16 | 491 | 484 | + 1.4 % |
| 20 | 477 | 467 | + 2.1 % |
| 24 | 468 | 453 | + 3.3 % |
| 32 | 456 | 452 | + 0.9 % |
| 48 | 444 | 432 | + 2.8 % |

*Table 2.* Fit of model to data.

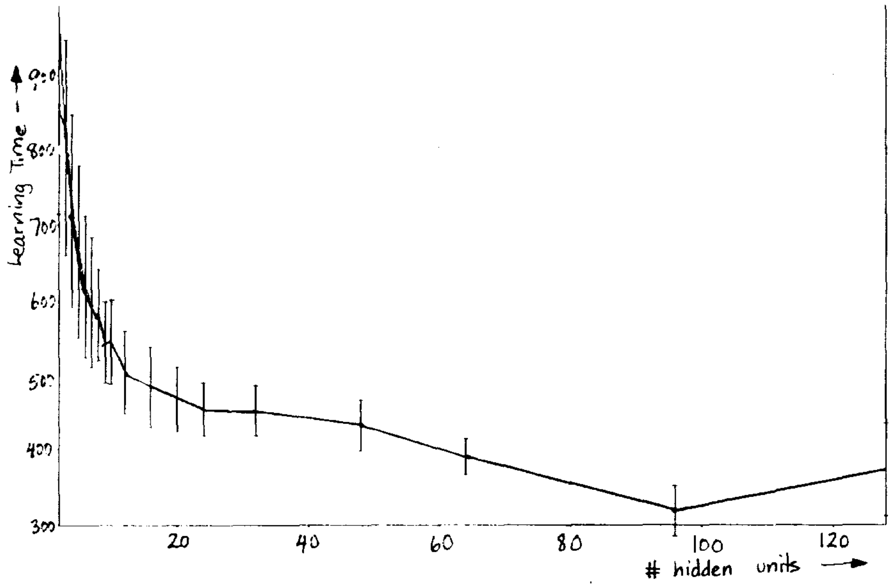


Figure 2: Graph of learning time as a function of the number of hidden units on the XOR problem.

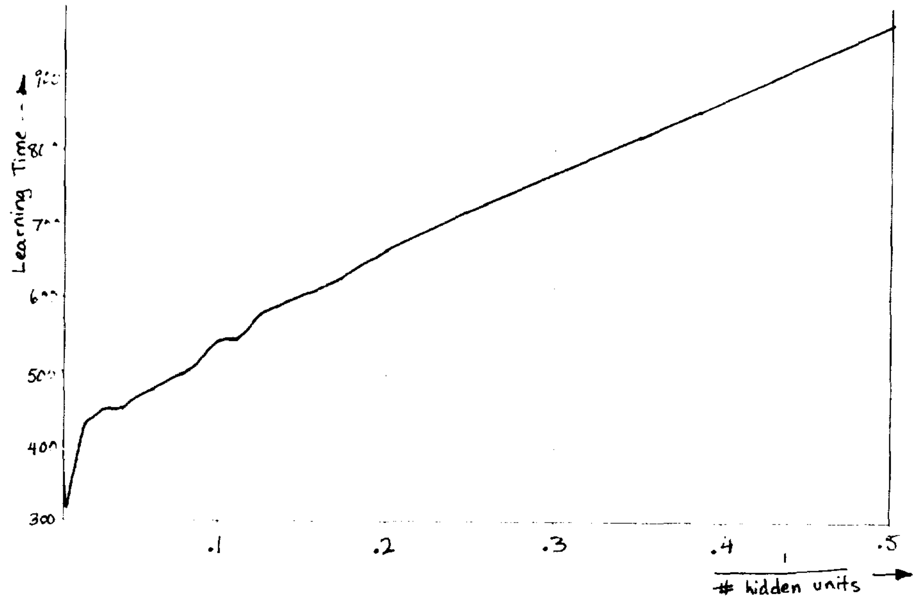


Figure 3: Graph of learning time as a function of the *inverse* of the number of hidden units on the XOR problem.

Further, the model was tested with new experimental data obtained independently of that from Tables 1 and 2. The experiment was performed in the same manner, but different values for the number of units in the hidden layer were used. The results are summarized in Table 3. The model was again able to predict the learning time for all new cases with an error variation not greater than 5 %.

| *hidden units* | *predicted* $\bar{t} \pm \sigma$ | *actual* $\bar{t}$ | *error* |
|---|---|---|---|
| 15 | 496 | 496 | 0.0 % |
| 30 | 458 | 441 | + 3.8 % |
| 45 | 445 | 440 | + 1.1 % |

*Table 3.* Fit of model to new data.

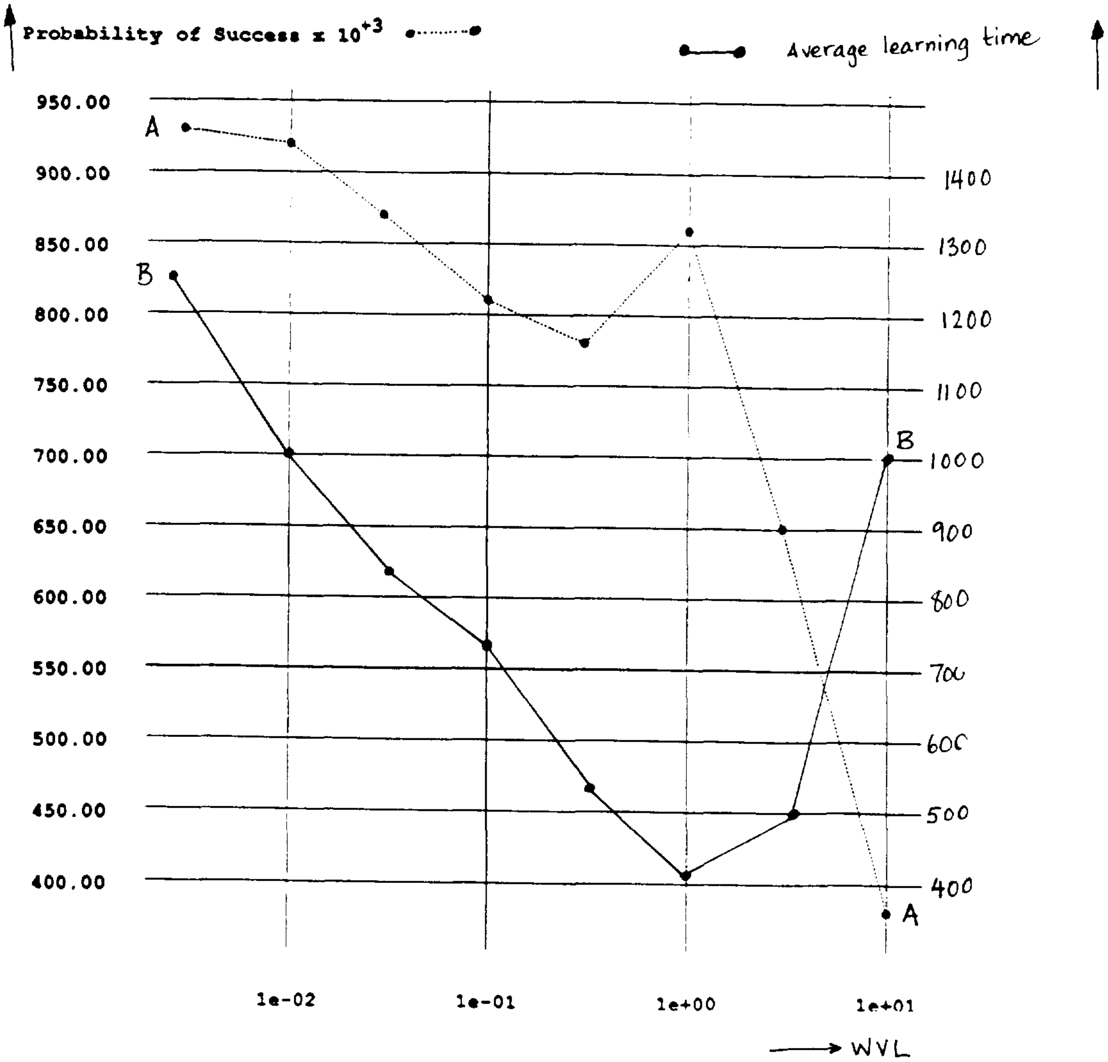


Figure 4: A. Probability of success as a function of initial weight vector length. B. Learning time as a function of initial weight vector length.

## 3.2 Learning time as a function of initial weight vector length

Weight values are usually initialized to "small" random values. We have found that the length of this initial weight vector is important and has a strong effect on measured learning time results, at least for the XOR problem with two hidden units. For all of our experiments, initial weights were chosen by randomly choosing a point in the unit-radius hypersphere according to a uniform probability distribution, and then normalizing the length of the weight vector to a user-specified length. (To achieve a uniform distribution over the unit-radius hypersphere, we chose points in the unit hypercube by choosing each coordinate independently from a uniform distribution, and then rejected points which were not within the unit hypersphere. While this method works well for a small problem like XOR which has a nine-dimensional weight space, it becomes impractical for much larger problems because too many of the randomly chosen points are rejected before a "hit".)

The learning algorithm used was batch back-propagation with a learning rate of 0.25 and a momentum of 0.9. The algorithm terminated with a "success" if the error was reduced to less than 0.125. Learning time for successful trials was measured in epochs, or passes through the entire training set. The algorithm terminated with a "failure" if it had not succeeded after 10,000 epochs.

Graph A in Figure 4 shows that for XOR, the probability of back-propagation terminating successfully does depend on initial random weight vector length. Eight initial

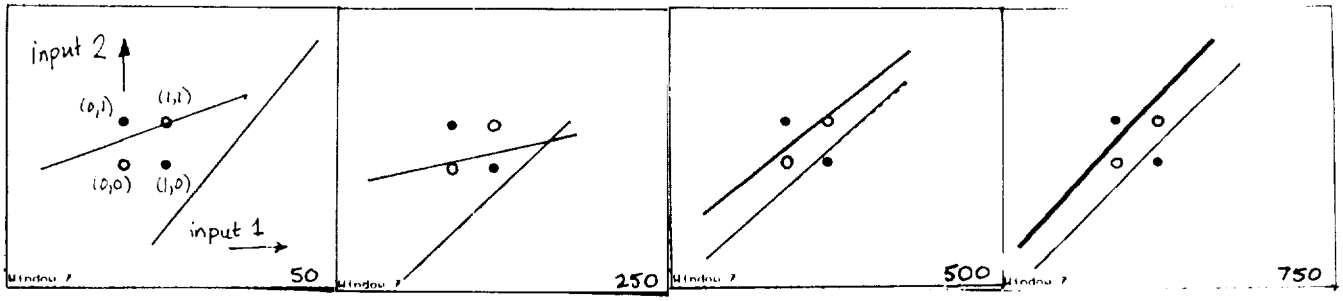


Figure 5: Hyperplane configurations in learning the XOR problem with two hidden units. Snapshots at epochs 50, 250, 500, and 750.

random weight vector lengths were chosen and these values appear on the WVL axis. For each length, 100 random initial weight vectors were generated and run through our learning algorithm. The percentage of these hundred runs which terminated successfully is plotted and is seen to decrease with increasing WVL.

Graph B in Figure 5 shows that average learning time does depend on initial weight vector length. Eight initial random weight vector lengths were chosen and these values appear on the WVL axis. For each length, 100 random initial weight vectors were generated and run through our learning algorithm. The average number of learning sweeps needed for the trials which terminated successfully is plotted, and shows that learning time increases if the length of the initial random weight vector is too small, as well as if it is too big.

# 4 Exploring the Error Function in Weight Space

We have considered several ways of looking at the error function in weight space. The error function is defined as the sum of the squares of the differences for each input in the training set of the desired output and the actual network output. For the two-input, two-hidden-unit XOR problem, we numerically explored the energy function in weight space, and derived the error equation analytically (for these results, see the forthcoming AI working paper [2]). We ran many backpropagation experiments. We watched the evolution over time of the hyperplanes representing the hidden units, and we performed a geometric analysis of final weight configurations. We found it very useful in practice to overlay the hyperplanes on a color image representing the function computed by the network (with a range of colors from blue to red representing values from 0.0 to 1.0). We now develope the hyperplane configuration approach.

Consider a sigmoid unit in a feedforward neural network, which computes the function

$$f(x) = \frac{1}{1 - e^{-x}},$$

where $x$ is the weighted sum of inputs to the unit. The unit will output a value of .5 whenever its weighted inputs sum to 0. Given weights on the input links, then, the locus in input space of values which will produce a 0 input to the node is a hyperplane. In the case of two inputs, this is a line. For binary output problems, a network that has learned to output correct responses to training set elements has a hyperplane configuration in which hyperplanes separate training inputs which have different outputs.

We plotted snapshots of the movement of these lines in input space over the course of learning. Figure 5 is an example of training a 2-hidden-unit network on the XOR problem. The four training set elements are represented in the input space by the four circles. The filled circles at (1,0) and (0,1) have a desired output of 1, and the open circles at (0,0) and (1,1) have a desired output of 0. The final solution, with the parallel lines separating the two classes of inputs, is in one of only two possible configurations (the other configuration has the parallel lines cutting the training examples the other way).

Figures 6 and 7 illustrate snapshots during the training of a 3-hidden-unit network on XOR. The final configuration in Figure 6 is similar to that of a 2-hidden-unit network; a glance at the output weights for each hidden unit reveals that the "extra" hyperplane has weight much smaller than the other two. On the other hand, the final configuration in Figure 7 is structurally dissimilar from that of a 2-hidden-unit network: a star of lines is formed instead of two parallel lines. All three of the hidden units are important in the network, and the magnitudes of the weights between them and the output node are approximately equal.

We can characterize the final configurations of the hyperplanes into informal equivalence classes. For example, the 2-hidden-unit solution with parallel lines is one equivalence class (the two possible orientations are considered to be in the same equivalence class, by symmetry). The star configuration is another equivalence class. Thus we found that increasing the complexity of the learning network not only allowed for variations of existing final configurations, but also added new structural equivalence classes. We are

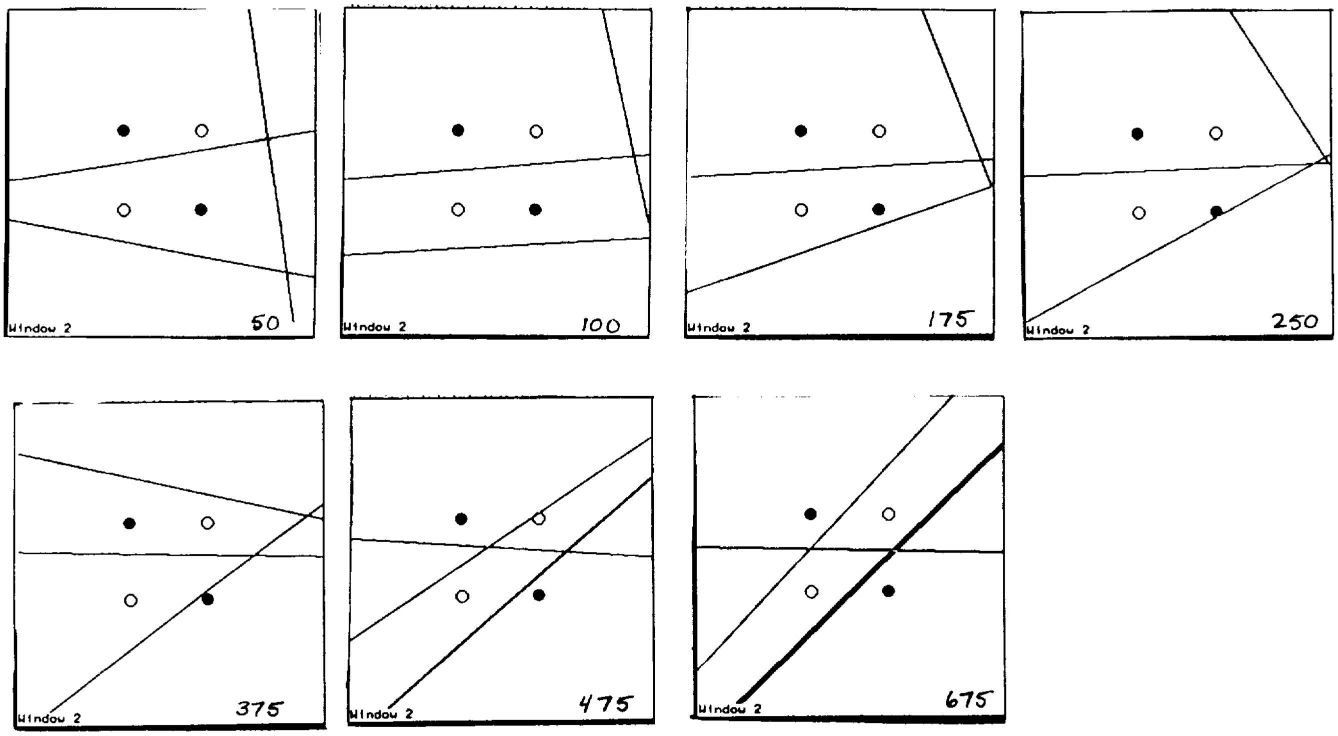


Figure 6: Hyperplane configurations in learning the XOR problem with three hidden units. Final configuration is similar to that of 2-hidden-unit network. Snapshots at epochs 50, 100, 175, 250, 375, 475, and 675.

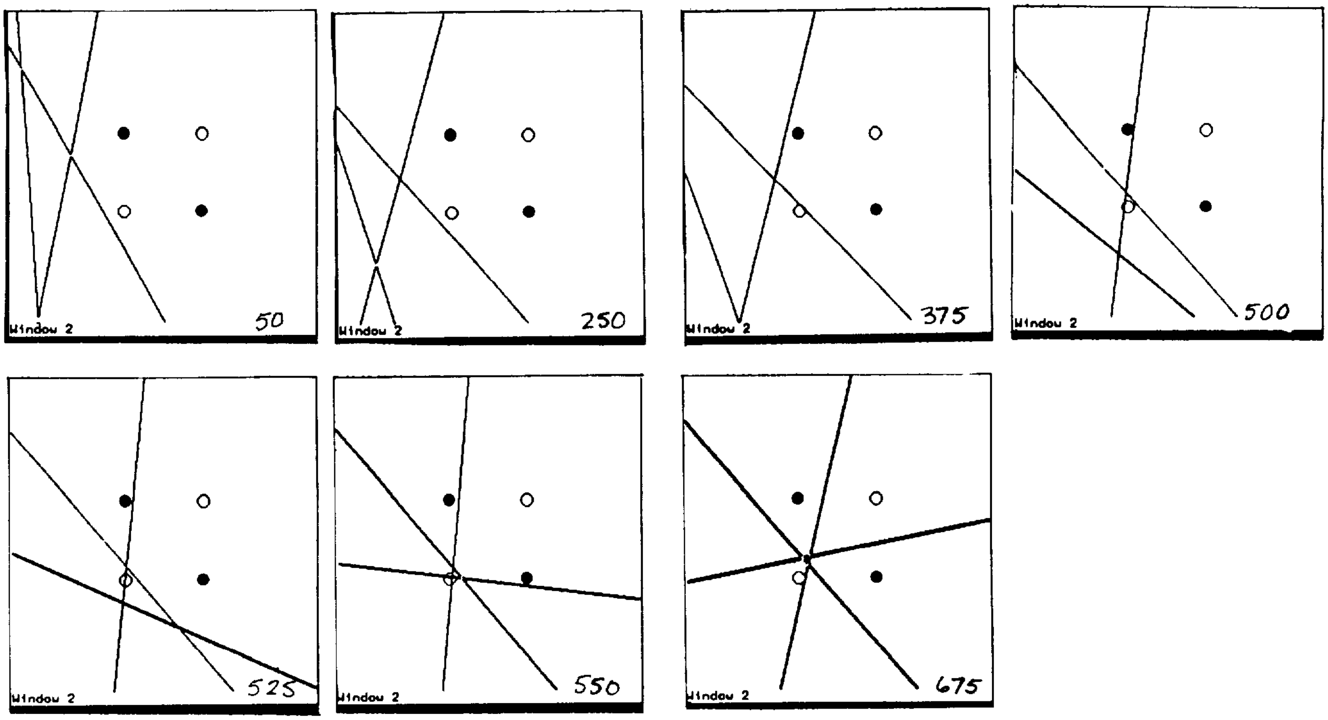


Figure 7: Hyperplane configurations in learning the XOR problem with three hiddenunits. Final configuration is *not* similar to that of 2-hidden-unit network. Snapshots at epochs 50, 250, 375, 500, 525, 550, and 675.

currently working on the problem of describing the regions of attraction in weight- or hyperplane-space for each of the final configurations. In particular, we noticed that there are long learning times corresponding to initial configurations which lie on the boundary of these regions of attraction, indicating that it takes some time to make the decision about which direction in which to go.

The motion of the hyperplanes during the course of learning (as well as the color map of the network function in input space) shows a lot about the shape of the error function. Slow movement of hyperplanes corresponds to low derivatives in the error function. Fast and often oscillatory movements of the hyperplanes indicate rapidly changing areas of the error function; the oscillation is due to overshoot of a local energy minimum along the direction of the gradient at a point.

## 5 Explaining Empirical Observations

The hyperplane configuration approach has provided answers or clues for various observed phenomena, as well as pointing out new questions we had not previously asked. We were particularly interested in such phenomena as speedup in learning with increased number of hidden units, "flat spots" in the error function, the initial shrinking of the weight vector, symmetry-breaking and hidden unit differentiation, and the success of the heuristic of removing hidden units after learning has been partially completed. By watching the hyperplanes as learning progressed, we were particularly struck by certain oscillatory effects, fast vs. slow movements, and analogies with attractive and repulsive forces.

Let's start by considering the experiments in which we varied the number of units in the hidden layer. We found a steady decrease in the learning time, up to a point, after which the learning time began to rise again. By looking at the hyperplanes and the color graph of the network function in input space, we could see that one of the explanations for decreasing learning time was that more final configurations were available to the network, and it had more freedom in choosing weight values with more hidden units and redundancy. For only two hidden units, the values of the weights could not vary much at all before violating the required error bound. Another thing we noticed was that the effective step size increased dramatically as the dimensionality of the weight space increased: the step size in the gradient descent algorithm is used by each weight in adjusting itself, so the more weights there are, the larger the total step size. Therefore, we were seeing a lot of oscillation in the large networks because there was a lot of overshoot of local minima in the direction of the gradient.

On many of our training sessions, we noticed that the length of the weight vector would often change rapidly over a brief period right away, and this would be accompanied by a rapid reduction in the error. After this there would be a leveling off of both signals. This was easy to understand when we looked at the plot of the network function over the input space: the first thing that happens is that the output layer weights are adjusted so as to bring the output function of the network to be approximately .5 everywhere. Usually the hyperplanes do not intersect the unit hypercube that is defined by the binary training vectors, because they are chosen at random. Then the hyperplanes would be slowly dragged in toward the unit hypercube; the farther away they were from the cube the slower they would move. Thus the initial location of the hyperplanes is a key factor in learning time.

Another interesting aspect was symmetry-breaking and hidden unit differentiation. We observed several cases where one or more of the hyperplanes was in a location near a "decision point". On one side of this point, the hyperplane would go in one direction, and on the other side of this point, it would be pulled in a different direction. Hyperplane movement near this decision point was generally quite slow. There were also times when more than one hyperplane was serving the same function (located in a similar place), and one was needed elsewhere. It would take a long time for the two hyperplanes to differentiate.

If two networks have been trained successfully on a problem, the smaller one tends to generalize better. However, the smaller networks also tend to become trapped in unacceptable local minima more readily during the training phase. Therefore researchers have investigated the possibility of removing hidden units after learning has been partially completed. Our conclusion is that this heuristic works well only under certain conditions, wherein "redundant" hidden units are chosen for elimination. Often useless or redundant units have smaller weight vector magnitudes, so this is a good heuristic.

We also did some experiments with larger-size problem such as parity of $n$ inputs and more complex classifications of two real-valued inputs. We found that the learning time of the networks tended to correspond to the number of hyperplanes that were needed to separate the inputs in different classes, and we used this as an informal definition of the complexity of the problem.

## 6 Speeding Up the Learning Algorithm and Future Work

Several approaches to obtaining speedup in training times have suggested themselves from our experiments. One is the choice of initial weights. By starting with the hyperplanes near the unit hypercube and separated from each other, we very much decreased the learning times. Various computational geometry algorithms are also attractive for determining initial hyperplane configurations; depending on the complexity of these algorithms as a function of input dimensionality and training set size, they might even become a replacement for back-propagation network training methods. For large training sets, we would like to consider alternatives such as those suggested by Stephen Omohundro [3].

We also implemented a steepest descent algorithm, and the more efficient Fletcher-Powell algorithm for determining the next direction in which to change the weights combined with a quadratic interpolation for optimum step size. This algorithm is parallelizable and achieves learning rates at least as good, and usually better, than others reported in the literature.

We are currently working on characterizing the regions of attraction of the final hyperplane configurations. In addition to studying the hyperplane configurations experimentally, we hope to make a theoretical statement about the hyperplane configurations of final solutions obtained by the back-propagation learning algorithm. We may be able to derive analytically the configurations which correspond to local minima of the error function in weight space. We are also interested in considering the movement of the hyperplanes in terms of forces between them and the training set inputs, as well as among hyperplanes themselves.

We would like to look at the behavior of other learning algorithms in hyperplane configurations. For example, we are particularly curious to try it on David Rumelhart's new scheme which adds terms to the error function that reduce the size of the network dynamically during learning.

## References

[1] G. Blelloch, "Scans as Primitive Parallel Operations", *Proceedings International Conference on Parallel Processing*, pp. 355-362, 1987.

[2] B. Moore, "Error Function Explorations, or Beating XOR to Death," forthcoming MIT AI Lab Working Paper.

[3] S. Omohundro, "Efficient Algorithms with Neural Network Behavior," Report no. UIUCDCS-R-87-1331, Department of Computer Science, Univ. of Ill. at Urbana-Champaign. April 1987.

[4] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning Internal Representations by Error Propagation," in D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Bradford Books/MIT Press, Cambridge MA, 1986.

[5] A. Wieland and R. Leighton, "Geometric Analysis of Neural Network Capabilities," *Proceedings of the IEEE First International Conference on Neural Networks*, pp. 385-392, 1987.

# THE IMPACT OF RENT'S RULE ON MASSIVE PARALLELISM

P. J. Koopman  D. P. Siewiorek
ECE Department  CS Department
Carnegie Mellon University
Pittsburgh, PA

**ABSTRACT:** Rent's Rule is an empirical relationship stating that the number of pins on a chip increases as the number of gates on the chip increases. In massively parallel systems, every extra pin is multiplied by the number of processors. This causes a rapid increase in system complexity, cost, and failure rate. The key to more efficient massively parallel systems is finding a way around Rent's Rule. By studying the effects of re-implementing a system of fixed complexity using different integration levels, we have found that Rent's rule does not apply to systems which place program memory on the same chip as the processor. This suggests that a focus for massively parallel systems might be to use processing elements simple enough to completely fit on a single chip, rather than faster but more complex processors that use external memory devices.

**Keywords:** Rent's Rule, integration level, system complexity.

## INTRODUCTION

Rent's Rule (Ref. 1) is an empirical relationship between the number of gates and the number of I/O pins a single chip. The relationship is given by:

$$IO = AS * G^{R}$$

In this equation, **IO** is the number of input/output pins on the chip. **AS** is the complexity of a single logic gate on the chip as measured by the number of inputs for the gate. **G** is the number of logic gates on the chip. **R** is the Rent Exponent, which is a circuit-dependent "magic" number between 0 and 1, which is often near 0.5.

The trend in VLSI processor design has been: given the availability of more silicon real estate, put more sophisticated functions or wider data paths into a single chip. Thus, memory chips have progressed from 256K bits to 1M bits to 4M bits. Also, microprocessors have evolved from 8 bits wide to 16 bits to 32 bits. These chips all obey the Rent's Rule prediction of a logarithmic increase in the number of pins as the number of gates on the chip increases. This increase in the number of pins has important implications for the builders of massively parallel systems.

## THE COST OF TOO MANY PINS

Since the innovation of standardized integrated circuits we have progressed from the introduction of the 14-pin dual inline package (DIP) to common use of pin grid array packages (PGAs) with hundreds of pins. The addition of extra pins to a chip has some obvious as well as hidden costs.

The most obvious cost is the manufacturing cost of the chip package itself. Small DIPs are very inexpensive to manufacture since they use stamped metal pins. As chips require more pins, DIPs become impractical, and packages such as leadless chip carriers (LCCs) are used. Each contact on an LCC costs more than a DIP pin, because it must be more precisely manufactured and placed around four sides of the package. At the high end of the spectrum, PGAs use precision-machined round pins that are precisely placed in a matrix on the back of the chip. It is not unusual for each pin of a PGA to be several times more expensive than an entire DIP package. Thus, there is a very steeply increasing cost curve for the entire chip as the number of pins is increased.

The direct cost of the chip package only begins to describe the costs of adding pins, however. At the on-chip level, every pin must have an on-chip pad. This pad consumes valuable silicon real estate. But, more importantly, each pad consumes power. As geometries become smaller, the amount of power used by a chip to drive its output pins can dwarf power consumption for on-chip logic. The problem is especially severe with CMOS technology, which is coming into favor for high density circuits.

At the system level, the footprint of the package on the printed circuit board increases as the number of pins on the package increases, costing valuable printed circuit board real estate. Increased printed circuit board sizes result in bigger cabinets and, ultimately, more boards with expensive and

CH2649-2/89/0000/0059$01.00 © 1988 IEEE

slow interconnect structures. Dense pin arrangements such as those found on PGAs further aggravate the problem by requiring expensive multi-layer boards.

Many indirect costs are associated with chips that have large pin counts. These costs include the use of very expensive automated chip testers when they are manufactured. Also, every extra pin in a finished computer reduces the over-all system reliability, since interconnect failure is a frequent system failure mode. (Ref. 2)

In a parallel computer, these pin costs are multiplied by the amount of parallelism in the design. Since the premise behind a massively parallel system is that more processors are better, all massively parallel designs will ultimately be limited in processing power by the number of processors that can be afforded within a given space/power/cost budget. The number of pins in each processing element within the system can therefore directly affect the ultimate computational power of a massively parallel processor.

## BREAKING THE RULE

One should not infer from the previous discussion that the use of VLSI chips with large pinouts is bad. These chips, while expensive, are less expensive than the total system cost of using a large number of less complex chips instead. The question is: can we do better?

Rent's Rule predicts that increased VLSI chip complexity will lead to an inexorable increase in pin count. A key to making massively parallel systems faster and more cost-effective is to find a way to break out of Rent's empirical relationship. One way to accomplish this goal is to find an implicit assumption in the relationship that can be altered.

There is an historic relationship between chip complexity and overall system complexity. As chips have become more dense, computer systems have not only become more highly integrated, but have also become more complex. Adding more complexity to a system makes sense in a uniprocessor environment, where the added complexity squeezes the most possible performance from the machine. Rent's Rule applies to computer systems as they have been built over the years, so it accounts for this implicit assumption. But what if we violate that assumption, and hold system complexity as a constant?

The answer to this question may be found by conducting an experiment that holds system complexity as a constant for varying integration levels. In order to do this, we built a hierarchical description of a 16-bit microprocessor system (Ref. 3) starting at the gate level. All circuit functions were ultimately reduced to combinations of 2-input NAND gates for simplicity. Then, we did a redesign of the system using six different integration levels ranging from SSI (all 2-input NAND gates) to high density VLSI (entire system on a single chip). Each integration level was chosen to correspond to a reasonable method for partitioning the system components. Figure 1 shows a graph of chip complexity versus pin count for the various implementations, as well as the curve for a Rent exponent of 0.38. There are more than six data points in this graph, since most implementations had several chips in the design. RAM chips are not shown as they obey a Rent curve with a different slope that clutters the diagram. Power supply pins are not accounted for since they vary with implementation technology.

Integration Levels 1 and 2, which correspond to SSI and MSI components, obey a classic Rent's Rule curve with an exponent value of approximately 0.38. Integration Level 4, which corresponds to a 3-chip system, also falls neatly on this curve. Integration Level 5, which corresponds to a standard micro-processor 2-chip system (processor chip and memory chip) is somewhat off the curve, but is still a reasonable fit. Integration Level 3 turned out to be an awkward level of integration, which forced a very poor partitioning of the system, resulting in a very high pin count for one of the chips.

The really interesting point on the graph is Integration Level 6. This design is nowhere near the curve! Integration Level 6 corresponds to a single-chip system, which incorporates program memory and the processing logic on the same chip. This implementation appears to break Rent's Rule.

## INTERPRETING THE RESULTS

Figure 2 shows a curve that helps us interpret the results of the experiment. If we ignore Integration Level 3 as a "bad data point", then what is really happening is that the designs obey Rent's rule quite well through Integration Level 4. Then, as we reach very high levels of system integration, the number of pins on the chips begins to decrease. If the entire system is on a single chip, only a few pins for system I/O are needed. While the microprocessor seems to be near the break in the curve, the break is not really noticeable until the system-on-a-chip approach is taken.

The results, once one thinks about the situation, are rather straightforward. A system-on-a-chip needs off-chip interconnection only for I/O, so it needs very few pins. Why hasn't this concept been exploited then? The reason is that it is of limited use in the uniprocessor world. Most high-performance uniprocessors are too complex to allow enough room for on-chip memory.

The situation in a massively parallel processor environment is quite different than in a uniprocessor environment. Since massive parallelism is cost effective only in applications which can achieve roughly linear speed improvements as processors are added, N processors that perform at 1/Nth the speed of a given uniprocessor are roughly equal in processing power to that same uniprocessor.

The approach that is supported by these findings is one of building relatively simple processor/memory systems that can fit on a single chip. Since these chips will be much less expensive to manufacture and use in a system, more processors can be included in a system.

There are several methods of implementing this strategy. One method is to simplify a given processor implementation as much as possible, probably sacrificing speed-enhancing hardware features for overall system size. With current technology (1 to 2 micron CMOS), this approach can lead to simple 16-bit processing elements with small program memories. Of course, appropriate software techniques to keep code size small are vitally important. This approach is probably the most attractive for MIMD machines.

Another possible method is to reduce the word-size of each processing element. The ultimate extension of this philosophy is bit-serial machines which can, in fact, have multiple processing/memory elements per chip. This approach is obviously well-suited to SIMD machines.

## CONCLUSIONS

In the near term, the challenge to achieving the maximum level of processing element integration is to find design styles and programming methodologies that can fit enough functionality onto available chip real estate to go beyond the Rent's Rule breaking point. Current architectures which may be able to do this include: bit-serial processors, which can pack several processors with memory onto a single chip; 8-bit microcontrollers, which are probably not powerful enough to be of interest in their currently available form; and stack-oriented processors, with their small program memory size requirements. In the future, chip sizes may increase enough to allow RISC processors to have a full-sized on-chip cache and slow serial interfaces to their program memories. CISC processors may eventually reach this point as well, but only if they are frozen at a particular complexity level.

Some parallel processor architectures, especially SIMD architectures, are clearly already embracing the philosophy of simple computational elements that can fit on a single chip. What we have explored are some of the theoretical underpinnings of this approach, and why it makes sense for massively parallel architectures.

## REFERENCES

1. Landman, B.S. and Russo, R.L., "On a Pin Versus Block Relationship For Partitions of Logic Graphs," *IEEE Transactions on Computers,* December 1971, **C20**(12), pp. 1469-1479

2. Siewiorek, D.P. and Swarz, R.S., *The Theory and Practice of Reliable System Design,* Digital Press, Bedford MA, 1982

3. Koopman, P.J., *CPU/16 Technical Reference Manual,* WISC Technologies, La Honda CA, 1986.

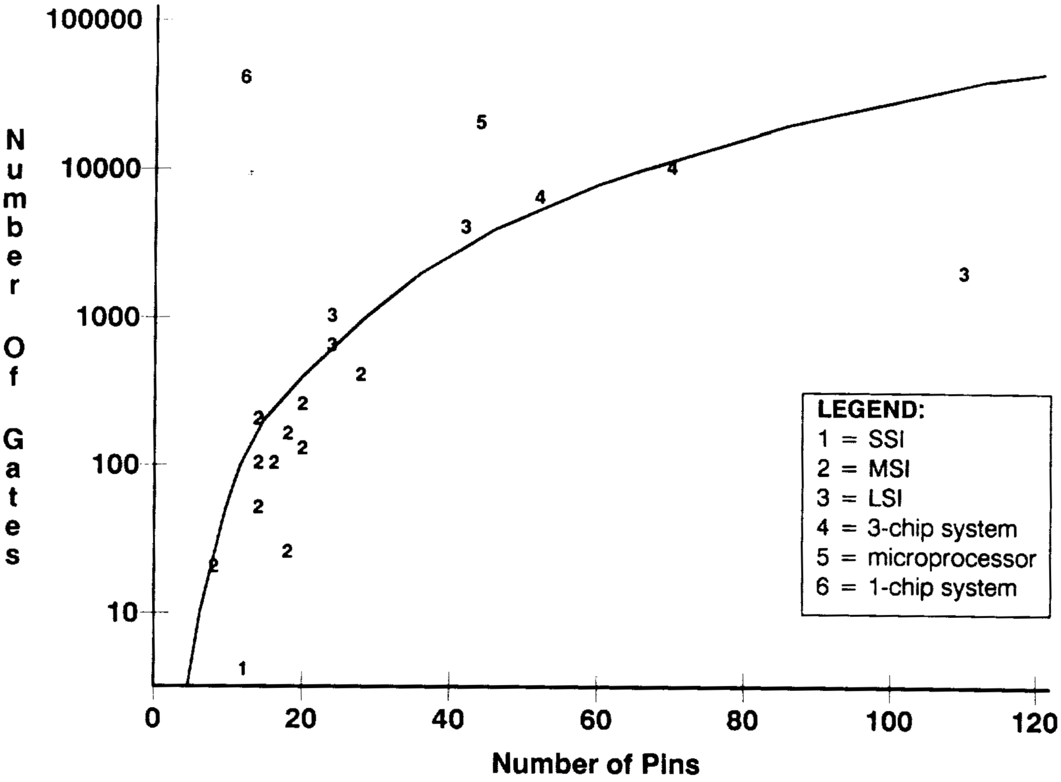


Figure 1. Experimental data shown with a Rent curve (R=0.38)

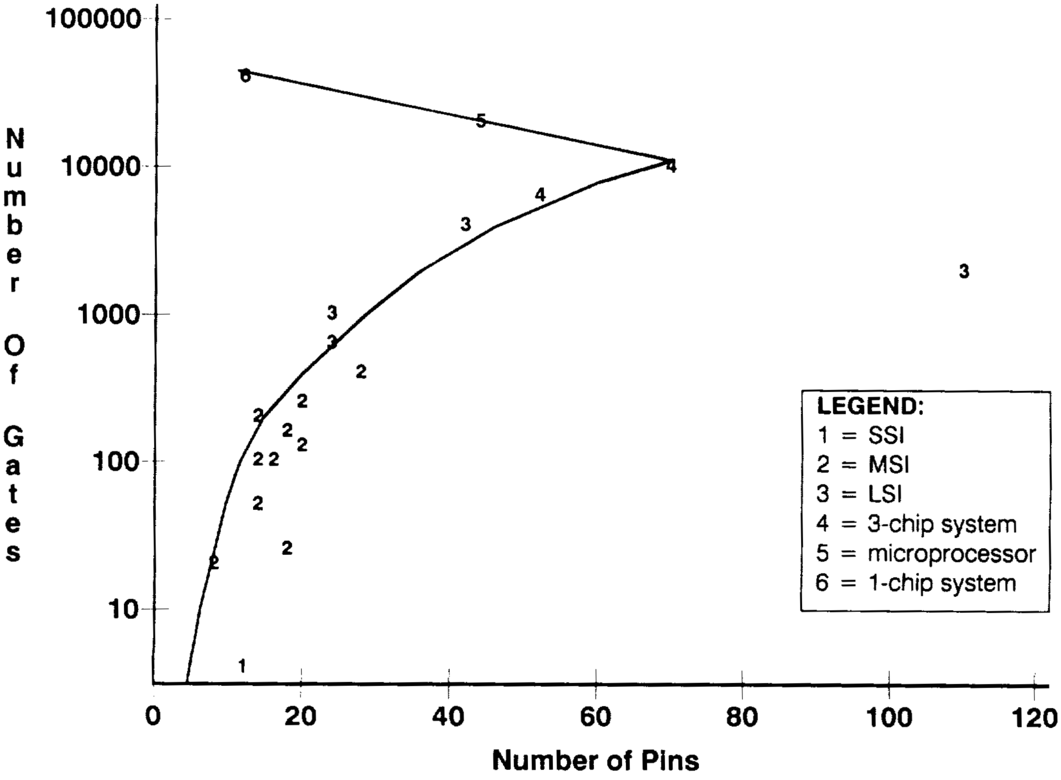


Figure 2. A different curve that better fits the data

# OVERVIEW AND EXTENSIONS OF A SYSTEM FOR ROUTING DIRECTED GRAPHS ON SIMD ARCHITECTURES *

Sherryl Tomboulian

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center, Hampton VA 23665

## ABSTRACT

Many problems can be described in terms of directed graphs that contain a large number of vertices where simple computations occur using data from adjacent vertices. A method is given for parallelizing such problems on an SIMD machine model that uses only nearest neighbor connections for communication, and has no facility for local indirect addressing. Each vertex of the graph will be assigned to a processor in the machine. Rules for a labeling are introduced that support the use of a simple algorithm for movement of data along the edges of the graph. Additional algorithms are defined for addition and deletion of edges. Modifying or adding a new edge takes the same time as parallel traversal. This combination of architecture and algorithms defines a system that is relatively simple to build and can do fast graph processing. All edges can be traversed in parallel in time $O(T)$, where T is empirically proportional to the average path length in the embedding times the average degree of the graph. Additionally we present an extension to the above method which allows for enhanced performance by allowing some broadcasting capabilities.

Keywords: routing algorithm, SIMD architecture, parallel processing, graph embedding, interconnection network

## INTRODUCTION

There are many problems that can be formulated as directed graphs. Such problems include circuit simulation, semantic networks and topography. Typically the *real-world* versions of these problems contain 100,000 vertices or more, and while the computations that occur at each vertex are simple, the size of the problems makes them computationally intensive. A natural way to parallelize these problems is a paradigm in which each processor is assigned a vertex in the graph, and there is some mechanism for realizing the arcs. This fine grained approach suggests the use of SIMD architectures, which can be built with many thousands of processors.

The problem with using SIMD architectures is that often they do not easily support generalized message passing schemes. This paper presents a method of embedding graphs in a class of SIMD architectures by using a special space-time labeling that supports message delivery and incremental addition of paths. The algorithms for this system are presented in (Refs. 16, 17). Basic concepts of the system will be reviewed, followed by an important generalization of the original method.

To maximize the number of processors which can be built, we choose the simplest hardware definition necessary to solve graph oriented problems. The machine model used is SIMD: there is a controller and a large number of slave processors which can execute the same instruction stream simultaneously. The processors have exclusively local memory, and they have no facilities for indirect addressing. The processors must be connected in a topology with the following requirements: (1) there must be some path between any two processors; (2) every neighbor link must be bi-directional, i.e. if A is a neighbor of B, then B must be a neighbor of A; and (3) the neighbor relations between processors must have a consistent invertible labeling. A more precise definition of the labeling requirements can be found in (Ref. 16). It suffices that most networks (Ref. 4) including grid, hypercube, cube connected cycles (Ref. 13), shuffle exchange (Ref. 14), and mesh of trees (Ref. 7) are admissible under the scheme. Additional requirements are that the processors be able to read from or write to their neighbors' memories, and that at least one of the processors acts as a serial port between the processors and the controller.

The Massively Parallel Processor (MPP) built by Goodyear Aerospace is an SIMD architecture with single bit processors arranged in a 128 by 128 processor grid (Ref. 2). The MPP is not the *perfect* machine for this algorithm since it is limited by its diameter being $\sqrt{N}$, but nevertheless is a good candidate.* A parallel machine design that fits our model well is the Boolean Vector Machine (BVM) being built at Duke University which is an SIMD machine that uses the cube connected cycles interconnection scheme (Ref. 13).

The Connection Machine, produced by Thinking Machines Corporation, is an SIMD architecture with 64K processors, each with 4K bits of memory, and complex routing hardware that supports arbitrary communication (Ref. 5). While a hardware router may be a preferred method for solving graph problems, many applications do not require this arbitrary communication facility and would profit by replacing the equivalent silicon area with more processors and using software for communication. By choosing a software alternative, problems that do not need generalized communication are more economical and graph oriented problems are still viable. Independent of the argument of whether one should or should not build an SIMD architecture with routing hardware, the fact remains that architectures such as the MPP are being built that do not have routing hardware, and routing software extends their usability.

## THE METHOD

In this section we present the concept of *conflict-free space-time* labeling, henceforth referred to as CFST-labeling. Using this labeling scheme, we present a simple algorithm for data movement and an algorithm for generating CFST-labeling of a graph incrementally.

It is necessary to distinguish between the graph problem being attacked and the computer model being used. The graph being embedded will be referred to using standard graph terminology with regards to vertex, edge, and degree. The machine elements are called processors and wires. Each vertex will be assigned to a different processor. Each edge in the graph will be realized by a path in the physical network which is a list of consecutive wires joining adjacent processors. Each wire specification that is part of a path is referred to as a link.

Traversing all the edges of the embedded graph in parallel will take more than one step since messages cannot be sent instantaneously but rather must be passed along through successive neighbors. Traversing all edges in parallel, referred to as the delivery phase, will be considered an uninterruptible operation that takes T steps. In addition to the spatial characterization of a path, a path will also be characterized by a relative temporal offset within the delivery phase. Rules are provided governing paths. Algorithms are presented to create paths

*THIS WORK WAS SUPPORTED BY THE NATIONAL AERONAUTICS AND SPACE ADMINISTRATION UNDER NASA CONTRACT NO. NAS1-18107 WHILE THE AUTHOR WAS IN RESIDENCE AT ICASE.

*See (Ref. 3) for an implementation of sorting on the MPP.

incrementally and to transfer a collection of messages through paths simultaneously.

We will begin by defining the data structures which will be resident at each processor.

```
ALLOCATED ---- boolean flag indicates that processor
               is assigned a graph vertex
VERTEX_LABEL --- label of graph vertex
HAS_NEIGHBOR[1..neighbor_limit] --- flag indicates
                                    existence of wires
SLOTS[1..T] OF       edge path information
     START----------new edge starts here
     DIRECTION------direction to send
                    {1..neighbor_limit,FREE}
     END-----------edge ends here
     ARC LABEL-----label of edge
```

The ALLOCATED and VERTEX_LABEL fields indicate that the processor has been assigned a vertex in the graph. The HAS_NEIGHBOR field is used to indicate whether a physical wire exists in the particular direction (e.g. in a flat grid, edge processors only have 3 neighbors, and corner processors 2, while internal processors have 4); for a completely regular topology it is superfluous. The SLOTS data structure is the key to the routing system. It is used in the delivery algorithm to instruct the processor where to send a message and in the labeling algorithm to insure that paths are constructed so that no collisions will occur. The SLOTS array is used to tell the processors what they should do on each relative time position within the delivery phase.

One of the characteristics of this algorithm is that a fixed path is chosen to connect two processors and once chosen it is never changed. For example, consider the grid in Figure 1.

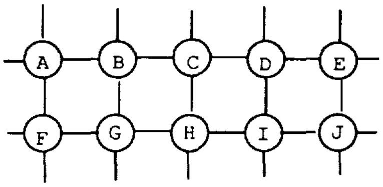


Figure 1.

If there is an edge between vertices in A and H, there are several possible paths: East-East-South, East-South-East, and South-East-East. Only one of these paths will be chosen between A and H, and that same path will always be used. For each edge, the corresponding path is not only fixed in space (i.e. the set of wires is constant), but is also fixed in time (the initial delay before the message starts down the path is constant). Once the starting time for the path has been fixed, it is never changed. Paths do not have to start on time 1, but can be scheduled to start at some relative offset within the delivery phase. Since there are no facilities for buffering, a message must proceed continuously along the specified directions without delay. For instance, if the path is of length 3 and it starts at time 1, then it will arrive at time 4; if it starts at time 2, it will be guaranteed to arrive at time 5. Further, it is necessary to place the paths so that no collisions occur; that is, no two paths can be at the same processor at the same instant in time. The rules for paths that fulfill these requirements are listed below.

- At most one link can enter a processor at a given time, and at most one link can leave a processor at a given time. It is possible to have both one coming and one going at the same time. Note that this does not mean that a processor can have only one link; it means that it can have only one link during a particular step in the delivery phase. It can have as many as T links going through it (since a delivery phase is length T by definition).
- Any path between two processors (u,v) representing an edge must consist of links at consecutive time steps. For example, if the path from processor u to processor v is {u,f,g,h,v}, then if the link from u-f is assigned time 1, f-g must use time 2, g-h time 3, and h-v time 4. Likewise if u-f occurs at time 5, then link h-v will occur at time 8.

When these rules are used to form paths, the SLOTS structure can be used to mark the paths. Each path goes through neighboring processors at successive time steps. For each of these time steps the DIRECTION field of the SLOTS structure is marked, telling the processor which direction it should pass a message if it receives it on that time slot. SLOTS serves both to instruct the processors how to send messages and to indicate that a processor is busy at a certain time slot so that when new paths are constructed it can be guaranteed that they won't conflict with current paths.

Consider the following example. Suppose we are given the directed graph with vertices A,B,C,D and edges $A \rightarrow C, B \rightarrow C, B \rightarrow D$, and $D \rightarrow A$ (Figure 2), and that vertices A,B,C, and D have been assigned to successive processors in a linear array. (A linear array is not a good network for this scheme but convenient for demonstration.) Initially all slots are free. We proceed to construct a CFST-labeling, placing each edge in the order it appears in the list above.

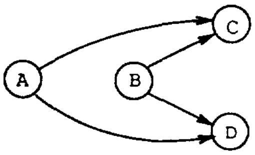


Figure 2.

```
A,B,C,D are successive members in a linear array

                 1---2---3---4
                 A---B---C---D
```

1. $A \rightarrow C$ can be completed with the map East-East, so Slots[A][1] = E, Slots[B][2]=E, End[C][2]=TRUE.
2. $B \rightarrow C$ can be done with the map East; it can start at time 1, since Slots[B][1] and End[C][1] are free.
3. $B \rightarrow D$ goes through C then to D; its map is East-East. B is occupied at time 1 and 2. It is free at time 3, so Slots[B][3]=E, Slots[C][4]=E, End[D][4]= TRUE.
4. $D \rightarrow A$ must go through C,B,A. using map West-West-West. D is free on time 1, and C is free on time 2, but B is occupied on time 3. The path can start from D on time 2. Slots[D][2]=W, Slots[C][3]= W, Slots[B][4]= W, End[A][4]=TRUE.

Every processor acts as a conduit for its neighbors' messages. No processor knows any message's source or destination, but each processor knows what it must do to establish the local connections.

Given that the paths satisfy the CFST-labeling rules, message delivery for graph problems is simple. The paths have been constructed so that there will be no collisions, and each path link uses consecutive time slots. The end of a path is specified by setting a separate bit that is tested after each message is received. A separate start bit in SLOT[$k$] indicates that a path starts at time $k$. The start bit is needed because the SLOTS array just tells the processors where to send a message, regardless of how that message arrived **. The start array indicates when a message originates, as opposed to arriving from a neighbor.

The following algorithm is basic to the routing system.

**Both the START and the STOP bits can be encoded as part of the DIRECTION field in SLOTS, but the presented method is simpler to explain and allows for more efficient execution.

```
PROCEDURE DELIVER

  for i = time 1 to T
     FORALL processors
   /* if an edge starts or passes through at this time */
        if SLOT[i].START = 1 or active = 1
           for j=1 to neighbor-limit
           begin
              if SLOT[i].direction= j
                 write message bit to
                     in-box of neighbor j;
                 set active = 0;
           end
           FORALL processor that just received a message
           if end[i]
               move in-box to message-destination;
           else
              move in-box to out-box;
              set active bit = 1;
```

This code follows the method described above. The time slots are looped through, and the messages are passed in the appropriate directions as specified in the SLOTS array. Two bits, in-box and out-box, are used for message buffering.

The time complexity of data movement is $O(T \times neighbor\_limit)$. Since the number of neighbors is assumed to be a small constant for each network, the complexity is $O(T)$. This suggests that networks with fewer neighbors have advantages. For instance, the hypercube network has $logN$ neighbors, and the cube connected cycles network has 3 neighbors. Empirical results (Ref. 17) shows that while hypercube uses a smaller $T$, CCC actually has a faster overall delivery time.

### Setting up Message Paths

One of the goals in developing this system was to have a method for adding new edges quickly. Paths are added so that they don't conflict with any old path. Once a path is placed it will not be re-routed by the basic placement algorithm; it will always start at the same spot at the same time. The basic idea of the method for placing a connection is to start from the source processor and in parallel examine all possible paths outward from it that do not conflict with pre-established paths. As the trial paths are flooding the system, they are recorded in temporary storage. At the end of this deluge of trial paths, if the destination processor has been reached, then a real path exists. Using the stored information a path can be backtraced and recorded. This is similar to the Lee-Moore routing algorithm (Refs. 6, 8) for finding a path in a system.

Suppose that the connection (u,v) is to be added. First it is assumed that processors for u and v have already been determined, otherwise (for now) assume a random allocation from a pool of free processors. It is necessary to find a path between u and v that does not conflict with any of the existing paths. The method for doing this is a type of flooding. A breadth-first search will be performed in parallel starting at the source processor. A record is kept of the trial paths resulting from this search. The paths must adhere to the CFST labeling rules, so a trial path must not conflict with paths that are already established. For instance, suppose a trial path starts at time 1 and moves to a neighboring processor, but that neighbor is already busy at time 1 (as can be seen by examining the DIRECTION-SLOT.) Since a path that would go through this neighbor at this time is not legal, the trial path would commit suicide, that is, it stops propagating itself. If the processor slot for time 2 was free, the trial path would attempt to propagate itself to that processor's neighbors at time 3.

Trial paths are recorded in a structure called TRIALSLOTS. A trial path knows if the next time slot is occupied by referring to the SLOTS data structure. If the destination processor is reached by a path, it will be a path that does not violate the rules. Therefore we can trace backwards from the destination processor using the markings in TRIALSLOTS and transfer this good path to the actual SLOTS structure.

## PERFORMANCE

Adding an edge (assuming one can be added), deleting any set of edges, or traversing all the edges in parallel, all have time complexity $O(T \times neighbor\_limit)$. If it is assumed that $neighbor\_limit$ is a small constant then the complexity is $O(T)$. Since T is related both to the time and space needed, it is a crucial factor in determining the value of the algorithms presented. Some analytic bounds on T were presented in (Ref. 16), but it is difficult to get a tight bound on T for general interconnection networks and dynamically changing graphs. For the case where the graph is known *a priori* an upper bound of $O(log^2N)$ can be achieved on a hypercube. This is obtained by applying a result by Nassimi and Sahni (Ref. 10) in which they present a method for data broadcasting in SIMD computers which conforms to the CFST-labeling rules.

Of major interest is the on-line case, where edges are added and deleted dynamically. A simulator was constructed to examine the behavior of the algorithms. Besides the simulated data, the algorithms mentioned were actually implemented for the Connection Machine. The data presented by the simulator is consistent with that produced by the real machine. The major result is that the size of T appears proportional to the average degree of the graph times the average path length in the embedding.

This is a highly significant result. If it is assumed that the average number of connections and the neighbor-limit are bounded by small constants then the time for a parallel traversal operation, is, the worst case, $O(diameter)$. This indicates that the algorithm performs optimally for routing random communication graphs, since a random graph can have connections between processors that are diametrically opposed. If it assumed that the diameter is $O(logN)$ then $C_T$ is $O(logN)$.

This bound indicates that the methods presented here are competitive with existing methods for parallel traversal in SIMD architectures. Some methods for SIMD parallel communication were mentioned in the introduction. Permutations can be done in $O(logN)$ time (Refs. 9, 10). Sorting can be done in essentially $O(log^2N)$ time, using (Ref. 12) or Batcher (Ref. 1) combined with Thompson (Ref. 15). These methods are restricted to permutations and sorting. One of the advantages of the method presented here is the ability to deal with graphs that are more general. Using the previous methods, if the connections specified a complete permutation, and additionally some processor also wanted to connect to two other processors, three entire permutations must be performed because the complexity is $diameter \times max\_number\_of\_connections$. In our system some items can have more connections than others without substantially increasing $T$. This result is achieved because the complexity of this method is based on the $diameter \times AVERAGE\_number\_of\_connections$, rather than the maximum number. The method used here also has the advantage that new connections can be added easily, unlike the other methods which require the entire set-up to be re-computed.

A further advantage of this method is the ability to exploit locality. Since the heuristic for $T$ is dependent on the *average path length*, situations where the embedding can be arranged so that processors connect to those in some neighborhood will produce smaller values of T.

## BRANCHING PATHS

There are many variations of this basic method that can be exploited. Among these are heuristics for picking shortest paths, assignment of node to processor, and choosing paths so as to avoid congestion. A discussion of these methods can be found in (Ref. 16).

A significant and important extension of the general CFST labeling rules involves a generalization that we refer to as *branching paths*. Branching paths is actually a form of broadcasting. Rather than having each arc represented by a different path, arcs starting from the same vertex can be combined for greater efficiency. However, this gain comes at the loss of two features.

The first is an obscure feature. In the basic CFST method presented, the system is identically free of conflicts whether it is run forwards or backwards. That is, rather than have an algorithm that starts at step 1 in the delivery cycle and initiates messages with the START bit marked, it is possible to write a *send backwards* algorithm which, using the same SLOTS structure, will start at time T initiating those links that *STOP* at the processors, and work backwards to time 1, reversing all the links, until the values get to the source. If one wished to implement a bi-directional graph rather than a directed graph, instead of actually putting in two sets of wires (from each pair $u \rightarrow v$ and then $v \rightarrow u$), one could just put in one set of wires, pass the values in the forward directions, then reverse it and pass the values back again.

The second and greater issue involves the nature of the messages that are passed along the arcs. If the application requires that a different message be passed along each arc, then it will not be possible to use this combining method. However, if all connecting processors are passed the same value, or the same value modified by a different constant, then this variation is viable. Many applications, including circuit simulation and neural networks fall into this class.

In the standard method, each arc in the original graph becomes a path in the embedding.

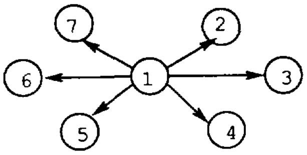


Figure 3.

For example, Figure 3 shows a graph in which one vertex is connected to all others. Mapped onto a linear arry, each arc becomes a path (Figure 4).

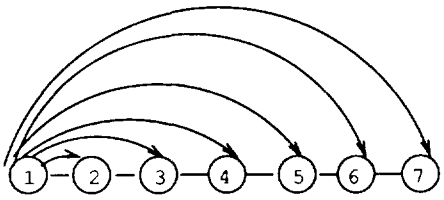


Figure 4.

The branching path method adheres to the CFST labeling rules, but alters the premise that each arc in the original graph is represented by a separate path. Instead, the set of arcs associated with each vertex can be represented by a set of branching paths. A branching path has two characteristics that differ from a standard path between two nodes. The first is that a path can "drop off" values at intermediate nodes in the path. For example, Figure 5 shows that a single path can be used to deliver messages to all intermediate vertices. Essentially, processors along the same route can share the path.

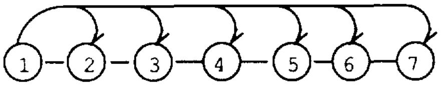


Figure 5.

In addition to path sharing, the other ability is to allow paths to branch out, or broadcast to their neighbors. For example, Figure 6 shows a grid in which processor A is connected to B and C. This is represented with one branching path.

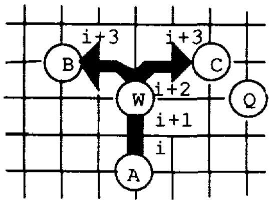


Figure 6.

More formally, as before, each vertex in the original graph is mapped to a processor in the network topology. The set of arcs associated with each vertex will be represented as a set of directed acyclic graphs (DAG). For each arc $(u,v)$, there must be a DAG, $D$, whose root is $u$ which has $v$ as either a leaf node, or an internal node. For any leaf node $w$ in the graph $D$ with root $u$, $(u,w)$ must be a an arc in the original graph. Further, in the family of DAGs which represent the arcs originating from the root $u$, each arc in the original graph can be represented only once. That is, if a graph has root $u$ and leaf $w$, then no other graph with root $u$ can have leaf $w$. Likewise, any internal node that is a destination must be marked to indicate that it is not just an intermediate node in the route, but rather a drop-off point. If a graph with root $u$ has a specially marked internal node $v$ representing arc $(u,v)$, then no other graph with root $u$ can have $v$ has a specially marked node, although $v$ can appear without a marking. As in the original method, each arc is only represented once, although paths can go through other nodes that are not part of their message destination. For example, in figure 6, A connects to B and C. Suppose that A were also connected to W. In this case W would have to be marked to indicate it not just an intermediate node in the graph. Further, if A were to connect to Q, this could be represented by a separate graph as a path going through W. But, W could only be a destination vertex in one of the two graphs.

Another aspect of this definition involves the nature of the messages. If all vertices that $u$ connects to receive the same value message then the definition mentioned above is fine. If all receive the same value message multiplied by a different constant, it is also fine, because each constant can be stored at the destination node, so all connecting arcs will be passed the same value, and multiplication will occur at the time of delivery. However, if connecting values are truly different, then no two arcs $(u,v)$, $(u,w)$ that have different arc weights can be represented by the same DAG. Hence, in the original case, where we assumed that each arc receives a different dynamic value, each arc would have to be represented by a separate DAG, which would be a DAG with only one leaf, which is our definition of a path. So the original method is simply a subset of this method.

Having defined this family of graphs that represent the arcs in the new embedding, we proceed to label them according to an extension to the CFST labeling rules. The first rule, non-exclusion, originally said that at most one connection can enter a processor at a given time and at most one can leave. Now we allow more than one to leave provided that the connection is defined as part of the same DAG. The second rule, of contiguous time, still holds. When a path splits into two or more branches, each of the connections proceeds at the same time, as illustrated previously in Figure 6.

A small clarification: strictly speaking, the branching paths do not have to be DAGS, that is, they don't have to be acyclic. That is, when a connection reaches a processor, it does so at some time $i$, and it is possible for the path to loop back on itself, which will occur at some later time $j$. While usually such loops are inefficient, sometimes they are used to avoid jams in the system, and further, they can occur naturally as part of the algorithms which finds paths. In terms of the static algorithm, a way to avoid the conceptual messiness of cycles is

## Algorithm Updates

The algorithms change surprisingly little. The algorithm for delivering the messages doesn't change at all. The loop which checks for the marked direction and passes the value in the appropriate direction may simply see, for instance, that the North bit is on, and so passes it north, and then checks that the West bit is on, and passes likewise within the same time step. Since, because of the SIMD nature of the machine, it is already necessary to loop through the different directions, no extra work is done.

The algorithm for finding a new path requires a small modification. In the original method, paths propagate from the source through neighboring processors which are not already busy. To encorporate the branching path method, during the spreading method all connections which start from the source are activated, as they would be during the normal delivery cycle, and these active processors are included as part of the path spreading. That is, if a processor would normally be active at time i if it were sending a message from the source as part of its normal delivery cycle, then on time $i+1$ it will attempt to propagate a new path to its neighbors. If the shortest path heuristic is used (Ref. 17), so that each trial path has a length associated with it, then a new path that is formed as a branch or continuation off an old path starts at length 0 from that branch point, rather than starting at length 0 from the source. For problems that do not require unique values to be passed and that have multiple connections per vertex, these minor changes allow for a much more efficient message delivery system.

### Analysis

The basic comments on analysis are the same as for the original case. A message delivery cycle still takes $O(T)$, where $T$ is the number of slots, but by using the branching paths, $T$ can be much smaller. As an example, we take the case of a fully connected graph embedded in a linear array. We know that the lower bound on $T$ is the cutwidth of the resulting embedding (Ref. 17). When embedding the fully connected graph in a line using the original method, the maximum cutwidth is $O(N^2)$, and in the branching path method it is $O(N)$. Hence, in this case the savings between the two methods is substantial. While this is an unusual example, it gives an idea of the advantages. Some empirical results on the on-line random graph examples, of the type done in (Ref. 17), tend to show an improvement of a factor of 2 to 3 over the original method. These results are preliminary and haven't been statistically validated.

## CONCLUSION

Some simple algorithms have been presented which allow arbitrary graphs to be embedded in SIMD architectures having a variety of topologies. The time for performing a parallel traversal and for adding a new connection appears to be proportional to the average path length in the embedding, times the average number of arcs in the graph being embedded. Since the average path length is no more than the diameter of the network, the method is competitive with existing methods for SIMD routing, with significant advantages for graphs that can exploit locality. Additional advantages are that there are no *a priori* requirements for the form of the data, the topological requirements are extremely general, and new arcs can be added without reconfiguring the entire system. The simplicity of the implementation and the flexibility of the method suggest that it could be an important tool for using SIMD architectures as graph processing machines.

## REFERENCES

1. K. Batcher, "Sorting Networks and their Applications," *Proceedings of AFIPS 1968 SJCC*, pp. 307-314.

2. K. Batcher, "Design of a Massively Parallel Processor," *IEEE Trans on Computers*, Sept 1980, pp. 836-840.

3. J.E. Dorband, "Sort Computation and Conservative Image Registration", Ph.D. Thesis, Pennsylvania State University, Dec. 1985.

4. T. Feng, "A Survey of Interconnection Networks," *Computer*, Dec 1981, pp.12-27.

5. W. Hillis, "The Connection Machine," MIT Press, Cambridge, Mass., 1985.

6. C. Lee, "An algorithm for path connections and its applications," *IRE Trans. Elec. Comput.*, Vol. EC-10, Sept. 1961, pp. 346-365.

7. T. Leighton, "Parallel Computation Using Meshes of Trees," *Proc. International Workshop on Graph Theory Concepts in Computer Science*, 1983.

8. E. Moore, "Shortest path through a maze," *Annals of Computation Laboratory*, Vol. 30, Harvard Univ. Press, 1959, pp. 285-292.

9. D. Nassimi and S. Sahni, "Parallel Algorithms to Set-up the Benes Permutation Network," *Proc. Workshop on Interconnection Networks for Parallel and Distributed Processing*, April 1980.

10. D. Nassimi and S. Sahni, "Benes Network and Parallel Permutation Algorithms," *IEEE Transactions on Computers*, Vol. C-30, No. 5, May 1981, pp. 332-340.

11. D. Nassimi and S. Sahni, "Data Broadcasting in SIMD Computers,"*IEEE Transactions on Computers*, Vol. C-30, No. 2, Feb 1981, pp. 101-106.

12. D. Nassimi and S. Sahni, "Parallel Permutation and Sorting Algorithms and a New Generalized Connection Network," *JACM*, Vol. 29, No. 3, July 1982, pp. 642-667.

13. F. Preparata and J. Vuillemin, "The Cube Connected Cycles: a Versatile Network for Parallel Computation," *Comm. ACM*, Vol. 24, No. 5, May 1981, pp. 300-309.

14. H. Stone, "Parallel processing with the perfect shuffle," *IEEE Trans. Computers*, Vol. C-20, No. 2, Feb. 1971, pp. 153-161.

15. C. Thompson, "Generalized Connection Networks for Parallel Processor Intercommunication," *IEEE Tran. Computers* Vol. C-27, Dec. 1978, pp. 1119-1125.

16. S. Tomboulian, "A System for Routing Arbitrary Communication Graphs on SIMD Architectures," Doctoral Dissertation, 1986,Dept. of Computer Science, Duke University, Durham, NC.

17. S. Tomboulian, "A System for Routing Directed Graphs on SIMD Architectures", ICASE Report No. 87-14, NASA Langley Research Center, Hampton, VA (updated 1988).

18. R. Wagner, "The Boolean Vector Machine," *IEEE 1983 Conference Proceedings of the 10th Annual International Symposium on Computer Architecture*, pp. 59-66.

[illegible]

P 13

# AN ANALYSIS OF DISJOINT PATH PROPERTIES IN DATA MANIPULATOR NETWORKS *

Wayne G. Nation Howard Jay Siegel

Parallel Processing Laboratory
School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907, USA

## ABSTRACT

A critical component of a supercomputer based on massively parallel processing is the interconnection network that provides communications among the system's processors and memories. The data manipulator network family is a class of multistage interconnection networks based on the PM2I interconnection functions. One interesting property of the data manipulator network family is the existence of multiple paths through the network for most source/destination pairs. The condition which must be present to have disjoint paths through the network for a given source/destination pair is shown, where disjoint paths are multiple paths with no links in common. For source/destination pairs which have no disjoint paths, a single fault can prevent communication between that source/destination pair. It is proven that the maximum number of disjoint paths for any source/destination pair is two and a method for finding the routing tags that specify these paths is given. The effect of a fault in a given stage of the network on the number of source/destination pairs that can be connected is also discussed.

Keywords: ADM, IADM, gamma network, data manipulator, redundant path networks, interconnection networks, parallel processing, supercomputers.

## 1. INTRODUCTION

Large-scale parallel processing is one approach to the design of supercomputers. The interconnection network in a massively parallel computer system is a critical component. The network provides communications among the processors and memories of the system. One family of networks that has been proposed for use in such systems is the data manipulator family.

The data manipulator network family is a class of multistage interconnection networks based on the PM2I interconnection functions [11]. In some cases, data manipulator networks have a single path from a source *S* to a destination *D* while in other cases multiple paths exist; i.e., the number of paths between a given S and D may vary from one S/D pair to the next. Having a variable number of paths impacts the network throughput, permuting ability, and routing tag control. These issues are under study [2-7, 9-11, 13-16].

Properties of disjoint paths between a given S and D in data manipulator networks are examined here. *Disjoint paths* are multiple paths from a given S to D which have no links in common. The condition that must be present to have disjoint paths through the network for a given S and D pair is presented. To avoid any fault in a path from S to D there must exist another disjoint path for the same S and D. If a fault develops in one of these disjoint paths, it can be avoided by choosing to use the other disjoint path. It is shown that disjoint paths are available for only half of the possible S and D pairs. This indicates the fault tolerance limitations of the data manipulator network family. Furthermore, it is proven that the maximum number of disjoint paths for any S/D pair is two. A method for finding the routing tags that specify these paths is given. The effect of a fault in a given stage of the network on the number of S/D pairs that can be connected is also discussed.

Section 2 introduces the data manipulator network and some of its variations that are in the literature. In Section 3, two routing tag schemes for specifying paths through these networks are described. Properties of disjoint paths in this network class are presented in Section 4. Finally, Section 5 is a general discussion of these results.

## 2. THE DATA MANIPULATOR NETWORK FAMILY

The *data manipulator network family* includes the data manipulator [2], the Augmented Data Manipulator (ADM) [12], the Inverse Augmented Data Manipulator (IADM) [6], and the gamma [8] multistage interconnection networks. The *data manipulator* [2] (Figure 1) consists of m stages ($N = 2^m$). Each *stage* is a column of N switches. There is also an (m+1)-st column of network output ports. The stages are ordered from m−1 to 0. At stage i switch j can pass data to switch j + $2^i$ modulo N of stage i−1 (i.e., PM$2_{+i}$), switch j of stage i−1 (i.e., straight), or j − $2^i$ modulo N of stage i−1 (i.e., PM$2_{-i}$).

* This research was supported by the Supercomputing Research Center, Lanham, MD, where H.J. Siegel was on leave when most of this research was done.

CH2649-2/89/0000/0069$01.00 © 1988 IEEE

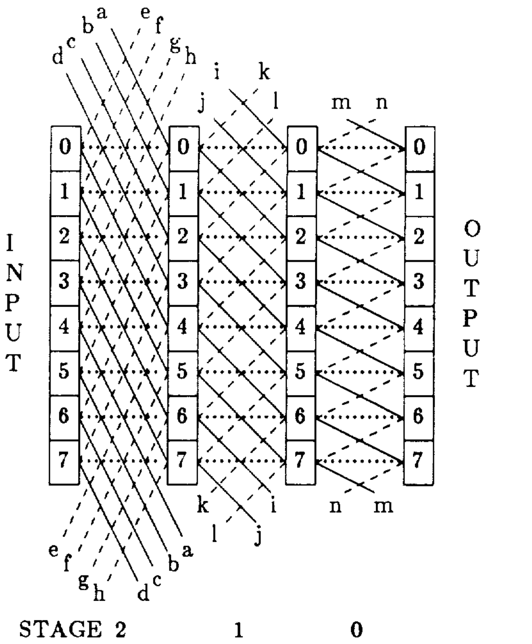


Figure 1: The data manipulator or Augmented Data Manipulator (ADM) network for N = 8.

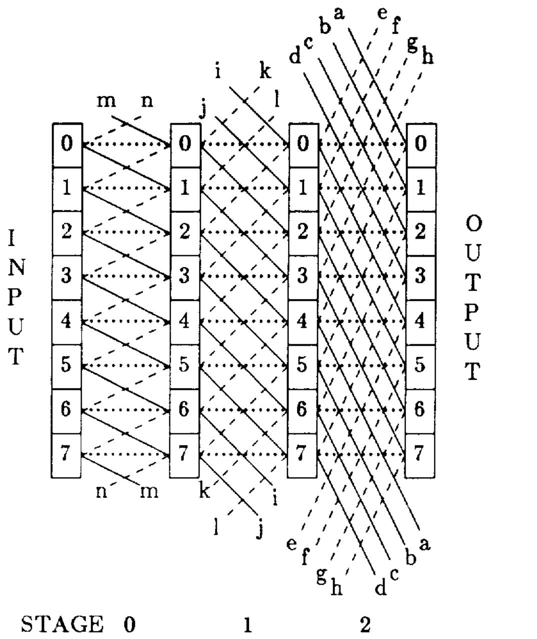


Figure 2: The Inverse Augmented Data Manipulator (IADM) or gamma network for N = 8.

A *switch* has three input links and three output links. Each switch can pass data from one of its input links to any one of its output links for a "one-to-one" setting. Data can also be passed from one of a switch's input links to two or three of its output links for a "broadcast" setting.

The *Augmented Data Manipulator* (*ADM*) is a data manipulator network constructed so that each switch can be set independently. The *Inverse Augmented Data Manipulator* (*IADM*) (Figure 2) is similar to the ADM except the stage ordering is reversed (stage 0 is the input stage). It is called the "Inverse" ADM because for any data permutation the ADM can do in one pass through the network, the IADM can do the inverse of that permutation [11]. The *gamma* network is a data manipulator network which has reverse stage ordering and individual switch control. The gamma network uses a 3×3 crossbar in each switch. This allows a gamma switch to perform several one-to-one settings at once; i.e., all of the switch inputs can each be connected to a unique switch output concurrently.

To route data from an input (source) S to an output (destination) D, the data must traverse links whose sum modulo N is (D−S) modulo N. As an example, for an ADM network with N = 8, the links traversed for S = 1 and D = 6 are: $+2^2$ ($+2^2$ link in stage 2), +0 (straight link in stage 1), $+2^0$ ($+2^0$ link in stage 0). The sum of the traversed links is $+2^2 + 0 + 2^0 = 5$. Four other paths exist that route data from 1 to 6: $+2^2$, $+2^1$, $-2^0$; and straight, $-2^1$, $-2^0$ ($-x = (N-x)$ modulo N. Assuming that the network is implemented with both $+2^{m-1}$ and $-2^{m-1}$ links at stage m−1 (even though ($+2^{m-1} = -2^{m-1}$) modulo N), then two more paths between 1 and 6 are: $-2^2$, straight, $+2^0$; and $-2^2$, $+2^1$, $-2^0$. In general, if $S \neq D$, there are multiple paths from S to D.

The ADM and IADM will be used to represent the data manipulator family. Because the discussion focuses on disjoint paths for individual S/D pairs, the gamma network's added capability from the 3×3 crossbar switch makes no difference. Thus, all properties derived for the IADM also apply to the gamma network.

The presentation to this point has assumed the networks to be unidirectionally connecting N processor/memory pairs, where processor/memory pair i is connected to both input port i and output port i of the network, $0 \leqq i < N$. However, the results of this paper apply to bidirectional implementations of the networks as well, where the networks can connect processors at the input side with memory modules at the output side and messages are routed in both directions.

## 3. ROUTING PATHS THROUGH THE ADM AND IADM

To specify an arbitrary path in an ADM network, a *full routing tag*, $F = f_{2m-1} \dots f_0$, can be used [6]. A stage i switch examines bits $f_i$ and $f_{m+i}$. If $f_i$ is zero, that stage i switch uses the straight link (the value of $f_{m+i}$ is ignored). If $f_i$ is a one then the switch will use the $+2^i$ link if $f_{m+i}$ is a zero and the $-2^i$ link if $f_{m+i}$ is a one. For example, for an N = 16 ADM network, the tag

$F = 00111011$ will route from 1 to 6 on the path $+2^3$, straight, $-2^1$, $-2^0$. The tag scheme using a full routing tag requires a 2m bit tag but can be used to specify any arbitrary path through the network.

A *natural routing tag* uses only one bit to specify the sign of the non-straight links used in the path, thus all the non-straight links traversed are of the same sign [6]. An m+1 bit routing tag T is formed by computing the *signed magnitude* difference between the destination and the source: $T = t_m \ldots t_0 = D-S$. The ***sign bit*** is $t_m$, where $t_m = 0$ indicates positive or zero (i.e., $D \geqq S$), and $t_m = 1$ indicates negative (i.e., $D < S$). Bits $t_{m-1} \ldots t_0$ equal $\delta$ the absolute value of D−S, the *magnitude* of the difference. The natural routing tag is interpreted in the same way as the full routing tag, except $t_m$ is used as the sign bit at every stage. For any natural tag T for S to D ($S \neq D$) an alternative routing tag from S to D can be computed that uses links of the opposite sign by taking the two's complement of T [6]. For example, for $N = 8$, $S = 1$, and $D = 6$, $T = 0101$, and the two's complement of $T = T' = 1011$. This is shown in Figure 3.

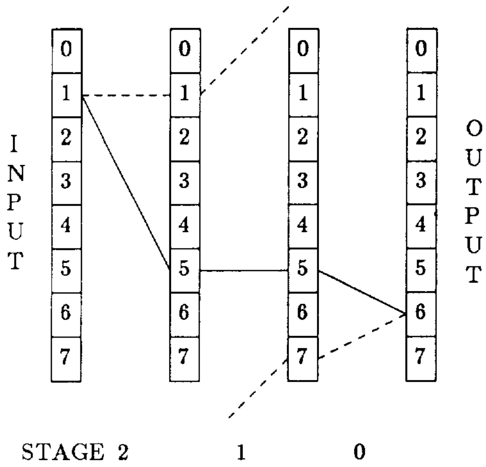


Figure 3: An $N = 8$ ADM network showing the two disjoint paths from $S = 1$ to $D = 6$. The solid line shows the positive dominant path specified by $T = 0101$. The dashed line shows the negative dominant path specified by $T' = 1011$.

A *positive dominant* routing tag is a natural routing tag with $t_m = 0$, while a ***negative dominant*** tag has $t_m = 1$. The *positive dominant path* is the path specified by the positive dominant tag and the ***negative dominant path*** is specified by the negative dominant tag. The two's complement of one sign dominant tag from S to D produces the sign dominant tag from S to D of opposite dominance.

A typical assumption made when studying the fault tolerance of multistage networks is that the network input switches and network output switches are not faulty (so that data can enter and leave the network), and the rest of the network is what must be made fault tolerant [1]. This simplifying, although somewhat unrealistic, assumption is adopted here also. It is shown that even with this assumption, the data manipulator has very limited fault tolerance.

## 4. DISJOINT PATH PROPERTIES OF THE DATA MANIPULATOR FAMILY

This section presents certain disjoint path properties of the data manipulator network family. In stage m−1, the $+2^{m-1}$ and $-2^{m-1}$ links from each switch are equivalent; i.e., $+2^{m-1} = -2^{m-1}$ modulo N. Thus, a Data Manipulator network could be implemented with a single link for each $\pm 2^{m-1}$ connection or separate links for each $+2^{m-1}$ and $-2^{m-1}$ connection. The results proven here are valid in either situation unless specified otherwise.

Theorem 1 and Corollary 1 show that, for any combination of S and D, $\delta$ ($|D-S|$) is odd if and only if the two paths formed by the sign dominant routing tags are disjoint. Theorem 2 and Corollary 2 show that all paths for a given S/D pair pass through no more than two switches in each stage (excluding I/O ports). While this has been conjectured before, it is formally proven here. It is stated in Theorem 3 that for $j > 0$, $2^j$ is the highest power of two for which $\delta$ is a multiple if and only if the sign dominant paths share straight links and switches in stages 0 to j−1. Theorem 4 proves that if there exists a path from S to D that uses straight links in stages 0 to j, then all paths from S to D use straight links in stages 0 to j. Corollary 3 (a generalization of Theorem 3) states that for $j > 0$, $2^j$ is the highest power of two for which $\delta$ is a multiple if and only if all paths from S to D share straight links in stages 0 to j−1. The results of the above are combined in Corollary 4 to prove that exactly two disjoint paths exist for S/D pairs if and only if $\delta$ is odd, while no disjoint paths exist if $\delta$ is even. The section concludes with the derivation of an expression for the number of S/D connections that are not possible given a straight link fault in any of the network stages (Theorem 5). The notation $t_{x/y}$ will be used to represent the string of bits $t_x t_{x-1} \ldots t_{y+1} t_y$, $x \geqq y$.

**Theorem 1:** Consider an ADM network for arbitrary N. $\delta$ is odd for a given S/D pair if and only if the two sign dominant tags form two link-disjoint as well as switch-disjoint paths through the network (excluding input and output switches).

**Proof:**
*Part 1*: $\delta$ odd → disjoint paths.
It is sufficient to prove that the two paths are switch-disjoint, for if the two paths do not share a switch in stage i then they cannot share a link leaving that stage. Without loss of generality, assume $D \geqq S$. Consider the positive dominant routing tag T from source S to desti-

nation D. Because $\delta$ is odd, T has the form $T = t_{m/1}1$. The negative dominant tag ($T'$ = two's complement of T) is therefore $T' = \bar{t}_{m/1}1$. Because of this relationship, when the negative dominant path takes the $-2^i$ link in stage i ($1 \leqq i \leqq m-1$), the positive dominant path will take the straight link in stage i. Likewise, when the positive dominant path takes the $+2^i$ link in stage i ($1 \leqq i \leqq m-1$), the negative dominant path will take the straight link in stage i. Consider the ADM for arbitrary N. Because $t_0 = 1$, the switch $P_i$ that the positive dominant path passes through in stage i ($0 \leqq i \leqq m-2$), where the path is followed from the destination D to the source, is given by:

$$P_i = \left[D - \sum_{j=0}^{i} t_j(2^j)\right] = \left[D - 1 - \sum_{j=1}^{i} t_j(2^j)\right] \text{modulo N.}$$

(When $i = 0$, $P_0 = D-1$.) Similarly, because $t'_0 = 1$ and $t'_k = \bar{t}_k$, $1 \leqq k \leqq m-1$, the switch $N_i$ that the negative dominant path passes through in stage i ($0 \leqq i \leqq m-2$) is given by:

$$N_i = \left[D + \sum_{j=0}^{i} t'_j(2^j)\right] = \left[D + 1 + \sum_{j=1}^{i} \bar{t}_j(2^j)\right] \text{modulo N.}$$

(When $i = 0$, $N_0 = D+1$.) If the paths ever meet at a switch in stage i, then the distance (difference) between $P_i$ and $N_i$ will be zero at that stage. This part of the proof is completed if

$$\left[N_i - P_i\right] \text{modulo } N \neq 0 \quad \text{for all i } 0 \leqq i \leqq m-2.$$

Using the above equations;

$$\left[N_i - P_i\right] =$$

$$\left[D - D + 1 + 1 + \sum_{j=1}^{i} t_j(2^j) + \sum_{j=1}^{i} \bar{t}_j\ (2^j)\right] \text{modulo N}$$

$$= 2 + \sum_{j=1}^{i} 2^j \text{ modulo N}$$

$$= (\ 2^{i+1}\ ) \text{ modulo } N \neq 0$$

$$\text{for } 0 \leqq i \leqq m-2.$$

Thus, the paths never meet at a switch (excluding the input switch (stage $m-1$) and output port), and this part of the proof is complete.

*Part 2*: disjoint paths ⟶ $\delta$ odd.

Due to the two's complement property of T and $T'$, $t_0 = t'_0$. If $t_0 = t'_0 = 0$ then the paths would not be disjoint (both would go straight in stage 0, connecting to the same network port). Therefore, if the paths are disjoint $t_0 = t'_0 = 1$, with the positive dominant path taking the $+2^0$ link in stage 0 and the negative dominant path taking the $-2^0$ link in stage 0. In order for $t_0 = t'_0 = 1$, $\delta = t_{m-1/0}$ must be odd. Thus, if T and $T'$ specify disjoint paths, $\delta$ must be odd.

□

An example of the disjoint paths for an $N = 8$ ADM for $S = 1$ and $D = 6$ is shown in Figure 3.

**Corollary 1:** Theorem 1 is true for the IADM network.

**Proof:**

*Part 1*: $\delta$ odd ⟶ disjoint paths.

The proof for the IADM is similar to the proof for the ADM with some factors that account for the different stage ordering. In particular, for the IADM, the paths meet at the stage 0 switch (the input switch) and the output port. To calculate $P_i$ and $N_i$ (for $1 \leqq i \leqq m-1$), the paths are followed from the source S to the destination, where:

$$P_i = \left[S + 1 + \sum_{j=1}^{i-1} t_j(2^j)\right] \text{modulo N}$$

$$N_i = \left[S - 1 - \sum_{j=1}^{i-1} \bar{t}_j(2^j)\right] \text{modulo N.}$$

Therefore,

$$\left[P_i - N_i\right] = (\ 2^i\ ) \text{ modulo } N \neq 0 \quad \text{for } 1 \leqq i \leqq m-1.$$

Thus, the paths never meet at a switch (excluding the input switch (stage 0) and the output port).

*Part 2*: disjoint paths ⟶ $\delta$ odd.

The proof follows from Theorem 1, proof of Part 2.

□

**Theorem 2:** In the ADM network, all paths for a given S/D pair pass through no more than two switches in each stage.

**Proof:** All possible paths emanating from the destination D back through stages 0, 1, ..., j pass through the set of stage j switches $A_j$, where:

$$A_j = \left\{ D \pm k \text{ modulo } N \ : 0 \leqq k \leqq (2^{j+1}-1) \right\}$$

An example of $A_j$ for an $N = 16$ ADM network with $j = 1$ and $D = 1$ is shown in Figure 4. All elements of $A_j$ represent $2^{j+2}-1$ consecutive switches modulo N in stage j (i.e., switches 0 and $N-1$ are consecutive). All possible paths emanating from the source S through stages $m-1$, $m-2$, ..., $j+1$ enter a set of stage j switches $B_j$, where:

$$B_j = \left\{ S \pm k \times 2^{j+1} \text{ modulo } N \ : 0 \leqq k \leqq 2^{m-2-j} \right\}$$

An example of $B_j$ for an $N = 16$ ADM network with $j = 1$ and $S = 10$ is shown in Figure 5. Due to the modulo N arithmetic, $S + k \times 2^{j+1} = S - k \times 2^{j+1}$ for $k = 2^{m-2-j}$. Each element of $B_j$ is at least $2^{j+1}$ switches apart from any other switch of $B_j$. A distance measure, $d(x,y)$, is defined as the shortest distance between two switching elements x and y of a stage; i.e.,

$$d(x,y) = \min(|x-y|, N - |x-y|).$$

So, $\forall\ x,y\ (x \neq y) \in B_j,\ d(x,y) \geqq\ 2^{j+1}$.

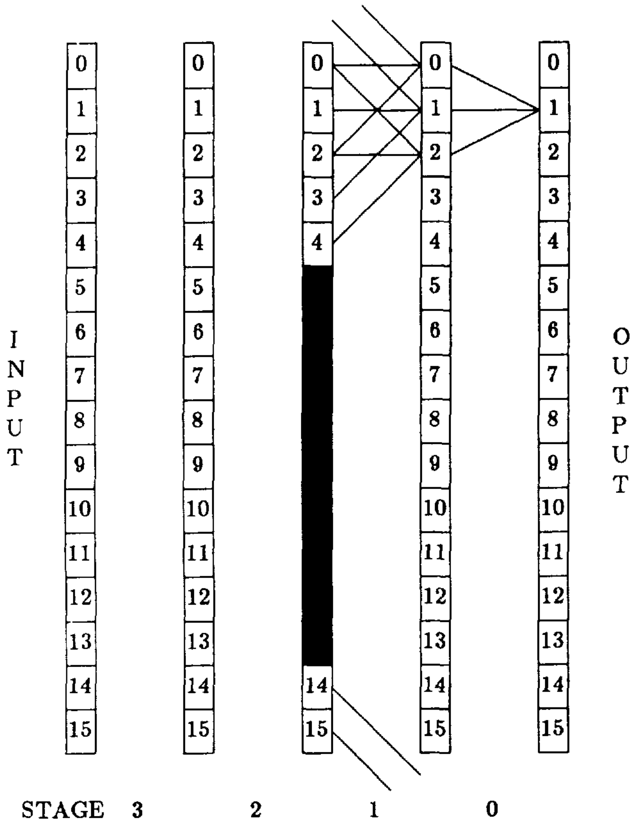


Figure 4: An N = 16 ADM network showing links used by all possible paths from stage 1 to D = 1. All stage 1 switches not in $A_1$ are blacked out.

Similarly for elements of $A_j$, $\forall$ x,y $(x \neq y) \in A_j$, $d(x,y) \leqq 2^{j+2}-2$. By definition, a stage j switch is on the path from S to D if and only if it is an element of both $A_j$ and $B_j$. The intersection of $A_j$ and $B_j$ $(A_j \cap B_j)$ must contain at least one element; otherwise, the network could not make a connection between S and D (which is known to be possible). It is a direct result of Theorem 1 that $\exists$ S,D such that $A_j \cap B_j = 2$ for $0 \leqq j \leqq m-2$.

Proof by contradiction is used to show that it is not possible for $A_j \cap B_j$ to contain three or more elements (i.e., the size of $A_j \cap B_j$ is either one or two). Assume $\exists$ $x,y,z \in A_j \cap B_j$ such that $x \neq y \neq z$. Consider the elements x and y. $2^{j+1} \leqq d(x,y) \leqq 2^{j+2}-2$. Also, because $x,y \in B_j$, $d(x,y) = k \times 2^{j+1}$ modulo N, for some k in the range $0 \leqq k \leqq 2^{m-2-j}$. Hence, to obey the bounds constraints on $d(x,y)$, k must equal 1, which implies $d(x,y) = 2^{j+1}$. Because $z \in A_j \cap B_j$, $2^{j+1} \leqq d(x,z) \leqq 2^{j+2}-2$ and $2^{j+1} \leqq d(y,z) \leqq 2^{j+2}-2$. Similarly, $d(x,z) = d(y,z) = 2^{j+1}$. This leads to a contradiction since $d(x,z) = d(y,z) = d(x,y) = 2^{j+1}$ only if x = z or y = z. Thus, the proof is complete.

□

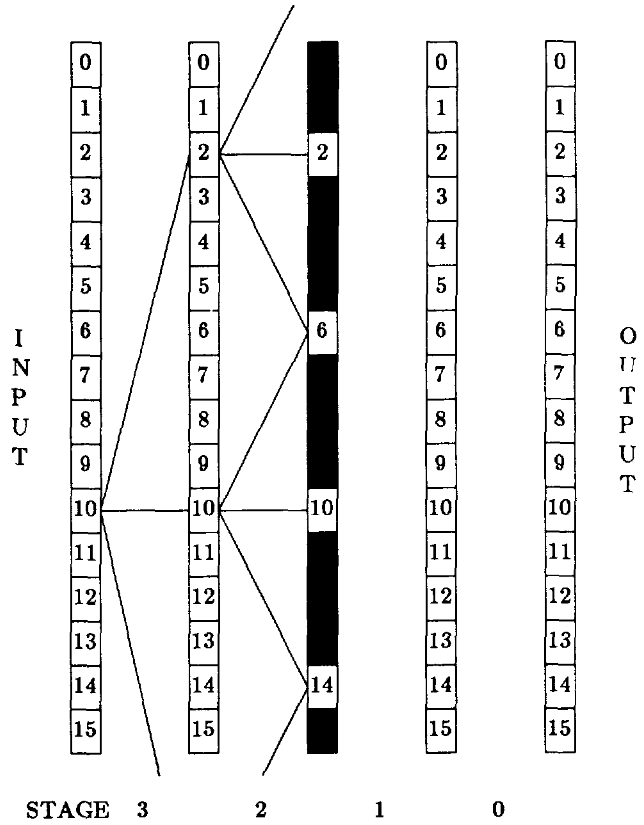


Figure 5: An N = 16 ADM network showing links used by all possible paths from S = 10 to stage 1. All stage 1 switches not in $B_1$ are blacked out.

**Corollary 2:** Theorem 2 is true for the IADM network.

**Proof:** The proof for the IADM is similar to the proof for the ADM with some factors that account for the different stage ordering. In particular, for the IADM, all paths emanating from the source S through stages 0, 1, ..., j pass through the set of stage j switches $A_j$, where:

$$A_j = \left\{ S \pm k \text{ modulo } N \ : 0 \leqq k \leqq (2^j-1) \right\}$$

Elements of $A_j$ represent $2^{j+1}-1$ consecutive switches modulo N in stage j. Thus,

$\forall$ x,y $(x \neq y) \in A_j$, $d(x,y) \leqq 2^{j+1}-2$.

All possible paths emanating from the destination D back through stages m−1, m−2, ..., j+1 enter a set of stage j switches $B_j$, where:

$$B_j = \left\{ D \pm k{\times}2^j \text{ modulo } N \ : 0 \leqq k \leqq 2^{m-1-j} \right\}$$

Each element of $B_j$ is at least $2^j$ switches apart from any other switch of $B_j$. Thus,

$\forall$ x,y $(x \neq y) \in B_j$, $d(x,y) \geqq 2^j$.

Proof by contradiction is used to show that it is not possible for $A_j \cap B_j$ to contain three or more elements (i.e., the size of $A_j \cap B_j$ is either one or two). Assume

$\exists$ x,y,z $\in A_j \cap B_j$ such that $x \neq y \neq z$. Consider the elements x and y. $2^j \leqq d(x,y) \leqq 2^{j+1}-2$. Also, because x,y $\in B_j$, $d(x,y) = k \times 2^j$ modulo N, for some k in the range $0 \leqq k \leqq 2^{m-1-j}$. Hence, to obey the bounds constraints on $d(x,y)$, k must equal 1, which implies $d(x,y) = 2^j$. Because $z \in A_j \cap B_j$, $2^j \leqq d(x,z) \leqq 2^{j+1}-2$ and $2^j \leqq d(y,z) \leqq 2^{j+1}-2$. Similarly, $d(x,z) = d(y,z) = 2^j$. This leads to a contradiction since $d(x,z) = d(y,z) = d(x,y) = 2^j$ only if $x = z$ or $y = z$. Thus, the proof is complete.

□

As long as $\delta$ is odd for a given S/D pair there are always two disjoint paths where the two paths are specified by the two sign dominant tags for that $\delta$. When $\delta$ is even, it can be shown that the paths formed by the sign dominant tags share a link in stage 0 and are thus nondisjoint. This fact can be extended to state that if $\delta$ is a multiple of $2^j$ the paths formed by the sign dominant routing tags will share links in stages 0, 1, ..., j−1 (i.e., the two sign dominant paths follow the same path in j of the m stages). A proof of this statement follows in Theorem 3.

**Theorem 3:** In a data manipulator class network, for $j > 0$, $2^j$ is the highest power of two for which $\delta$ is a multiple if and only if the two sign dominant paths formed for S/D pairs with the given $\delta$ share straight links (and switches) in stages 0 through j−1 and are disjoint in stages j to m−1.

**Proof:** In the special case of $j = m-1$, it is necessary to assume that the $+2^{m-1}$ and $-2^{m-1}$ links are distinct in order to have disjoint paths in stage m−1.

*Part 1*: $\delta$ multiple of $2^j \longrightarrow$ sign dominant paths share straight links in stages 0 to j−1.

If $2^j$ is the highest power of two for which $\delta$ is a multiple, then the j+1 lower order bits of $\delta$ can be written as $\delta_{j/0} = 10^j$, where $0^j$ means a string of j '0's. Next consider the tags T and T′ for the given $\delta$. From Section 3, $t_{m-1/0} = \delta$. Because the j+1 lower bits of T are $t_{j/0} = 10^j$, the j+1 lower bits of T′ are also $10^j$ (a property of two's complement numbers). In this case both sign dominant paths use straight links in stages 0, 1, ..., j−1 and $\pm 2^j$ links in stage j. Depending on the stage ordering of the network under consideration, stage 0 is either at the input of the network (for the IADM) or at the output of the network (for the ADM). In either case, the sign dominant paths must meet at the input and output ports of the networks considered. For the ADM, because the two sign dominant paths meet at the network output ports and use straight links in the last j stages, the two paths must share links (and switches) in those j stages of the network. For the IADM, because the two sign dominant paths meet at the network input switches and use straight links in the first j stages, the two paths must share links (and switches) in those j stages of the network. If $\delta = 0$, $t_{m-1/0} = 0^m$ and thus only straight links are used in traversing the network, forcing the two sign dominant paths to share links and switches in every stage of the network.

Due to the properties of two's complement numbers, if $T = t_{m/j+1}10^j$, then $T' = \bar{t}_{m/j+1}10^j$. Therefore, as was shown in the proof of Theorem 1, when $t'_k = \bar{t}_k$, the two sign dominant paths do not share a link in stage k, $j \leqq k \leqq m-1$.

*Part 2*: sign dominant paths share straight links in stages 0 to j−1 $\longrightarrow$ $\delta$ is a multiple of $2^j$.

Because both sign dominant paths use straight links in stages 0 to j−1 and are disjoint in stages j to m−1, $t_{j/0} = t'_{j/0} = 10^j$. Thus, $\delta_{j/0} = 10^j$, and $2^j$ is the highest power of two for which $\delta = X^{m-(j+1)}10^j$ is a multiple.

□

An example of this theorem for an N = 16 ADM network with S = 7 and D = 11 ($\delta = 4$) is given in Figure 6.

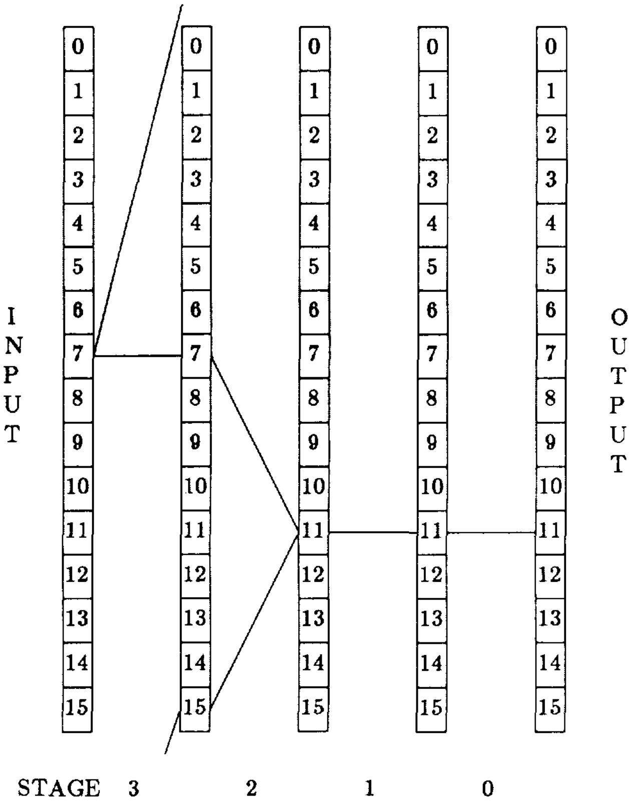


Figure 6: An N = 16 ADM showing the sign dominant paths for S = 7 and D = 11 ($\delta = 4$). The highest power of two for which $\delta$ is a multiple is j = 2. Thus, the sign dominant paths share links in two stages (stages 0 and 1).

**Theorem 4:** **For the data manipulator network family, if there exists a path from S to D that uses straight links in stages j to 0, all paths from S to D use straight links in stages j to 0.**

**Proof:** In general,

$$D = S + \sum_{i=0}^{m-1} f_i 2^i \text{ where } f_i \in \{ -1, 0, +1 \} .$$

Because there exists a path from S to D with stages j to 0 set to straight, $f_i = 0$ for $0 \leqq i \leqq j$. Thus,

$$D = S + \sum_{i=j+1}^{m-1} f_i 2^i .$$

This implies $s_i = d_i$ for $0 \leqq i \leqq j$. If it can be shown that $s_i = d_i$ for $0 \leqq i \leqq j$ implies the switches on all paths from S to D must be set to straight in stages j to 0, then the proof is complete.

This will be proven by contradiction. Assume that for $0 \leqq k \leqq j$, stage k is the lowest numbered stage with a switch on a path from S to D which is set to a non-straight state. Because stages 0 to k−1 are set to straight, they cannot affect $d_k$, and stages k+1 to m−1 affect bits $d_{k+1}$ to $d_{m-1}$, but cannot affect $d_k$. Therefore, the non-straight state of the switch in stage k forces $d_k = \overline{s}_k$. Thus, the assumption that stage k, $0 \leqq k \leqq j$, is set to a non-straight state implies $s_k \neq d_k$, which contradicts $s_i = d_i$ for $0 \leqq i \leqq j$. Therefore, the assumption is false and the proof is complete.

□

**Corollary 3:** In data manipulator class networks, for $j > 0$, $2^j$ is the highest power of two for which $\delta$ is a multiple if and only if all paths from S to D share straight links (and switches) in stages 0 through j−1 and there exist disjoint paths in stages j to m−1.

**Proof:** The proof follows from Theorems 3 and 4.

□

**Corollary 4:** For the data manipulator family of networks: (1) there are exactly two disjoint paths between a given S/D pair if and only if $\delta$ is odd, and (2) for $\delta$ even, there are no disjoint paths.

**Proof:**
*Proof of (1)*: Theorem 1 and associated Corollary 1 prove that the $\delta$ for S/D pairs is odd if and only if disjoint paths exist for those S/D pairs. Theorem 2 and associated Corollary 2 state that no more than two switching elements are used in any stage for paths between S and D. Therefore, there are exactly two disjoint paths between a given S/D pair if and only if $\delta$ is odd.
*Proof of (2)*: Follows from Theorem 3.

□

It has been shown previously [6] that any faulty non-straight link can be avoided by a reroute technique involving the two sign dominant paths. It is possible now to derive an expression for the number of S/D pairs in a data manipulator network that are blocked given a straight link fault in stage j. A faulty straight link cannot be avoided by any reroute technique if both sign dominant paths must both use that link (or switch). Theorem 5 presents a way of calculating the number of S/D pairs that cannot communicate given a straight link fault at switch P in stage j of an ADM network. Furthermore, Theorem 5 goes on to show a way of enumerating these S/D pairs that cannot communicate due to that fault.

**Theorem 5:** For the data manipulator network family, a straight link fault in stage j at switch $P = p_{m-1/0}$ will prevent $2^{m-(j+1)}$ S/D pairs from communicating. The S/D pairs that are blocked are of the form $S = s_{m-1/j}p_{j-1/0}$ (i.e., all inputs which agree with P in the lower order j bit positions) and D = P.

**Proof:**
Case 1: S/D pairs where the faulty straight link is not used by either the positive or negative dominant path (or both). These S/D pairs can still communicate.
Case 2: S/D pairs where both the positive and negative dominant paths share the faulty straight link at stage j. This implies $t_j = t'_j = 0$, which implies $t_{j/0} = t'_{j/0} = 0^{j+1}$ due to the properties of two's complement numbers. Thus, the positive and negative dominant paths share straight links in stages j to 0. Because there exists a path that uses straight links in stages 0 to j, all paths must use the same straight links in stages 0 to j (Theorem 4). Thus, if a faulty straight link is found in stage j then all S/D pairs whose sign dominant paths share this link cannot communicate. Consider the ADM network. If the faulty straight link is at switch P, then the destination of all S/D pairs that are blocked is also P. For these S/D pairs, $\delta_{j/0} = 0^{j+1}$. The set of possible sources is all sources which agree with P in the low order j+1 bit positions; i.e., D = P, $\delta = |D-S|$, and $\delta_{j/0} = 0^{j+1}$ imply $p_{j/0} = s_{j/0}$. This set of sources takes the form $s_{m-1/j+1}p_{j/0}$ where $s_{m-1/j+1}$ can take on any of $2^{m-(j+1)}$ values. Therefore, the S/D pairs that are blocked by the fault are the $2^{m-(j+1)}$ pairs of the form $S = s_{m-1/j+1}p_{j/0}$ and D = P. The proof for the IADM network is similar with the result that $2^{m-(j+1)}$ S/D pairs of the form S = P and $D = d_{m-1/j+1}p_{j/0}$ (where $d_{m-1/j+1}$ can take on any value) are blocked by a straight link fault in stage j at switch P.

□

## 5. SUMMARY

Several important properties of the data manipulator family of networks have been presented. Specifically, the difference $\delta$ ($|D-S|$) between the source port number S and the destination port number D is odd if and only if there exist two disjoint paths from S to D (Theorem 1 with associated Corollary 1). These two paths are the sign dominant paths specified by the sign dominant routing tags. Using this, a fault in one of the disjoint paths can be avoided by choosing the other sign dominant path. Theorem 2 (with associated Corollary 2) states that no more than two switching elements are used in each stage for all paths between a given S and D. For $j > 0$, the sign dominant paths share straight links and switches in stages 0 to $j-1$ and are disjoint in stages j to $m-1$ if and only if $2^j$ is the highest power of two for which $\delta$ is a multiple (Theorem 3). Theorem 4 proves that if there exists a path from S to D that uses straight links in stages 0 to j, then all paths from S to D must use those same straight links in stages 0 to j. Corollary 3 generalizes Theorem 3 by showing that for $j > 0$ all paths from S to D share straight links in stages 0 to $j-1$ if and only if $\delta$ is a multiple of $2^j$. The above results are combined to prove that exactly two disjoint paths exist for S/D pairs if and only if $\delta$ is odd, while no disjoint paths exist if $\delta$ is even (Corollary 4). This property limits the data manipulator family when considering their use as fault tolerant networks. Lastly, given a straight link fault in stage j, there are $2^{m-(j+1)}$ enumerable S/D pairs that are blocked due to that fault (Theorem 5).

Thus, various properties of disjoint paths for the data manipulator family have been shown. Studies of network characteristics, such as this, should aid system designers in choosing networks appropriate for their needs.

## 6. REFERENCES

1. G. B. Adams III, D. P. Agrawal, and H. J. Siegel, "Fault-tolerant multistage interconnection networks," *Computer,* Vol. 20, No. 6, June 1987, pp. 14-27.
2. T. Y. Feng, "Data manipulating functions in parallel processors and their implementations," *IEEE Transactions on Computers,* Vol. C-23, No. 3, March 1974, pp. 309-318.
3. M. D. P. Leland, "On the power of the augmented data manipulator network," *1985 International Conference on Parallel Processing,* August 1985, pp. 74-78.
4. D. Lee and K.Y. Lee, "Control algorithms for the augmented data manipulator network," *1986 International Conference on Parallel Processing,* August 1986, pp. 123-130.
5. K. Y. Lee and W. Hegazy, "The extra stage gamma network," *Thirteenth Annual International Symposium on Computer Architecture,* June 1986, pp. 175-182.
6. R. J. McMillen and H. J. Siegel, "Routing schemes for the augmented data manipulator network in an MIMD system," *IEEE Transactions on Computers,* Vol. C-31, No. 12, December 1982, pp. 1202-1214.
7. D. S. Parker and C. S. Raghavendra, "The gamma network: a multiprocessor interconnection network with redundant paths," *Ninth Annual Symposium on Computer Architecture,* April 1982, pp. 73-80.
8. D. S. Parker and C. S. Raghavendra, "The gamma network: a multiprocessor interconnection network with redundant paths," *IEEE Transactions on Computers,* Vol. C-33, No. 4, April 1984, pp. 367-373.
9. D. Rau, J. Fortes, and H. J. Siegel, "Destination Tag Routing Techniques Based on a State Model for the IADM Network," *Fifteenth Annual Symposium on Computer Architecture,* May 1988, pp. 318-324.
10. S. M. Reddy and V. P. Kumar, "On multipath multistage interconnection networks," *1985 International Conference on Distributed Computing Systems,* May 1985, pp. 210-217.
11. H. J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies,* Lexington Books, D. C. Heath and Co., Lexington, MA, 1985.
12. H. J. Siegel and S. D. Smith, "Study of multistage SIMD interconnection networks," *Fifth Annual Symposium on Computer Architecture,* April 1978, pp. 223-229.
13. A. Varma and C. S. Raghavendra, "Performance analysis of a redundant-path interconnection network," *1985 International Conference on Parallel Processing,* August 1985, pp. 474-479.
14. A. Varma and C. S. Raghavendra, "On permutations passable by the gamma network," *Journal of Parallel and Distributed Computing,* Vol. 3, No. 3, March 1986, pp. 72-91.
15. H. Yoon, K. Y. Lee, and M. T. Liu, "Performance analysis and comparison of packet switching interconnection networks," *1987 International Conference on Parallel Processing,* August 1987, pp. 542-545.
16. H. Yoon, K. Y. Lee, and M. T. Liu, "A new approach to internetworking of integrated services local networks," *1987 International Conference on Distributed Computing Systems,* September 1987, pp. 390-397.

# DATA STRUCTURES FOR ASSOCIATIVE SUPERCOMPUTERS[1]

J. L. Potter

Kent State University
Kent, Ohio 44242

## ABSTRACT

The parallel associative supercomputer model uses associative data structures and parallel associative searches to eliminate the need for complex address computation. This paper describes the simplicity of programming associative supercomputers and expands conventional associative data references into a generalized associative data structure reference mechanism which encompasses the arrays, data structures and data typing constructs of conventional languages.

Keywords: Associative computing, Massive Parallelism, Parallel Data Structures

## INTRODUCTION

### Associative Supercomputing

Associative supercomputing is a model of computation, where each datum has its own dedicated processor. It is a single instruction, multiple data (SIMD) model with a cellular memory, an exclusive read/write and a restricted concurrent read capability. In a cellular memory each cell contains memory, a dedicated processor and at most one association of data. Only one word in a cell may be accessed at a time. All cells may be accessed concurrently. The exclusive read/write capability means that each memory cell can be accessed by only one processor at a time. The concurrent read restriction means that only one memory word from a single memory cell may be read concurrently by all processors. The restricted concurrent read capability is equivalent to a global broadcast capability.

Associative computing exploits massive fine grain parallelism in a natural way. Massively parallel associative computers such as the MPP[2] with 16k processors and the Connection Machine[3] have been built. These computers eliminate the need for time sharing a single central processing unit with a multitude of data elements thus avoiding the classic memory - CPU bottleneck. Associative computing uses massive parallel searching in place of address calculation, reducing programming complexity. This paper describes a method for implementing data structures in the ASP language based on the associative computing model. See Potter [1987].

### Background

The concept of associativity has been present in computer science for many years. For example, Jacks [1971] and Findler [1979]. The most prevalent realization is associative triples in AI. The standard definition of an association is an ordered triple of object, attribute and value. Many similar definitions for associations have been formulated. See, for example, Simon [1970] and Savitt [1967]. The association list and ASSOC function in the LISP language are of course associative concepts simulated in software on conventional sequential hardware.

Kohonen [1978] rejected the standard definitions of associativity as needing to be defined in a more general fashion (p. 5). He proposed a model of association in which a collection of triples forms an associative memory and an entire triple is retrieved when any portion of it is used to query the memory.

Kohonen's model is expanded in associative computing. Triples of data are replaced by associations of any number of items. An arbitrary number of different kinds of associations may be stored in memory. Thus in associative programming, sets of collections of items form an associative memory and an entire record of items (i.e. an association) is selected when any subset

[1] This research was supported in part by ONR grant N00014-85-k0010.

[2] Manufactured by Loral Aerospace Division, Akron, Ohio.

[3] Manufactured by Thinking Machines Inc., Cambridge, Mass.

of its items are used to query the memory. When an association in memory is so selected, it is not moved to a central processor, but is processed in situ.

Associative computers were first developed at Goodyear Aerospace in the early 1970s. See Batcher [1977]. Foster's book [1976] describes the basic components of elementary associative computers. Associative computers should not be confused with content addressable (associative) memories (CAMs). CAMs do not have in situ processing. They retrieve data for delivery to a cpu just as conventional memories do. They are limited in use due to their relatively high cost.

Inherent in associative programming is the concept that each association has its own dedicated processor and that computation is effected by repeatedly selecting associations to be processed and retrieved. To date, no formal mechanism has been developed for associative languages to enable the specification of the equivalent of data structures in conventional languages. In the past data structures had to be implemented at the assembly language level. For example, Potter [1983], Reed [1985] and Potter [1985]. This paper expands on the concepts used for assembly language data structures, building a completely general hierarchy of data structures which can be used in any higher order language.

**ASSOCIATIVE PROGRAMMING**

**Background**

The impact of associative programming can be best explained by analyzing the fundamental components of a program. A program contains two major types of information, the procedural component and the identification component. The procedural part specifies the operations to be performed and the order in which they are to be executed. The identification component of a program selects the data to be operated on by the procedural component. The identification component uses the data's address within a memory (i.e. its position in the program's data structure) to select it.

The positional information content of a program's data structure is established by two mappings. The first mapping is between the problem data and the logical data structure used by the algorithm. The second mapping is between the logical data structure and the physical organization of the computer's memory. A third mapping is required to map the physical organization into a time sliced sequence of scalar data elements. These mappings are established by the programmer and are often the most crucial aspect of program development. Figure 1 illustrates the mapping sequence which is incorporated into the addressing function component of conventional programs.

In the simplest conceptualization, a different addressing function is required for fetching each individual piece of data required by an algorithm. However, these simple addressing functions are combined into larger more comprehensive and complex functions using looping and address modification (indexing) techniques. The loop construct, for example, is used extensively to time share the CPU among the many identical records of a file. An important aspect of selecting a data structure for a sequential computer is to pick one which allows the addressing functions to be efficiently folded so that the loop construct can be used.

Associative computers reduce the complexity of addressing functions without recursion and without limiting the logical data structure, thus they are easier to program

```
PROBLEM logical LOGICAL   physical PHYSICAL   time     CPU
DATA    -------->DATA     --------->DATA      ------->DATA
        mapping STRUCTURE mapping  STRUCTURE sharing STRUCTURE

MATRIX          TWO                  ONE              SEQUENTIAL
                DIMENSIONAL          DIMENSIONAL      SCALARS
                -------------        -----
(1 5 3)         | 1 | 5 | 3 |        | 1 |            1,5,3,4,2,6
(     )         |---+---+---|        | ---
(4 2 6)         | 4 | 2 | 6 |        | 5 |
                -------------        | ---
                                     | 3 |
                                     | ---
                                     | 4 |
                                     | ---
                                     | 2 |
                                     | ---
                                     | 6 |
                                     | ---
```

**Figure 1** - A Conventional Program Message

```
PROBLEM logical  LOGICAL
DATA  ---------->DATA
        mapping  STRUCTURE

MATRIX            TWO                STRUCTURE PROBLEM
                  DIMENSIONAL             CODE DATA
                 -------------           ---------
(1 5 3)          | 1 | 5 | 3 |           |1,1| 1 |
(     )          |---+---+---|           |---|---|
(4 2 6)          | 4 | 2 | 6 |           |1,2| 5 |
                 -------------           |---|---|
                                         |1,3| 3 |
                                         |---|---|
                                         |2,1| 4 |
                                         |---|---|
                                         |2,2| 2 |
                                         |---|---|
                                         |2,3| 6 |
                                         |---|---|
```

**Figure 2** - Associative Data Mapping

than conventional computers. First, every data record has its own dedicated processor. Thus, the need for a "time sharing" factor in the address function is eliminated. Second, the physical mapping component of the address function is replaced by parallel (associative) searching. Finally, as described later, the logical mapping relationship is stored associatively as Structure Codes with the data elements eliminating the need for run time address calculations.

For example, in Figure 2, the logical portion of the address function consisting of the matrix row and column indices are stored with the data elements as structure codes. Since the data structure codes are dependent only on the logical mapping, the programming task is reduced to 1) directing the computer in the sequential execution of the fundamental steps of the algorithm and 2) the manipulation of the logical data structure codes. The artifacts of time sharing the CPU and the physical sequential organization of memory are eliminated.

## ASSOCIATIVE DATA STRUCTURES

The data structures, arrays and data types of conventional languages can all be mapped onto the general concept of associations. In data structures, the address function is a constant consisting of a path name. The path name is constant because of the requirement in conventional computers that the address be determined at compile time. Unlike data structures, variables can be used in the address function of arrays because the declared regular structure allows run time address calculation. Run time calculation requires that indices be numerical. However, certain languages such as PASCAL use data typing to map non-numerical address values into numerical ones at compile time. The association data structure in associative programming handles all three types of data organizations.

When the implicit address functions of conventional languages are stated explicitly in a content addressable computer, the explicit address function values state specifically the positions of the object in the data structure space generated by the address function. For this reason, the individual explicit address function values are referred to as structure codes. Thus in Figure 3, "age" and "size" are the structure codes for "50" and "large" respectively. The structure codes are discussed as if they were a unique type of data item. In reality they are not. They are just like the other data items in an associative object in that they can be searched for and manipulated by all ASP associative programming statements. Structure codes are unique only in that they contain structural information on how one problem data element relates logically to the other problem data elements.

### Data Structures as Extended Associations

The associative concept is most commonly introduced in terms of attribute value pairs. All conventional data organizations techniques can be viewed as extensions of the attribute value pair concept. Specifically, an array can be thought of as an attribute value pair with a compound attribute consisting of a constant portion, the array name, with variable modifiers, the indices, as shown in Figure 4. The "dimension" of the array determines the number of modifiers.

On the other hand, a "data structure," as shown in Figure 5, is an attribute value

| structure code | data element |
|---|---|
| attribute | value |
| age | 50 |
| size | large |
| color | blue |
| patient | jones |

**Figure 3** - A Simple Scalar Structure

```
attribute|modifier1|modifier2|value
---------+---------+---------+-----
    A    |    I    |    J    | 3

    A    [    I    ,    J    ]= 3
```

**Figure 4** - Compound Attribute

pair with a compound value. The fields of the data structure constitute the multiple values. Thus arrays and data structures can both be viewed as generalizations of attribute value pairs. In fact, all multi-attribute, multi-valued data objects can be viewed as extended associations.

**Simultaneous Multiple Data Organizations**

In an associative memory there is no hard distinction between the attribute and the value portion of an attribute value pair. That is, the same datum can be retrieved by searching for the matching attribute or the matching value. For example (color $) and ($ blue) would both select (color blue).

Thus in effect, either the attribute portion or the value portion can be defined as the constant portion of an address function. It is only by convention that the attribute portion is considered the address function.

By extension to associative triples, any one component of the triple can be considered to be the address function with the other two components being the compound value, as shown in Figure 6. Moreover, any combination of two components can be considered to be a compound address function with the third component a simple value. In general, if there are n components in an object there are:

$$\sum_{k=1}^{n-1} C^{n}_{k}$$

sets of address functions.

In an associative computer, all of these address functions are available to the programmer simultaneously. There is no a priori reason to select one set of address functions and its inherent data organization over any other. Therefore, all can be used at the programmers discretion, intermingled in any way without any need for reordering. Multiple simultaneous data organizations are impossible in conventional and parallel sequential computers, since the data structures must be sorted to be efficiently accessed and they can be organized only one way at a time. Frequently auxiliary data structures such as linked lists are employed to overcome this limitation of conventional computers.

```
        struct emp {
                     int emp#;
                     int birth_year;
                     int birth_day;
                    };


attribute|subvalue1|subvalue2 |subvalue3
---------+---------+----------+---------
   emp   |  emp#   |birth_year|birth_day
```

**Figure 5** - An Associative Data Structure

**Structure Codes for Generalized Array Data Structures**

Arrays are the canonical forms of data structures. As shown in Figure 7, their address functions form a natural hierarchy of complexity. Scalars are zero dimensional arrays. They are represented by the class of address functions consisting only of constants. The class of address functions for one dimensional arrays consist of constants plus one variable. Two dimensional arrays have two-variable address functions, etc. The most common example of address functions for arrays, are the row-major and column-major ordering functions generated automatically for indexed arrays by most high order languages such as FORTRAN, PASCAL and C.

One dimensional arrays can be stored using a straight forward extension of scalar structure codes. The structure code consists of the object name (the constant portion of the address function) and the position of the value in the construct (the variable portion of the address function). The variable component for one dimensional arrays is simply the ordinal position of the data element in the array. Thus, for example, the one dimensiona object A = (1 5 4 3 2) would have the structure code shown in Figure **8**.

The structure code for two dimensional arrays is a natural extension of one dimensional arrays as shown in Figure 9 . The extension of structure codes to higher dimensional arrays is obvious. The composition and manipulation of these canonical array structure codes to make structure codes for complex compound data structures is considered next.

| add. funct. | compound | value |
|---|---|---|
| sofa<br>table<br>chair | color<br>size<br>weight | red<br>big<br>heavy |

a - Object
Address Function

| compound | address function | value |
|---|---|---|
| sofa<br>table<br>chair | color<br>size<br>weight | red<br>big<br>heavy |

b - Attribute
Address Function

| compound | value | addres functi |
|---|---|---|
| sofa<br>table<br>chair | color<br>size<br>weight | red<br>big<br>heavy |

c. - Value
Address Function

**Figure 6** - Multiple Data Orginazation

| ADDRESS FUN TYPE | EXAMPLE | DATA STRUCTURE |
|---|---|---|
| constant | a | scalar |
| constant+$x_1$ | a($x_1$) | 1 dimensional |
| constant+$x_1$+$x_2$ | a($x_1$,$x_2$) | 2 dimensional |
| constant+$x_1$+..+$x_n$ | a($x_1$,..+$x_n$) | n dimensional |

**Figure 7** - A Hierarchy of Address Functions

One dimensional arrays are logical data structures which are natural for use with several common problem data structures such as vectors, lists and strings. Two dimensional arrays are logical data structures which are natural for dealing with matrices and imagery. The mapping from these problem data structures to the logical data structure is the identity mapping. Consequently, for ease of reading, where no confusion can arise, the terms vector and matrix will be used interchangeably for one dimensional and two dimensional arrays respectively.

### Associative Data Structure References

It is not uncommon to consider matrices as collections of vectors. Thus if the constant portion of the structure code shown in Figure 9 is modified to include "row position." The constant address "B 1" is shared by two values representing the vector (5 3) and "B 2" represents (7 6). Similarly, if the constant portion is modified to include "column position" instead of "row position," "B 1" represents (5 7)$^{-1}$ and "B 2" represents (3 6)$^{-1}$.

| structure code | | data element |
|---|---|---|
| constant part | variable part | |
| object name | element position | value |
| A | 1 | 1 |
| A | 2 | 5 |
| A | 3 | 4 |
| A | 4 | 3 |
| A | 5 | 2 |

**Figure 8** - A One Dimensional Array

An important property of structure codes is the ability to reorganize them as illustrated above. The "." operator will be used to indicate the basic code grouping and can be thought of as a concatenation operator. The symbol, "$", is used as a place holder. Thus the code B.1.$ represents the vector (5 3), B.$.2 represents ( 3 6)$^{-1}$, etc.

### DATA STRUCTURE CODE MANIPULATION

The concept of combining data structures to form new data structures at run time is common in some languages such a LISP. For example, lists can be grouped together to form lists of lists, etc. This can be done because of the generalized method of data storage for lists. However, in conventional languages, this capability is not easily extended to other types of data structures such as arrays. In associative computing, it is possible to create new data structures from existing data structures at run time for all types of data organizations. That is, structures such as

| structure code | | | data element | |
|---|---|---|---|---|
| constant part | variable part | | | |
| object name | row | col | value | |
| | position | | | |
| B | 1 | 1 | 5 | |
| B | 1 | 2 | 3 | B=(5 3) |
| B | 2 | 1 | 7 | (7 6) |
| B | 2 | 2 | 6 | |

**Figure 9** - A Two Dimensional Array

arrays of arrays of lists of arrays can be generated, decomposed and manipulated with ease.

In order to describe how the structure codes for two arbitrary data structures can be combined to generate the structure codes for a combined data structure, several definitions are necessary. Let $DS_j$ be a data structure of dimension r with address function $A_j$. Then $A_j = a_j0.a_j1..a_jr$ are the r+1 components of the structure code. By convention, the 0th component is the constant portion which is the name of the data structure. Let $A_j(m)$ stand for the structure code of $A_j$ for the mth element of $DS_j$. Let $0^1$ denote the constant value 0, $0^2$ denote 0.0, $0^3$ denote 0.0.0, etc. Then $0^n$ denotes the constant zero structure code for a function with n components. Similarly, let $A^n(x)$ denote the first (left most) n components of a structure code. The depth of a component is equivalent to the number of components to its left.

Then if $DS_l$ is the complex data structure obtained by inserting data structure $DS_k$ with dimension s, as the mth element of $DS_j$ with dimension r, at depth d, the address function $A_l$ for $DS_l$ has dimension d+s, and is given by

$$A_l(x) = A_j(x).0^{d+s-r} \qquad \text{for } x \mathrel{!=} m$$

$$A_l(x) = A_j^d(x).A_k(y) \qquad \text{for } x = m, \text{ for all } y \text{ in } DS_k$$

The data structure insertion operation is denoted by:

receiving_data_structure||
[element,depth]inserted_data_structure.

If a complex structure is to be built by a number of insertions, they may occur in any order, i.e. if $m_1 \mathrel{!=} m_2$, then

$$(A_l \,||[m_1,d_1]\, A_k) \,||[m_2,d_2]\, A_j = (A_l \,||[m_2,d_2]\, A_j) \,||[m_1,d_1]\, A_k.$$

Figure 10 gives an example. $DS_a$ is an "empty" vector with address function $A_a$ = (1 2). $DS_b$ and $DS_c$ are both matrices with the same address function $A_b$=$A_c$=(1.1, 1.2, 2.1, 2.2). The composition A=($A_a$ ||[1,1] $A_c$) ||[2,1] $A_b$ is shown. Clearly, arbitrarily complex hierarchical data structures can be composed from the basic canonical forms.

Figure 12 shows the conventional nested loop statements required by a conventional language (C, Fortran, Pascal, etc.) to perform the same operation. Note that the data must be physically moved (resorted)so that the physical memory layout maps correctly onto the logical layout. This requires that the number of items in all arrays be known at run time. In associative programming languages which use structure codes, the address function is modified as specified by the address composition function above. The number of data items is immaterial and the intent of the data reorganization is clear. The new address function is not hidden inside a number of loops which need to be untangled.

|     | Aa | Value |
|-----|----|-------|
| DSa | 1  | nil   |
| DSa | 2  | nil   |

DSa=(nil nil)

|     | Ab  | Value |
|-----|-----|-------|
| DSb | 1.1 | 7     |
| DSb | 1.2 | 14    |
| DSb | 2.1 | 3     |
| DSb | 2.2 | 8     |

DSb=(7 14)
(3 8)

|     | Ac  | Value |
|-----|-----|-------|
| DSc | 1.1 | 9     |
| DSc | 1.2 | 15    |
| DSc | 2.1 | 6     |
| DSc | 2.2 | 2     |

DSc=(9 15)
(6 2)

| OBJ<br>NAME | VEC<br>POS | MAT<br>R | <br>C | VAL |
|---|---|---|---|---|
| A | 2 | 1 | 1 | 7 |
| A | 2 | 1 | 2 | 14 |
| A | 2 | 2 | 1 | 3 |
| A | 2 | 2 | 2 | 8 |
| A | 1 | 1 | 1 | 9 |
| A | 1 | 1 | 2 | 15 |
| A | 1 | 2 | 1 | 6 |

A =
((9 15) (7 14))
((6 2) (3 8))

**Figure 10** - A Vector of Matrices

| OBJECT<br>NAME | VECTOR<br>POSITION | VECTOR<br>POSITION | VECTOR<br>POSITION | VALUE |
|---|---|---|---|---|
| LIST | 1 | 0 | 0 | THIS |
| LIST | 2 | 1 | 0 | IS |
| LIST | 2 | 2 | 1 | A |
| LIST | 2 | 2 | 2 | LIST |

LIST = (THIS (IS (A LIST)))

**Figure 11** - A List

The structure code mechanism is completely general. Lists, for example, are simply a special case of data structures. They are "vectors" whose elements are atoms or other lists. Address function composition can be applied to list structure codes to generate the structure codes for any complex nested

```
        for (i=1, i<2, i++)
          for (j=1, j<2, j++)
            { a[2,i,j] = b[i,j];
              a[1,i,j] = c[i,j];
            }
```

**Figure 12** - Conventional Data Reorganization

list. Figure 11 illustrates the structure codes for a list. Since as described above, arrays and data structures are both just generalizations of associations, the application of this technique to data structures is straight forward although not as intuitive.

S = "A STRING"

| OBJECT NAME | POSITION | VALUE |
|---|---|---|
| S | 1 | A |
| S | 2 | |
| S | 3 | S |
| S | 4 | T |
| S | 5 | R |
| S | 6 | I |
| S | 7 | N |
| S | 8 | G |
| S | 9 | null |

**Figure 13** - A String

**Synonymous Data Structures**

In some applications, it is desirable to view data structures in two or more ways. For example, a string can be thought of as a single variable containing a list of characters or as an array of characters. As can be seen in Figure 13, this dual approach to referencing strings is a natural artifact of using associative addressing techniques. The string as a whole can be accessed by the structure code S.$ while the nth character in the string can be accessed by S.n. Note that this capability is due to the parallel associative implementation of structure codes and does not require multiple variable declarations or equivalences.

**Associative Stack and Queues**

Other commonly used data structures, such as, stacks, queues, and linked storage can also be handled in the associative model. Stacks and queues are simply variable length vectors. A stack push is accomplished by adding a new (larger) ordinal position to the vector. A pop is simply the selection of the largest ordinal position of the vector and the return of its associated value. Queues and linked lists can likewise be easily implemented. See Figure 14.

The time tag column is shown intentionally out of order to illustrate that ordering is immaterial. In reality the nature of stack and queue operation is to order items naturally and as a result, the time tags would normally be in sequential order. In addition, the time tags are shown to be

| TIME TAG | VALUE | TIME TAG | VALUE | TIME TAG | VALUE |
|---|---|---|---|---|---|
| 0 | 10 | 0 | 10 | 0 | 10 |
| not | used | 5 | 17 | not | used |
| 2 | 15 | 2 | 15 | 2 | 15 |
| 4 | 20 | 4 | 20 | not | used |
| 3 | 35 | 3 | 35 | 3 | 35 |
| 1 | 100 | 1 | 100 | 1 | 100 |
| before | | after push 17 | | after pop->20 | |

associative stack

| | | |
|---|---|---|
| | 17 | |
| 20 | 20 | |
| 35 | 35 | 35 |
| 15 | 15 | 15 |
| 100 | 100 | 100 |
| 10 | 10 | 10 |
| before | after push 17 | after pop->20 |

conventional stack

| TIME TAG | VALUE | TIME TAG | VALUE | TIME TAG | VALUE |
|---|---|---|---|---|---|
| 0 | 10 | 0 | 10 | not | used |
| not | used | 5 | 17 | not | used |
| 2 | 15 | 2 | 15 | 2 | 15 |
| 4 | 20 | 4 | 20 | 4 | 20 |
| 3 | 35 | 3 | 35 | 3 | 35 |
| 1 | 100 | 1 | 100 | 1 | 100 |
| before | | after queue 17 | | after next->10 | |

associative queue

| | | |
|---|---|---|
| | 17 | |
| 20 | 20 | |
| 35 | 35 | 20 |
| 15 | 15 | 35 |
| 100 | 100 | 15 |
| 10 | 10 | 100 |
| before | after queue 17 | after next->10 |

conventional queue

**Figure 14** - Associative Stacks and Queues

sequential integers, in reality, they may be any ordered sequence of unique values - numeric, alphabetic or alphanumeric (Alphabetic and alphanumeric codes would be retrieved in ASCII sorted order).

It should be emphasized that lists, queues and stacks are artifacts of conventional sequential programming, and that in an associative programming environment the need for these structures is eliminated.

## CONCLUSIONS

This paper has presented a unified approach for representing arbitrarily complex data structures in content addressable memories and associative computers. This approach to data structures in associative computers has the advantages of 1) automatically extracting fine grain parallelism, 2) eliminating much of the complexity of the non-algorithmic address computation in program development, 3) allowing multiple data structures to be associated with each datum, 4) allowing the data structures themselves to be modified, and 5) allowing information exchange between vastly different program languages such as LISP, PROLOG, OPS5, FORTRAN and PASCAL.

Some areas for future research are:
1) defining arithmetic operations on complex data structures as a natural extension of element by element arithmetic of vectors and matrices, 2) the utilization of multiple distinct structure codes in the same datum. In general, there can be a different structure code for every logical hierarchical data structure to which the datum belongs. This aspect may be particularly useful for semantic networks and frames in AI applications, 3) the development of universal operators for the manipulations of structure codes. For example, the operator "root" will generate the structure code for the root of a tree from the structure code of any of its nodes (See Potter,1985). and 4) the investigation of mathematical properties of addressing functions and structures codes.

## REFERENCES

1. Batcher, K. E., Multidimensional Access Memory in STARAN,"in IEEE COMPUTER, February, 1977, pp. 174-177.

2. Findler, N. V. (ed.), "Associative Networks - Representation and Use of Knowledge by Computers," Academic Press, New York,1979.

3. Foster, C. C., Content Addressable "Associative Networks - Representation and Use of Knowledge by Computers," Academic Press, New York,1979.

4. Jacks, E. L. (ed.), "Associative Information Techniques," Elsevies, New York, 1971.

5. Kohonen, T., Associative Memory: A system-theoretical approach," Springer-Verlag, Berlin, 1977.

6. Potter, Jerry L., "Alternative Data Structures for Lists in Parallel Associative Computers," in THE PROCEEDINGS OF THE 1983 ICPP, Bellaire, Michigan, August 23-26, 1983, pp.486-491.

7. Potter, Jerry L., "Specialized SIMD Instructions for Associative Processing," in PROCEEDINGS ON THE 1985 INTERNATIONAL CONFERENCE ON CIRCUIT DESIGN, Port Chester, New York, October 7-10, 1985, pp. 490-493.

8. Potter, Jerry L., "An Associative Model of Computation," The Second International Conference on Super-Computing, May 4-7, 1987, San Francisco, Ca.

9. Reed, B. Jr., "An Implementation of Lisp on a SIMD Parallel Processor," in AEROSPACE APPLICATIONS OF AI, Dayton, Ohio, September 16-19, 1985.

10. Savitt, D. A., H. H. Love, Jr. and R. E. Troop, 1967 Sprint Joint Computer Conference, p. 87.

11. Simon, H. A. and A. Newell, "Information-Processing in Computers and Man," in PERSPECTIVES ON THE COMPUTER REVOLUTION, Z. W. Pylyshyn, ed., Prentice-Hall, Englewood Cliff, N.J., 1970.

# Parallel Implementations of the Simplex Algorithm

Richard Marciano, Teodor Rus

Department of Computer Science
The University of Iowa
Iowa City, IA 52242

### Abstract

Three parallel implementations of the simplex algorithm on three different parallel architectures, are presented and compared. Each machine is the representative of one class of parallel computers. Performance comparisons and the major difficulties encountered by the user of these machines are given.

The potential for parallel programming of the array processors is investigated with the MPP machine. The multiprocessor systems with asynchronous shared memory are studied by implementing the simplex algorithm on the Encore machine in both the process creation by *fork()* and tasking environment. The class of supercomputers represented by the Alliant FX-8 "mini-supercomputer" where a Fortran compiler can parallelize and vectorize DO loops is considered.

**Keywords:** array processor, parallel programming, performance, simplex algorithm, multiprocessor, vectorization.

## 1 The Simplex Algorithm

The simplex algorithm was developed by Dantzig[DANT63] for finding the solution of a linear programming problem. Its simplicity and elegance made it the essential numeric tool for solving optimizing linear problems. Therefore, it was (and still is) the object of intense study [VAJD60], [FICK61], [BORG80]. Our paper is a contribution toward efficient implementations of the simplex algorithm on parallel processors available today. The general form of a linear programming problem can be expressed as follows:

*Maximize the linear function* $f = c_1x_1 + c_2x_2 + \ldots + c_nx_n$ *where* $c_1, c_2, \ldots c_n$ *are given real numbers called costs and* $x_1, x_2, \ldots, x_n$ *are unknowns subject to the linear restrictions*

$$\begin{array}{rcl} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1i}x_i + \ldots + a_{1n}x_n & \leq & b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2i}x_i + \ldots + a_{2n}x_n & \leq & b_2 \\ \ldots & & \ldots \\ a_{j1}x_1 + a_{j2}x_2 + \ldots + a_{ji}x_i + \ldots + a_{jn}x_n & \leq & b_j \\ \ldots & & \ldots \\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mi}x_i + \ldots + a_{mn}x_n & \leq & b_m \end{array}$$

*and* $x_i \geq 0, i = 1, 2, \ldots n,$ *where* $a_{ji} \in R, i = 1, 2, \ldots, n, j = 1, 2, \ldots, m.$

The standard simplex algorithm [BUND84] consists of a sequence of iterations that for a given solution $\mathcal{X}^0 = (x_1^0, x_2^0, ..., x_n^0)$ of the linear programming problem improves $f$ until the optimum solution is obtained (if one exists, otherwise the absence of a solution is specified). Let $\mathcal{X}^i$ be the solution before iteration $i$. A new solution $\mathcal{X}^{i+1}$ is constructed at iteration $i$ from $\mathcal{X}^i$ with the property that $f(\mathcal{X}^{i+1}) \geq f(\mathcal{X}^i)$. The construction of $\mathcal{X}^{i+1}$ from $\mathcal{X}^i$ is performed by the following sequence of operations:

1. Find the variable $x_c$ which generates the best contribution to the value of $f$ if introduced in the solution. The index $c$ of this variable is given by the maximum coefficient of the function $f$ at this iteration.

2. Find the variable of $\mathcal{X}^i$ that needs to be replaced by $x_c$. The index of this variable is given by the smallest number $b_j/a_{jc}, j = 1, 2, \ldots, m$, for $a_{jc} > 0$. Let it be $b_r/a_{rc}$.

3. Transform the matrix of the initial problem by a Gaussian elimination using element $a_{rc}$ as a pivot, i.e., perform the operation $\forall j, i, j \neq r, i \neq c, a_{ji} := a_{ji} - a_{ri}a_{jc}/a_{rc}$.

Computationally the algorithm can be presented as in figure 1.

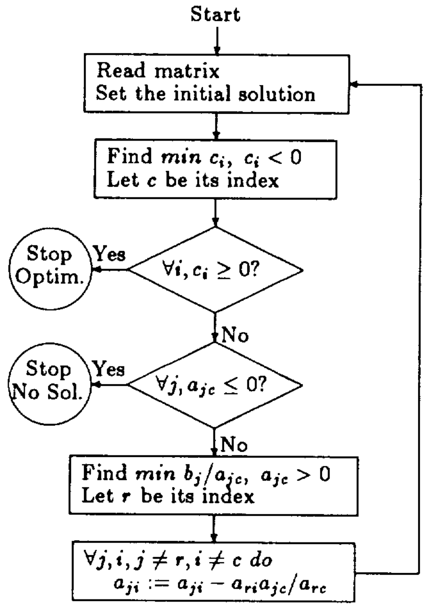

Start
Read matrix
Set the initial solution
Find $min\ c_i,\ c_i < 0$
Let $c$ be its index
Stop Optim.
Yes
$\forall i, c_i \geq 0?$
No
Stop No Sol.
Yes
$\forall j, a_{jc} \leq 0?$
No
Find $min\ b_j/a_{jc},\ a_{jc} > 0$
Let $r$ be its index
$\forall j, i, j \neq r, i \neq c$ do
$a_{ji} := a_{ji} - a_{ri}a_{jc}/a_{rc}$

Figure 1: Flow of control

CH2649-2/89/0000/0085$01.00 © 1988 IEEE

In order to use this algorithm to solve a linear programming problem the set of $m$ linear inequalities defining the problem is first converted into a set of $m$ linear equations by introducing at most $m$ slack variables and by changing the sign of all free terms such that $b_j \geq 0, j = 1, 2, \ldots, m$. The optimizing function $f$ is then added as line 0 of the linear system of equations thus obtained in the form $C - f = 0$ where $C$ is its optimal value (originally 0), i.e., $-c_1x_1 - c_2x_2 - \ldots - c_nx_n = 0$. The initial feasible solution is then obtained by introducing $m$ new variables whose coefficients in the function $f$ are set to zero. The matrix of this system with the free terms in column 0 is organized as the two dimensional array called ***simplex tableau***:

$$\begin{array}{ccccccccc}
a_{00} & a_{01} & \cdots & a_{0i} & \cdots & a_{0n} & a_{0n+1} & \cdots & a_{0n+m} \\
a_{10} & a_{11} & \cdots & a_{1i} & \cdots & a_{1n} & a_{1n+1} & \cdots & a_{1n+m} \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
a_{j0} & a_{j1} & \cdots & a_{ji} & \cdots & a_{jn} & a_{jn+1} & \cdots & a_{jn+m} \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
a_{m0} & a_{m1} & \cdots & a_{mi} & \cdots & a_{mn} & a_{mn+1} & \cdots & a_{mn+m}
\end{array}$$

The simplex tableau is automatically constructed by the procedure reading the simplex matrix.

Using the simplex tableau defined above, the simplex algorithm can be formulated as the following sequence of steps:

1. Find the pivot column, i.e., perform the computation:

$$C = \begin{cases} -1, & \text{if } T[0,i] \geq 0 \text{ for } i = 1, 2, \ldots, n+m; \\ c \geq 1, & \text{if } T[0,c] = min(T[0,i]),\ T[0,i] < 0, \\ & \qquad i = 1, 2, \ldots, n+m. \end{cases}$$

   If $C = -1$, the optimum solution was found. Otherwise step 2 follows.

2. Find the pivot line, i.e., perform the computation:

$$L = \begin{cases} -1, & \text{if } T[j,C] \leq 0 \text{ for } j = 1, 2, \ldots, m; \\ r \geq 1, & \text{if } T[r,C] = min(T[j,0]/T[j,C]), \\ & \qquad T[j,C] > 0, j = 1, 2, \ldots, m. \end{cases}$$

   If $L = -1$ there is no solution. Otherwise step 3 follows.

3. Transform the simplex tableau by the formula:

```
for $i = 0, 1, \ldots, n+m,\ i \neq C$ do
    for $j = 0, 1, \ldots, m,\ j \neq L$ do
        T[j,i] := T[j,i] - T[L,i]*T[j,C]/T[L,C];
```

Clearly parallelism can only be found within each of these three steps. In order to obtain maximum speed, the granularity of parallelization needs to be controlled by the user according to the architecture of the machine. This is done by allowing the user to define the unit of parallelization as being a contiguous block of $H$ lines and $K$ columns of the simplex tableau. The number $N$ of contiguous blocks T[H,K] (i.e., parallel jobs) in which the simplex tableau T can be partitioned is determined by:

$$p = \lfloor (n+m)/K \rfloor + sg(rest((n+m)/K))$$
$$q = \lfloor m/H \rfloor + sg(rest(m/H)), N = p * q.$$

The constants H and K that determine $p, q$ are dependent on the type of hardware and its computation power.

## 2 Implementation on the MPP

The MPP machine[NASA88] is an array processor that operates under the control of a conventional VAX-11/780 front-end (figure 2) and consists of three main units:

1. Array Processing Unit, APU, a two dimensional 128 x 128 mesh with wrap around connections between processing elements in the same row or column denoted by PE(i,j), $i, j = 0, 1, \ldots, 127$. Each PE(i,j) is 1-bit processor containing 1,024 bits of random access memory denoted here by PEM(i,j)[0..1023].

2. Array Control Unit, ACU, which executes scalar operations and controls the operations performed by the APU. The ACU, figure 2, is actually composed of three units:

   - Main Control Unit, MCU, which is the local memory of the MPP used to store an MPP program and its scalar data.
   - I/O Control Unit, IOCU, which controls the flow of data in and out of the APU, in particular data transfers between the APU and STM.
   - Processing Element Control Unit, PECU, which controls the execution of the array operations in the MPP program.

3. Staging Memory, STM, which is a large storage unit of 32 megabytes connected to the APU via a fast 128 bit data path. It is used to buffer data due to limited memory capacity of the APU.

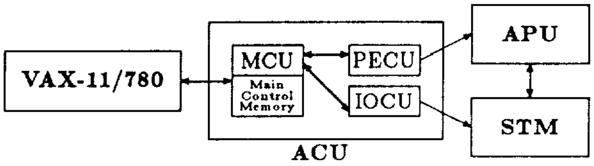


Figure 2: MPP Diagram

An MPP program is a sequential program which contains array operations, I/O operations and scalar operations. The special feature of the ACU is that all of its three control units can operate simultaneously to allow overlapping of the three types of operations found in an MPP application program.

The software support for parallel processing implemented on the MPP allows a user to develop a program using ***parallel arrays*** and operations on parallel arrays as computation units. A parallel array is an abstraction for the APU, i.e., an array of size 128 x 128 of a given type (integer, real, or boolean). An array operation (i.e., having parallel arrays as operands) is simultaneously performed by every PE(i,j) of the APU, each PE(i,j), $i, j = 0, 1, \ldots, 127$ acting in a lock step fashion on the corresponding memory components of the parallel array operands stored in its memory area.

The Pascal language has been extended with new constructs supporting array processing and implemented on the MPP under the name MPP Pascal.

MPP Pascal supports all the Pascal data types. In addition it has been extended with the ***parallel array*** as a predefined data type supporting arithmetical and logical operations and the ***stager array*** as a predefined data type supporting information exchange between APU and STM. Additional language constructs operating on parallel arrays are provided in MPP Pascal allowing parallel array management in a high level fashion such as: *max, min, sum, prod, shift, rotate, transpose, rowbroad, colbroad, insert, extract.*

A parallel array may not be indexed directly. This is why to perform an array operation using selected PE-s of the APU, MPP Pascal provides the special *where* masking statement:

$$\textbf{where} < mask > \textbf{do} < S_1 > \textbf{otherwise} < S_2 >.$$

The *mask* is a boolean expression that evaluates to a parallel array of type boolean mapped on a bit-plane. Each element (i,j) in this bit-plane specifies a processor PE(i,j) of the APU enabling (mask(i,j)=1) or disabling (mask(i,j)=0) its execution.

The operations of information exchange between front-end, array memory and stager memory are shown on figure 3.

The stager memory is treated as an extended array memory of 512 parallel arrays of reals or integers. The unit of transfer is a parallel array.

The synchronization operations *waitq*, which idles the MCU until the PECU has finished, *waitio* which idles the MCU until the completion of the I/O transfer initiated by the IOCU occurs, allow the three components of the MPP to operate concurrently.

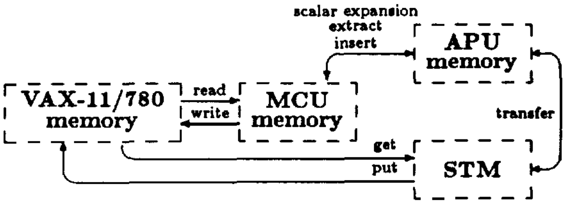


Figure 3: MPP Data flow

A typical MPP application program consists of two parts: an MPP part that runs in the MPP control unit (MPP Pascal), and a host part (Fortran, C, Pascal) that runs on the host. The MPP part is usually the "main" program consisting of data transfers and large scale array computations. The host part generates data files to be read in by the MPP main program and serves as a driver for the MPP program. The communication between MPP part and host part is performed by a tool called CAD. Each part is compiled separately on the host and resulting object modules are linked together to produce a program image that can be directly loaded into the MPP for execution. This discussion will be illustrated further with the MPP simplex implementation.

In order to execute a program the user invokes its executable image through the CAD user interface. Access to the MPP is done on a first come first served basis. Thus, the interaction between the host part and the main part of an MPP program during its execution observes a master-slave relationship.

### 2.1 Simplex Algorithm on MPP

The structure of array memory and stager memory determines the parallelization strategy of the simplex algorithm. It consists of splitting the simplex tableau in as many contiguous subtableaux T[128,128] as possible.

The implementation of the simplex algorithm on the MPP consists of two programs, a program running on VAX called *Prepare_data* and a program running on the MPP called *Simplex*. The program *Prepare_data* written in Fortran performs as follows.

1. Read the dimensions $m, n$ of the simplex tableau maintained as a VAX file and determine constants $p, q$ by the rules:

$$p = [(n+m)/128] + sg(rest((n+m)/128))$$

$$q = [m/128] + sg(rest(m/128))$$

2. Reorganize the simplex tableau T[0..m,0..n+m] as an array of parallel arrays stored on VAX file F2 in the following format:

$$\begin{array}{cccc} P(1,1) & P(1,2) & \ldots & P(1,p) \\ P(2,1) & P(2,2) & \ldots & P(2,p) \\ \ldots & \ldots & \ldots & \ldots \\ P(q,1) & P(q,2) & \ldots & P(q,p) \end{array}$$

$$P(i,j) = T[(i-1)q..(i-1)q+127, (j-1)p..(j-1)p+127].$$

3. Use CAD to invoke *Simplex*, and to wait for its execution.

The program *Simplex* is an MPP Pascal program. It uses the following type declarations:

```
type
    ParAr23  = parallel array[1..23,0..127,0..127] of real;
    ParAr1   = parallel array[0..127,0..127] of real;
    ParArInt = parallel array[0..127,0..127] of integer;
    ParArBol = parallel array[0..127,0..127] of boolean;
    StAr1    = stager array[0..127,0..127] of real;
    StAr512  = stager array[1..512,0..127,0..127] of real;

program Simplex(input,output,row_index,col_index,T1,T2);
%include 'type.dat'
%include 'procedures.dat'
var
  T1 : text; T2 : file of StAr1; A : ParAr23;
  T : StAr512; C_ind, R_ind, Pivcol, Pivrow : integer;
  K1, K2, Flag : integer; DONE, IMPOSSIBLE : boolean;
begin
  zeroarr;
  DONE := false;
  IMPOSSIBLE := false;
  Load(T1,T2,T,A,Flag);
  if (Flag <> 0) then
    repeat
      if (Flag = 23) then
        begin
          PivCol(A,C_ind,Pivcol,K2);
          if (C_ind <> -1) then
            begin
              PivRow(A,R_ind,Pivrow,C_ind,Pivcol,K1,K2);
              if (R_ind <> -1) then
                Update(A,R_ind,Pivrow,C_ind,Pivcol,K1,K2);
              else NoSolution := true;
            end
          else Done := true;
        end
    if (Flag = 512) then
      begin
        StPivCol(T,C_ind,Pivcol,K2);
        if (C_ind <> -1) then
          begin
            StPivRow(T,R_ind,Pivrow,C_ind,Pivcol,K1,K2);
            if (R_ind <> -1) then
              StUpdate(T,R_ind,Pivrow,C_ind,Pivcol,K1,K2);
            else NoSolution := true;
          end
        else Done := true;
      end
  until (Done or NoSolution)
end.
```

The *Load* procedure recomputes the constants K1 and K2 by the rules shown above and reads the file F2 into the array memory or stager memory depending upon its size. Therefore, the simplex on the MPP operates in two modes distinguished by the variable *Flag*. When *Flag* ≤ *28* the entire simplex tableau is stored in the array memory and *PivCol, PivRow, Update* are then used. When *28* < *Flag* ≤ *512* the simplex tableau is stored in the stager memory and parallel arrays need to be *swapped-in and swapped-out* in order to be processed and updated. The procedures *StPivCol, StPivRow, StUpdate* similar to *PivCol, PivRow, Update* need to be used in that case.

Let us suppose for sake of clarity that the simplex tableau is small enough to be entirely mapped onto one parallel array $A[1,,]$. Each entry $(i,j)$ in the tableau is thus mapped onto its own processor PE(i,j). Once pivot column and row have been determined, the tableau updating can be carried out simultaneously by all PE-s in one array operation. To perform this updating each PE(i,j) needs to access three tableau items, $(i,j)$, $(piv\_row,j)$ and $(i,piv\_col)$. The last two tableau items are not accessible by PE(i,j) and data communication and exchange between processors PE(i,j), PE(piv_row, j) and PE(i, piv_col) is necessary. This is performed by creating two new parallel arrays (BrPC and BrPR) both constructed by using *shift* and *broadcast* array functions as carried out by the *Update* procedure. PE(i,j) now has access to the three corresponding tableau items and the tableau updating is performed by $A[1,,] := A[1,,] - BrPR \times BrPC$, figure 4. When the tableau maps over more than one parallel array in the array memory, this data broadcasting scheme is applied iteratively to each parallel array.

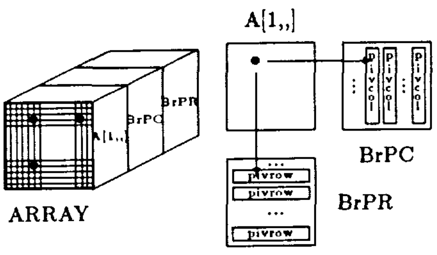


Figure 4: Simplex tableau updating operation

### 2.2 Performance Measurements

Performance measurements of the simplex implementation discussed above are given in the table 1. The lines of the table are labeled by the number of iterations required to find the solution while the columns are labeled by the number of parallel arrays required to store the simplex tableau. The time in seconds taken by the MPP to solve a problem of the size the number of parallel arrays recorded in the column j and performing the number of iterations recorded in the line i is recorded in the table entry (i,j).

| | 1 | 4 | 9 | 16 | 25 | 36 | 49 | 64 |
|---|---|---|---|---|---|---|---|---|
| 3 | 0.003 | 0.006 | 0.011 | 0.016 | 0.43 | 0.57 | 0.73 | 0.91 |
| 6 | 0.006 | 0.013 | 0.022 | 0.033 | 0.84 | 1.10 | 1.40 | 1.80 |
| 44 | 0.047 | 0.102 | 0.173 | 0.260 | 6.13 | 8.18 | 10.0 | 12.0 |

Table 1: Performance measurements on MPP

## 3 Implementation on Multimax

A new class of computers [GORD87] called *Multimax* emerges as multiprocessor computers using as components microprocessors that have the speed and functionality of mid-range supercomputers. The Encore Multimax is a modular system designed as a component of the *Encore Computing Continuum*[ENCO87], which provides a true multiprocessing and distributed environment. The Encore Continuum uses tightly coupled multiprocessing, distributed, and intelligent control of I/O devices and clustering of Multimax systems. A multimax cluster incorporates from 2 to 20 32-bit processors each provided with 32K byte cache of fast static RAM, 4 to 32 Mbytes of fast shared memory and configurable I/O devices.

The Multimax support for parallel program development and execution consists of a library of functions that extend the collection of system calls supported by Unix[1] and allow the user to create parallel processes in a program, schedule them for execution while sharing resources, and control their interaction. A user can take advantage of these functions creating and managing a process environment or a tasking environment.

### 3.1 Simplex with Process Environment

The process environment is provided by the *fork()* system call that allows a program to create processes in the user program. The function *MakeProce* was designed by us in order to allow the simplex user to create a variable number of processes.

Process interaction is done by all processes having access at the variable declared shared. There are two classes of system calls in the parallel library allowing the user to declare shared objects:

1. When shared memory is statically managed the user proceeds as follows:
   - Declare a C-language data structure, say *data* and/or a pointer to it, say *datapt*.
   - Call the function *share()* in the parallel library to make *data, datapt* shared under the form

     datapt = share(0, sizeof(data));

[1] Unix is a trademark of Bell Labs

2. When shared memory is dynamically managed then the user proceeds as follows:

   - Provides the memory area to be manipulated dynamically by the program using the call

     alloc = share_malloc_init(size);

     which returns a pointer to an area of memory of size "size".
   - Manage dynamically the memory pointed to by alloc using calls of the form

     datapt = share_malloc(sizeof(data));
     share_free(datapt);

Process synchronization is done by using the lock data types supported by the Encore Multimax[RUS88]:

- **Lock**: is a binary semaphore supporting the operations *spin_init(lock, flag), spin_create(flag), spin_unlock(lock), spin_condlock(lock)*, where *flag* shows the state of the lock.

- **Barrier**: allows a fixed number of processes to synchronize at a given point in a program. It supports the operations *barrier_create(count, state), barrier_init(lock, count, state), barrier(lock)* where count is the number of processes that need to arrive at the barrier before it opens.

- **Semaphore**: is general semaphore supporting the operations *semaphore_init(lock, state), semaphore_create(state), semaphore_wait(lock), semaphore_signal(lock).*

- **Event**: provides a barrier at which a variable number of processes can wait having two states, *event_posted* and *event_cleared.* It supports the functions *event_create(state), event_init(lock, state), event_post(lock), event_clear(lock), event_wait(lock).*

The *lock* parameter is a pointer to an object of type the type supporting the function using it, *state* is *SPIN_BLOCK, PROCESS_BLOCK, TASK_BLOCK*, showing the mechanism implementing wait, *count* is an integer and *flag* is PAR_LOCKED or PAR_UNLOCKED.

The lock variables used in a program need to be created in shared memory. All operations supported by the lock data types specified above are atomic. In addition, the parallel library provides the function *timer_init()* and *timer_get()* which allow the timing of the program execution.

The structure of a parallel program under process environment is illustrated by the following sketch of the simplex implementation.

```
#include <stdio.h>
#include <parallel.h>
#define cols 1200
#define lines 1200
struct shared_area
  {
   double pivot, a[lines][lines+cols];
   int m, n, H, K, Procs, Jobs, JobCount;
   int C, L, p, q, ColCount, RowCount;
   BARRIER barr;
   LOCK lock;
   } *glob;
```

```
int IdProc = 0;
main (int argc, char *argv[])
    {
     int i, State = SPIN_BLOCK;
     glob = share (0, sizeof(*glob));
     /* Read matrix, parameters, and initialize data */
     spin_init(&glob- >lock, PAR_UNLOCKED);
     barrier_init(&glob- >barr, &glob- >Procs, State);
     IdProc = MakeProcs(&glob- >Procs-1);
Start: PivCol(&glob- >K,&glob- >C);
     barrier(&glob- >barr);
     if (&glob- >C < 0) { PrintSolution(); exit()}
     PivRow(&glob- >H,&glob- >L);
     if (IdProc == 0)
       &glob- >JobCount = 0;
     barrier(&glob- >barr);
     TransformL(&glob- >L, &glob- >K);
     barrier(&glob- >barr);
     if (&glob- >L < 0) { NoSolution(); exit()}
     i = Monitor(&glob- >JobCount);
     while (i < &glob- >Jobs)
           {
             Update(i/p, i%p, &glob- >H, &glob- >K);
             i = monitor(&glob- >JobCount);
           }
     barrier(&glob- >barr);
     TransformC(&glob- >C, &glob- >H);
     barrier(&glob- >barr);
     goto Start;
   }
```

The functions *PivCol(), PivRow(), Update(), TransformL(), TransformC()*, and *Monitor()* implement the three steps of the simplex algorithm, perform matrix transformations and ensure computation consistency, respectively.

### 3.2 Simplex with Tasking Environment

Since process creation is a very costly operation, the tasking mechanism was developed to support parallel program development on the Encore Multimax. A task is a function provided with its own stack and thus capable of being executed in parallel with other tasks. A parallel program using the tasking environment consists of a collection of tasks that can be executed in parallel. There is a special task called *master* that starts the execution of the program initiating other tasks. Each task in turn can start other tasks. The tasking environment of a program is thus defined by the memory size *Mem* used to allocate stacks for the tasks, the number of processes *Procs* that run tasks in parallel and the master task, *Master.* The tasking environment of a program and the start of the master task are set up with a call to the function *task_init*

task_init(Mem, Procs, Master, Stack, Argc, Arg0,...,Argn);

The starting of a task specified by a function *Func* in the tasking environment (by master task and/or by other tasks) is performed by the call

task_start(Func, Stack, Argc, Arg0, ..., Argm);

which allocates *Stack* bytes as the stack of this task from *Mem*, transmits arguments on the stack and starts a process to execute the code of *Func* on this stack, if there exists a processor available for this purpose.

The tasking environment is controlled by the program using the following tasking primitives: ***task_suspend()*** that suspends its caller; ***task_resume(name)*** makes the task *name* reschedulable; ***task_stop()*** terminates its caller; ***task_join()*** waits for all tasks initiated by its caller to terminate; ***task_self()*** returns the task identification number of its caller.

There are two restrictions imposed on parallel program development by the tasking approach: the code of a function designed as a task needs to be provided in the program text before the invocation of that task and the locks need to be created and initialized in the main program. The consequence is a bottom-up approach for program development. The structure of a parallel C language program using the tasking environment is illustrated by the following sketch of the tasking version of the simplex algorithm.

```
#include < stdio.h >
#include < parallel.h >
#define Stacks 20000
#define Stack 500
#define cols 1200
#define lines 1200
double pivot, a[lines][lines+cols];
int m, n, Procs, JobCount, Jobs, H, K;
int answer, ColCount, RowCount, C, L, p, q;
LOCK *lock;
transform()
   {
    int i;
    i = Monitor(JobCount);
    while (i < Jobs)
          {
           Update(i/p, i % p, H, K);
           i = Monitor(JobCount);
          }
   }
master ()
   {
     int i;
     Start: ColCount = 0;
     for (i = 0, i < Procs, i++)
         task_start(Stack, PivCol, 1, C);
     task_join();
     if (C < 0) { answer = 1; return }
     RowCount = 0;
     for (i = 0, i < Procs, i++)
         task_start(Stack, PivRow, 1, L);
     task_join();
     if (L < 0) { answer = -1; return }
     JobCount = 0;
     for (i = 0; i < Procs; i++)
         task_start(Stack, transform, 0);
     task_join();
     goto Start;
   }
```

```
main (int argc, char *argv[])
   {
    /* Read matrix, parameters and initialize data */
    lock = share (0, sizeof(LOCK));
    spin_init(lock, PAR_UNLOCKED);
    Set_timers();
    task_init (Stacks, Procs, master, Stack, 0);
    Get_timers();
    if (answer == 1) PrintSolution();
    else PrintNoSolution();
   }
```

### 3.3 Performance Measurements

The performance measurements of the simplex implementation on the Multimax using the process environment and the tasking environment closely follow the same pattern. Therefore in tables 2 and 3 we only present the performance of the program implemented in the process environment which is slightly better than for the tasking environment. Table 2 illustrates the variation of the time to solve a problem whose simplex tableau was 512 × 512 (i.e., $256 \times K$ elements), with the number of processes running in parallel and the granularity of their interaction. The lines of this table are labeled by the number of processes running in parallel and the columns are labeled by the granularity of the process interaction. The granularity is expressed by the size of the contiguous subtableaux of the simplex tableau transformed by a process independently of the other processes. This is given in the number of parts in which the lines and columns of the simplex tableau are divided. The time in seconds needed by the Encore parallel processor to solve the problem is recorded in the entry (i,j). However, examining the behavior of the algorithm on a large number of problems we observed that the best time was provided by job size (16, 260) with 12 processors. Therefore, the last column of table 2 records the behavior of the algorithm for this process interaction granularity.

| | 1 | 2 | 4 | 8 | 16 | 32 | (16,260) |
|---|---|---|---|---|---|---|---|
| 1 | 33.6 | 9 | 20 | 27.2 | 32.4 | 35.3 | 17.8 |
| 3 | 33.9 | 9.4 | 7.2 | 9.8 | 11.1 | 11.7 | 6.6 |
| 6 | 34.4 | 9.7 | 5.3 | 5.8 | 6.3 | 6.5 | 4.2 |
| 9 | 34.8 | 10.2 | 3.6 | 4.6 | 5 | 5.2 | 3.6 |
| **12** | 35.2 | 10.7 | 4.1 | 4.5 | 4.6 | 4.7 | **3.5** |
| 15 | 35.7 | 11.1 | 4.7 | 4.5 | 4.5 | 4.6 | 3.9 |
| 18 | 39.9 | 12.8 | 6 | 4.9 | 5.3 | 5.3 | 4 |

Table 2: Granularity study

| | 1 | 4 | 9 | 16 | 25 | 36 | 49 | 64 |
|---|---|---|---|---|---|---|---|---|
| 3 | 0.6 | 2 | 2.5 | **3.5** | 4.6 | 5.5 | 6.7 | 8.2 |
| 6 | 1 | 2.1 | 3.5 | 5 | 6.6 | 8.5 | 10.5 | 12.9 |
| 44 | 2.9 | 7.1 | 13.2 | 21.9 | 36.4 | 44.3 | 62.5 | 73.2 |

Table 3: Encore performance measurements

## 4 Implementation on Alliant

The Alliant FX8 is a register to register machine equipped with 8 MC68000 compatible vector processors, 11 interactive processors for input and output, and a 64 megabyte memory subsystem [ARGO86]. Parallel processing on the Alliant is performed by pipelining vector operations and by parallel processing of the 8 vector processors.

Parallel programming support is provided by the Concentrix operating system (Unix-based) and the FX/Fortran language which supports the array data type. Like MPP Pascal, FX/Fortran has been extended with a set of intrinsic array functions : *min(Ar), max(Ar), size(Ar), etc...*, as well as a very similar conditional array assignment statement allowing masking on an array assignment:

**where** < *cond* > < *st*1 > **otherwise** < *st*2 > **end where**

A Fortran optimizing compiler generates parallel streams of control and vector operations unfolding DO loop operations under programmer control.

Four modes of program execution are available on the Alliant FX8, concurrent execution, vector execution, vector-concurrent execution, and concurrent-outer vector-inner execution. Typically a programmer tunes the program execution by inserting compiler directives in the Fortran code. These directives might for example either turn off vectorization for a specific loop or rather force concurrency specifying that there are no data dependencies in a loop. The information allowing the programmer to inject compiler directives in his program is provided by the compiler itself. There are only 7 groups of possible directives. The syntax of a compiler directive is **CVD$**[*s*] ***directive*** where:

$$s = \begin{cases} G, & \textit{directive} \text{ applies globally (i.e., to the end of file);} \\ R, & \textit{directive} \text{ applies to end of the current routine;} \\ L, & \textit{directive} \text{ applies to end of the current loop (default).} \end{cases}$$

The available ***directives*** are ( * indicates the default value): *ASSOC, NOASSOC**, telling the compiler to perform the optimization of the associative operations; *CNCALL, NOCNCALL**, allowing subroutine and function references in loops optimized for parallel execution; *CONCUR*, NOCONCUR*, forcing (or inhibiting) the optimization for concurrency irrespective of data dependency; *DEPCHK*, NODEPCHK*, telling the compiler to check (or to inhibit the checking) for data dependencies between loop iterations; *LSTVAL*, NOLASTVAL*, telling the compiler to generate code to save last values of original indexes and promoted scalars of optimized loops and arrays; *SYNC*, NOSYNC*, telling the compiler to check for synchronization problems between loop iterations; *VECTOR*, NOVECTOR*, telling the compiler to optimize (or to inhibit the optimization) for vectorization.

### 4.1 Simplex on Alliant FX8

In the Alliant FX8 simplex implementation through compiler directives, we inhibited vectorization of certain loops and strategically forced concurrency by inhibiting data dependency checks. A sketch of the Fortran version of the simplex algorithm fully tuned and optimized and running on all eight processors follows:

```
        program simplex
        integer L, C, m, n
        real mat(1200, 1200)

CVD$R NOLSTVAL
C Read matrix and initialize data
        SetTimer
99
C Find Pivot col and find Pivot row
C Divide Pivot row
CVD$L NOSYNC
        do 80 j = 1, m
        if(j.ne.L) then
CVD$L NOSYNC
            do 40 k = 1, n+m
              if(k .ne. C) then
                mat(j,k)=mat(j,k)-mat(L,k)*mat(j,C)
              endif
40          continue
            endif
80      continue
C Divide Pivot col
        goto 99
C GetTimer
C Print solution or lack of solution
        stop
        end
```

In this implementation all the code is brought in the main program to avoid subroutine calls.

### 4.2 Performance Measurements

In order to compare the performance of the simplex algorithm implemented on the three machines, MPP, Encore Multimax, and Alliant FX8, we run the program on the same set of problems and organize the results in the same way. The behavior of the algorithm on the Alliant FX8 is given in table 4.

| | 1 | 4 | 9 | 16 | 25 | 36 | 49 | 64 |
|---|---|---|---|---|---|---|---|---|
| 3 | 0.015 | 0.093 | 0.39 | 0.79 | 1.3 | 1.8 | 2.5 | 3 |
| 6 | 0.026 | 0.15 | 0.74 | 1.6 | 2.5 | 3.6 | 4.9 | 5.9 |
| 44 | 0.11 | 0.8 | 4.6 | 9.5 | 16.7 | 24.5 | 33.6 | 41.2 |

Table 4: Alliant performance measurements

## 5 Instead of Conclusion

The conclusions of the experiments we performed with the simplex algorithm implemented on the three different computers are twofold. On the one hand they regard the efficiency of the algorithms implemented on the new parallel processing architectures measured by the speed-up obtained by their parallelization and on the other hand they regard the user convenience of the various parallel processing architectures measured by the difficulties implied in their programming.

The speed-up of *Machine1* versus *Machine2*, (*Machine 1, Machine 2* are A for Alliant, E for Encore, and M for MPP) while solving a problem requiring a given number of iterations for various sizes of the simplex tableau is recoded in a line of a speed-up tableau labeled by *Machine1:Machine2* in table 5. The size of the simplex tableau used in our experiments is measured in number of parallel arrays required to accommodate it.

| | 1 | 4 | 9 | 16 | 25 | 36 | 49 | 64 |
|---|---|---|---|---|---|---|---|---|
| 3 - *A:E* | 40 | 20.4 | 7.7 | 5.6 | 4.4 | 3.9 | 3.5 | 3.5 |
| 3 - *M:A* | 5.2 | 14.8 | 36.1 | 48.2 | 3 | 3.2 | 3.4 | 3.3 |
| 3 - *M:E* | 207 | 302 | 278 | 268 | 13.3 | 12.3 | 12 | 11.6 |
| 6 - *A:E* | 38.5 | 14 | 4.7 | 3.1 | 2.6 | 2.4 | 2.1 | 2.2 |
| 6 - *M:A* | 4.3 | 11.6 | 33.6 | 48.5 | 3 | 3.3 | 3.5 | 3.3 |
| 6 - *M:E* | 167 | 162 | 159 | 152 | 7.9 | 7.7 | 7.5 | 7.2 |
| 44 - *A:E* | 26.4 | 8.9 | 2.9 | 2.3 | 2.2 | 1.8 | 1.9 | 1.8 |
| 44 - *M:A* | 2.3 | 7.8 | 26.6 | 36.5 | 2.7 | 3 | 3.4 | 3.4 |
| 44 - *M:E* | 61.7 | 69.6 | 76.3 | 84.2 | 5.9 | 5.4 | 6.2 | 6.1 |

Table 5: Speed-up for 3, 6 and 44 iterations

The simplex problems for these sizes were actually automatically generated from smaller problems. Therefore, instead of definitive conclusions we present our findings as the following three observations:

- The simplex algorithm provides a natural application in which operations on matrices are used. Therefore, array and vector processors should perform better than the general multiprocessor machines. This was confirmed by the speed-up of the algorithm implemented on the three machines.
- The second conclusion shows that even for problems that are naturally suited for array and vector operations, the control of the granularity of process interaction allows the shared memory multiprocessor to become comparable in efficiency to the vector processor in the case of large size problems.
- The third conclusion shows that the performance of the vector processor provided with parallel execution becomes comparable with that of the array processor when the the size of the problem is large. This is due to the cost of array transfer between array memory and stager memory.

Parallel processors clearly allow the simplex algorithm to become an efficient tool in solving linear programming problems. Therefore, comparing the standard version of the simplex algorithm [TARJ83],[DANT79] with the newly discovered polynomial time algorithms [ASPV79], [BORG80](pp. 18-22) may provide different data when executed in parallel environments. So, further study of the parallel implementations of the simplex algorithm and its comparison with the parallel implementations of these newer methods are necessary.

Each of the three different philosophies of handling parallel processing has its specific type of user difficulties. The major difficulties in programming an *array processor* result from the promotion of the array (which is a defined type in most programming languages) to a predefined data type. However, the predefined type "array" does not coincide with the array type existing in most languages nor with the matrix type existing in mathematics. Therefore, in order to take advantage of the machine's potential for parallel processing both experience and the language support developed in this respect provide the necessary help. The major difficulties in developing parallel programs for a *multiprocessor machine* result from the requirement to explicitly manage the implicit process type in the program. This task is performed by the multiprogramming (multiprocessing) operating system operating on a sequential program. Again, experience, the development of concepts and their encapsulation in appropriate data types in the language seem to provide the real help. As for developing parallel programs through the *compiler* the major difficulties result from the compiler-programmer-processor interaction which requires the programmer to have knowledge of architecture, compilers, and the behavior of the algorithm. Therefore, this could be only a temporary solution used to successfully parallelize existing code that would otherwise be too expensive to redesign.

## 6 Acknowledgments

We would like to express our acknowledgments to Prof. G. Carmichael who provided us with the possibility to learn and use the MPP machine, to Prof. E. Haug and D. Golden, for allowing us to use the Alliant FX8 and Encore Multimax in the HSCF of the University of Iowa, Iowa City, and to Daniela Rus, for the valuable observations and suggestions leading to improvements of our work.

## References

[ARGO86] Argonne National Laboratory, *Using the Alliant FX/8*, ANL/MCS-TM-69, Rev. 1, Mathematics and Computer Science Division, September 1986.

[ARGO87] Argonne National Laboratory, *Using the Encore Multimax*, ANL/MCS-TM-65, Rev. 1, Mathematics and Computer Science Division, February 1987.

[ASPV79] Aspval, B., Stone, R. E., "Khachian's linear programming algorithm", J. Algorithms, 1 (1980) pp 1-13

[BORG80] Borgwardt, K. H., *The Simplex Method*, A Probabilistic Analysis, Springer-Verlag, New York 1988.

[BUND84] Bunday, B. D., *Basic Linear Programming*, Edward Arnold, London 1984.

[DANT63] Dantzig, G. B., *Linear Programming and Extensions*, Princeton Univ Press, Princeton, NJ 1963

[DANT79] Dantzig, G. B., "Comments on Khachian's Algorithm for linear programming", Tech. Report SOR 79-22, Dept. Operations Research, Stanford Univ, Stanford, CA, 1979

[ENCO87] Encore Multimax, *Using the Encore Multimax* Argonne National Laboratory, MCS-TM-65, pp. 1-1, 1987.

[FICK61] Ficken, F. A., *The Simplex Method of Linear Programming*, Holt, Rinehart and Winston, New York, 1961.

[GORD87] Gordon, Bell, "The Multi - A New Computer Class", *Using The Encore Multimax*, Argonne National Laboratory, MCS-TM-65, pp. 5-8, 1987.

[NASA88] NASA Goddard Space Flight Center, Greenbelt, Maryland, *MPP Pascal Programmer's Guide*, March 1988.

[RUS88] Rus, T., "Language Support for Parallel Programming", Proceedings of Computer Standards Conference, pp. 21-23, March 21-23, Washington, D.C., 1988.

[TARJ83] Tarjan, R. E., *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics Philadelphia, Pennsylvania 1983.

[VAJD60] Vajda, S., *Linear Programming and the Theory of Game*, John Wiley and Sons, New York, 1960.

# REGION GROWING ON A HIGHLY PARALLEL MESH-CONNECTED SIMD COMPUTER

Marc Willebeek-LeMair
School of Electrical Engineering
Cornell University
New York, USA

Anthony P. Reeves
Department of Computer Science
University of Illinois at Urbana-Champaign
Illinois, USA

## ABSTRACT

The region growing paradigm for image segmentation groups neighboring pixels into regions depending upon a predetermined homogeneity criteria. A parallel method for region growing on a highly parallel SIMD mesh computer is presented. The approach is based upon a parallel merging paradigm, which involves the selection of the best of all merge possibilities for all regions concurrently. A key requirement of any parallel region growing scheme is the ability to concurrently compute functions on irregular shaped regions. A set of general primitive functions for region growing have been defined and techniques to implement these functions on an SIMD processor have been developed. These techniques make use of an embedded tree data structure to represent regions. The results of implementing a parallel split and merge region growing algorithm on the Massively Parallel Processor are discussed. The approach is shown to be efficient primarily for images involving large numbers of regions.

**Keywords:** Non-uniform load distribution, Parallel processing, SIMD, Segmentation and 2-D description, Parallel region growing, and Split and merge.

## INTRODUCTION

Region growing is a general technique for image segmentation. Frequently, the basic scheme is to combine pixels with adjacent pixels to form regions; regions are then merged with other regions to "grow" larger regions. The association of neighboring pixels and neighboring regions in the region growing process is often governed by a homogeneity criterion that must be satisfied in order for the pixels and regions to combine. The homogeneity criteria are application dependent and may be dynamic within a given application.

In addition to the homogeneity criterion the order in which regions are merged can have an important effect on the final result; many sequential region growing algorithms ignore this fact. We introduce a parallel merging paradigm in which a merge decision is based upon the best of all alternatives for all regions simultaneously.

Highly parallel SIMD processors have been shown to be very effective for regular algorithms such as image filtering and the FFT. This work extends the domain of the SIMD processor to the irregular processing characteristics of region growing algorithms which exhibit non-uniform and unpredictable load distibutions. Our results indicate that, especially for the case of a very large number of small objects, powerful parallel region growing techniques can be implemented in a reasonable amount of time. Section two of this paper outlines the principles of region growing with emphasis on the split and merge algorithm. Section three discusses the characteristics of the SIMD architecture and section four introduces a parallel merging paradigm and considers the parallel implementation of a region growing technique on an SIMD architecture. The algorithm complexity is discussed in section five. An example of the parallel region growing technique is presented through a split and merge implementation on the Massively Parallel Processor (MPP) in section six.

## REGION GROWING

Region growing is a technique for partitioning an image by linking individual pixels into groups of pixels called regions. The merging of pixels or regions to form larger regions is usually governed by a pre-defined homogeneity criterion that must be satisfied. In this section we first define a region and discuss its properties, next, we review the concept of homogeneity criteria, and finally, we present a split and merge algorithm as an example of a region growing technique.

### Regions

A region might correspond to a world object or part of one. As defined by [1], a region is a four-connected, two-dimensional area that is allowed to be non-simply connected (contain holes). A single pixel cannot belong to more than one region. These properties stated more formally are as follows.

A region R is considered to be a set of points with the following properties:

(1) $x_i$ in a region R is connected to $x_j$ iff there is a sequence $\{x_i,...x_j\}$ such that $x_k$ and $x_{k+1}$ are connected and all the points are in R.

(2) R is a connected region if the set of points x in R has the property that every pair of points is connected.

(3) I, the entire image = $\bigcup_{k=1}^{m} R_k$.

(4) $R_i \bigcap R_j = 0$, $i \neq j$.

### Homogeneity Criteria

When grouping pixels into regions it is usually necessary that the groups satisfy some sort of homogeneity criteria. Therefore, the grouping of neighboring pixels into regions is dependent on the characteristics of the individual pixels. Once pixels have been combined to form a region, the region assumes certain properties based on the combined characteristics of the pixels as a group (e.g. area or texture). A homogeneity criteria can be designed to specify such things as the maximum range or gradient allowable within a region, etc. This criteria will then be used as a test to determine whether or not a given group of pixels can be classified as a region.

As an example, the pixel range homogeneity criteria H(R) is defined as follows:

$$H(R) \equiv \begin{cases} true, & \textit{if for all point pairs x and y in R,} \\ & f(x)-f(y)<T. \\ false, & \textit{otherwise.} \end{cases} \qquad (1)$$

This particular criterion requires that the range between the minimum and maximum values within a region R, not exceed a threshold T.

A variety of homogeneity criteria have been investigated for region growing, ranging from statistical techniques, which involve the distribution of pixel grey levels, to state-space approaches, which represent regions by their boundaries [1,2,3].

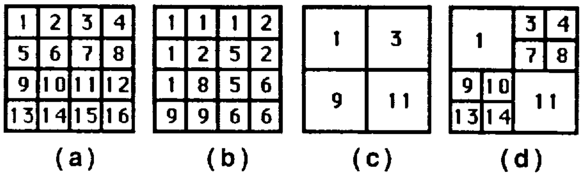
A problem with many region growing schemes is that large regions require an excessive number of merge steps. A computationally expedient technique called split and merge [4] addresses the large region problem with a preprocessing split phase.

CH2649-2/89/0000/0093$01.00 © 1988 IEEE

## The Split and Merge Approach

The split and merge technique requires two types of operations; in our algorithm a fast split phase is followed by one or more merge phases. The split stage rapidly partitions an image into square regions which conform to a first homogeneity criterion; then a region growing technique is used to merge these square regions into larger regions which conform to a second homogeneity criterion. High speed is achieved by the top down approach that rapidly deals with very large regions. Such regions require many iterations to achieve with just a region growing paradigm.

**Splitting**--The first phase of the algorithm is concerned with dividing the image up into homogeneous, square regions of varying dimensions. Beginning with the entire image as the area in question, an area is checked for homogeneity. Should the homogeneity check fail, the area is split into four quadrants. Each of the quadrants is then checked the same way and recursively split until the homogeneity requirement is satisfied. If an area passes the homogeneity test then it is considered a region and left alone until the merge stage of the algorithm. The scheme is illustrated in Figure 1. For this case the homogeneity criterion requires that the range of the pixel values in a region must not be greater than 2. Initially each pixel is labeled with a unique identifier number (ID). While splitting, pixels are assigned a region ID number corresponding to the pixel ID of the pixel located in the upper left hand corner of the quadrant to which they are associated.



|  |  |  |  |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 |

|  |  |  |  |
|---|---|---|---|
| 1 | 1 | 1 | 2 |
| 1 | 2 | 5 | 2 |
| 1 | 8 | 5 | 6 |
| 9 | 9 | 6 | 6 |

**Figure 1.** Split and Merge splitting phase. (a) Pixel ID numbers. (b) Pixel values. (c) Region ID's after first split. (d) Region ID's after second and final split.

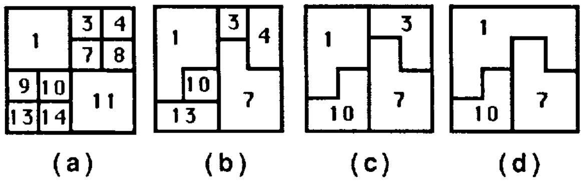


**Figure 2.** Split and Merge merging phase. (a) Region ID's after splitting phase. (b) Region ID's after first merge. (c) Region ID's after second merge. (d) Region ID's after third and final merge.

**Merging**--In the merge phase, adjacent regions are merged to form non-square regions. Region pairs that satisfy the homogeneity criterion are allowed to merge. Figure 2 presents an example of the merging stage for the regions split in Figure 1. In this case, the same homogeneity criterion as for the split phase is used again. Now, however, merged region pairs assume the smaller ID of the two regions.

For many merging criteria, including the one used in the example, the order of merging is important and this affects both the execution time and the final result. An approach which involves increasing the threshold value, in stages, in the merge phase, has been found to improve the quality (i.e., produce less regions) in the final result for region growing using both range images [13] as well as grey level images [5]. Certain constraints imposed on the merging order, as mentioned above, help to improve on the final results, other constraints, however, are imposed in the parallel merging strategy to avoid violating the homogeneity criteria. These constraints will be addressed later in the section on parallel region growing.

## THE MESH-CONNECTED SIMD ARCHITECTURE

Many highly parallel computer architectures designed for low level image processing applications have been proposed and implemented [6,7,8]. These designs exhibit a variety of interconnection schemes between processors and the processors themselves range from simple processing elements in the SIMD arrangements to much more sophisticated processors in the MIMD systems. The architecture of interest in this paper is the highly parallel (tens of thousands of binary processors) mesh-connected SIMD processor array. The SIMD mesh consists of an array of identical processing elements (PE's) with near-neighbor connections. We will be considering the 4-connected case in which processors are connected to their neighbors to the north, east, south, and west. The architecture we are considering is illustrated in Figure 3. The array of PE's is controlled by a host computer that issues it instructions when a parallel array computation needs to be performed and is linked to memory via a bi-directional I/O path. Each PE contains an ALU with a limited amount of local memory.

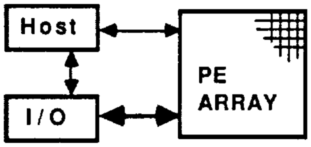


**Figure 3.** General organization for a mesh-connected SIMD architecture.

The SIMD-mesh architecture is particularly well suited for low-level image processing. The physical layout of the processor array allows for a direct mapping of pixels to PE's. Furthermore, the type of computations inherent in low-level processing, such as image filtering and edge detection, involve the analysis of a pixels local neighborhood.

## The Massively Parallel Processor Architecture

The Massively Parallel Processor [11] is a SIMD mesh-connected computer arranged in a two-dimensional 128x128 array of PE's. The PE's are bit-serial, allowing for a flexible data format and efficient utilization of resources. Each PE can perform high speed arithmetic and implement all sixteen possible boolean functions. In addition, each processor contains a mask register that can be set to inhibit execution of an instruction. PE's are connected to their near-neighbors to the north, east, south, and west. For data to be transmitted from one point of the array to another, it must be routed via a path linking both points through adjacent processors. The MPP is equipped with a built-in sum-OR tree which combines the output from all PE's in a tree of OR elements. This can be used to check for termination or convergence of a repeated sequence of instructions.

## PARALLEL REGION GROWING

### Parallel Merging Paradigm

When merging regions in parallel, in order to obtain a correct result, it is necessary that the merge sequence be ordered. Sequential region growing algorithms are most often based on a "first merge" paradigm. In this approach pixels are scanned one at a time, left to right, top to bottom, and combined so long as they meet the homogeneity requirements. The "best merge" paradigm requires that regions only merge with the neighboring region that best satisfies the homogeneity requirements. This not only imposes an ordering to the merge sequence, but tends to yield better results by minimizing the increase in range with each merge. The best merge paradigm is based on the following rules:

(1) Each region can only merge with one other region at a time; that being the neighbor which best satisfies the homogeneity criteria.

(2) A tie is broken, arbitrarily, by selecting the neighbor with the larger ID.

(3) A merge choice must be mutual for two regions to merge.

The parallel region merging paradigm can be modeled using an undirected graph.

> *Let G = (V,E) be an undirected graph with weighted edges. The vertices, V, of the graph correspond to the regions in the image.*

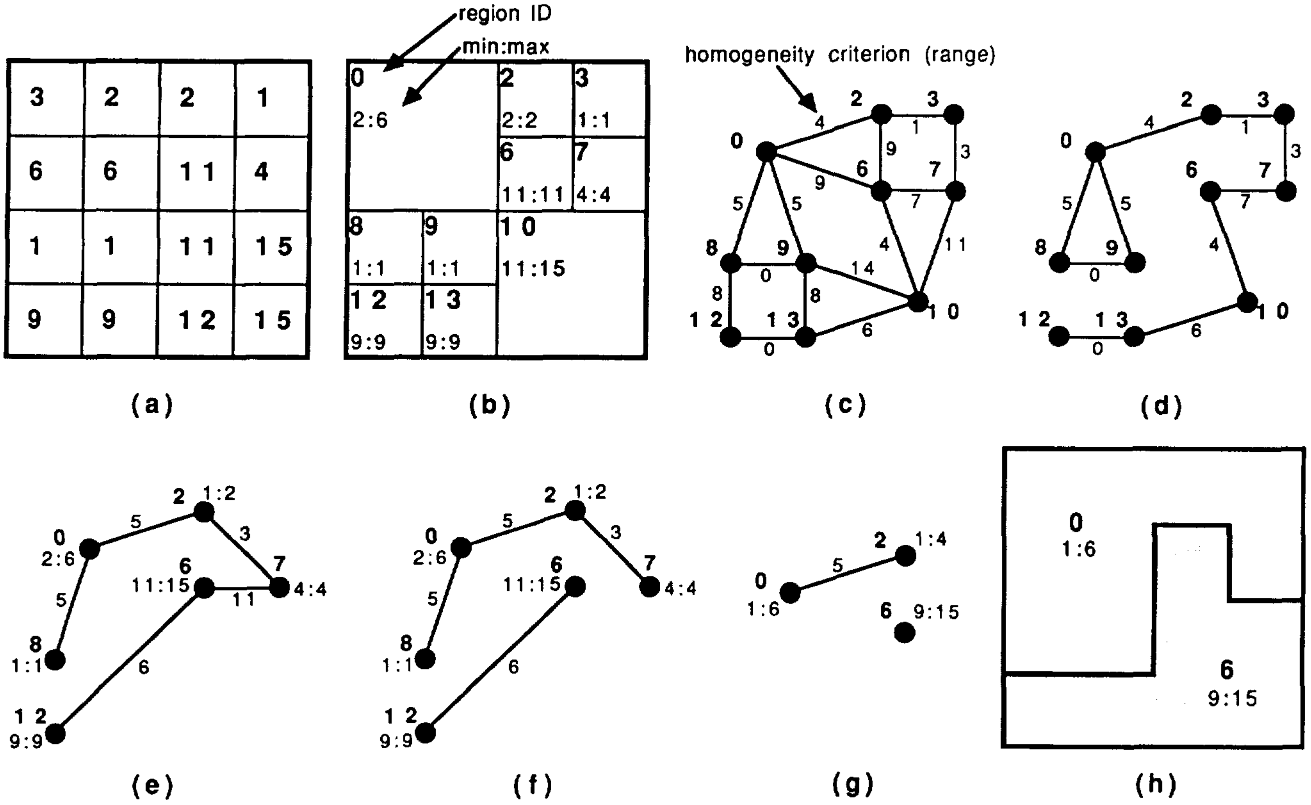


**Figure 4.** Parallel merging graph model. In the example shown the homogeneity range threshold has been set at 6. Pixel values are given in (a), and the result of the split phase is represented in (b). Edges exceeding the allowable range are deleted from the graph. Merges are performed by combining vertices that share an edge that is of lowest value for both vertices. After combining the vertices all edges touching the newly formed vertices are updated with the new range values. Merging terminates when there are no edges left in the graph. The result is shown in (h).

*The set of edges, E, is comprised of the edges (v,w) such that the regions corresponding to vertices v and w share a common boundary. The edge weight, given by*

$$e_{v,w} = h(v,w),$$

*equals the value of the homogeneity criterion evaluated for the regions represented by v and w.*

Using the model described above, the process of merging regions in parallel is performed as follows:

*For all edges E, in graph G, merge those vertices, v and w, for which $e_{v,w}$ is the edge of minimum weight for both vertices v and w. For vertices with more than one edge having the same minimum weight, the edge connecting with the vertice of highest value is selected. Only edges weighted within the homogeneity threshold are considered.*

*Two vertices, v and w, are merged by deleting edge (v,w) and relabeling all edges (i,w), connecting to w, as (i,v). All edges connected to v must then be updated with the new values for $e_{i,v}$.*

*The process continues until there are no edges weighted below the homogeneity threshold remaining in G.*

The parallel merge process is illustrated in Figure 4. In this example we make use of the range homogeneity criterion with a threshold, $T = 7$. Figure 4(a) gives the pixel values of a 4x4 image. The result of the split phase of a split and merge region growing process is presented in Figure 4(b). At this point each region is represented by a unique identification number (ID) and the values of the minimum and maximum pixels within the region. Using this initial partitioning, the regions are represented in the form of a graph (Fig 4(c)).

In the case of the range homogeneity criterion, the homogeneity value $h(v,w)$ for two regions can never decrease. Hence, edges weighted with a value greater than the allowed threshold can be deleted from the graph. A new region assumes the smaller region ID of the two regions being merged.

In a single merge step multiple region pairs can merge without conflicting with each other (Fig 4(d)). This illustrates the degree of parallelism inherent in the merging paradigm. After each merge step, once all edge weights have been updated (Fig 4(e)), those edges exceeding the criterion threshold can once again be removed (Fig 4(f)). The merge process is terminated when there are no edges left in the graph (Fig 4(h)). The final result is illustrated in Fig 4(i).

## Parallel Region Growing Implementation

A parallel region growing strategy is affected by both the form of the homogeneity criteria and the constraints of the parallel processor architecture. In this section a set of representative local processing functions which can be used to realize a large number of homogeneity criteria are defined. Their implementation on the mesh-connected SIMD architecture, outlined in the previous section, is considered.

**Parallel Region Growing Primitives--** A general set of primitive operations, which can be used on a multi-processor system to implement parallel region growing algorithms, is described in this section. The strategy for mapping image points onto processors in a multi-processor system may vary from one architecture to another. Conceptually, a region is characterized by the values and spatial locations of its individual points, as well as the properties of all individual points combined. Therefore, a suitable representation of a region would be: a unique region ID number assigned to all points in the region in conjunction with a region descriptor that contains all relevant global information needed to describe the region. Figure 5(a) illustrates the partition of an image, where each region in the partition is described using the representation suggested above. We assume that the region descriptor information is located within a given processor. However, there is no guarantee or constraint that confines all image points within a region to a single processor.

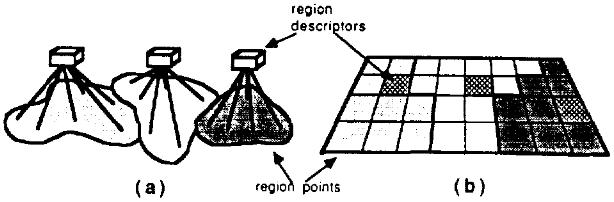


**Figure 5.** Region representation. (a) A generalized representation of a region consists of a spatial distribution of the region points relative to their locations in the image, along with a corresponding region descriptor for each region. (b) A region representation on a mesh of processing elements consists of each region pixel mapped to its own PE and a designated PE to serve as the region's descriptor.

The primitive operations required by a parallel region growing algorithm are as follows. First, in order to adequately characterize a region, a method to accumulate information from all region points to the region descriptor is needed. Second, in order to update region points concerning changes in status (i.e. region ID due to merging), information held at the descriptor must be distributed out to all region points, particularly boundary points. Third, in order to interact with all neighboring regions in parallel, a method for exchanging information between all neighboring regions sharing a common boundary is needed. These tasks are fundamental in a parallel region growing process and can be accomplished using the following set of primitives.

(1) **Reduction primitive:** reduces information from many points in a region to a single value and records it at the region descriptor. Typical reduction functions used are minimum, maximum, and sum.

(2) **Distribution primitive:** distributes a value from the region descriptor to all points in the region.

(3) **Exchange primitive:** exchanges information between points across a common boundary shared by neighboring regions.

Having defined a set of general parallel region growing primitives, we now consider their implementation on a mesh-connected SIMD architecture. When processing images on a highly parallel array of processors each pixel in the image can be directly mapped to a processor in the array. Initially each processor will only have information about a pixel's value and its spatial location in the image (this is straight forward with this architecture since a pixel's location in the processor array is a direct spatial mapping of its location in the image). To adhere to the region representation outlined above, as pixels are joined to form regions, a given PE within each region is designated to be the region descriptor (Fig 5(b)). What we require is a systematic way of choosing the descriptor PE and a method to efficiently implement the primitives described above. The nature of the primitives, particularly the reduction and distribution functions, suggest that a tree structure could be an efficient mechanism to incorporate into the region representation outlined thus far. This reasoning helped us arrive at the embedded tree data structure described in the following section.

**The Embedded Tree Data Structure--** An embedded tree structure is used to represent a region in an image. The tree structure is embedded within a region's boundaries on the PE array. Each region PE is assigned to a vertex in that region's embedded tree. A natural choice of PE to be designated as region descriptor is the PE located at the corresponding location of the tree root.

A **tree** is defined in [9] as a directed acyclic graph containing exactly one vertex, called the root, which no edges enter. Every other vertex has exactly one entering edge and there is a path (which is easily shown unique) from the root to each vertex. A sample tree structure is presented in Figure 6(a). The arrows indicate the edge directions. Furthermore, if (v,w) is a directed edge of the tree, then v is called the *parent* of w and w the *child* of v. A vertex with no children is called a *leaf*. The *depth of a tree* is the length of the longest path from the root to a leaf. For example, in Figure 6, vertex a is the parent of vertices b, c, and d, and conversely vertices b, c, and d are children of vertex a. Vertices e, f, i, j, k, and l are all leaves, and the depth of the tree is 3.

An **embedded tree** is a tree confined within a specified boundary, and linked in such a way that all points within the boundary correspond to a unique vertex in the tree. An example of an embedded tree is shown in Figure 6(b). Since each point in the image, or a region for that matter, is mapped to its own processor, we can think of the processors themselves as the tree vertices. Furthermore, edges of the tree correspond to interconnections between adjacent processors.

We chose the tree data structure because of its fan-in, fan-out nature and because it is easily extended to cover a two-dimensional arbitrary shaped region. The tree's fan-in and fan-out qualities make it very efficient for implementing the reduction and distribution primitives. By using a tree to link together the various pixels in a region, where the pixels form the vertices of the tree and the region descriptor is located at the tree root, it becomes possible to broadcast information from the region descriptors to other region pixels, or conversely, to accumulate region information at the descriptor PE's, within all regions simultaneously. What is important, however, is the way the tree is constructed. The primitives' efficiency is directly dependent on the depth of the tree. When processing regions of an image in parallel, the complexity of the functions employing the tree structure is bounded by the maximum tree depth of all regions in the image. Therefore, it is essential that the trees be constructed in such a way so as to minimize the tree depth of all regions.

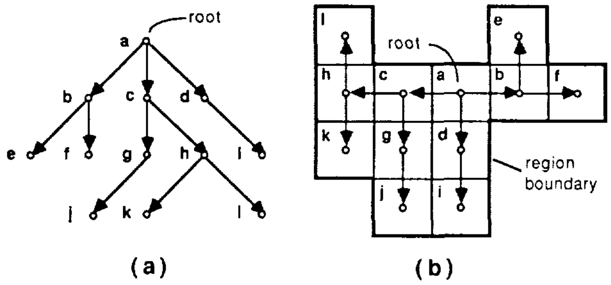


**Figure 6.** (a) Tree data structure. (b) Embedded tree data structure.

**Embedded Tree Generation--** An iterative shrinking algorithm is used to create an efficient tree structure. This technique systematically removes pixels around a region's border and creates pointers linking the removed pixels to still existing neighboring pixels. The shrinking process continues until only a single pixel of each region remains. This remaining pixel is defined as the tree root or region descriptor. This strategy will centralize a tree root within a region.

The shrinking algorithm makes use of the near-neighbor connection scheme existing in the mesh. A pixel can be removed in the shrinking process only if certain conditions regarding its neighbors are true. This requires that each PE investigate its neighbors' values in order to make a decision on whether or not to "shrink". Initially all PE's in the array contain a binary one. As pixels are removed in the shrinking process these ones are changed to zeroes. Two things about a PE's neighbors need be known in order to determine whether or not the conditions required for shrinking exist, (1) their region ID number and (2) their binary value. The set of conditions shown in Figure 7 pertain to the shrinking of the central pixel into its neighbor to the south. Shrinking may occur into any one of a pixels four near-neighbors. To determine whether a pixel may shrink to the west, north, or east, the conditions need to be adjusted accordingly. In one iteration of the shrinking algorithm the conditions are checked for each of the four neighbor directions

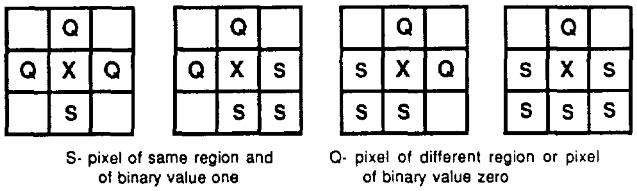

Q
Q X Q
S
Q
Q X S
S S
Q
S X Q
S S
Q
S X S
S S S
S- pixel of same region and of binary value one
Q- pixel of different region or pixel of binary value zero

**Figure 7.** Shrinking Algorithm. In order for pixel X to shrink to the south one of the near-neighbor arrangements shown must exist.

in turn. Should a pixel satisfy the conditions in any one of these directions, its binary value is set to zero and a set of pointers are created. Each processor contains a pointer set which is encoded in four bits; one bit for each possible direction. The resulting embedded tree is represented by a double-linked set of *parent* and *child* pointers. For example, if *parent-pointer[d]* is true for PE(i,j) then the pixel at (i,j) is linked to its parent in the direction corresponding to d. We can ascertain that at the parent node, *child-pointer[d]* will be true for the value of d corresponding to the opposite direction. The region PE with no *parent-pointers* is the root of the region tree, and those PE's with no *child-pointers* are designated as the tree leaves. The result of applying the tree generating algorithm is illustrated in Figure 8. The region pixels are linked together by the *parent-pointers*, shown in 8(c), created during the shrinking process. A corresponding set of *child-pointers* exist.

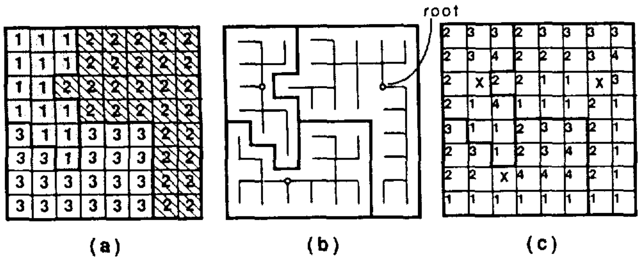


**Figure 8.** Embedded tree data structure. (a) Regions are represented by a unique ID number. (b) Embedded trees are created by the shrinking algorithm. (c) Parent-pointers are assigned according to the following encoding: 1=N, 2=E, 3=S, 4=W.

**Regions with Holes--** The algorithm just described cannot generate embedded trees for non-simply connected regions (i.e. regions with holes). Rather than converging to a single point, the algorithm converges to a single element wide, ring of region pixels which surround the enclosed region. This ring is similar to a medial axis of the region to which it belongs. The result of applying the shrinking algorithm to a region with a hole is shown in Figure 9(a). Since a root node must be chosen from the remaining pixels contained in the ring, it is most efficient to choose that pixel with maximum distance to the region boundary. This would help to minimize the height of the region's tree. Should two or more pixels possess the same height, the pixel with the highest ID is selected, arbitrarily, to be the region root. The remaining pixels in the ring are then, systematically, linked to each other in the direction of the chosen root. The ring uncoupling is illustrated in Figure 9(b). Pixels neighboring the root are uncoupled from the ring and linked to the root first. Their neighbors in the ring are removed next, and so on, until all ring pixels have a linked path to the root. This solution will also work for regions with more than one hole. Unfortunately, however, the presence of holes can lead to rather inefficient (deep) tree structures.

## General Parallel Region Growing Algorithm

Region growing, as mentioned earlier, is achieved by merging regions to form larger regions that conform to a pre-defined homogeneity criterion. Using the embedded tree structure, two regions can be merged together by discarding the embedded trees of the individual regions and regenerating a single more efficient tree for the combined regions. Merging is performed in iterations and terminates when no more regions are able to merge without violating the homogeneity criterion. The parallel merging is accomplished by regions exchanging information with their neighbors and accumulating this information at the region descriptors for evaluation. Regions can then select a merge partner based on the homogeneity criterion being used. Following the parallel region growing paradigm proposed earlier, although more than one neighbor may be suitable for merging, merging is performed in a hierarchical fashion giving preference to those regions that best satisfy the criterion by allowing them to merge first.

The order of merging and a solution to the merge contention problem are important aspects in the parallel approach. Both aspects are addressed by the "best merge" paradigm which states that it is not possible for a region to merge with more than one other region during a single merging iteration. The resulting region could otherwise be in violation of the homogeneity requirements. Two common merge contention situations are illustrated in Figure 10. Furthermore, merging is not performed unless both regions select each other as merge partners. A region that was unable to merge during a given iteration because its selection was not mutual, may succeed in a subsequent iteration.

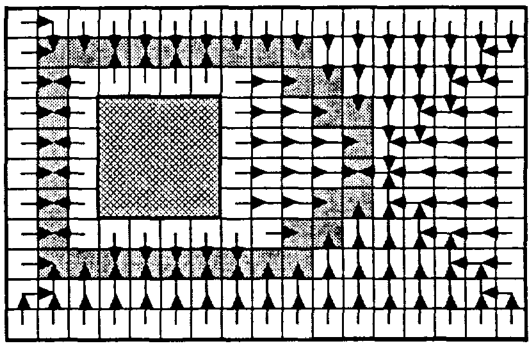


**Figure 9(a).** The result of the shrinking algorithm performed on a region with a hole is a ring in the shape of a medial axis.

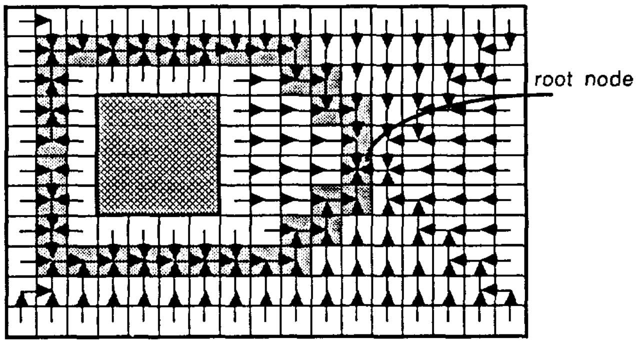


**Figure 9(b).** The ring is uncoupled in such a way that the region root is located at the point on the ring that is furthest from the boundary.

## ALGORITHM COMPLEXITY

The region growing algorithm complexity depends upon both the implementation of the region growing primitives and the number of iterations required to arrive at the desired result. The cost of implementing a region growing primitive, Embed, Dist, or a reduction, is $O(d)$ where d is the maximum distance across any region. The Exchange primitive is implemented in constant time.

The speed of the primitives also depends on the region shape and topology. Convex blob-like regions containing $O(d^2)$ elements are processed in $O(d)$ time. Non-simply connected regions require slightly more time than simple regions due to the additional time taken to compute the embedded tree. The complexity for these regions is still $O(d)$ and, in general, the cost increase will be less than 2. Strange concave shapes, such as spirals, require the most time. Here d is the longest path between two pixels in a region which does not go outside the region. However, these regions rarely occur in practice. The ideal cost complexity is *O(log d)* but, this cannot be achieved because of routing limitations on a mesh-connected architecture.

The number of iterations also depends upon the region dimension d. In an ideal situation, merging of a two-dimensional region can be achieved in *O(log d)* parallel merges starting with each pixel as a region; however, if only one merge occurs during each iteration then the worst case complexity is *O(d)*.

There are two heuristics which can greatly reduce the number of merge iterations. The first is the split phase which was mentioned earlier. The split algorithm only requires *O(log d)* computations for the region growing primitives (although routing is still *O(d)*) and *O(log d)* iterations to complete. Second, in many practical applications there is a "background" behind a collection of objects. This background, being a single region, could dominate the algorithm cost. However, in most cases it is possible to remove the background from consideration by a fairly simple thresholding technique. The algorithm cost then depends on the span of the largest object which remains.

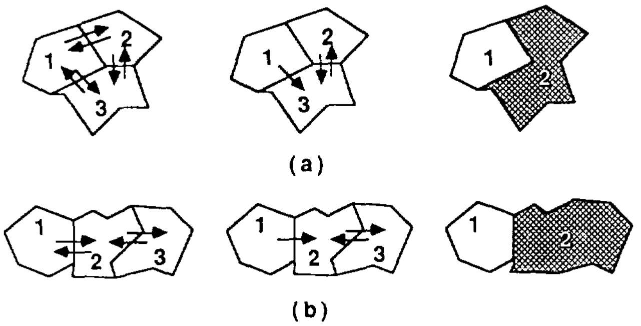


**Figure 10.** Merging order. Contention for merging is indicated by the arrows in the diagram. (a) and (b) represent two different situations that may develop. In both cases regions first choose a neighbor based on the homogeneity criterion and resolve ties by selecting the neighbor with the higher ID. Two regions may only merge if the choice is mutual.

## PARALLEL SPLIT AND MERGE ON THE MASSIVELY PARALLEL PROCESSOR

A parallel split and merge algorithm has been implemented on the Massively Parallel Processor (MPP), using the parallel region growing technique outlined in the previous sections.

### The Parallel Split and Merge Algorithm

In order to implement the parallel region growing algorithm, a procedure for generating the embedded tree data structures and a set of primitives closely resembling those outlined earlier, were designed. These primitives are as follows:

**EMBED(ridm, pptrs, cptrs, maxtd):**

> This procedure receives as input the region ID matrix, **ridm**, and performs the region shrinking and embedded tree generation. It returns a matrix of parent and child pointers (**pptrs,cptrs**), and a value proportional to the maximum tree depth, **maxtd**. Region ID zero is reserved to mask out regions that need not be processed.

**MIN(pptrs, maxtd, pixval, rootval):**

**MAX(pptrs, maxtd, pixval, rootval):**

> These procedures receive as input the tree parent pointers, **pptrs**, the maximum tree depth, **maxtd**, and the pixel values, **pixval**, and perform reduction functions. They deliver the region minimum and maximum, respectively, to the region roots (**rootval**).

**DIST(cptrs, maxtd, rootval, pixval):**

> This procedure receives as input the tree child pointers (**cptrs**), the maximum tree depth (**maxtd**), and the values contained at the region roots (**rootval**) and distributes them throughout the regions (**pixval**).

**EXCHANGE(ridm, pixval, direction, neighbval):**

> This procedure receives as input the region ID matrix (**ridm**) and the pixel values (**pixval**) to be exchanged across the region border. The parameter **direction** is needed to discriminate between neighboring regions bordering in different directions. The pixel values of neighboring regions are returned in **neighbval**.

Since all regions are square in the split phase of the algorithm, it is not necessary to create embedded trees to represent the regions. On the MPP the regions in the split phase will have dimensions mxm where m is a power of 2. Hence, the upper-left corner PE of each region can be designated as the region root. Then, by initially treating each pixel in the image as an independent region, an iterative merging technique can be used to create larger square regions comprised of four subregions (one in each quadrant). This has the same effect as a top-down split but is more efficient. Using the homogeneity criterion suggested earlier, eqn (1), all information needed to describe a region can be accumulated at the root during the region growing process. In effect, a larger region's properties (e.g. minimum and maximum values) are obtained from the properties of its four subregions. There is no need to re-evaluate the minimum or maximum values of all pixels in a region, but simply to calculate the minimum and maximum of the values held at the roots of the four subregions. A larger region is created by merging its four quadrants only if all four quadrants are homogeneous regions and the combined properties of the subregions satisfy the homogeneity criterion. Hence, reduction operations only require a constant number of steps to perform and the distribution operation can be performed in *O(log(m))* steps. However, since the MPP only has near-neighbor connections the cost of routing is still *O(m)*. This could be reduced if a more complex interconnection network between PE's were available (e.g. *O(log(m))* given a hypercube interconnection). In any case, this is much more efficient than using the embedded trees, and only requires a slight modification of the parallel primitives described above. The modified primitives for square regions are described below.

**SQRMIN(iteration,pixval,rootval):**

**SQRMAX(iteration,pixval,rootval):**

> These procedures receive as input the splitting iteration number to determine the dimensions of the regions being processed and the locations of the region roots. A region's minimum and maximum are computed by shifting the minimum and maximum values of its subregions to the region root. The results are returned in **rootval**.

**SQRDIST(dimensions,rootval,pixval):**

> The parameter **dimensions** is an array containing the dimensions of each region. This information is used to broadcast the root value to all PE's within the region. The result is returned in **pixval**.

Both phases of the split and merge algorithm are described below. Each phase of the algorithm is first outlined by dividing it into a series of steps. The corresponding pseudo-code for each phase is then broken down into the same steps. The **WHERE (condition)** construct is a mask that only enables those PE's for which **condition** is true. The **ANY (condition)** construct returns a value of true if **condition** holds true for any PE in the system. The homogeneity criterion suggested earlier, eqn.(1), is used.

The Split phase algorithm

1. Since initially all pixels are treated as independent regions, the region minimums and maximums are set to equal the pixel values and the split iteration is set to one.
2. Each region's maximum and minimum values are computed.
3. The region ranges are calculated at the roots.
4. If any regions display a range within the allowed threshold, the new region dimensions are set at the roots.
5. The iteration value is incremented and, unless either the iteration value exceeds log(n) (n is the image dimension) or no merges occurred during the last iteration, control is returned to step 2.

6. The regions' root PE ID's are distributed to all region PE's using the region dimensions contained at the region roots.

The Split phase pseudo-code

```
     /* Variable declarations */
     idm, pid, rootnode : PARALLEL ARRAY [1..128,1..128]
     OF [0..16383];
     pixval, regmin, regmax, T, range : PARALLEL
     ARRAY [1..128,1..128] OF [0..255];
     dimension : PARALLEL ARRAY [1..128,1..128] OF [1..128];
     iteration : [1..7];
     merge : BOOLEAN;

1.   /* Initializations */
     regmin ← pixval;
     regmax ← pixval;
     iteration ← 1;

2.   REPEAT
       /* Compute the region minima and maxima. */
       SQRMIN(iteration, regmin, regmin);
       SQRMAX(iteration, regmax, regmax);

3.     /* Calculate the new region ranges. */
       rootnode ← f(pid,iteration);
       WHERE rootnode
         range ← regmax- regmin;

4.     /* Calculate new dimension for homogeneous regions. */
       merge ← ANY(range < T);
       IF (merge) {
         WHERE ((range < T) AND rootnode)
           dimension ← $2^{iteration}$;
       }

5.     /* Terminate when either no merges occurred in the previous
       iteration or the entire image has been spanned. */
       iteration ← iteration + 1;
     UNTIL ((iteration > log(n)) OR NOT merge);

6.   /* Broadcast the region ID's to all PE's contained within
     their boundaries. */
     SQRDIST(dimension, pid, idm);
```

The Merge phase algorithm

1. Embedded trees for all regions are constructed using the region ID's.
2. Minimum and maximum values and region ID's are exchanged between neighboring regions, to determine the range of combined regions, and possible merges.
3. Lowest merge ranges are accumulated at the region roots and distributed to all region PE's.
4. In case more than one neighboring region yields the same "lowest merge range", the region with the higher ID is selected.
5. Choices of merge selection are exchanged with neighbors. If two regions select each other, both regions merge by assuming the smaller ID of the two.
6. Unless no merges occurred during the last iteration, control returns to step 1.

The Merge phase pseudo-code

```
/*
Variable declarations.
*/
ridm, regmins, regmaxs, mergeneighbid, mergechoices, mergechoice,
neighbmergechoice : PARALLEL ARRAY [1..128,1..128]
OF [0..16383];
pixval, mintoroot, maxtoroot, mergerange, neighbmin, neighbmax,
newmergerange, bestrangeval : PARALLEL ARRAY [1..128,1..128]
OF [0..255];
cptr, pptr : PARALLEL ARRAY [0..5,1..128,1..128] OF BOOLEAN;
merger : PARALLEL ARRAY [1..128,1..128] OF BOOLEAN;
maxtd, i : INTEGER;

1.   /* Create embedded trees for all regions and calculate
     the region minima and maxima. */
     EMBED(ridm, pptrs, cptrs, maxtd);
     MIN(pptrs, maxtd, pixval, mintoroot);
     MAX(pptrs, maxtd, pixval, maxtoroot);
     DIST(cptrs, maxtd, mintoroot, regmins);
     DIST(cptrs, maxtd, maxtoroot, regmaxs);

2.   /* Initialize the merge range to the threshold value. */
     mergerange ← T;

     /* Exchange ID, minima, and maxima information
     with all neighbors. */
     FOR (i ← 1 TO 4) {
       EXCHANGE(ridm, regmin, i, neigbmin);
       EXCHANGE(ridm, regmax, i, neigbmax);
       EXCHANGE(ridm, ridm, i, neighbid);
       /* Determine the homogeneity range value of each neighbor. */
       WHERE (neighbmax < regmax)
         neighbmax ← regmax;
       WHERE (neighbmin > regmin)
         neighbmin ← regmin;
       newmergerange ← neighbmax-neighbmin;
       /* Record the range and ID of the region
       yielding the best range. */
       WHERE ((newmergerange < mergerange) AND
                 (ridm <> neighbid)){
         mergerange ← newmergrange;
         mergeneighbid ← neighbid
       };
     };

3.   /* Accumulate the best merge range at the root. */
     MIN(pptrs, maxtd, mergerange, mintoroot);
     DIST(cptrs, maxtd, mintoroot, bestrangeval);

4.   /* Select the neighbor yielding the best range value and arbitrate
     ties by selecting the neighbor with the largest ID. */
     WHERE (bestrangeval = mergerange)
       mergechoices ← mergeneighbid;
     MAX(pptrs, maxtd, mergechoices, maxtoroot);
     DIST(cptrs, maxtd, maxtoroot, mergechoice);

5.   /* Exchange merge choice information with neighbors.
     Choices must be mutual for a merge to occur. */
     FOR (i ← 1 TO 4) {
       EXCHANGE(ridm, mergechoice, i, neighbmergechoice);
       EXCHANGE(ridm, ridm, i, neighbid);
       WHERE ((ridm = neighbmergechoice) AND
                 (mergechoice = neighbid))
         merger ← true;
       haveamerge ← ANY(merger);
       /* For merging regions, arbitrarily, select the smaller
       of both ID's as the new region's ID. */
       IF (haveamerge) {
         WHERE (merger AND (neighbid < ridm))
           ridm ← neighbid;
         MIN(pptrs, maxtd, ridm, mintoroot);
         DIST(cptrs, maxtd, mintoroot, ridm);
       };
     };
```

## Results

A high-level language (Parallel Pascal [12]) implementation of the region growing primitives was run on the MPP. A series of timing tests were conducted using images of square NxN regions with the set of dimensions: N = 8, 16, 32, and 64 (Table 1). These test sets were representative of blob-like regions with d=N.

**Table 1.** Measured Timing Results for Primitives.

| Dimension | Time (ms) | | | |
|---|---|---|---|---|
| N | embed | min/max | dist | exchange |
| 8 | 12.5207 | 1.9675 | 1.5291 | 0.0660 |
| 16 | 22.2983 | 3.5137 | 2.7260 | 0.0660 |
| 32 | 41.8582 | 6.6043 | 5.1177 | 0.0660 |
| 64 | 80.9739 | 12.7876 | 9.9032 | 0.0660 |

The following set of timing expressions, given in milliseconds, were derived from these results.

$$T_{embed}(N) = 1.222N + 2.741 \quad (2a)$$

$$T_{min/max}(N) = 0.1932N + 0.4210 \quad (2b)$$

$$T_{dist}(N) = 0.1495N + 0.3322 \quad (2c)$$

$$T_{exchange}(N) = 0.0660 \quad (2d)$$

The measured values do not deviate from the above equations by more than 2 percent.

An estimated performance analysis was conducted assuming an optimal machine-level encoding of the primitives. The clock cycle time for the MPP is 100 nanoseconds. Furthermore, it was assumed that loop set-up in the host could be overlapped with array computations. This is possible on the MPP since a FIFO buffer exists between the host and the PE array which allows the host to perform serial calculations while the array is busy processing instructions in the buffer. The analysis yielded the following expressions.

$$\hat{T}_{embed}(N) = 0.1570N + 0.0197 \quad (3a)$$

$$\hat{T}_{min/max(N)} = 0.0168N + 0.0034 \quad (3b)$$

$$\hat{T}_{dist}(N) = 0.0104N + 0.0034 \quad (3c)$$

$$\hat{T}_{exchange}(N) = 0.0358 \quad (3d)$$

The estimated results range between 2 to 15 times faster than the measured results. This is probably due to two main reasons. First, the Parallel Pascal code generator produces inefficient code since it currently does not perform any code optimizations; second, the host may not be able to generate instructions fast enough to keep the processor array busy; especially for boolean data operations. (Conditional branches which are dependent on processor array values cause the FIFO buffer to empty, however no such branches occur in the implementation of the primitives).

A similar comparison analysis was conducted for the split and merge phases of the algorithm. The measured results for the blob-like regions are summarized in Table 2.

**Table 2.** Measured Timing Results for Split and Merge Phases.

| Dimension | Time (ms) | |
|---|---|---|
| N | Split | Merge |
| 8 | 2.7276 | 38.1122 |
| 16 | 3.4329 | 67.1308 |
| 32 | 4.2128 | 125.1579 |
| 64 | 5.2824 | 241.2121 |

These results yielded the following expressions.

$$T_{split}(N) = 0.01518N + 0.5839\log(N) + 0.8077 \quad (4a)$$

$$T_{merge}(N) = 3.627N + 9.104 \quad (4b)$$

The first term in the split phase expression represents the cost due to the shift operations needed to route information through an NxN mesh connected region. This could be reduced to *O(logN)* given a hypercube interconnection scheme. The second term in the expression is related to the number of arithmetic operations performed to create an NxN region. The third term includes the overhead operations that are independent of N. The merge phase results are primarily a product of the primitive operations and therefore yield an expression of *O(N)*. The estimated performance using an optimal encoding is as follows.

$$\hat{T}_{split}(N) = 0.0045N + 0.100\log(N) + 0.052 \quad (5a)$$

$$\hat{T}_{merge}(N) = 0.3746N + 0.3737 \quad (5b)$$

For the split phase the estimated performance was three to five times faster than the measured results. Since the merge phase of the algorithm is highly dependent on the parallel primitive implementation, we could predict that the estimated performance will be approximately ten times faster than measured results. This is in fact the case as demonstrated by the expression above.

## CONCLUSION

A technique for region growing on a highly parallel SIMD computer has been described. This technique is based on a "best merge" paradigm which imposes an ordering to the parallel merge sequence. Three fundamental primitives for region growing have been defined: reduction, distribution, and neighbor exchange. Furthermore, it has been shown that the split algorithm on the target architecture is efficiently implemented by region growing using special primitives for processing square regions.

A simple homogeneity criteria, pixel range, has been used to demonstrate the basic parallel techniques. However, arbitrary complex homogeneity functions can be computed with this scheme. Additional reduction functions such as PRODUCT and MEAN may be implemented with a similar efficiency to the primitives MAX and MIN. An embedded tree structure has been introduced to implement the region growing primitives on a mesh-connected SIMD system.

## REFERENCES

[1] D. H. Ballard and C. M. Brown, *Computer Vision*, Prentice-Hall, Englewood Cliffs, New Jersey, (1982).

[2] A. Rosenfeld and A. Kak, *Digital Picture Processing*, Academic Press, New York, New York, (1982).

[3] S. W. Zucker, "Survey, region growing: childhood and adolescence," *Computer Graphics and Image Processing* 5, pp. 382-399, (1976).

[4] S. L. Horowitz and T. Pavlidis, "Picture segmentation by a directed split-and-merge procedure," *Proceedings 2nd IJCPR*, pp. 424-433, (August 1974).

[5] J. P. Gambotto and O. Monga, "A parallel and hierarchical algorithm for region growing," *Proceedings of the 1985 IEEE Conference on Computer Vision and Pattern Recognition* pp. 649-652 (1985).

[6] A. P. Reeves, "Survey, parallel computer architectures for image processing," *Computer Vision Graphics and Image Processing* 25, 68-88 (1984).

[7] A. P. Reeves, "Highly parallel computer architectures for scientific applications,"

[8] J. Kittler and M. J. B. Duff, *Image Processing System Architectures* , Research Studies Press, Letchworth, England, (1985).

[9] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms* , Addison-Wesley, Reading, Mass., (1974).

[10] A. P. Reeves, "A systematically designed binary array processor," *IEEE Transactions on Computers* C-29, pp 278-287 (1980).

[11] K. E. Batcher, "Design of a massively parallel processor," *IEEE Transactions on Computers* C-29(9) pp.836-840 (September 1981).

[12] A. P. Reeves, "Parallel Pascal: an extended Pascal for parallel computers," *Journal of Parallel and Distributed Computing* 1 pp. 64-80 (1984).

[13] R. W. Taylor, M. Savini, A. P. Reeves, "A fast algorithm for range image segmentation," *1987 IEEE Systems Man and Cybernetics Annual Conference* (October 1987).

# HYPERCUBE ALGORITHMS SUITABLE FOR IMAGE UNDERSTANDING IN UNCERTAIN ENVIRONMENTS

*T.L. Huntsberger, A. Sengupta*

Intelligent Systems Laboratory
Department of Computer Science
University of South Carolina
Columbia, SC 29208

## *ABSTRACT*

Computer vision in a dynamic environment needs to be fast and able to tolerate incomplete or uncertain intermediate results. An appropriately chosen representation coupled with a parallel architecture addresses both concerns. The wide range of numerical and symbolic processing needed for robust computer vision can only be achieved through a blend of SIMD and MIMD processing techniques. The 1024 element hypercube architecture manufactured by NCUBE of Beaverton, Oregon has these capabilities, and was chosen as the test-bed hardware for development of highly parallel computer vision algorithms.

This paper presents and analyzes parallel algorithms for color image segmentation and edge detection. These algorithms are part of a recently developed computer vision system which uses multiple valued logic to represent uncertainty in the imaging process and in intermediate results [HUNT86]. Algorithms for the extraction of three dimensional properties of objects using dynamic scene analysis techniques within the same computer vision system framework [HUNT87, HUNT88] are briefly examined. Results from experimental studies using a 1024 element hypercube implementation of the algorithms as applied to a series of natural scenes will also be reported.

## INTRODUCTION

Several supercomputers such as the NEC SX-2, the Cray Research CRAY-2, and the ETA-10 offer the possibility of extremely high speed execution of various computer vision algorithms. However, the synchronization of numerical processing and symbolic manipulations of higher level vision information has been proven to be a difficult problem. The massively parallel architecture of the connection machine addresses some of the underlying problems with processing for image understanding [BALL83, HARR86]. The large number of processors needed for such a machine for computer vision would dictate a possibly prohibitive cost. A modular pipeline architecture such as the PIPE design allows computer vision tasks to be partitioned [KENT85].

The low level operations of computer vision such as edge detection and component labeling are efficiently accomplished on parallel arrays of processors such as the ZMOB [BANE81, KUSH81, RIEG81], the homogeneous multiprocessor [DIMO81, RAMA86] and systolic arrays [KUNG83] among others. Reeves gives a review of the current designs for parallel architectures for image processing [REEV84]. The development of reconfigurable architectures like the PASM design [SIEG81] has led to more flexibility in the range of tasks possible on a given architecture of this type. Since the nature of image understanding is more abstract at higher levels, vastly different tools are needed to perform higher level operations such as object identification. Putting more intelligence into the operating system for control in image understanding tasks was recently suggested in [DELP85].

The large bandwidths (100 MFLOPS) typically needed for feature extraction algorithms has severely limited response times in most vision systems. Many architectures are capable of local image operations, but become very inefficient for higher order symbolic processing. This is due not only to the architecture design itself, but also to the utilization of algorithms and processing techniques not optimized for parallelism. Recent results for a number of both low and high level vision algorithms implemented on a hypercube architecture indicate the flexibility inherent in the design [BOND88, JONE88, WILL88].

## IMAGE SEGMENTATION ALGORITHM

An iterative algorithm which performs clustering in an image color space is used for image segmentation [HUNT85a]. This clustering in color space is done with the fuzzy c-means algorithm generalized by Bezdek [BEZD81]. The performance of this algorithm on a Single Instruction Single Data (SISD) machine has been disappointing, with a typical runtime of from hundreds of minutes to 40 plus hours depending on the hardware used. A recently developed integer version of the algorithm coupled with lookup tables for the distance metric and exponentiation gives an order of magnitude increase in performance on a SISD machine, with the tradeoff of increased storage requirements [CANN86]. Upon investigation, it was found that certain portions of the algorithm were highly parallel in nature and that a significant performance benefit can be achieved by exploiting this inherent parallelism [HUNT85b].

The clustering procedure has two phases: cluster center generation and membership value determination from these cluster centers. This means that there is an inherent sequential limitation to the algorithm, since each phase relies on the output of the other. Expressions for the cluster centers and membership values are given below:

$$v_i = \frac{\sum_{k=1}^{n} (\mu_{ik})^m x_k}{\sum_{k=1}^{n} (\mu_{ik})^m}, \tag{1a}$$

for the cluster center $v_i$ and

$$\mu_{ik} = \frac{1}{\left[ \sum_{j=1}^{c} \left[ \frac{d_{ik}}{d_{jk}} \right]^{\frac{2}{(m-1)}} \right]} \tag{1b}$$

for the membership value $\mu_{ik}$, where $c$ is the number of clusters, $d_{ik}$ is a distance metric, $m$ is a weighting exponent and $n$ is the total number of points being analyzed. The values used in the segmentation phase of the computer vision system are $n = 9$, $m = 2.0$ and $c = 2$ with a Euclidean distance metric in RGB color space.

The scheduling table for the parallel implementation of the cluster generation phase of the segmentation algorithm indicates that generation of cluster centers can be accomplished in 13 machine cycles using a 16 node hypercube configuration, as opposed to 177 operations on a sequential machine. Partitioning the tasks along these lines differs from the multistage pipeline, which was the original application-specific design [HUNT85b]. Analysis of the membership value determination portion of the algorithm indicates that 16 cycles will be required on the same hypercube configuration, as opposed to 212 operations on a sequential architecture.

The implementation of the clustering algorithm discussed above has been designed to only allow interprocessor communication between nearest neighbor nodes in the hypercube network. This will greatly cut down on communication overhead times and should allow accurate

CH2649-2/89/0000/0101$01.00 © 1988 IEEE

reports of running times, since the direct peak interprocessor communication speed on the NCUBE/10 is 180 Megabytes/sec. Each run of the clustering algorithm will typically take 5 iterations to converge. Thus, there are typically about 1945 floating point operations involved in the production of the membership values and cluster centers for each pixel for a sequential analysis, or put another way, 11.7M floating point operations for each frame of size 240X256 full color pixel data. Recent results of parallel algorithm development on the NCUBE/10 have indicated a relatively slow sustained processor floating point performance of 0.14 to 0.26 single precision MFLOPS [GUST88]. Allocation of the full 1024 nodes as 16 element sub-cubes gave a near real-time segmentation rate of 1.2 seconds per frame.

Although the MIMD algorithm just discussed exhibits a possible parallel speed-up of about 82 percent, message buffering for interprocessor communication within the AXIS operating system on the NCUBE/10 hypercube totally negates this speed-up. As observed by the Sandia group, unless overlapped communication cycles can be built into the algorithm, interprocessor transfers typically take about 365 microseconds per four byte transfer [GUST88]. As the operating systems on hypercube architectures evolve, this speed bottleneck will be eliminated with unbuffered nearest neighbor communication protocols included at the algorithm level.

The hypercube architecture is flexible to allow alternate mappings of algorithms. The possible parallel implementations of the segmentation algorithm for the hypercube would be decomposition of the image data with very little interprocessor communication needed, or a 5 stage pipeline. Both of these approaches were investigated, with the expected results. If the image data set is decomposed into windows of size 9 X 6 pixels, with a single window on each of 1024 processors, the segmentation time is reduced to 40 milliseconds per 240 X 256 full color frame. Another version of the data set decomposition technique would assign two processors to each window, with each processor computing a single cluster center and then exchanging the results. Despite the simultaneous calculation, the communication overhead drove the segmentation time up to 67 milliseconds per frame.

The five stage pipeline design allows each iteration of the algorithm to be computed in one stage of the pipe. For five processors arranged in a ring pipeline the number of interprocessor communication cycles is 35 with overlapping and a segmentation rate of 54 milliseconds per frame. This time is about of factor of 1.4 times slower than the image data decomposition method. If a five stage pipeline with two processors in each stage is constructed for the segmentation process, the time increases to 92 milliseconds due to the extra interprocessor communication cycles.

## PARALLEL EDGE DETECTION ALGORITHM

Once the membership values to the color cluster centers are determined, the process of edge detection is done using these membership values. Local homogeneity properties of a color image are evaluated with set operations. An edge operator has been developed that is independent of thresholds and that treats the pixel color characteristics as a vector [HUNT85b]. The information for the location and strength of color edges is resident in the hypercube processors after the image segmentation process. This leads to efficient edge detection for input to higher level processes, such as shape representation [HUNT86], due to the elimination of reloading time for the membership values.

Ambiguities in edge strength and location can arise from many factors. Among these are noise in the sensors and motion blur of region boundaries. Membership values are close to one within a homogeneous color region, and drop close to zero after crossing the boundary of the region. The corresponding change from one to zero will occur in the membership values of the adjacent region. If regions are treated as image subsets, the spatial intersection between these subsets can be defined as an edge.

As such, we defined an edge operator based on local pixel characteristics obtained from the cluster analysis. This operator has the form

$$HOMOG_i = \mu_{ia} - \mu_{ib}, \tag{2}$$

and computes the relative homogeneity of pixel $i$ with respect to its adjacent pixel. Here, $\mu_{ia}$ and $\mu_{ib}$ are the membership values associated with pixel $i$ to the image subsets $a$ and $b$. The location of an interclass edge is the point at which the operator

$$EDGELOC_{jk} = HOMOG_j - HOMOG_k, \tag{3}$$

experiences a zero-crossing, where $j$ and $k$ are labels for two adjacent pixels. Both operators are only defined for two adjacent pixels, where the index $a$ in (2) is the set label corresponding to the maximum membership value for pixel $j$ and index $b$ is the set label corresponding to the maximum membership value in pixel $k$.

From the homogeniety information derived in equation (2), it is possible to express the strength of the edge in terms of a membership value to the set of ideal step edges. This edge strength is defined as

$$\mu_{EDGE:i} = \frac{\left| HOMOG_i - HOMOG_j \right|}{2}, \tag{4}$$

where $i$ and $j$ are two adjacent pixels, and 2 is a normalization factor. A diffuse edge is characterized by values of $\mu_{EDGE:i}$ intermediate between zero and one, being closer to one as the interface between two adjacent color regions becomes more distinguishable. This operator is computed only at the zero crossings of operator (3), i.e. where evidence for an edge exists.

Determination of edge location and strength was only implemented in the data decomposition hypercube mapping scheme used in the segmentation phase of processing. The edge location using equations (2) and (3) and the same size windows as in the segmentation step took 9.7 milliseconds per frame. It is anticipated that this step can be interleaved with the segmentation phase. Determination of edge strength varied depending on the edge frequency found in the previous step. Typical times on our full color natural images ran on the order of 2 to 4 milliseconds per frame.

## DYNAMIC SCENE ANALYSIS

In this section we analyze a possible parallel technique for the determination of the optic flow field in a sequence of images. This technique exploits the link between contour and region deformations that is inherent in the behavior of moving objects as viewed by a monocular observer. The segmentation method mentioned in a previous section is extended to deal with dynamic scenes. We consider a packet of $k$ frames at a time for analysis, $k$ normally being four or five. The clustering analysis is done for the first frame of this $k$ frame sequence. The color cluster centers obtained are used as reference centers for the calculation of the region characteristics for subsequent frames.

These centers allow a connected components analysis to be performed using the technique discussed in Jones [JONE88]. The typical case timing on a 256X256 image was 636 milliseconds on a 6 dimensional 7 MegaHertz NCUBE system. After the connected components calculation is performed, further calculation of principal components can be done using the butterfly accumulation algorithm [JONE88].

Changes in these principal components are used as features for spatiotemporal deformation studies. A series of simultaneous linear equations is derived from the deformation of four of the low order principal components. Sequential versions of these algorithms are discussed in [HUNT87, HUNT88].

## DISCUSSION

The 1024 element hypercube architecture allows the integration of image processing and computer vision modules under a single structure. Studies of the parallel implementation of some recent dynamic scene analysis work is also under way [HUNT87, HUNT88]. The performance improvement over the sequential algorithms just in the segmentation phase of the system indicate that close to real-time three dimensional analysis of moving objects can be accomplished on this type of architecture. We are presently implementing these algorithms on a 1024 element hypercube (NCUBE/10). In addition, parallel algorithms are being developed for the higher level operations such as model base matching [HUNT86].

## ACKNOWLEDGEMENTS

One of the authors (TLH) would like to express his thanks to Judson Jones of the Oak Ridge National Laboratory for his helpful comments and software assistance.

## REFERENCES

[BALL83]
D.H. Ballard, G.E. Hinton and T.J. Sejnowski, "Parallel visual computation," *Nature*, Vol. 306, pp. 21-26, Nov. 1983.

[BEZD81]
J.C. Bezdek, *Pattern Recognition with Fuzzy Objective Function Algorithms*, Plenum Press, New York, 1981.

[BOND88]
A.H. Bond and D. Fashena, "Parallel vision techniques on the hypercube computer," *Proc. Third Conf. Hypercube Concurrent Computers and Applications*, Pasadena, CA, Jan 1988, pp. 1007-1010.

[CANN86]
R.L. Cannon, J.V. Dave and J.C. Bezdek, "Efficient implementation of the fuzzy c-means clustering algorithms," *IEEE Trans. PAMI*, Vol. PAMI-8, pp. 248-255, March 1986.

[DELP85]
E.J. Delp, H.J. Siegel, A. Whinston and L.H. Jamieson, "An intelligent operating system for executing image understanding tasks on a reconfigurable parallel architecture," in *Proc. CAPAIDM*, Miami Beach, FLA, Nov. 18-20 1985, pp. 217-224.

[DIMO81]
N.J. Dimopoulos, "On the structure of the homogeneous multiprocessor," *IEEE Trans. Computers*, Vol. C-34, pp. 141-150, 1981.

[GUST88]
J.L. Gustafson, G.R. Montry and R.E. Benner, "Development of parallel methods for a 1024-processor hypercube," to appear in *SIAM Journ. Scientific and Stat. Computing*, Vol. 9, No. 4, July 1988.

[HARR86]
J.G. Harris and A.M. Flynn, "Object recognition using the connection machine's router," in *Proc. IEEE Conf. Comp. Vis. and Patt. Recogn.*, Miami Beach, FLA, June 22-26, 1986, pp.134-139.

[HUNT85a]
T.L. Huntsberger, C.L. Jacobs and R.L. Cannon, "Iterative fuzzy image segmentation," *Pattern Recognition*, Vol. 18, pp. 131-138, Apr. 1985.

[HUNT85b]
T.L. Huntsberger and W.R. Wood, "FLASH, A parallel architecture for computer vision in uncertain environments," in *Proc. CAPAIDM*, Miami Beach, FLA, Nov. 18-20 1985, pp. 280-283.

[HUNT86]
T.L. Huntsberger, C. Rangarajan and S.N. Jayaramamurthy, "Representation of uncertainty in computer vision using fuzzy sets," *IEEE Trans. Computers, Special Issue on Multiple Valued Logic*, Vol. C-35, pp. 145-156, Feb. 1986.

[HUNT87]
T.L. Huntsberger and S.N. Jayaramamurthy, "Determination of the optic flow field from deformations of region properties," *Patt. Recogn. Letters*, Vol. 6, pp. 169-177, 1987.

[HUNT88]
T.L. Huntsberger and S.N. Jayaramamurthy, "Determination of the optic flow field in the presence of occlusion, *Patt. Recogn. Letters*, in press.

[JONE88]
J.P. Jones, "A concurrent on-board vision system for a mobile robot," *Proc. Third Conf. Hypercube Concurrent Computers and Applications*, Pasadena, CA, Jan 1988, pp. 1022-1032.

[KENT85]
E.W. Kent and S.L. Tanimoto, "Hierarchical cellular logic and the PIPE processor: Structural and functional correspondence," in *Proc. IEEE Workshop Comp. Arch. Patt. Analys. Image Database Management*, Miami Beach, FLA, Nov. 18-20, 1985, pp. 311-319.

[KUNG83]
H.T. Kung and M.S. Lam, "Real time signal processing VI," *Proc. Soc. POIE*, San Diego, CA (Aug. 1983), pp. 143-158.

[KUSH81]
T. Kushner, A.Y. Wu and A. Rosenfeld, "Image processing on ZMOB," in *Proc. IEEE Workshop CAPAIDM*, Hot Springs, Va. (Nov 11-13, 1981), pp. 88-95.

[RAMA86]
D.V. Ramanamurthy, N.J. Dimopoulos, K.F. Li, R.V. Patel and Al-Khalili, "Parallel algorithms for low level vision on the homogeneous multiprocessor," in *Proc. IEEE Conf. CVPR*, Miami Beach, FLA, June 22-26, 1986, pp. 421-423.

[REEV84]
A.P. Reeves, "Parallel computer architectures for image processing," *CVGIP*, Vol. 25, pp. 68-88, Jan. 1984.

[RIEG81]
C. Rieger, "ZMOB: Doing it in parallel," in *Proc. IEEE Workshop CAPAIDM*, Hot Srings, Va., Nov 11-13 1981, pp. 119-124.

[ROSE84]
A. Rosenfeld, "The fuzzy geometry of image subsets," *Patt. Recogn. Letters*, Vol. 2, pp. 311-317, 1984.

[SIEG81]
H.J. Siegel, L.J. Siegel, F.C. Kemmerer, P.T. Mueller, Jr, H.E. Smalley and Smith, S.D., "PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Trans. Computers*, Vol. C-30, pp. 934-947, Dec. 1981.

[WILL88]
M. Willebeek and A.P. Reeves, "Region growing on a hypercube multiprocessor," *Proc. Third Conf. Hypercube Concurrent Computers and Applications*, Pasadena, CA, Jan 1988, pp. 1033-1042.

# Efficient Scan Operators for Bit-Serial Processor Arrays

C. M. Fiduccia and R. M. Mattheyses
General Electric R & D Center
Schenectady, NY 12301

R. E. Stearns
SUNY at Albany
Albany, NY 12222

## Abstract

A fast algorithm is presented for broadcasting a word of length $w$, on an $n \times n$ SIMD array of bit-serial processing elements, in time $O(n + w)$. Data-skewing problems caused by SIMD restrictions are solved by assuming that each processing element contains a *shift register* and an *activity flag* that allows each processing element to conditionally ignore instructions. The broadcasting algorithm is then extended to a fast segmented-scan (prefix) algorithm that runs in time $O(n + (w + t) \log n)$, where $t$ is the time needed to perform the arbitrary, user-defined, operation on which the scan is based. Because of the versatility of scan operations, many algorithms written for more powerful SIMD computers, such as the Connection Machine, can easily be adapted to bit-serial arrays. Slightly less efficient algorithms are also presented for processing elements that lack shift registers.

**Keywords**: Broadcast, parallel prefix, scan, SIMD arrays, bit-serial algorithms, meshes.

## I. Introduction

The Connection Machine is a powerful Single Instruction Multiple Data (SIMD) computer consisting of 64K bit-serial processing elements (PEs) connected by a global router. In addition to providing general interconnections among the PEs, much like a telephone system, the router can be used to implement sorting and various segmented-scan (prefix) operations [Hil, Ble]. Due to the relative efficiency of scans, many Connection Machine algorithms use them heavily in conjunction with some sorting and a minimal number of general permutations. Algorithms written in this style can often be ported directly to bit-serial processor arrays for which efficient scan implementations exist.

In this paper we present a fast parallel algorithm for computing segmented scans on a two-dimensional $n \times n$ SIMD array of bit-serial processors. The algorithm takes $O(n+(w+t)\log n)$ time to perform a scan on $n^2$ data items, where $w$ is the word length and $t$ is the time to perform the binary operation on which the scan is based.

The scan algorithm is adapted from a novel SIMD broadcasting technique that takes $O(n + w)$ time on an $n \times n$ bit-serial array in which each PE has an *activity flag*, that determines whether that PE will execute or ignore the current instruction, plus a *shift register* that is used to solve data-skewing problems. Machines such as the MPP [Bur] and the GE Cross-Omega chip [GE] have both of these facilities. In practice, our bit-serial technique is considerably faster than the straight-forward word-level method, which requires $O(wn)$ time, and has the desirable property that it is essentially independent of word length, for a large array, since $n + w \approx n$.

A slightly less efficient broadcasting technique is also presented for PEs without shift registers. However, the algorithm is sufficiently more complex that it makes a good case for including shift registers in future designs of bit-serial SIMD arrays. Indeed, it can be argued that, in addition to including shift registers, the data paths within a PE can be easily designed so that broadcasting takes place at clock speeds; i.e., in about $n + w$ cycles. Such mesh arrays would then become competitive with machines, such as the DAP [Flan etal], which resort to global busses for broadcasting.

For applications, such as linear programming or Gaussian elimination, in which broadcasting is an essential part of the pivoting operation, our fast bit-serial algorithm speeds up broadcasting by nearly a factor of $w$. This reduces the broadcasting time to a small fraction of the pivoting time, rather than dominating it.

## II. Fast Broadcast

Consider the problem of broadcasting the values in a given row, of a bit-serial SIMD array, to the $m$ rows below it. If the values are words of length $w$, the obvious word-level row-at-a-time algorithm takes $O(wm)$ time. Given the restrictions of the SIMD model, it is not clear that a significantly faster algorithm exists. In this paper we show how the inclusion of a shift register in each PE, together with the exploitation of the bit-serial nature of the array, can be used to reduce this time to $O(m + w)$. If shift registers are not available, we show how it can be done in time $O(m + w \log w)$ using a more complicated algorithm. We note that both of these results give speedups on the order of $w$, when $m$ is sufficiently large. This is quite sig-

CH2649-2/89/0000/0105$01.00 © 1988 IEEE

nificant in practice. For example, consider the problem of performing a pivoting operation on a 512 × 512 array using 32-bit words. If the bit-serial arithmetic in a PE takes time $O(w^2)$, then the broadcast time (using our algorithms) is on the same order as the time to do arithmetic, whereas with the word-level $O(wm)$ algorithm, communication time would completely dominate the pivoting operation.

Implicit in our algorithms is the ability of each PE to conditionally ignore the current instruction. This is done via an *activity flag*. We also assume that each PE has its own shift register, and that the PE's activity flag also controls the shift register. This means that, in an inactive PE, the contents of the shift register do not change. On the other hand, we view the bit-serial mesh as being a global connection network that is *external* to the PEs and is thus influenced only by the global controller. Thus, the movement of bits along the mesh, into and out of the one-bit mesh registers located "under" the PEs, is unconditional. An alternative view is that every PE, active or not, must participate in moving mesh data, although an inactive PE can neither read nor write its mesh register. The major SIMD restriction is that all PEs must use the *same address* whenever a reference is made to a bit in the local memory of a PE. No indexing is allowed within a PE.

Since the broadcasting of each word in the source row is confined to one column, we confine our attention to a single column of the PE array. The bit-level algorithms in the sequel may best be understood by referring to Figure 1, where a typical column of the array is depicted. The basic idea is to exploit the bit-serial nature of the mesh by allowing each sending PE to *stream* its $w$ bits down the bit-serial mesh, while each receiving PE copies the bits as they go by.

Note that there is no word-level description of this bit-streaming process. If we can make it work in a SIMD environment, the broadcast will clearly take only $O(m+w)$ time. The major problem is that a given bit, of the word being broadcast, arrives at different PEs at different times. This causes skewing problems, because all active PEs must use the same memory address to store each incoming bit. In this section, we show how to solve the skewing problem by assuming that each PE has a shift register. We then show how this problem can also be solved, without shift registers, by accepting the fact that bits are first stored in a skewed fashion, and then using an $O(w \log w)$ algorithm to align them.

## A. With a shift register

Assume that each PE has a bit-array $B[0 : w - 1]$ in its memory, serving as a $w$-bit buffer. Initially, a sender has a word in its buffer; eventually, every receiver will also have a copy of this word in its buffer. In addition, each PE has two one-bit flag registers $S$ and $R$ to indicate various conditions. If a PE is a sender then $S = 1$ otherwise $S = 0$. The value of $S$ is assumed to be given as part of the input to the broadcasting routine and is not modified by the routine. The $R$ flag is used to indicate that a PE has received its copy of the word; at that point, $R = 1$. To deal with the SIMD restrictions, we find it useful to also treat each sender as a receiver. Thus, when $R = 1$ in all PEs, a copy of the word exists in every PE. The subtleties involved with detecting this global condition are discussed below.

Associated with each PE, there is also a one-bit mesh-register M which the PE can read or write when it is active. The global instruction **advance mesh** causes the bit contents of all mesh register to unconditionally "shift down" one row. That is to say, each PE, active or not, simultaneously sends the bit in its M register to the PE below and replaces it with the bit in the M register of the PE above. It goes without saying that the "down" direction is used only for illustrative purposes and that any mesh movement could be used in its place. Indeed, it should be clear that the entire broadcast process can be carried out not only along any direction of a PE array of any dimension $d \geq 1$, but also along any set of node-disjoint paths of any graph.

Finally, we assume that each PE has a shift register SHR of length $l \geq w$, where $l = O(w)$. The instruction SHR(X, Y) simultaneously shifts the register's contents to the right, reads a new bit from bit-register X into the left end, and writes the old right bit into bit-register Y. The purpose of the shift register is to capture the bits streaming down the mesh and shift them until they occupy the rightmost $w$ bits of the register. This solves the skewing problem, provided each PE can be made to conditionally stop once the $w$ bits are right justified in its shift register. Conditional stopping is accomplished by preceding the $w$ bits in the broadcast stream with an extra **header bit**. Being up front, it is the first to emerge from the right end of the shift register, and serves to deactivate the PE. This event can be detected by, say, using the value 1 for the header in conjunction with an initially cleared shift register. Careful analysis reveals that it suffices to clear only the leftmost $l - w$ bits of the shift register, since the rightmost $w$ bits are shifted out before the halting condition is tested. It follows that *no clearing* is necessary when $l = w$.

The parallel SIMD pseudo-code for the entire process is given below.

```
clear leftmost $l - w$ bits of shift register
M ← S
SHR(M,R)
for k ← 0 to $w - 1$ do
    advance mesh
    when S do M ← B[k] endwhen
    SHR(M,R)
endfor
while not all R do
        advance mesh
        when not R do SHR(M,R) endwhen
endwhile
shift rightmost $w$ bits into buffer B[0:$w - 1$]
```

The line M ← S performs two functions. Since S=1 for a sender and S=0 for a receiver, the command effectively clears the mesh and then writes the header bit S=1 on the mesh iff the PE is a sender.

The line SHR(M,R) immediately reads the header bit into the shift register. The second argument R serves no logical purpose at this point; however, the fact that it can be chosen as R is of considerable importance. It should be noted that all instances of the SHR command use only the arguments M and R, in that order. This means that the data-path connecting M, R, and the shift register is never changed. This implies that it can be efficiently implemented in hardware. Such a concurrent data-path would allow the algorithm to stream bits at the clock rate. In such an implementation, the bit register R could be essentially identified with the (complement of) the PE's activity flag.

The *for loop* also serves two purposes. It is used, by a sender, to move its $w$ bits $B[0], \ldots, B[w-1]$ onto to the mesh. This portion of the code is factored out of the while loop, below it, because the first $w$ steps of the algorithm are the only ones that need to make memory references. During this time, the initial rightmost $w$ bits of the shift register, assumed to be garbage, are shifted out by all PEs and ignored. Again, the argument R serves no logical purpose, until $k = w - 1$, when (if $l = w$) the header bit emerges from the shift register into R. Because the leftmost $l - w$ bits were cleared (set to 0), it follows that the first 1 to eventually emerge from the right end of a shift register must be the header bit. Its recognition is therefore assured. When this event occurs, the shift register has the desired $w$ bits right-justified and the PE deactivates itself. If $l > w$, the leftmost $l - w$ bits are considered to be junk.

The *while loop* keeps the bit-stream going, until each receiver has a right-justified copy of the desired $w$ bits in its shift register. Of course, during this time, any PE that has already received its copy (R=1=true) remains inactive. Note that a *when statement* activates only those PEs that satisfy its predicate; only those PEs execute the *entire body* of the when statement.

The predicate *not all R* that controls the while loop needs some discussion. It should be viewed more as a specification than an actual evaluation of a predicate. Its real intent is to keep the bit-stream going until every PE has received the $w$ bits and has deactivated itself. We note that there is no harm in letting the loop continue past this point, and that being able to do so offers a flexibility that allows the algorithm to be used on various PE arrays and under various circumstances. Certainly, if the PE array has a *global flag* [GE, Hil], so that each PE has an input to a global AND gate, say, then the loop condition can actually be evaluated as needed. Since such a massive gate would likely be pipelined, the termination condition would be detected after the event actually occurred. As pointed out above, this will cause no harm. A more likely situation is that the distance $m$, that a word needs to be broadcast, or an upper bound for $m$, is known in advance or can be precomputed. In such a case, the while loop can be replaced with a for loop (counter). One need only guarantee that the number of iterations is sufficiently large so as to allow all PEs to deactivate themselves; such a scheme needs no global gate.

The clearing step, the for loop, and the final shifting step can all be done in $O(w)$ time. The while loop takes $O(m)$ time since the last bit sent must be streamed down into the shift register of the farthest receiver. The entire process thus takes $O(m + w)$ time.

It is worth noting that, if the R register is identified with the PE's activity flag, the when statement, in the while loop, can be replaced by the unconditional instruction SHR(M,R). This is possible because the condition *not R* is then equivalent to *if the PE is active*, and hence only active PEs would participate. The point is that, if PEs have the data path depicted in Figure 1, the body of the while loop can be performed at clock speeds.

## B. Without a shift register

If the PEs do not contain shift registers, the streaming idea still works; however the solution to the skewing problem is a bit more complicated. The SIMD restriction that all PEs use the same memory address, together with the fact that a given bit of the word being broadcast reaches different PEs at different times, means that the bit will be stored at different addresses in different PEs. However, the pattern is regular enough to be managed by a SIMD algorithm.

During the first stage, as the bits are streaming by, the global controller generates a sequence of addresses that *cycle* through the $w$ locations of the buffer $B[0 : w - 1]$. This cyclic addressing process continues until every PE has a (cyclically skewed) copy of the $w$ bits. Each PE is responsible for deactivating itself, once it has stored $w$ bits in its buffer, to prevent the bits from being overwritten with garbage. Each PE must also maintain an **offset counter** that keeps track of how much that PEs buffer needs to be cyclically shifted to correct its skew. This is done with an additional $\log w$-bit counter C in each PE. Once all PEs have received copies of the $w$ bits, a second stage of the algorithm is performed, in order to correct the skew in each buffer, by cyclically shifting its contents as specified by that PE's offset value.

There are two barriers to this approach in a bit-serial SIMD environment. Although C is a counter, we do not want to simply increment it in the obvious way. In the worst case, carry propagation would take $O(\log w)$ time, resulting in an $O(m \log w)$-time broadcasting algorithm. A more efficient counting scheme is discussed below. Once the offsets have been determined, the straightforward method of correcting the cyclic skews of the buffers is to allow each PE that has a given skew $s$, $1 \leq s < w$, to correct it by cyclically shifting the contents of its buffer $s$ units. Since any such a shift needs to move all $w$ bits of the buffer, it takes $O(w)$ time for any value of $s$. Doing this, with a SIMD algorithm, for all $w - 1$ values of $s$ takes $O(w^2)$ time. We show below how it can be done in $O(w \log w)$ time.

In addition to these complications, we must still deal with the problem of detecting when a given PE should deactivate itself, so as not to overwrite the $w$ bits in its buffer $B[0 : w-1]$. For this purpose, we again resort to the use of a header bit. We essentially clear the buffer and begin to cyclically store the incoming bit stream. Before storing a bit, we check the bit already there. The first time a value of 1 is seen, we are assured that it is the header bit, and can thus deactivate the PE.

The parallel SIMD pseudo-code for the entire process is very similar to the previous code.

```
M ← S
for k ← 0 to $w - 1$ do
    advance mesh
    when S do M ← B[k] endwhen
    B[k] ← M
endfor
R ← S
k ← 0
C ← 0
while not all R do
      advance mesh
      when not R do
            R ← B[k]
            B[k] ← M
            increment C from global k
      endwhen
      k ← k+1 (mod $w$)
endwhile
align the circular buffer B[0:w-1]
```

Note that there is no need to initially clear the buffers. Since the instruction M ← S overwrites the garbage in the mesh, the for statement will store $w$ of these "clean" mesh bits into the buffers, before the testing of register R begins. This effectively overwrites and ignores the initial contents of the buffers.

The global counter $k$ serves two purposes. It specifies the single (SIMD) memory location $B[k]$, where each PE stores the bit copied from the mesh, and it is also used to update each of the offset counters C. After reaching location $B[w-1]$, $k$ returns to location $B[0]$ and continues in this cyclic fashion until all PEs are deactivated. Each counter C simply tracks $k$ until its PE deactivates itself; its sole function is to remember the last value of k. When a PE deactivates itself, its C points to the buffer location where the broadcast word starts. Rather than letting each PE cyclically increment its own C register, in a bit-serial fashion, the global controller need only broadcast the bits of C that need to be modified to keep it current. This can be done in a SIMD manner since ***all active PEs have the same value*** of C, and hence need the same update. Although as many as $\log w$ bits of C may need updating, to increment C, the total number of bits that need to be changed, to perform $w$ increments of C, is at most $2w$ [FidMat], rather than $w \log w$. Thus, on average, an update modifies only two bits, and the streaming process, although not progressing at a constant rate, is slowed down by a factor two.

When the streaming process finally stops, each PE has a copy of the $w$ broadcast bits in its circular buffer $B[0 : w-1]$, together with a pointer C to the location $B[C]$ of the first bit. Again, there is no harm in letting the loop go past the point when all PEs have been deactivated. The contents of all buffers can then be aligned, in $d = \log w$ iterations, by stepping through the $d$ bits $C[d-1], \ldots, C[0]$ of C. At the $i$th iteration, any PE that has bit C[i]=1 cyclically shifts its entire buffer an amount $2^i$. After $d$ iterations, taking $dw = w \log w$ time, all PEs have their buffers aligned and the broadcast is complete.

The entire process takes $O(2m + w + w \log w)$ time.

## C. Segmented broadcasting

The above descriptions of the two broadcasting algorithms were given as if only *one* PE per column were sending a word of length $w$ to the $m$ PEs below it. In fact, both algorithms work for ***any number*** of senders. The real meaning of register S is that any PE with S=1 will send its word to all PEs (if any) between itself and the next PE, below it, that also has S=1. The crucial step that makes this work is that every advance of the mesh is followed by a write to the mesh. As a consequence, every sender overwrites the bits coming from above and substitutes its own. This effectively cuts the mesh into segments, at the will of the user, according to the values given to S in each PE. The term $m$ should then be interpreted as the length of the longest segment. Moreover, the values of S in one column of the PE array can be chosen completely independently of those in other columns. Any assignment to S is valid, no matter what the dimension of the PE array. This means that we are not restricted to simply broadcasting rows, in two-dimensional array, or planes, in three-dimensional arrays.

The only uncertainty in this description is how the boundary PEs are connected. If the mesh wraps around, the top PE in a column may be naturally viewed as being below the bottom PE in that column. Since this interpretation gives a potentially infinite column, there should be at least one sender per column to guarantee that the PEs in that column will eventually be deactivated. If the mesh does not wrap around, any PE above the topmost sender will receive garbage from the boundary of the mesh; however, because of the segmentation, all other PEs receive well-defined values.

As an important example of segmentation, consider a column with $n = 2^k$ PEs, numbered 0 through $2^k - 1$, say. If we choose those PEs whose indices are multiples of $2^i$ as senders, we effectively segment the column into $2^{k-i}$ segments, each of length $2^i$, and can concurrently broadcast a different value in each segment. Thus, for $i = 1$, each of the $n/2$ PEs an even index will send its value to the PE immediately below it. This idea is used in the next section to build a broadcasting tree that serves as the skeleton of the scan algorithm.

# III. Fast scan

In this section, we first describe how one-dimensional broadcasting can be made into a fast bit-serial scan algorithm. We then show how to extend the scan algorithm to two dimensions. The generalization to higher-dimensional arrays is immediate.

Let $X = (x_1, \ldots, x_N)$ be a list (vector) whose elements are from some domain $D$. Assume that $D$ has an ***associative*** operation on it, which may be though of as multiplication. The product of two elements $x$ and $y$ is denoted, as usual, by $xy$. Multiplication need not be commutative.

Given an input list $X = (x_1, \ldots, x_N)$, the ***scan operator*** $\sigma$ produces the list $\sigma(X) = (\pi_{11}(X), \ldots, \pi_{1N}(X))$, whose $i$th element is the **prefix product** $\pi_{1i}(X)$, where, in general, $\pi_{ij}$ denotes the **interval product** $\pi_{ij}(X) = x_i x_{i+1} \ldots x_j$. We also use the notation $\pi(X)$ to denote the product $\pi_{1N}(X)$ of the entire list. Note that $\sigma(X)$ is a list, whereas $\pi(X)$ is a scalar.

By appropriate choice of the multiplication operation, the scan operator can be made to perform a variety of useful functions. Its versatility may be further enhanced when a bit-vector $B = (b_1, \ldots, b_N)$ is associated with the input $X$. The vector $B$ is used to segment $X$, by interpreting each 1 in $B$ as the beginning of a new segment, going from left to right. The scan function can then be extended to a list of pairs, over $\{0, 1\} \times D$, by restricting each prefix product that represents the $i$th output element to the segment that contains the $i$th input element $x_i$. That is to say, if $B$ segments $X$ into $X_1, X_2, \ldots, X_s$, say, then $\sigma_B(X) = (\sigma(X_1), \sigma(X_2), \ldots, \sigma(X_s))$

The recursion $\pi_{1i}(X) = \pi_{1,i-1}(X)x_i$ makes it clear that a scan can be performed sequentially in time $O(N)$; however, the computation is *not* inherently sequential. In [LadFish] it is shown that there exists a parallel prefix circuit of size $O(N)$ and depth $O(\log N)$. A simpler approach easily yields a circuit of size $O(N \log N)$ and depth $O(\log N)$. We now use this simpler approach to derive a scan algorithm for a one-dimensional bit-serial array. The algorithm is then extended to two-dimensional arrays.

Let $m$, $1 \le m \le N$, be a "midpoint" that partitions the input $X$ into two lists $X_1 = (x_1, \ldots, x_m)$ and $X_2 = (x_{m+1}, \ldots, x_N)$. Note that for every $i > m$, we have $\pi_{1i}(X) = \pi_{1m}(X)\pi_{m+1,i}(X)$. This suggests that to compute $\sigma(X)$, we concurrently compute the scans $\sigma(X_1)$ and $\sigma(X_2)$, and then simply "lift" every element of $\sigma(X_2)$ by premultiplying it by the last element $\pi_{1,m}(X_1) = \pi(X_1)$ of $\sigma(X_1)$. This gives the recursion

$$\sigma(X_1, X_2) = (\sigma(X_1), \pi(X_1)\sigma(X_2))$$

where $a(b_1, \ldots, b_n) \equiv (ab_1, \ldots, ab_n)$.

For our purposes, the main observation is that the last element $\pi_{1m}(X_1) = \pi(X_1)$ of $\sigma(X_1)$ needs to be ***broadcast*** to all the elements of $\sigma(X_2)$. This three-step process of computing $\sigma(X_1)$ and $\sigma(X_2)$, broadcasting $\pi(X_1)$ to $\sigma(X_2)$, and multiplying $\pi(X_1)$ by the elements of $\sigma(X_2)$, is depicted in Figure 2. Although the technique is recursively defined, it can be easily expressed as a $\log N$-stage iterative SIMD algorithm. The main problem is to determine, at each stage, which PEs need to broadcast. This can be decided on the fly, assuming that each PE knows its own index, or that each PE can precompute a *logN*-bit value that specifies at which stages that PE should be broadcasting.

Using our fast bit-serial broadcasting algorithm, and assuming that multiplying two words of length $w$ takes time $t = t(w)$, the time $T(N)$ to perform a scan, on a list of length $N$, satisfies the recursion: $T(1) = 0$,

$$T(N) = \max(T(m), T(N-m)) + \max(m+w, N-m+w) + t$$

Assuming $N = 2^k$, and that at each stage of the recursion we cut each sublist in half, the time to perform a scan on a list $X$, of length $N$, whose elements are placed one per PE, on a one-dimensional mesh, is

$$T(N) = (N/2+w+t) + (N/4+w+t) + \ldots \approx N + (w+t)\log N$$

Thus, the use of our fast streaming broadcast method gives an $O(N)$ scan algorithm, rather than $O((w+t)N)$, for values of $w$, $t$; and $N$ that are likely to be encountered in practice.

## A. Segmented scan

In this section, we consider how to perform the scan operation on a list $X$ when we are also given a bit-vector $B$ that specifies a segmentation of $X$ [Ble]. Rather than giving a new scan algorithm, we simply show how to define a new multiplication operation that yields the desired result.

We first take the bit-vector $B$, with elements in $\{0, 1\}$, and the list $X$, with elements in $D$, and combine then into a single list of pairs $X' = ((b_1, x_1), \ldots, (b_N, x_N))$, with elements in the cartesian product $D' = \{0, 1\} \times D$. We now define the product of any two pairs $(a, x)$ and $(b, y)$, in $D'$, as

$$(a, x)(b, y) \;\equiv\; if\ (b = 1)\ then\ (b, y)\ else\ (a, xy).$$

If $D$ has has a unit element $I$, the new product is equivalent to $(a + b, x^{\bar{b}}y)$, where $x^0 = I$ for all $x \in D$, + stands for boolean OR, and the bar denotes boolean negation. Note that when b=1, meaning that $y$ is at the beginning of a new segment, we have $x^{\bar{1}}y = x^0y = Iy = y$. If, on the other hand, b=0, we get $x^{\bar{0}}y = x^1y = xy$. Thus, b=1 causes a segmentation, while b=0 does not. This means that by doing a conventional scan $\sigma(X')$ on the list of pairs (using the new multiplication), and then projecting out the boolean component in the resulting list of pairs, we get the desired segmented scan $\sigma_B(X)$ of the original list $X$. By case analysis, one can easily verify that the new multiplication is associative.

## B. Two-dimensional scan

In this section, we consider the problem of doing a scan on a list $X$ of length $N = mn$, whose elements are placed, one per PE, in raster-scan fashion, on a two-dimensional $m \times n$ bit-serial array. Thus, the first $n$ elements of $X$ are in the first row, the next $n$ elements are in the second row, $\ldots$, and the last $n$ elements are the last row. Note that in this section, the row and column indices of the PE array go from 1 to $m$ and from 1 to $n$, respectively.

Although the PE array is two-dimensional, we wish to treat $X$ as a one-dimensional list and perform a scan operation on it, such that the result $\sigma(X)$ has its $i$th element in the same PE that holds $x_i$. The algorithm has five steps.

**Step 1**[horizontal scan]: In parallel, for $i = 1$ to $m$, perform a left-to-right scan $\sigma(x_{(i-1)n+1}, \ldots, x_{in})$ of each row $i$.

After this step, the $(i,j)$th PE contains the interval product $\pi_{(i-1)n+1,(i-1)n+j}(X)$. In particular, the last column of the array contains the products

$$p_1 = \pi_{1,n}(X),\ p_2 = \pi_{n+1,2n}(X),\ \ldots, p_m = \pi_{(m-1)n,mn}(X).$$

Note that the interval products in the first row are the desired prefix products, those in the second row need to be lifted by $q_1 = p_1$, those in the third row by $q_2 = p_1 p_2$, and that, in general, those in the $(i+1)$th row need to be lifted by the factor $q_i = \pi_{1i}(p_1, \ldots, p_m)$. Clearly, $(q_1, \ldots, q_m) = \sigma(p_1, \ldots, p_m)$, so that a vertical top-to-bottom scan of the last column of the PE array is required.

**Step 2**[vertical scan]: Perform a top-to-bottom scan of the values in the last column of the array to compute the "lifting factors" $q_1, \ldots, q_m$.

Since $q_i$ is needed in row $i + 1$, we need to do:

**Step 3**[vertical shift]: In parallel, shift $q_1, \ldots, q_{m-1}$, in the last column of the array, down one row: $q_i \leftarrow q_{i-1}$.

Since each interval product in row $i$, $2 \leq i \leq m$, needs to be premultiplied by (the new value of) $q_i = \pi_{1,(i-1)n}(X)$, which is now at the right end of row $i$, we need to do:

**Step 4**[horizontal broadcast]: In parallel, for $i = 2$ to $m$, broadcast the lifting factor $q_i$ to every element in row $i$.

There are two ways to perform this step. We can simply do a right-to-left broadcast, or, if the mesh wraps around horizontally, we can first shift the last column to the right, into the first column, and then do a *left-to-right* broadcast. The advantage of the second method is that it uses the same left-to-right broadcasting direction as the scan in step 1.

Now that each PE has the appropriate lifting factor, we can do:

**Step 5**[lift]: In parallel, in every PE, in rows 2 through $m$, multiply the lifting factor by the interval product computed in step 1.

The product computed in the $(i,j)$th PE will then be the prefix product $\pi_{1,(i-1)n}(X)\pi_{(i-1)n+1,(i-1)n+j}(X) = \pi_{1,(i-1)n+j}(X)$, as desired.

The times taken by the five steps are: $T_1 = O(n + (w+t)\log n))$, $T_2 = O(m + (w+t)\log m)$, $T_3 = O(w)$, $T_4 = O(n + w)$, and $T_5 = O(t)$. The total time to perform the entire scan on the list $X$, of length $N = mn$, is thus $O(m + n + (w+t)\log(mn))$. If $m = n = \sqrt{N}$, the time is $O(2\sqrt{N} + (w+t)\log N)$.

Its clear that the technique used above for a two-dimensional mesh can be easily extended to a multi-dimensional mesh. Thus, if $N = n^d$, the scan time, on a $d$-dimensional mesh, for a list $X$ of length $N$, is $O(dn + (w+t)\log N)$.

A final observation is in order, for the realistic situation in which the length $N$ of the list $X$ is larger than the number of PEs in the array. Suppose that the array is $m \times n$ and that $N \leq smn$, for some integer $s > 1$. Assuming each PE has sufficient memory to hold $s$ elements, we can again distribute $X$ in a raster-scan fashion, putting $s$ adjacent elements of $X$ per PE. Each PE can then perform a scan on its $s$ elements, in time $O(st)$, using the obvious sequential algorithm. The $s$th result, in each PE, is then used by the above scan algorithm, to compute the necessary $mn$ lifting factors. Since all $s$ elements in a given PE need to be lifted by the same factor, an additional $O(st)$ time is needed to compute the final results. The total parallel time is thus $O(m + n + (w+t)\log(mn) + 2st)$. When $m = n$, this is $O(2\sqrt{N} + 2st)$, for realistic values of $t$ and $w$, showing that $s$ can be $O(n/t)$ without increasing the time by more than a factor of two.

# References

[Ble] Blelloch, G., "Scans as Primitive Parallel Operations," *Proc. 1987 International Conference on Parallel Processing*, 1987.

[Bur] Burkley, J.T., "MPP VLSI Multiprocessor integrated circuit design," in *The Massively Parallel Processor*, J. L. Potter (Ed.), The MIT Press, Cambridge , Massachusetts, 1985.

[FidMat] Fiduccia, C.M. and Mattheyses, R.M., "Bit-Serial Counting in Linear-Time," in preparation.

[GE] General Electric Company, "The Cross-Omega Machine," DARPA proposal, November 1984.

[Flan etal] Flanders,P.M., Hunt,D.J., Reddaway, S.F., and Parkinson, D., "Efficient High Speed Computing With the Distributed Array Processor," in *High Speed Computer and Algorithm Organization*, Kuck, D.J., Lawrie, D.H., and Sameh, A.H., Academic Press, New York, 1977.

[Hil] Hillis, W.D., *The Connection Machine*, The MIT Press, Cambridge, Massachusetts, 1985.

[LadFish] Ladner, R.E. and Fisher, M.J., "Parallel Prefix Computation," *JACM*, 27(4), 1980, pp. 831-838.

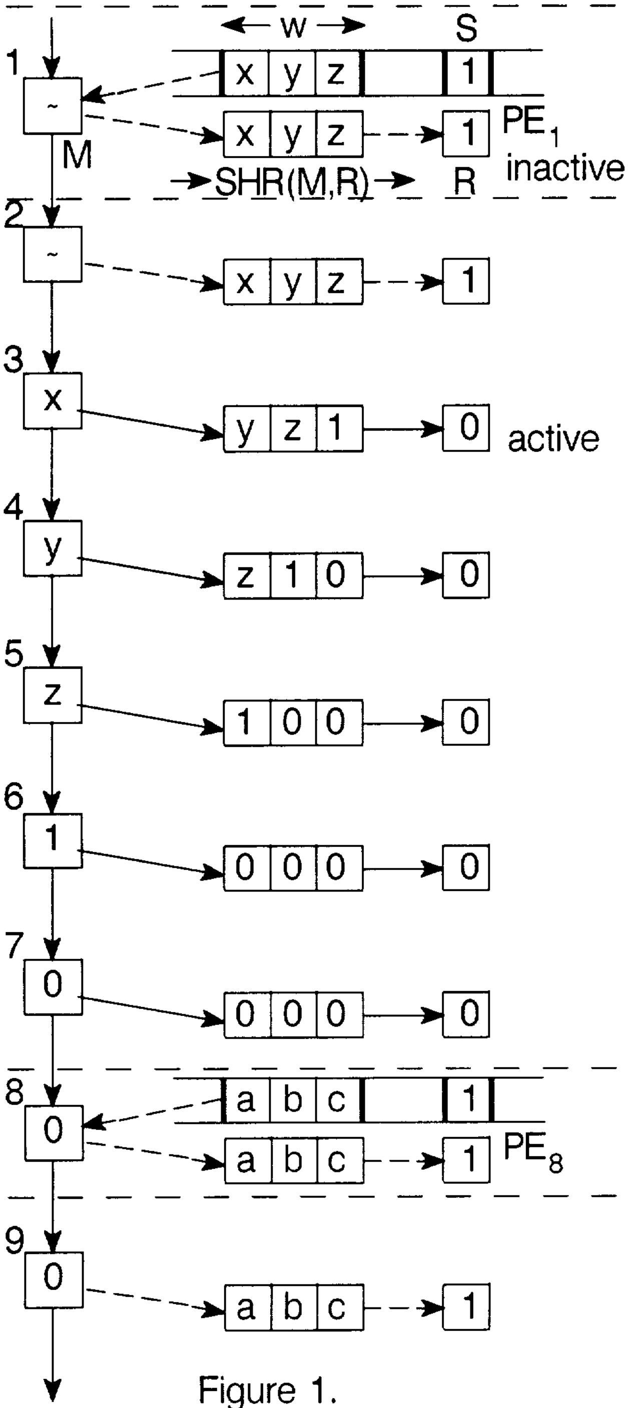


Figure 1.
Segmented Broadcast

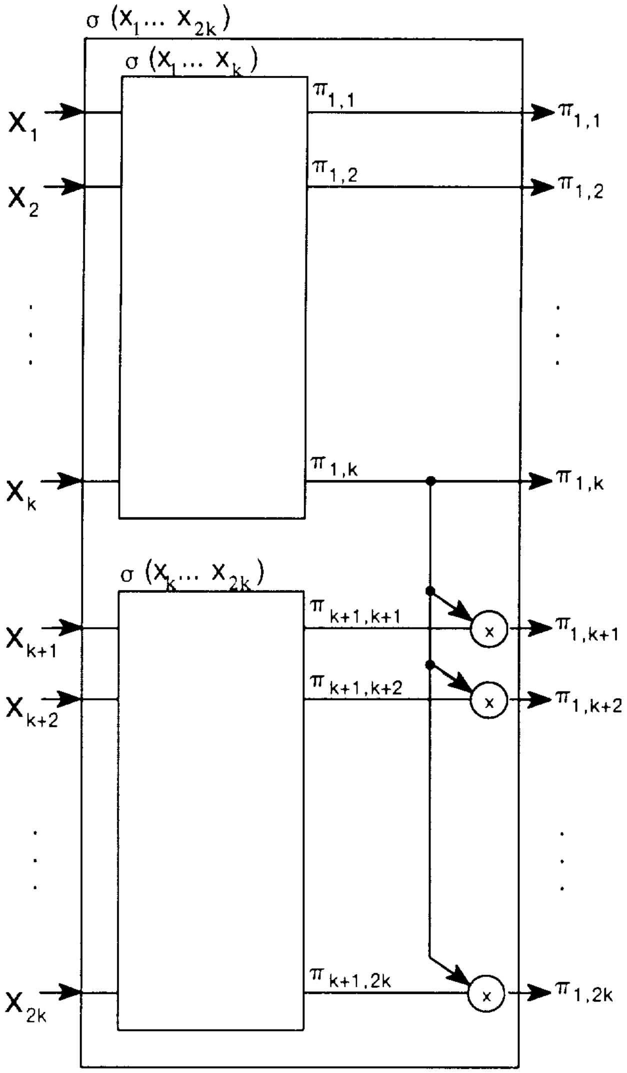


Figure 2.
Scan Using Broadcast

# SECTION I: ALGORITHMS

## Part 2: Poster Presentations

# AN EFFICIENT METHOD FOR THE REPRESENTATION AND TRANSMISSION OF MESSAGE PATTERNS (summary)

*P.J. Bernhard*
Department of Computer Science
Clemson University
Clemson, S.C. 29634

*D.J. Rosenkrantz*
Department of Computer Science
SUNY Albany
Albany, N.Y. 12222

**Abstract**

In this paper we describe a formalism for the compact representation of message patterns for multistage interconnection networks. In this formalism a descriptor called an (s,d)-mask is used to represent a message pattern, or rather, a set of messages. We show that when message patterns are represented in this way a number of their properties can be determined in polynomial time. This includes determining if a message pattern creates conflicts or congestion. In addition, we show that the minimum round partitioning problem, which in general is NP-complete, can be solved in polynomial time for any message pattern which can be represented by a single (s,d)-mask. This generalizes a known result to a more general class of message patterns and a more general class of networks.

Keywords: Omega network, routing, computational complexity, SIMD, parallel processing.

## 1 Introduction

In [La73] Lawrie proposed the Omega network as an interconnection network for a multiprocessor system. For this network a particular message can be represented by a source-address, destination-address pair, abbreviated as an (s,d) pair, where $s$ is the binary address of the source of the message and $d$ is the binary address of the destination of the message. Hence, a *message pattern* can be represented by a set of (s,d) pairs, where each (s,d) pair corresponds to one message.

In this paper we develop a formalism, called the *mask language*, for the representation and transmission of message patterns on Omega networks. In this formalism a message pattern can be represented by a single descriptor called a *(s,d)-mask.* This representation has a number of advantages. For example, a single (s,d)-mask can represent a number of (s,d) pairs which is *exponential* in the size of the (s,d)-mask. Hence, it saves space and, in the context of multiprocessor communication, a single (s,d)-mask can be broadcast to all processors rather then sending the entire set of (s,d) pairs to their respective processors. In addition, we show that when a message pattern is represented by an (s,d)-mask, a number of properties of the message pattern can be determined in polynomial time simply by examining the (s,d)-mask rather than the entire corresponding message pattern. Since a message pattern can be exponentially large compared to its corresponding (s,d)-mask, this fact illustrates one of the main advantages of representing message patterns in the mask language. In addition, we show that the mask language defines a class of message patterns for which the minimum round partitioning problem can be solved in polynomial time for a general class of networks called *bundled Omega networks* .

## 2 The Omega Network

Following Lawrie [La75], an N-input N-output Omega network (also called an $N \times N$ Omega network), where N= $2^m$, consists of $m$ identical stages. Each stage consists of a *perfect shuffle* wire interconnection [St71] followed by N/2 switching elements. In Figure 1(a) we show an $8 \times 8$ Omega network, and in Figures 1(b)-1(g) we show the possible states for each of the switches. Figure 1(b) shows the "straight through" state where the input signals are sent directly to the corresponding outputs, 1(c) shows the "interchange" state where the input signals are first interchanged before being sent to the outputs and Figures 1(d)-1(g) show "incomplete" states. For example, in Figure 1(d) a signal is passed from the upper input to the upper output while nothing is on the lower input or lower output. Note how the model here differs from the one in [La75] since switches are not allowed to "broadcast" messages. In Figure 1(a) we have labeled the interconnection links for each stage, from the top down, with a $log_2$N bit binary address. We have also numbered the stages and shown a path through the network from input 000 to output 011.

A particular path through the network can be represented by a source-destination pair, abbreviated as an

CH2649-2/89/0000/0115$01.00 © 1988 IEEE

(s,d) pair, where the source $s= s_0s_1... \ s_{m-1}$ is the binary address of the input at the first stage, the destination $d= d_0d_1... \ d_{m-1}$ is the binary address of the output at the last stage and $m= log_2N$ . Careful examination of the network shows that the *path code* $s_0s_1... s_{m-1}d_0d_1... \ d_{m-1}$ completely determines a unique path through the network. Specifically, if we define an $m$ bit window $W_i$ as the bit pattern beginning t bit position $i$ of the path code, we see that at stage $i$ in the network, where $0 \leq i \leq m$, the path which goes from $s_0s_1... s_{m-1}$ to $d_0d_1... \ d_{m-1}$ makes use of the link with address $W_i= s_is_{i+1}... \ s_{m-1}d_0d_1... \ d_{i-1}$[RV86]. For example, Figure 1(a) shows a path from 000 to 011. For this path $W_2$=001 and at stage 2 the path makes use of the link with address 001.

The fact that a path code uniquely determines a path through the network enables communication conflicts in the network to be detected easily. Two messages that are being transmitted through the network will conflict if and only if they require use of a common link in the network. Hence, in light of the window property mentioned above, two (s,d) pairs are said to **conflict** if and only if there exists an $i$ such that the two (s,d) pairs have the same bit pattern on window $W_i$. For example, Figure 1(a) shows the paths (000,011) and (100,000). Both (s,d) pairs have $W_1$=000, accordingly at stage 1 both pass through the link with address 000.

This concept of an Omega network can be generalized by the addition of a new parameter $b$ called the **bundle size** of the network. Specifically, we define a (b)N×(b)N Omega network, where N=$2^m$, to have bundle size $b$ if each switch in the network has two bundles of inputs and two bundles of outputs, each of size $b$. For example, in Figure 2 we show a (3)4×(3)4 Omega network. Each bundle in the network may carry $b$ or fewer signals into a switch. Hence, a total of at most $2b$ signals may be input to a switch at any given time. Similarly, each output bundle may carry $b$ or fewer signals out of a switch. For each input bundle, the incoming signals may be sent to the upper or lower output bundle. However, all the signals on a given input bundle don't necessarily have to go to the same output bundle. Some may go to the upper output bundle, while others may go to the lower. Similarly, two signals on different input bundles may go to the same output bundle. The only constraint is that at most $b$ signals can use a particular output bundle at any given time. If more than $b$ require use of the same output bundle then we say that **congestion** occurs. The definition of the standard Omega network is a special case of the generalized definition, where $b$=1. Similarly "conflict" is just a special case of "congestion". Bundled networks have also been considered in [SH87], where a bundled network was referred to as a *dilated* network.

Recall that two (s,d) pairs are said to conflict at stage $i$ in the network if and only if they have the same bit pattern on window $W_i$. However, when $b > 1$ the fact that two pairs have the same bit pattern on window $W_i$ doesn't necessarily imply that congestion occurs. In order for congestion to occur at stage $i$, at least $b$+1 pairs must have the same bit pattern on window $W_i$. For example, consider the paths (0000,1000) and (1100,1001) on a (2)16 × (2)16 Omega network. These pairs have the same bit pattern 0010 on $W_2$ and, hence, the bundle at stage 2 with address 0010 is full because the two paths are in use at the same time. If we now consider the path (0100,1010) we see that at stage 2 this also requires use of the bundle 0010. Hence, if all three paths were required to be in use at the same time, congestion would occur. An example of a bundled Omega network with $b$=16 is in the proposed G.E. Cross Omega machine [H86].

Finally, define a **message pattern** to be a set of (s,d) pairs. Each (s,d) pair in the set represents the fact that a message is to be sent from input $s$ to output $d$ of the network. Note that this definition imposes no restrictions on what type of message pattern the set represents. For example, many pairs may have the same source or the same destination. Furthermore, any number of conflicts may exist in the set.

# 3 Definition of the Mask Language

Define the mask language as follows. **Symbols** used in the language will include constants and literals. **Constants** are 0 and 1, **literals** include variables " $x_0$" , " $x_1$" ," $x_2$" , etc. and their complements. A **mask** is any sequence of symbols such as 0001, 1, 11, $x_0 10 x_1 1$, $x_0x_1\overline{x}_2$, etc. The length of a mask $M$ is the number of symbol occurrences in the mask. Each mask has an implicit universal quantifier to the left of the mask for each variable in the mask, where the variables are quantified over the set {0,1}. Hence, a mask containing the variables $x_0$, $x_1$,..., $x_{r-1}$ is said to **represent** the set $S$ of $2^r$ addresses, each specified by one of the $2^r$ functions from the variables $x_0$, $x_1$,..., $x_{r-1}$ to the set {0,1}. For example, the mask $x_0 1 x_1 0$ represents the set of addresses {0100, 0110, 1100, 1110} . Furthermore, each address in the set is said to be **covered** by the corresponding mask. In the case where a mask contains no variables, such as the mask 101, then the mask represents the set which contains only itself {101}.

An **(s,d)-mask** consists of a left hand side and a right hand side, where each is a mask of the same length. Examples of (s,d)-masks are (001,010), ($1x_0$, 01) and ($x_0 10 x_1 x_2$, $x_1 10 x_2 x_0$). As with masks, an (s,d)-mask has an implicit universal quantifier to the left of the (s,d)-mask for each variable contained within. Hence, the (s,d)-mask is said to represent the corresponding set of (s,d) pairs. For example, the (s,d)-mask $M$=($x_0$10, $\overline{x}_0 1 x_1$) represents the set $S$={(010, 110), (010, 111), (110, 010), (110,011)}.

## 4 (s,d)-Masks and Detecting Congestion

In this section we describe a structure called a *conflict-cube*[RV86]. Each (s,d)-mask has a conflict-cube associated with each window of the (s,d)-mask. As we shall show, the conflict-cubes associated with a given (s,d)-mask can be used to determine a number of properties of the corresponding message pattern.

Suppose that $M$ is an (s,d)-mask and let $V$ be the set of variables which occur or whose complements occur in $M$. Furthermore, let $V_j$ be the set of variables which occur or whose complements occur in window $W_j$ of the (s,d)-mask, where $0 \leq j \leq m$. The **conflict-cube** $S_{M,j}$ of $M$ corresponding to window $W_j$ is the set $S_{M,j}= V\text{-}V_j$. Note that this definition is a slight variation of the one given in [RV86].

Now let $S$ be the message pattern corresponding to $M$. Then the following property of $S$ holds[RV86].

**Fact 1.**Consider an Omega network with bundle size $b$=1. Then the number of messages which conflict on a particular link at stage $j$ is given by $2^k$, where $k$ is the cardinality of the corresponding conflict-cube $S_{M,j}$. Hence, a message pattern represented by a single (s,d)-mask will contain conflicts if and only if it has a nonempty conflict-cube.

In following sections we will show how conflict-cubes can be used to determine a number of properties of (s,d)-masks and their corresponding message patterns. Furthermore, conflict-cubes can be exploited in the solution to the minimum round partitioning problem for any message pattern which can be represented by a single (s,d)-mask.

## 5 Detecting Conflicts in an (s,d)-mask

The Omega network is a blocking network and, as such, does not allow the transmission of arbitrary message patterns. Specifically, it does not allow the transmission of message patterns which give rise to communication conflicts. Hence, algorithms for detecting communication conflicts and strategies for dealing with communication conflicts have become the focus of numerous researchers. As stated in Section 1, one of the advantages of the mask language is that many properties of message patterns can be determined simply by examining (s,d)-masks rather than the entire corresponding message pattern. The following lemmas illustrate this for the detection of conflicts and congestion in a message pattern represented by one or more (s,d)-masks. It should be noted that for the lemmas and theorems in this paper we give short sketches of the proofs. We refer the interested reader to [B88] for the detailed versions.

**Lemma 2.** Let $M$ be an (s,d)-mask of length $m$. Then determining if the message pattern corresponding to $M$ contains communication conflicts can be done in $O(m)$ time.

**Proof.** (sketch) By Fact 1 in Section 4, a given (s,d)-mask will contain conflicts if and only if it has a nonempty conflict-cube. Hence, an algorithm for detecting conflicts would operate by scanning the (s,d)-mask from left to right checking for a nonempty conflict-cube. The key to the algorithm lies in the fact that each window is examined using only a constant amount of time, thus ensuring that the algorithm operates in linear time.□

In [BR87] an algorithm is discussed which will determine if a given set $S$ of (s,d) pairs contains communication conflicts. The algorithm operates in time O($m^2p$), where $p =| S |$ . Given that $p$ could be exponential in $m$, the length of a corresponding mask pair, Lemma 2 illustrates one of the main advantages of using (s,d)-masks for representing message patterns. We now consider the more general case of detecting *congestion* in an Omega network with bundle size $b \geq 1$.

**Lemma 3.** Let $M$ be an (s,d)-mask of length $m$ and $b \geq 1$ be a bundle size. Then determining if the message pattern corresponding to $M$ is congestion-free for an Omega network with bundle size $b$ can be done in $O(m)$ time.

**Proof.**(sketch) As in the proof of Lemma 2, an algorithm for detecting congestion would scan the (s,d)-mask from left to right examining the conflict cubes at each window. However, it follows from Fact 1 in Section 4, that in order for congestion to occur on an Omega network with bundle size $b > 1$, the (s,d)-mask must contain a conflict-cube of size $k$, where $2^k> b$. □

In addition to message patterns representable by a single (s,d)-mask, we consider message patterns which require more than one (s,d)-mask for their representation. Hence, it becomes important to be able to detect communication conflicts and congestion in a *set* of (s,d)-masks. In the following, a set of (s,d)-masks is said to be *disjoint* if no (s,d) pair is covered by two different (s,d)-masks in the set.

**Theorem 4.** Let $S$ be a set of disjoint (s,d)-masks where $n =| S |$ and $m$ is the length of each (s,d)-mask in $S$. Then determining if $S$ is conflict-free can be done in O( $m^2n^2$) time.

**Proof.** (sketch) An algorithm for detecting if a set of (s,d)-masks contains conflicts would operate by scanning all of the (s,d)-masks, at the same time, from left to right. As it scans it would examine the set of (s,d)-masks on each window. For each window it would compare each pair of (s,d)-masks to see if they conflict on that window. This can be determined by a reduction to an instance of 2-SAT, which can be solved in O(m) time[GJ79]. Since each pair of (s,d)-masks must be compared on each window, a total of O($n^2$) 2-SAT instances must be solved for each window. Hence, each window requires an O($mn^2$)

operation. Since there are a total of $m+1$ windows to be examined, this gives a total running time of $O(m^2n^2)$.□

**Theorem 5.** Let $b \geq 1$ be a bundle size and $S$ a set of $n$ disjoint (s,d)-masks, where $n$ is fixed and each (s,d)-mask in S is of length $m$. Then determining if the set $S$ is *congestion-free* for an Omega network with bundle size $b$ can be done in O( $m^2$) time.

**Proof.** (sketch) As in Theorem 4 an algorithm for testing for congestion will check each of the $m+1$ windows. In addition, for each window each of the $2^n$ subsets of the $n$ masks must be checked to see if a subset of the (s,d)-masks covers a set of conflicting (s,d)-pairs, at least one (s,d)-pair per mask. The test of each such subset requires that an instance of 2-SATISFIABILITY be solved, which requires $O(m)$ time. However, since the bundle size of the network may be greater than 1, an additional counting step must be performed. Since there are $m+1$ windows and since $n$ is fixed the running time of the algorithm is O( $m^2$). It should be noted that since each window requires $2^n$ 2-SATISFIABILITY instances to be solved the constant on the running time is exponential in $n$. □

As with Lemma 2, the above results illustrate one of the main advantages of using (s,d)-masks. Specifically, the corresponding algorithms which operate on sets of (s,d) pairs, instead of (s,d)-masks, may require an exponential increase in time.

## 6 Minimum Round Partitioning for (S,D) Masks

Suppose that a message pattern is to be transmitted on an Omega network. In addition, suppose that it has been determined that the message pattern creates congestion. One strategy for dealing with this situation is to partition the corresponding set of messages into disjoint, congestion-free subsets, called **rounds,** and then transmitting the set of messages by successively transmitting the messages in each round. Clearly, in order to minimize the total time for message transmission, it is important to minimize the total number of rounds.

The problem of partitioning a set of (s,d) pairs into a minimum number of rounds is referred to as the **minimum round partitioning problem.** This problem has previously been considered by a number of authors. For example, in [A83] upper and lower bounds for the problem have been established. In [WF80] and [DF87] heuristics for the problem are given. An algorithm is given in [RV86] which will construct a "partitioning function" for a set of messages when the message pattern is represented as a "bit permute complement" permutation. And in [BR87] the computational complexity of the problem was considered. For a number of special cases the problem was shown to be solvable in polynomial time, however, it was shown in general to be NP-hard. Here, we show that the problem can be solved in polynomial time when the message pattern can be represented by a single (s,d)-mask.

**Theorem 6.** Let $S$ be a message pattern which can be represented by a single (s,d)-mask. Then $S$ can be partitioned into a minimum number of congestion-free rounds for an Omega network with bundle size $b \geq 1$ in linear time.

**Proof.** The algorithm for performing the partitioning exploits two facts related to message patterns which can be represented by a single (s,d)-mask. The first is that for any such message pattern the (s,d)-mask can be computed from the set $S$ in linear time[B88]. And the second is that the minimum number of rounds required by the message pattern is equivalent to $2^k$, where k is the cardinality of the largest conflict cube for the corresponding (s,d)-mask. Given the message pattern as input, the algorithm will compute the (s,d)-mask and then determine how many rounds are required by examining the associated conflict-cubes. Using this information it will then partition the message pattern. Each of these steps can be performed in linear time. Hence, the result follows. □

## 7 Conclusion

In this paper we have described a formalism for the compact representation of message patterns. We have shown that when message patterns are represented in this formalism a number of their properties can be determined in polynomial time, simply by examing representative (s,d)-masks rather than the message patterns themselves. This fact is important since a message pattern may be exponentially large compared with its corresponding (s,d)-mask. In addition, we have shown that the minimum round partitioning problem, which in general is NP-complete, can be solved in polynomial time for any message pattern that is representable by a single (s,d)-mask. This generalizes a known result [RV86] to a more general class of message patterns and a more general class of networks.

## References

[A83] Agrawal, D.P., "Graph Theoretical Analysis and Design of Multistate Interconnection Networks," IEEE Trans. Comput., vol. C-32, no. 7,July 1983, pp. 637-648.

[B88] Bernhard,P.J., "Algorithmic Aspects of Message Transmission Strategies for Multistage Interconnection Networks," Comput. Sci. Dept., SUNY Albany, Albany, N.Y.

[BR87] Bernhard,P.J. and Rosenkrantz,D.J., "The Complexity of Routing Through an Omega Network," *Proc. Twenty-Fifth Annual Allerton Conf. on*

*Communication, Control and Computing,* Sept. 1987. Also appears as a technical report: Comput. Sci. Dept., SUNY Albany, Albany, N.Y., TR 87-12, 1987.

[DF87] Deogun,J.S. and Fang,Z., "A Heuristic Algorithm for Conflict Resolution Problem in Multistage Interconnection Networks," *Proc. of the 1987 International Conf. on Parallel Processing,* Aug. 1987. pp.475-478.

[GJ79] Garey,M.R. and Johnson,D.S., "Computers and Intractability: A Guide to the Theory of NP-Completeness," W.H. Freeman and Co., 1979.

[H86] Hardy,R.M., personal communication.

[La73] Lawrie,D.H., "Memory-processor connection networks," Univ. Illinois, Urbana-Champaign, Dep. Comput. Sci., Rep. 557, Feb. 1973.

[La75] Lawrie,D.H., "Access and Alignment of Data in an Array Processor," IEEE Trans. Comput., Vol. c-21, no. 12, Dec. 1975, pp. 1145-1155.

[RV86] Raghavendra,C.S. and Varma,A.,"Fault-Tolerant Multiprocessors with Redundant-Path Interconnection Networks," IEEE Trans. Comput., Vol. C-35, No. 4, April 1986, pp. 307-316.

[SH87] Szymanski, T.H. and Hamacher, V.C., "On the Permutation Capability of Multistage Interconnection Networks," IEEE Trans. Comput., Vol. C-36, July 1987, pp. 810-822.

[St71] Stone,H.S., "Parallel processing with the perfect shuffle," IEEE Trans. Comput., vol. C-20, Feb. 1971, pp. 153-161.

[WF80] Wu,C.L., Feng,T.Y., "On a Class of Multistage Interconnection Networks," IEEE Trans. Comput., vol. C-29, no. 8, August 1980, pp.694-702.

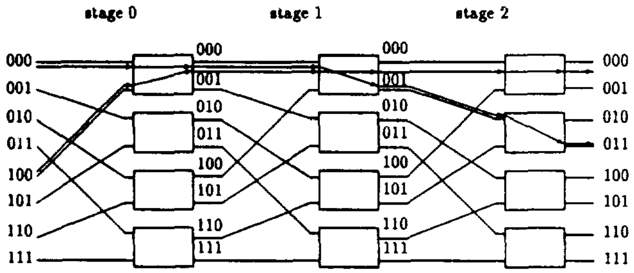


**Figure 1(a):** An Omega network.

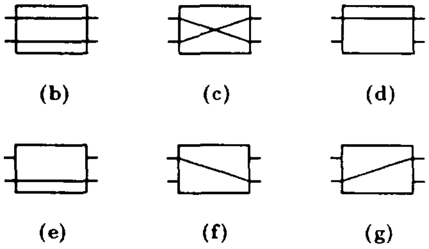


**Figure 1(b)-(g):** Possible switch states.

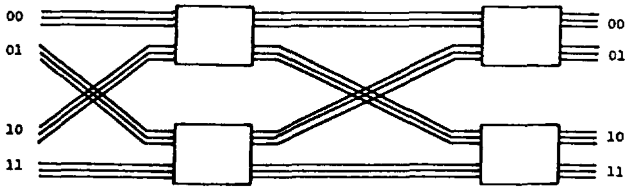


Figure 2: A 3(4) by 3(4) Omega Network

# MINIMUM SPANNING TREE ON THE HMESH ARCHITECTURE*

R. V. Boppana C. S. Raghavendra
Dept. of Electrical Engineering–Systems
University of Southern California
Los Angeles, CA 90089

**Abstract**

A fast algorithm to compute minimum spanning tree of a given undirected graph on Hierarchical MESH connected computer (HMESH) is presented. The time complexity of the algorithm is $O(\log^2 n)$, where $n$ is the number of nodes in the graph. HMESH is a broadcast bus VLSI architecture which consists of $n \times n$ processing elements (PE's) in a mesh connected structure and a hierarchy of broadcast buses in each row and column of the mesh structure such that each broadcast bus is connected to exactly $k$ PE's, where $k$ is a small constant. Later, we will show that with simple modifications to the algorithm, MST of $n$ node graph can be found on HMESH of size $p \times p$ in $O(\lceil n/p \rceil^2 \log n \ \log p)$ time. It is also shown how to compute *connected components* and *transitive closure* of a given undirected graph in $O(\log^2 n)$ with a few modifications to the algorithm presented for computing minimum spanning tree.

## 1 Introduction

The *minimum spanning tree* of a given undirected, connected graph $G = (V, E)$, where $V$ is a non-empty set of $n$ nodes and $E$ is a set of $e$ unordered pairs of nodes called edges, with cost or weight assigned to each edge in the graph, is the connected subgraph of $G$ whose total edge cost is minimum. The problem of finding *minimum spanning tree* has some practical applications e.g., the problem of connecting various cities by high ways so that each city has a path to other cities, directly or through another city, with minimum cost, routing a common signal to different points in a VLSI chip efficiently, efficient broadcasting in networks etc. The problem of finding the *connected components* of a given undirected graph is to color all the vertices in the same component of $G$, with some unique color. So, the problem of finding the *connected components* of a given undirected graph $G$ can be seen as the problem of finding the spanning forest of $G$ with weights of all edges being equal. The problem of finding connected components has some applications in areas like pattern recognition etc.

The classical methods of finding, sequentially, the *minimum spanning tree* of a given undirected graph are Prim-Dijkstra's [1,2] method, Kruskal's [3] method, and Sollin's [4] method. Of these three, Sollin's method is most suitable for parallel computation of the *minimum spanning tree* of a graph. The problem of obtaining a parallel solution to compute the *minimum spanning tree* and the *connected components* of a given undirected graph $G$ with $n$ nodes and $e$ edges, has been studied extensively in the literature [5,6,7,8,9,10,11,12,13].

Some fast and efficient parallel algorithms are proposed in [8,7,9,10,11]. In [8] an $O(\log^2 n)$ parallel algorithm to compute the connected components is presented. They use PRAM model with CREW (Concurrent Read, Exclusive Write) capability. Modifications to the algorithm in [8] resulted in faster algorithms with time complexity $O(\log n)$ to compute *minimum spanning tree* [9,10], and *connected components* [11]. However, they all use PRAM model with CRCW (Concurrent Read, Concurrent Write) capability. The algorithm proposed in [11] uses $O(\log(n + 2e))$ processors, and computes the *connected components* of $G$. The algorithm proposed in [9] uses $n^3$ processors, and computes minimum spanning tree of $G$.

Many researchers use PRAM models to evaluate the time complexity of their parallel algorithms. However these PRAM models are not realizable in practice with the current technology. So, some researchers have proposed various VLSI architectures as practical models for synchronized parallel computation [29,25,26,27,28]. In particular linear array and two dimensional array processor architectures are given considerable attention [30], [18],[17],[16],[14], [6]; because, the regularity of the architectures makes them suitable for VLSI implementation. One disadvantage of these architectures is that they have large diameter (i.e. worst case communication delay between any two processing elements in the architecture). Attempts to overcome this, in particular for two dimensional mesh connected computers, are met with some success [15,23,19,20,32]. However, mapping algorithms to these architectures is non-trivial and, in some cases, is highly complex [19,15]. Moreover, the resultant architecutes are not practical for VLSI implementation. As a solution to these problems the Hierarchical Mesh structure (HMESH) was proposed [21]. HMESH is a highly modular architecture, which allows easy mapping of algorithms [24]. In this paper we will present an algorithm to compute the *minimum spanning tree* using HMESH (described later) in $O(\log^2 n)$ time. With simple modifications the same algorithm can be used to find *connected components* and *transitive closure* of $G$.

## 2 Architecture of the HMESH

In mesh connected computers, the solution time complexities are dominated by the interprocessor communication times. In a 2-dimensional MCC with $n^2$ PE's, time required for movement of data between two farthest PE's is $O(n)$. Therefore, many problems on the MCC will have $O(n)$ complexity, e.g., finding maximum of $n$ numbers. Since the MCC is a well suited structure for various problems, attempts are made to reduce the delay involved with long data movements, by adding broadcast buses [15,22,19]. In [21] the hierarchical mesh connected computer (HMESH) has

*This research is supported by the NSF Presidential Young Investigator Award No. MIP 8452003, a grant from AT&T Information Systems, and a grant from TRW Inc.

CH2649-2/89/0000/0121$01.00 © 1988 IEEE

been proposed by generalizing the idea of multiple broadcast buses with a view to certain practical aspects, such as limiting the number of PE's that can be connected to a bus. The main idea is to provide multiple buses in each row and column of a 2-MCC such that there are some finite number of PE's connected to each bus (figure 1.)

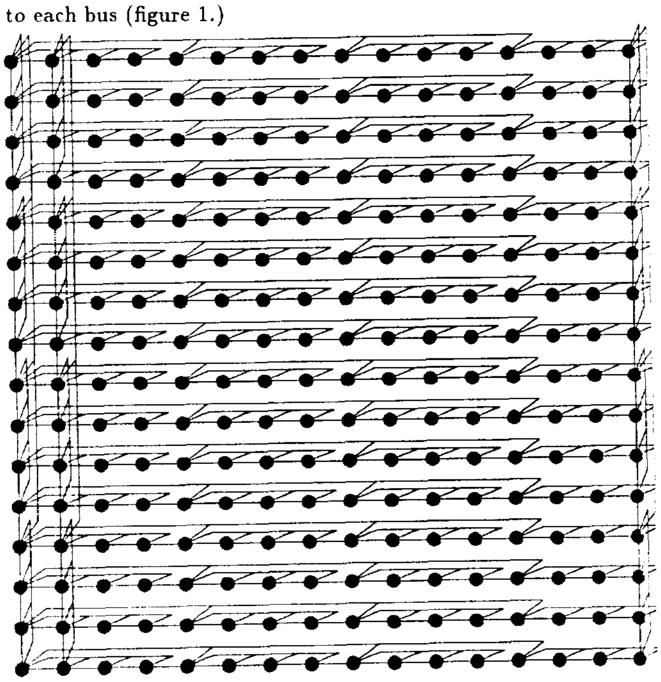


Figure 1: The Hmesh architecture with $n = 16$ and $k = 4$.(Only first two and last column buses are shown for clarity)

The hierarchical mesh is an SIMD architecture consisting of $n \times n$ processing elements with four nearest neighbor connections. Each PE consists of a few registers and is capable of performing arithmetic and logic computations. For routing data to long distances, the array of PE's are interconnected by a system of hierarchical broadcast buses. The PE's are numbered as $(i, j)$ where $1 \leq i, j \leq n$. In each row and column PE's are grouped and each group of size $k$ share a common bus. The least index numbered PE's are again grouped in the next level, and again groups of $k$ PE's share a bus. This construction is repeated until the top level is reached with one group $\leq k$ PE's which would require only one bus. There will be $\lceil \log_k n \rceil$ levels of buses in each row and column. The architecture for 16x16 PE's is shown in figure 1.

The hierarchy of multiple buses allow fast data transfer between a pair of PE's. It takes at most $O(\log n)$ bus transfers for data to be routed from a source PE to a destination PE. Of course multiple PE's can be transferring data to other PE's simultaneously as long as different buses are used. However, there is potentially high parallelism in transfer of data between PE's. Also, a single PE can broadcast its data to all other PE's in $O(\log n)$ time. This can be accomplished by first broadcasting to local PE's, then to subsequent level of PE's, and after $\log n$ steps to the entire row of PE's. Then the same procedure can be repeated in all the column of PE's. In the algorithm for MST we repeatedly use te row or column broadcast operation (contents of the register $x$ of $PE_{ij}$ will be sent to all the PE's in row $i$,) and min operation (minimum of the contents of register(s) of PE's in a column or row is found) repeatedly. The time taken by either of these operations is $O(\log n)$ (here $k$ is treated as a constant.) These two operations are discussed in detail in [31,5].

Many parallel algorithms use divide and conquer technique which maps naturally to this architecture. Therefore, we can expect algorithms for this architecture to be much simpler and yet very efficient.

## 3 Minimum Spanning Tree

In this section we present the algorithm MST to compute the minimum spanning tree of a given graph $G$. We will also show that with simple modifications to the algorithm, MST's of graphs larger than HMESH can be found with correspondingly larger time complexity.

### 3.1 Discussion

The algorithm uses the well known Sollin's technique to compute the minimum spanning tree in parallel [4]. The basic idea is to group nodes in the same component using the edge weight information, and to make all the nodes in a group to have one identification number (also called color of the group, but it is really the node number of one of the nodes in the group) so that any two nodes can immediately identify whether they are in the same group or not. A group in which all the nodes have one color is called a super vertex.

At the beginning of the algorithm, each node is a super vertex by itself, and its color is same as its node number. Each node tries to hook to a node, to which it is connected by the minimum weight edge. This is called hook operation. As a result of hook operation there could exist a long chain of nodes in which a node is trying to another node and one or more nodes are trying hook to this node. Cycles between two nodes are avoided by making one of the nodes to point it to itself and the other to point to it. This chain is condensed into a star shape format so that all the nodes point to a single node, called the leader of the super vertex which also gives the color of the super vertex. This condensing operation is called shortcut operation. In case of long chains it may take several steps before they are condensed into one supervertex. In the meanwhile, a supervertex is free to hook to another supervertex or to a chain of nodes that is being condensed. It should be noted that a chain of nodes will never participate in a hook operation until it is condensed into a supervertex. This process is repeated until no supervertex changes and there exists no chains of nodes.

### 3.2 The algorithm MST

We shall now present the algorithm to compute minimum–cost spanning tree of an undirected graph $G$ (see figure 2.) The initial conditions and terminating conditions are as given below.

**Initial Conditions**

- Each $PE_{ij}$ for $1 \leq i, j \leq n$ has the following registers: (a) A register $C_{ij}$ to keep the color of the node j. $C_{iC_{ii}}$ indicates the color of the node $C_{ii}$. (b) register $W'_{ij}$ to keep the cost of the edge connecting the new neighbor vertex. (c) A register $C'_{ij}$ to keep the color of the new neighbor vertex.
- Each non diagonal $PE_{ij}$ for $1 \leq i, j \leq n$ and $i \neq j$ has the following registers/flags in addition to the above mentioned ones: (a) A flag $E_{ij}$ to indicate whether there is an edge $(i, j)$ present in the given graph. i.e. adjacency matrix is assumed to be the input to the algorithm. (b) A register $W_{ij}$ to keep the cost of edge $(i, j)$. (c) A flag $S_{ij}$ to indicate whether the edge $(i, j)$ is present in the spanning tree.

- Each $PE_{ii}$ for $1 \leq i \leq n$ has the following registers/flags in addition to those mentioned above: (a) A flag $Q_i$ to indicate whether the vertex $i$ has participated in the present iteration or not. This is useful in testing for the completion fo the algorithm. (b) Two registers $S1_i$ and $S2_i$ to keep the index information of the PE that supplied the most min. cost edge to connect super vertex $i$ to a neighbor super vertex in the present iteration.

**Terminating Conditions**

- All diagonal PE's in the same component will have the same color. i.e. , $C_{ii} = C_{jj}$ iff vertices $i$ and $j$ are in the same connected component.
- All the edges in the spanning tree are indicated by setting the corresponding '$S$' flag true. i.e. , $S_{ij} = true$ if edge $(i, j)$ is in the spanning tree.

Time complexity of the algorithm is, shown in [31], $O(\log^2 n)$. Proof of correctness also is given in [31].

### 3.3 Finding MST with Smaller HMESH

In this section we will show that MST of a given graph G(V,E) can be found on HMESH of size smaller than $n$, the number of nodes in $G$.

Let the number of rows and columns in the HMESH be $p$ where $p > 1$. With simple modifications to the algorithm MST given earlier it is possible to find MST of $G$. Now each PE in the HMESH will have $r = \lceil n/p \rceil$ times the original number of registers, and will keep the information about $r$ nodes. Information about node $i$ will be kept in PE's in row $i \ mod \ p$ and column $i \ mod \ p$. Each step in the algorithm is modified to execute the same operation for all the nodes having the same index ( given as $x \ mod \ p$ where $x$ is the node number ) and hence are taken care of by PE's in the row or column given by the index number. All the subscripts appearing in the algorithm are now their actual value modulo $p$ so that correct PE's are accessed.

It is shown in [31] that the resulting time complexity is $O(\lceil n/p \rceil^2 \log n \ \log p)$.

## 4 Related Problems

In this section we will show how to compute *connected components* and *transitive closure* of a given graph $G$ using the algorithm MST.

By taking the adjacency matrix as the weight matrix, the algorithm MST can be used to compute *connected components* of the given graph $G$. However, to compute *transitive closure* of $G$ we first compute the *connected components* of $G$ and then step **3** given below is performed. The *transitive closure* of the graph is given by the boolean matrix formed by falgs $S_{ij}$ for all $1 \leq i, j \leq n$.

```
3       foreach $i$ do
           if $C_{ij} = C_{ii}$ then $S_{ij} = true$
```

**Algorithm 1** *MST*

```
/* The following terminology is used in the algorithm.
forall indicates that all the PE's are active in the computation. foreach i indicates that all the PE's in a row are active. foreach j indicates that all the PE's in a column are active. broadcast is done along that column. The broadcast value is given by the diagonal PE. */
1       /* Necessary parameters are initialized now */
1.1     forall $PE_{ii}$ do { $C_{ii} = i$; }
1.2     forall $PE_{ij}$ $j \neq i$ do $S_{ij} = false$;
2       repeat
2.1       forall $PE_{ii}$ do { $Q_i = false$;}
2.2       column–broadcast $C_{jj}$;
2.3       /* This is a shortcutting operation */
2.3a        foreach $i$ do {
2.3a1         $C_{ii} = C_{iC_{ii}}$;
2.3a2         if $C_{ii} \neq C_{iC_{ii}}$ then $Q_{C_{iC_{ii}}} = true$;}
2.3b        column–broadcast $C_{jj}$;
2.4       /* This is a hooking operation */
2.4a        foreach $i$ do
2.4a1         if $C_{ii} = C_{iC_{ii}} \wedge \neg Q_{C_{ii}}$ then do {
2.4a2           $S2_i = min_j\{j|W_{ij}$ is min. $\wedge C_{ij}$ is min.
                                  $\wedge E_{ij} \wedge C_{ij} \neq C_{ii};\}$
2.4a3           $W'_{iC_{ii}} = W_{iS2_i}$;
2.4a4           $C'_{iC_{ii}} = C_{iS2_i}$ and mark PE $(i, C_{ii})$ active; }
2.4a5         else $C'_{ii} = 0$;
2.4b        foreach $j$ do
2.4b1         if there exists an active PE in the column do {
2.4b2           $Q_j = true$;
2.4b3           $S1_j = min_i\{i|C'_{ij}$ is min. $\wedge W'_{ij}$ is min.$\}$;
2.4b4           $S_{S1_j S2_{S1_j}} = true$;
2.4b5           $S_{S2_{S1_j} S1_j} = true$;
2.4b6           $C'_{jj} = C'_{S1_j j}$ }
2.4c        column–broadcast $C'_{jj}$
2.4d        foreach $i$ do {
2.4d1         if $(C'_{ii} \neq 0) \wedge (i = C'_{iC'_{ii}})$ then
2.4d2             $C'_{ii} = min\{i, C'_{ii}\}$
2.4d3         else if $C'_{ii} \neq 0$ then $C_{ii} = C'_{ii}$; }
        until $(\neg Q_i$ for all $i)$
```

Figure 2: Algorithm to compute *minimum spanning tree*

## References

[1] R. C. Prim, *Shortest Connection Networks and Some Generalizations.* Bell Systems Technical Journal 36, Nov. 1957, pp 1389–1401.

[2] E. W. Dijkstra, *A note on two problems in connection with graphs.* Numerisch Math, Vol. 1, No. 5, pp 269–271.

[3] J. B. Kruskal, *On the shortest spanning subtree of a graph and the tarvelling salesman problem.* Proc. Amer. Math. Society, 1956, Vol. 7, No. 1, 48–50.

[4] Sollin, An algorithm attributed to Sollin in *Programming, Games and Transportation Networks.* by Berge, C., and Choulia-Houri, A. Wiley, NY 1965.

[5] D. Nath, S. N. Maheshwari and P. C. P. Bhatt, *Efficient VLSI Networks for Parallel Processing Based on Orthogonal Trees.* IEEE Trans. on Comput. Vol C-32. No. 6. June 1983.

[6] M. J. Atallah and S. R. Kosaraju, *Graph Problems on a Mesh-Connected Processor Array.* JACM Vol. 31, No. 3, July 1984.

[7] F. Y. Chin, J. Lam and I. Chen, *Eficient Parallel Algorithms for Some Graph Problems.* Communications of ACM, Vol. 25, No. 9, Sept. 1982.

[8] D. S. Hirschberg, A. K. Chandra and D. V. Sarwate, *Computing Connected Components on Parallel Computers.* Communications of ACM, Vol. 22, No. 8, August 1979.

[9] D. S. Hirscberg and D. J. Volper, *A Parallel Solution for the Minimum Spanning Tree Problem.* Proceedings of the 1983 Johns Hopkins Conference on Info. Science and Systems, pp 680-684.

[10] F. T. Leighton, *Parallel Computation Using Meshes of Trees.* Proceedings of the Workshop on Graphtheoretic Concepts in Computer Science, July, 1983.

[11] Y. Shiloach and U. Vishkin, *An $O(\log n)$ Parallel Connectivity Algorithm.* Journal of Algorithms 3, 1982, pp 57-67.

[12] D. Nath and S. N. Maheshwari, *Parallel Algorithms for the Connected Components and Minimal Spanning Tree Problems.* Information Processing Letters, Vol. 14, No. 1, March 1982.

[13] J. L. Bentley, *A parallel Algoritm for Constructing Minimum Spanning Trees.* Tech. Report, Deptt. of Computer Sci. and Mathematics, CMU, August 1979.

[14] S. H. Bokhari, *MAX: An Algorithm for Finding Maximum in an Array Processor with a Global Bus.* Proc. 1981 International Parallel Processing Conference, pp 302-303.

[15] S. H. Bokhari, *Finding Maximum on an Array Processor with a Global Bus.* IEEE Transactions on Computers, Vol. C-33, No. 2, February 1984, pp 133-139.

[16] D. A. Carlson, *Performing Tree and Prefix Computations on Modified Mesh-Connected Parallel Computers.* Proc. 1985 Int. Conf. on Parallel Processing, August 1985, pp 715-718.

[17] P. S. Gopalakrishnan, I. V. Ramakrishnan, L. N. Kanal, *An Efficient Connected Components Algorithm on a Mesh-Connected Computer.* Proc. 1985 Int. Conf. Parallel Processing, August 1985, pp 711-714.

[18] H. T. Kung, C. D. Thompson, *Sorting on a Mesh Connected Computer.* Communications of the ACM, 1977.

[19] V. K. Prasanna Kumar, C. S. Raghavendra, *Array Processor with Multiple Broadcasting,.* Proc. 12th Annual Symposium on Computer Architecture, June 1985, pp 2-10.

[20] V. K. Prasanna Kumar, C. S. Raghavendra, *Image Processing on an Enhanced Mesh Connected Computer.* Proc. IEEE Workshop on Computer Architecture for Pattern Analysis and Image Database Management, November 1985, pp 243-247.

[21] C. S. Raghavendra, *HMESH:A VLSI Architecture for Parallel Processing.* CONPAR 86, Lecture Notes in Computer Science, Springer–Verlag.

[22] Q. F. Stout, *Broadcasting on Mesh Connected Computers.* 1982 Conference on Information Sciences and Systems, pp. 85–90.

[23] Q. F. Stout, *Mesh Connected Computers with Broadcasting.* IEEE Trans. on Computers, pp. 826–830, 1983.

[24] Suresh B. C. and C. S. Raghavendra, *Geometric Algorithms on HMESH Architecture.* 1987 Workshop on Pattern Analysis and Image Processing (to appear).

[25] Bentley, J. L. and H. T. Kung, *A tree machine for searching problems.* Proceedings of the 1979 Intl. Conf. on Parallel Proc., Aug. '79, pp 257–266.

[26] H. T. Kung and C. E. Leiserson, *Systolic Arrays (for VLSI).* In Sparse Matrix Proc. 1978, Society for Industrial and Applied Mathematics, 1979, pp 256-282.

[27] F. P. Preparata and J. E. Vuillemin, *The cube-connected cycles: a versatile network for parallel compuation.* Proc. 12th Annual IEEE symposium on Foundatioins of Computer Sciencs, 1979.

[28] G. Barnes et.al., *The Illiac IV computer.* IEEE Trans. on Comput., Vol. c–17, No. 8, pp 746–757, Aug. 1968.

[29] H. Stone, *Parallel Processing with Perfect Shuffle.* IEEE Trans. on Comput., Vol. C–20, No. 2, Feb. 1971.

[30] V. C. Hamacher, *Machine Complexity Versus Interconnection Complexity in Iterative Arrays.* IEEE Trans. on Comput., Vol. C–20, 1971, pp 321–323.

[31] R. V. Boppana and C. S. Raghavendra, *Minimum Spanning tree on the HMESH architecture.* Tech. report CRI–88–20, Computer Research Institute, Univ. of Southern California, Los Angeles.

[32] R. Miller and Q. F. Stout, *The Pyramid Computer for Image Processing.* Proc. 7th International Conf. on Pattern Recog., 1984.

# Optimal Mesh Algorithms For VLSI Routing[1]

(Extended Abstract)

Shing-Chong Chang
Department of Electrical Engineering
Systems Research Center
University of Maryland
College Park, MD. 20742

Joseph JáJá
Department of Electrical Engineering
Institute for Advanced Computer Studies
Systems Research Center
University of Maryland
College Park, MD. 20742

**Abstract**

We develop optimal mesh algorithms for several VLSI routing problems, such as river routing between rectangles, routing within a rectilinear polygon and wiring module pins to frame pads. We assume that the mesh consists of a $\sqrt{n} \times \sqrt{n}$ processors, where $n$ is the input size. Each processor has a constant amount of memory. All our algorithms run in time $O(\sqrt{n})$.

## 1 Introduction

The recent advances in the VLSI technology allow the fabrication of highly complex systems on single chips. Sophisticated software tools are needed to successfully design such systems. In particular, the routing phase is a critical and time-consuming part of the overall design process. Unfortunately, it turns out that most routing problems are NP-complete and hence no efficient solutions seem to be likely. There are few exceptions, however. For example, various river routing (one-layer) problems, the two-layer channel routing with no constraints, and few routing problems in the knock-knee model are known to have efficient solutions ([D et al],[MP],[O],[P],[PL]). Our goal is to develop a good set of techniques to obtain fast and efficient parallel routing algorithms.

In this paper, we consider several basic problems in VLSI routing such as river routing between rectangles, routing within a rectilinear polygon, and wiring module pins to frame pads. The known strategies to handle these problems seem to be inherently sequential. We develop new techniques that lead to optimal parallel algorithms. Our basic model of parallel processing is the two dimensional array.

Some of the well-known parallel techniques, such as path doubling, prefix computation, list ranking and sorting, are used extensively in our parallel routing algorithms. All of these techniques have efficient mesh implementations. We will briefly introduce these techniques below. Path doubling is a basic technique used to solve many problems involving lists and graphs. For example, given a set of linked lists, we can determine the sink reachable from each node by iterating the process of changing the successor of a node to the successor of the successor (effectively doubling the length of the path from the node to its successor after each iteration). Given $n$ elements $a_0, a_1, \ldots, a_{n-1}$ and an associative operation $*$, the prefix computation consists of evaluating the $n$ partial products $s_i = a_0 * a_1 * \ldots a_i$, for $0 \leq i \leq n-1$. Finally, sorting is the process of rearranging a sequence of values in ascending or descending order. All the techniques mentioned above can be implemented in $O(\sqrt{n})$ time on a $\sqrt{n} \times \sqrt{n}$ mesh.

## 2 Definitions

We assume that the reader is familiar with the basic definitions related to river routing, routing within a rectilinear polygon (See for example [D et al],[LP],[P],[SD],[T]), and wiring module pins to frame pads (See [BP]). Here we will introduce these problems briefly. The class of general river routing problems involves routing between ordered sequences of terminals such that the final layout is planar. Figure 1 shows an example of a river routing problem and a wiring achieving the minimum separation between the two boundaries. A more general version of the river routing problem is to perform planar routing where the ports lie on the boundary of a simple rectilinear polygon. Figure 2 shows an example of this routing problem.

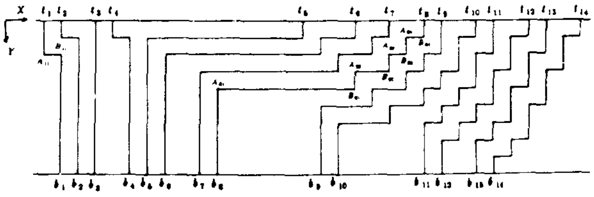

X
Y
$t_1$ $t_2$ $t_3$ $t_4$ $t_5$ $t_6$ $t_7$ $t_8$ $t_9$ $t_{10}$ $t_{11}$ $t_{12}$ $t_{13}$ $t_{14}$
$b_1$ $b_2$ $b_3$ $b_4$ $b_5$ $b_6$ $b_7$ $b_8$ $b_9$ $b_{10}$ $b_{11}$ $b_{12}$ $b_{13}$ $b_{14}$

Figure 1: Basic river routing problem between two parallel boundaries.

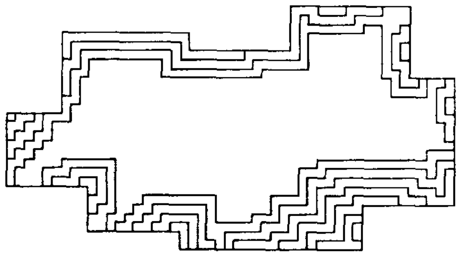


Figure 2: Routing within a rectilinear polygon.

[1] Supported in part by NSA Contract No. MDA-904-85H-0015, NSF Grant No. DCR-86-00378 and by the Systems Research Center Contract No. OIR-85-00108

The problem of wiring module pins to frame pads is given by a triplet $< \mathcal{M}, \mathcal{F}, \mathcal{N} >$, where $\mathcal{M}$ is an arbitrary rectilinear polygon representing a module, $\mathcal{F}$ is a rectangle representing a frame, and $\mathcal{N}$ is a set of two-terminal nets such that one terminal is on $\mathcal{M}$ and the other is on $\mathcal{F}$. We assume that $\mathcal{F}$ contains $\mathcal{M}$ and that each boundary segment of $\mathcal{M}$ is parallel to a frame edge. We are supposed to determine a one-layer routing of $\mathcal{N}$ whenever it exists. Figure 3 is an instance of a such problem and its final wiring.

In this abstract, we will concentrate on the one-layer model. Section 3 presents a summary of the river routing algorithms and section 4 addresses the problem of wiring module pins to frame pads.

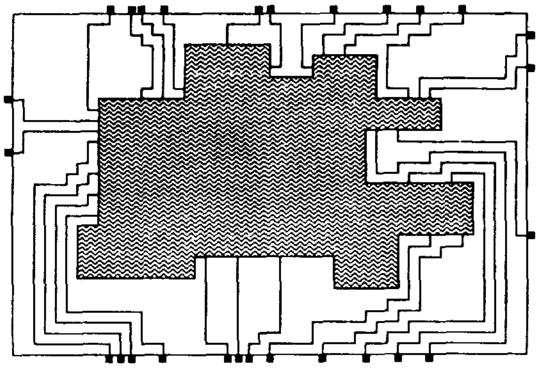


Figure 3: Routing between module pins and frame pads.

## 3 River Routing

We can partition the nets into blocks such that the wiring problem is reduced to wiring each block simultaneously. For any *right* block, such as $< N_5, N_6, \ldots, N_{14} >$ in Figure 1, we have following lemma.

**Lemma 1**: ([CJ1]) Let $N_i$ be a net in a right block and let $\hat{j}$ be the minimum $j \leq i$ such that $t_j + (i - j - 1) \geq b_i$. Then the coordinates of the characteristic bend points of $N_i$ (bend points closest to the bottom row) are $A_{i1} = (b_i, i - \hat{j} + 1)$ and $B_{i1} = (t_{\hat{j}} + i - \hat{j}, i - \hat{j} + 1)$. The characteristic bend points uniquely define the overall wiring.

**Theorem 1**: ([CJ1]) The characteristic bend points of the $n$ input nets as well as the minimum channel separation can be determined on a $\sqrt{n} \times \sqrt{n}$ mesh in $O(\sqrt{n})$ time.

In the routing problem of nets within a simple rectilinear polygon, our strategy for the routing problem will consist of identifying a set of net groups and the *representative net* of each group then performing the wiring of each such net with the nets "covered" by it separately.

**Lemma 2**: ([CJ1]) The total number of bend points of all the representative nets is $O(n)$, where $n$ is the number of nets. We can identify the proper groups and find the representative nets in time $O(\sqrt{n})$ on a $\sqrt{n} \times \sqrt{n}$ mesh of processors.

Let $N =< x, y >$ be a net in a group whose representative is $N_r$. With the number of nets between $N$ and $N_r$, we can determine a ***bounding perimeter*** such that the wiring of $N_r$ cannot lie inside it. We claim that the following lemma is true.

**Lemma 3**: ([CJ1]) The union of all the bounding perimeters of all the nets within a group determines the contour of the group and hence determines the wiring of the representative net. If the number of nets in the group is $n$, then the union can be determined in time $O(\sqrt{n})$ on the mesh.

**Theorem 2**: ([CJ1]) Detailed routing of $n$ nets within a simple rectilinear polygon can be done in time $O(\sqrt{n})$ on a $\sqrt{n} \times \sqrt{n}$ mesh of processors.

The problem may be unroutable if (1) The graph determined by the nets when restricted to lie within the rectangle is *nonplanar.* (Figure 4-1) (2) The wiring of all the nets requires more area. (Figure 4-2) Case (1) can be detected easily by the techniques such as path doubling , prefix computation and sorting. Our approach to case (2) is to partition nets into blocks, then determine *the wiring capacity* and the *wiring density* between blocks. Based on this information, we can detect the routability ([CJ1]). Below we give the algorithm to determine whether or not there is enough space between the blocks to wire the remaining nets.

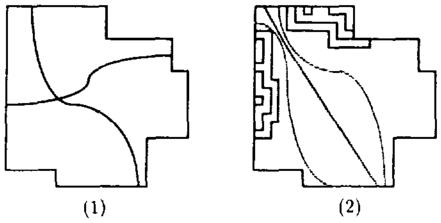


Figure 4: Routability testing: (1) Planarity tesating (2) Area testing.

*Algorithm Density and Capacity*

*Input:* The wiring of cover nets of the blocks and the terminals of the remaining nets.

*Output:* Determine whether or not there is enough space between the blocks to wire the remaining nets

1. Cut the rectilinear polygon boundary at some fixed point and straighten it into a horizontal line ([CJ1]).

2. Assign weights to the terminals of each remaining nets outside the blocks as follows: +1 to the terminal with smaller coordinate in the line, -1 otherwise. Order these terminals according to their line coordinates and compute the *rank* ( summation of all the weights proceed it, including itself) of each terminal. Each block is assigned the *rank* of terminal adjacent to the left terminal (in the horizontal line) of its cover net. The density between two blocks is equal to the difference in their ranks.

3. Compute the intersection point of each 45 degree line segment emitting from the convex corners of each block and the block contour or the original boundary of the rectilinear polygon ([CJ1]). The capacity between blocks can then be calculated easily.

4. Test whether the capacity is at least as large as the corresponding density. Otherwise the problem is unroutable.

**Theorem 3**: Testing the routability of $n$ nets within a simple rectilinear polygon could be done in time $O(\sqrt{n})$ on a $\sqrt{n} \times \sqrt{n}$ mesh of processors.

## 4 Wiring Module Pins to Frame Pads

For any routing problem $< \mathcal{M}, \mathcal{F}, \mathcal{N} >$, we partition the nets into groups such the nets in the same group share a 45 degree diagonal. For example, the groups of Figure 5 are given by: $G_1 = \{N_1, N_2, N_3, N_4, N_5, N_6\}$, $G_2 = \{N_7\}$, $G_3 = \{N_8\}$, $G_4 = \{N_{10}\}$, $G_5 = \{N_{11}, N_{12}\}$, $G_6 = \{N_{15}, N_{16}, N_{17}\}$, $G_7 = \{N_9\}$ and $G_8 = \{N_{13}, N_{14}\}$. For each net, we call the intersection point of the routing with the diagonal as the *intermediate terminal.*

The wiring of each net consists of two parts. The first one starts from the module terminal and routes as close to the module boundaries as possible until it reach the intermediate terminal. The other part is between the intermediate terminal and the frame terminal and stays as close to the frame edges as possible. The first part can be determined by the techniques of determining the union of rectilinear polygons. For the second part, we move the intermediate points vertically to a horizontal line $L$ such that the separation distance is enough to solve the corresponding river routing problem. With the above methods, we can determine the wiring of each net.

The following is an algorithm to determine the intermediate terminals of those group such as $G_1, G_3$ and $G_5$ in Figure 5. Obvious changes can be made to get other intermediate terminals.

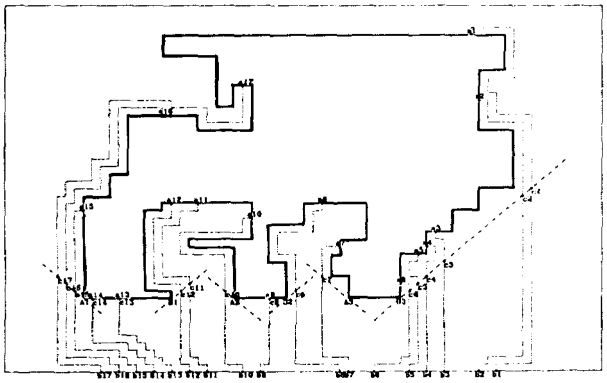


Figure 5: Example of routing problem with frame terminals on the bottom frame edge.

*Algorithm Intermediate Terminals*

*Input*: Corners of module boundary, module terminals of a group of nets, and the corner and the diagonal of the group.

*Output*: The intermediate terminal of each net in the group.

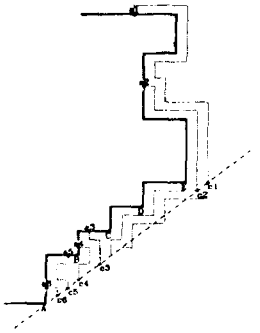
1. Rank each terminal of the group according to its order counterclockwise around the boundary. Call the corresponding rank of a net *sequence number.*

2. Calculate the distance between each convex corner and the diagonal of the group. For each convex corner $C$, determine an integer $k$ such that each net with sequence number $\geq k$ intersects the diagonal near $C$. $k$ will be called the *bounding value* of $C$.

3. For each net $N$, determine the closest convex corner $Q$ with bounding value less than or equal its sequence number. $Q$ is called the *bounding point* of $N$. From this information, determine the intermediate terminals of all the nets in the group.

Figure 6 shows the lists obtained by the above algorithm for group $G_1$.

**Lemma 4**: ([CJ2]) The *intermediate terminals* of all the nets can be determined in time $O(\sqrt{n})$ with an array of $\sqrt{n} \times \sqrt{n}$ PEs.



|  | a1 | a2 | a3 | a4 | a5 | a6 |
|---|---|---|---|---|---|---|
| Sequence Number : | 6 | 5 | 4 | 3 | 2 | 1 |

|  | A | B | C | D | E |
|---|---|---|---|---|---|
| Bounding Value : | 1 | 4 | 6 | 6 | 5 |

|  | N1 | N2 | N3 | N4 | N5 | N6 |
|---|---|---|---|---|---|---|
| Bounding Point : | E | E | B | A | A | A |

Figure 6: Determination of Intermediate Terminals.

**Theorem 4**: Given an instance of above routing problem, we can determine the routing in time $O(\sqrt{n})$ with an array of $\sqrt{n} \times \sqrt{n}$ PEs, where $n$ is the input length.

For the routability testing, we can find out the routing of the outermost net in each group then detect the intersection between these nets, module and frame boundary by the methods of [MS]. The following is the algorithm for the set of nets whose frame terminals are on the bottom frame edge. Obvious changes can be made for more general nets.

*Algorithm Routability Testing*

1. Partition the nets with one terminal on the bottom frame edge into groups and identify the corresponding corners and diagonals.

2. Determine the outer contour of each group as well as the intersection points (*intermediate terminals*) of the routing with the corresponding diagonal assuming a greedy strategy as close to the module as possible (greedy-in).

3. Move the intermediate points vertically to a horizontal line $L$ such that the separation distance is enough to solve the corresponding river routing problem. Find the characteristic bend points of nets corresponding to the induced river routing problem.

4. Determine if there is any intersection between the wirings of any two different nets or between the wiring of any net and the module or frame boundary.

Step 4 involves a set of cases. Each of these cases will be reduced to testing the intersection between two sets of line segments ([CJ2]).

**Theorem 5**: Given an instance of the routability testing problem, we can test whether a solution exists in time $O(\sqrt{n})$ with an array of $\sqrt{n} \times \sqrt{n}$ PEs, where $n$ is the input length.

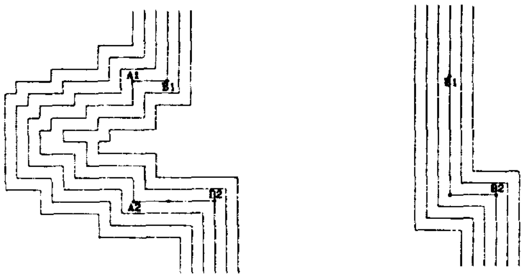


Figure 7: Transformation of U-wires with on obstacles.

Suppose that in addition to the input $< \mathcal{M}, \mathcal{F}, \mathcal{N} >$, a wiring of all the nets in $\mathcal{N}$ is also provided. Our next subproblem is to modify the given wiring in such a way that the total wire length is minimized. A *U-Wire* is a sequence of three successive segments resulting from two successive 90 degree turn clockwise or counterclockwise. A U-wire is ***reducible*** if the line segment one unit from the base is not occupied by another wire or module edge, or is occupied by the base of a reducible U-wire. It is clear that shapes more complicated than reducible U's have to be considered if a fast parallel algorithm is desired. We can shrink U-wires whenever possible as follows. Let $W$ be a U-wire with initial segment $e_1 = (B_1, A_1)$ and last segment $e_2 = (B_2, A_2)$ with (say) the x-coordinate of $B_1$ less than or equal the x-coordinate of $B_2$. In addition, suppose there is no obstacle inside $W$. Then we can apply the transformation shown in Figure 7 to shorten $W$. If $W$ has an obstacle inside it, then we find a maximal set of U-wires with the same obstacle and apply the transformation shown in Figure 8. The algorithm consists of identifying U-wires and reducing them whenever possible. This process has to be repeated $O(\log n)$ times to remove all the reducible U-wires such that the resulting wiring is of minimum length. We can use the divide and conquer technique. After the first iteration, the problem is divided into two subproblems of half the size. Repeat this strategy recursively for each subproblem for at most $O(\log n)$ times. We have following result ([CJ2]).

**Theorem 6**: Given an initial wiring, we can change this wiring so that the resulting wiring is of minimum total length in time $O(\sqrt{n})$ with an array of $\sqrt{n} \times \sqrt{n}$ PEs, where $n$ is the input length.

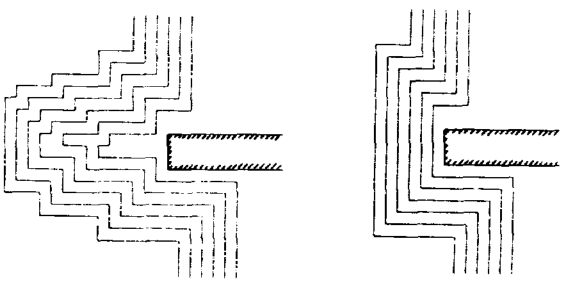


Figure 8: Transformation of U-wires with an obstacles.

## 5 References

[BP] B. Baker and R. Pinter, "An Algorithm for the Optimal Placement and Routing of a Circuit Within a Ring of Pads," Proceedings of the 24th Symposium on Foundations of Computer Science, Nov. 1983, pp. 360-370, Tucson, Az.

[CJ1] S. C. Chang and J. JáJá, "Parallel Algorithms For River Routing," Proc. International Conference on Parallel Processing, 1988, pp. 9-13.

[CJ2] S. C. Chang and J. JáJá, "Parallel Algorithms for Wiring Module Pins to Frame Pads", Technical Report UMIACS-TR-88-2, CS-TR-1970, University of Maryland, College Park, Jan. 1988.

[CS] R. Cole and A. Siegel, "River routing every which way, but loose," 25th FOCS, Oct. 1984, pp. 65-73.

[D et al] D. Dolev, K. Karplus, A. Seigel, A. Strong and J. Ullman, "Optimal wiring between rectangles," Proc. 13th Annual ACM Symposium STOC, May 1981, pp. 312-317.

[LM] C. E. Leiserson and F. M. Maley, "Algorithms for routing and testing routability of planar VLSI layouts," 17th ACM STOC, May 1985, pp. 69-78.

[LP] C. Leieserson and R. Pinter, "Optimal placement for river routing," SICOMP 12(3), August 1983, pp. 417-462.

[M] F. M. Maley, "Toward a mathematical theory of single-layer wire routing," 5th MIT conference on Advanced Research in VLSI, March 1988, pp. 277-296.

[Mi] A. Mirzaian, "Channel routing in VLSI," STOC, May 1984, pp. 101-107.

[MS] R. Miller and Q. Stout, "Mesh Computer Algorithms for Computational Geometry," Technical Report 86-18, Dept. Computer Science, State University of New York at Buffalo, July 1986.

[O] Ohtsuki, T., "Layout Design and Verification," Advances in CAD for VLSI, vol. 4, North-Holland, 1986.

[P] R. Pinter, "River routing: methodology and analysis," Proceedings of the third CALTECH Conference on Very Large Scale Integration, March 1983, pp. 141-163.

[PL] Preparata, F. and W. Lipski, "Optimal Three-Layer Channel Routing," IEEE Trans. on Computers, C-33, pp. 427-437, 1984.

[SD] A. Seigel and D. Dolev, "The separation for general single layer wiring barriers," Proceedings of the CMU Conference on VLSI Systems and Computations, October 1981, pp. 143-152.

[T] M. Tompa, "An optimal solution to a wire routing problem," Proceedings of the 12th Annual Symposium on Theory of Computing, April 1980, pp. 161-176.

# A UNIFORM AND RECONFIGURABLE FRAMEWORK FOR THE MULTIDIMENSIONAL FOURIER TRANSFORM

Ron Coleman and Michael Post
Department of Electrical Engineering and Computer Science
Polytechnic University
Brooklyn, New York 11201
(718) 260 - 3440

Alan Waksman
Plex Systems Research, Incorporated
New York City, NY 10016
(212) 686- 9600

## ABSTRACT

The computational demands and real-time constraints of the multidimensional Fourier transform (FT) make it an ideal candidate for attack by massive parallelism. The appeal of parallelism has lead to the consideration of a variety of architectures including systolic arrays, data flow architectures, arrays of digital signal microprocessors and so-called hybrid architectures. These designs have been advance for d ≤ 2 dimensions or rely complex and inflexible hardware such as array transposers and peripheral rotation networks. Moreover, they include specific optimizations for the FT or assume the need for course-grain, high-speed computational resources. The *Hypercomputer* supercomputer is a reconfigurable, massively parallel architecture, a 9,072 processor prototype currently planned. The hypercomputer architecture family is based on arrays of a simple and autonomous unit logic entity, the universal cell. Physically wired in a uniform, eight-degree mesh, the universal cell is a pipelined, 8-bit microarchitecture. The mesh array is supported by a separate 3D, parallel IO network. The surprising conclusion we demonstrate in this paper is that simulated *d*-dim FT algorithms on the hypercomputer are fast and achieve optimal area and time complexity even though the approach is uniform and does not incorporate any specific architectural optimizations for the FT. For an *N*-cube signal space of *d* dimensions, we derive detailed performance models for the *k* cell hypercomputer where $N^d \leq k$ and $N \leq k < N^d$. We also present numerical results comparing a simulated hypercomputer and the AT&T Signal Processing Ensemble Parallel Computer Architecture.

## INTRODUCTION

The application of the Fourier transform (FT) has undergone explosive growth aided primarily by the discovery of more efficient algorithmic implementations like the fast Fourier transform (FFT) (Ref. 1,2) and more recently, by advances in Very Large Scale Integration (VLSI). As such, the FT has emerged as one of the most widely studied, computationally intensive algorithms in the literature. Much of this interest has focused on accelerating its performance in a unidimensional and multidimensional signal space. Several special-purpose and parallel architectures have been suggested including systolic arrays (Ref. 3), data flow architectures (Ref. 4), digital signal processors as well as arrays of digital signal processors (Ref. 5,7,8), and so-called *hybrid* VLSI architectures (Ref. 9). In all, the above proposals have been advanced for d ≤ 2 dimensions, or rely on complex and inflexible hardware such as array transposers and peripheral rotation networks. Moreover, these architectures include specific optimizations for the FT or assume the necessity of course-grain, high-speed computational resources.

In contrast, we present an alternative, general design based instead on reconfiguration and massive parallelism; that is, in the context of the Hypercomputer supercomputer (Ref. 6), a 9,072 processor prototype currently planned at Plex Systems Research, Inc. in New York City. The surprising conclusion we show is that simulated multidimensional FT algorithms on the hypercomputer are fast, achieve optimal space and time complexity even though the approach is uniform and does not incorporate any specific architectural optimizations for the FT. Furthermore, each cell in our design is simple and fine grain (8-bits), and is to be implemented with high latency circuits ( ≥ 200 ns). We exploit these hardware limitations through massive parallelism at the instruction level and the algorithm level. Through the reconfiguration, we address specialized computation and communication, tailoring the degree of both to accomodate each other as well as the FT of different problem sizes and with different space-time constraints.

## HYPERCOMPUTERS

The hardware foundation of the hypercomputer family centers on a uniform, eight-degree mesh of universal cells. This mesh array is supported by a three dimensional, global IO network which provides real-time IO in the planar dimension and distributed configuration and synchronization along the polar dimension. Physically, each cell is an autonomous, 8-bit microarchitecture with a four stage, pipelined data path and a 128x32-bit control store, general purpose registers, control registers and flags, and an interface to its neighbors through locally shared regions collectively called the *synapse*. These simple hardware features is the physical basis of low level reconfiguration which captures the functionality of processor elements, switch lattices, adjunct memory, etc. in a single conceptual entity. These attributes also serve a much more powerful abstration concept we call *computational holism* in which objects or *actors* (Ref. 10) pool their limited resources functionally and hierarchically, forming cooperative associations of cell ensembles or "chunks" that behave as a single, continuous whole. The software counterpart of this model is embodied in an object-oriented, visually interactive environment, called *hyperware* which composes arbitrarily abstract and complex actors.

## ALGORITHM

The discrete Fourier transform (DFT) for the *N*-cube signal space of *d* dimensions is computed by Equation 1 where $S(n_1,n_2,...n_d)$ is a datum in

$$X_{k_1k_2 \cdots k_d} = \sum_{n_d} \cdots \sum_{n_2}\sum_{n_1} S_{n_1n_2 \cdots n_d} \cdot W_{N_1},\ W_{N_2} \cdots,\ W_{N_d} \qquad (1)$$

the signal space, $N = N_1 = N_2 = ...N_d$, N is a power of 2 and $W_N = \exp(j2\pi n_i k_i/N_i)$ for $n_i,k_i = 0,1,...N_{i-1}$. Equation 1 can be re-written in a form which is more suitable for direct implementation (Ref. 9) as Equations 2-4 where *R* is a cyclic permutation function: that is, $R(S(a_1,a_2,...a_n)) = S(a_2,a_3,...a_n,a_1)$. Functionally, R is a *perfect shuffle* routing map. Physically, R is a rotation of the signal cube through one dimen-

CH2649-2/89/0000/0129$01.00 © 1988 IEEE

$$X_{n_2 n_3 \cdots n_d k_1} = R\left(\sum_{n_1=0} S_{n_1 \cdots n_d} W_{N_1}\right) \tag{2}$$

$$X_{n_3 \cdots n_d k_1 k_2} = R\left(\sum_{n_2=0} X_{n_2 n_3 \cdots n_d k_1} W_{N_2}\right) \tag{3}$$

$$X_{k_1 k_2 \cdots k_d} = R\left(\sum_{n_d=0} X_{n_d k_1 k_2 \cdots k_{d-1}} W_{N_2}\right) \tag{4}$$

sion of the d dimensional signal space. For d = 2, this is the simple matrix transposition. Thus, computing the 2-dim FT using cyclic rotations is equivalent to the row-column method which computes the 1-dim DFT on all the rows then computes another 1-dim DFT on the columns of the row results. We note, in the general case, the d-dim FT is computable solely in terms of 1-dim DFTs, each interlaced by rotating the signal cube. Since the 1-dim DFT requires O($N^2$) multiplications, the d-dim FT (from Equations 2-4) has requires O($dN^{d+1}$) multiplications. It can be seen, then, that a successful real-time implementation of the FT (following Equations 2-4) will depend on fast multiplication. In many of the designs listed above, the need for fast multiplication is the justification for low latency circuits.

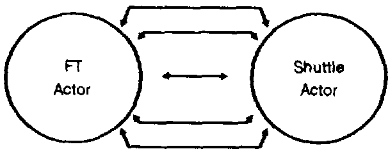


*Figure 1. Cooperating subproblems for the d-dim FT.*

To achieve the theoretical lower bound complexity, O(d log N) (Ref. 12), Equations 2-4 are implemented to take full advantage of the inherent opportunities for parallelism: first, the parallel, fast Fourier transform (ρFFT) is used instead of the serial DFT and second, in the (d-1)-dim plane, $N^{d-1}$ ρFFTs are computed in parallel. However, in practice, this lower bound cannot be achieved without communication overhead costs since the speedup gained through parallelism is obtained at the expense increased data movement in the ρFFT and the cube rotation. The cost of this overhead is architecture dependent. For our implementation, we present overhead results in the "Performance" section.

## IMPLEMENTATION

In view of the above algorithm (Equations 2-4), the d-dim FT on the hypercomputer presents a challenge, enough to easily overwhelm any single cell. However, when local groups of cells chunks pool their limited resources through cooperative computation, complicated tasks like the d-dim FT can be viewed as various levels of abstraction which as a whole, simplifies the task enormously. At the highest level, we analyse the d-dim FT problem as two smaller cooperative subproblems in computation and communication which map to two tightly coupled, complex actors: the *FT actor* and the *shuttle actor*. See Figure 1. These complex actors are hierarchically integrated as a composition of progressively simpler, more fundamental actors which "bottom-out" at the level of primitive actors: i.e., actors whose behavior is derived directly from a single universal cell.

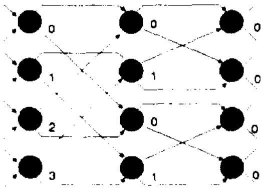


*Figure 2. N = 8 parallel FFT.*

### FT Actor

The FT actor is composed of $N^{d-1}$ 1-dim ρFFT actors. A multistage version of the ρFFT actor for $N = 8$ is shown in Figure 2. The ρFFT version we implement is recirculating: that is, there is one stage which behaves like multiple stages by way of an interconnection network to "circulate" the data flow. In our case, this recirculation is carried out by the shuttle actor. We prefer the recirculating network of actors because of its significant area advantage compared to the multistage version: that is, for the ρFFT, *O(N)* vs. *O(N log (N/2)*. The ρFFT actor is composed of N/2 *butterfly actors* (e.g., the black objects in Figure 2) each of which compute *log N* output pairs, *(A + B)* and $(A\text{-}B)w^k$ (see Figure 3). *A, B* are inputs and $w^k$ is a "twiddle factor" constant which corresponds to the $N^{th}$ root of unity for the complex number $(x + iy)^N$.

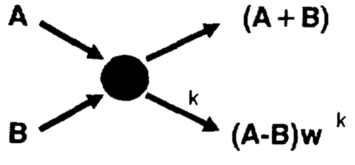


*Figure 3. Butterfly operation.*

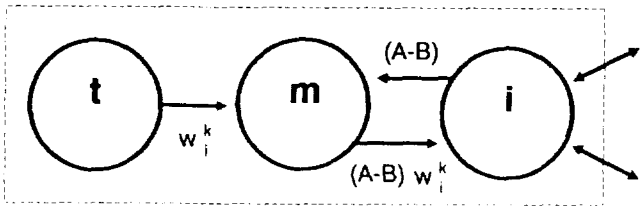


*Figure 4. Butterfly actor.*

A butterfly actor is composed of three primitive actors: *i*, *m*, and *t*. See Figure 4. The main task for the i-actor is to provide a control interface between the rest of the butterfly actor and its counterparts in the shuttle actor. As a secondary task, the i-actor computes *(A +B)* of the butterfly output after forwarding *A,B* to the m-actor. Since the FT-shuttle network is recirculating, the i-actor also retains one of the outputs, either *(A + B)* or $(A\text{-}B)w^k$ depending upon its logical address in the butterfly network. The t-primitive actor provides an active store for the twiddle factors and forwards to the m-actor one constant for each of the *log(N/2)* outputs. The twiddle factors are stored in a *register chain* table using the general purpose registers and are retrieved using *very long shifts*. Although very long shifts are slow (2-bits per cycle), the computations of $w^k$ in the t-actor and (A + B) in the i-actor are completely overlapped by the computation of $(A\text{-}B)w^k$ in the m-actor. The impact of this bottleneck in the butterfly can be reduced if *parallel arithmetic* (Ref. 11) is employed to accelerate the multiplication step. In this scheme, the butterfly actor is reconfigured to perform two 16x32-bit or four 8x32-bit partial multiplications in parallel, then integrate the partial results.

### Shuttle Actor

The shuttle actor is a general purpose, message-processing assembly of simpler, *S* and *N* actors which allow parallel data flow from one input to another output. On behalf of the FT actor, the shuttle actor recirculates the data flow for both butterfly and perfect shuffle rotation networks: the actor reconfigures itself as necessary. The S actor pipelines data generally "southbound" and the N actor pipelines data generally "northbound." Variations on these themes allow data to be transferred in lock-step fashions that are function routing dependent. For example, Figure 5 shows the step-wise data flow for the N = 16 butterfly network. A similar data flow is used for the perfect shuffle rotation network.

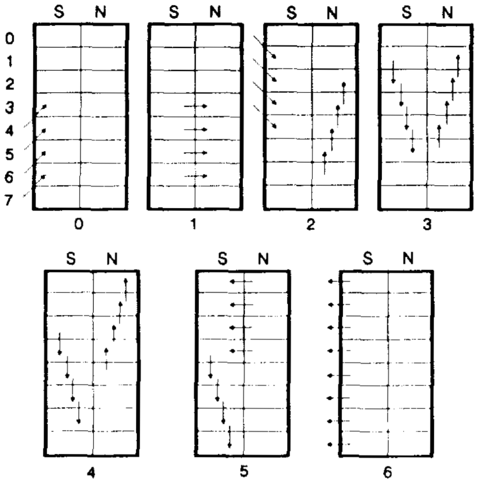


*Figure 5. Data flow for N = 16 butterfly.*

### 32 x 32 Fourier transform

We now present a detailed example of the FT and shuttle actors composed for the two dimensional, 32 x 32 Fourier transform. The actor architecture is presented in Figure 6 which shows the two dimensional signal input of 32 rows to each of 32 pFFT actors. These pFFT actors are 16 butterfly actors wide. Figure 6 also shows the rotation of the signal space for only one row output: namely, the worst-case scenario in which the points (0,31) and (31,0) are transposed. This corresponds to a movement of 480 cells.

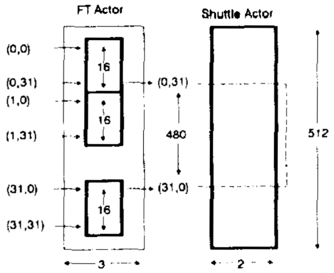


*Figure 6. 32 x 32 two dimensional Fourier transform.*

## PERFORMANCE

In this section, we derive the area and time performance of the multidimensional FT on the hypercomputer and compare these data with existing production systems: specifically, commercial DSP microprocessors and the AT&T Signal Processing Ensemble or ASPEN architecture.

### Area, Time

The area performance for the N-cube signal space is determined by the butterfly actor type (i.e., the configurable degree of parallelism, *q*, used by the built-in subactor multiplier), the shuttle actor area and the *d* dimensional cube volume. These relationships are expressed in Equation 5. This

$$A_{Total} = (q+2)\frac{N^d}{2} \qquad (5)$$

area complexity, *O(N$^d$)* is optimal (Ref. 12). The run-time performance for the N-cube signal space is determined by the pipelined computation time for the 1-dim pFFT ($T_{FT}$) along each dimension and the communication delay in the 1-dim pFFTs ($D_{FT}$) and the rotation ($D_{rot}$). (All times are reported in system clock cycles except where noted otherwise.) The pipelined computation time is given in Equation 6 (where $t_{c(q)}$, measured

$$T_{FT} = t_{c(q)}\, d \log(N/2) \qquad (6)$$

empirically, is the butterfly stage computation time for the *q* version of the butterfly actor). The values for $t_{c(q)}$ are given in Table 1. The total

*Table 1. Empirical values for $t_{c(q)}$.*

| q | $t_{c(q)}$ |
|---|---|
| 1 | 507 |
| 2 | 309 |
| 4 | 177 |

delay, $T_D = dD_{FT} + (d-1)D_{rot}$, is elaborated in Equation 7 (where $t_D = 6$ is the cell-to-cell routing delay).

$$T_D = t_D\left(d\frac{\log^2(N/2)}{2} + (d-1)\frac{N^d-2N}{2}\right) \qquad (7)$$

The above performance model assumes a *k* cell hypercomputer such that $N^d/2 \le k$. However, this number of cells may be impractical for the problem size such that only $N/2 \le k < N^d/2$ cells are available. To illustrate our principle for dealing with this problem we suggest a model for the simpler case of *k = (q + 2)N/2* where *k* is now just large enough to compute the 1-dim FT directly. We employ the global IO network which is able to input data through the planar dimension or the polar dimension in parallel. See Ref. 6 for details. We use this network to pump inputs into the mesh array and carry outputs (e.g., temporary results) away from the mesh array. So, unlike the case where $N^d/2 \le k$, the data flow now recirculates between the 1-dim FT actor to the global IO network. For *k = (q + 2)N/2*, we cycle a single dimension through the k cell array at a time. The new computation time is now given by Equation 8 and the

$$T_{FT} = t_{c(q)}\, d N^{d-1} \log(N/2) \qquad (8)$$

delay, Equation 9 where $t_G$ is the delay steps to input or output a single

$$T_D = t_D\left(d\frac{\log^2(N/2)}{2} + (d-1)(2Nt_G + t_{rot})\right) \qquad (9)$$

datum items and $t_{rot}$ is the delay steps to rotate the signal cube. The final performance of this system will depend upon the architecture of the global IO network.

### Comparisions

We use the above models to compute the performance of our prototype machine with a 200ns clock for the 1-dim 1024 point FT and the 2-dim 32x32 point FT. Note, the 1-dim 1024 point FT is unaffected by $N/2 \le k < N^d/2$. We compare these results with available data in the literature for commercial 32-bit digital signal processors (Ref. 7) and the prototype AT&T Signal Processing Ensemble or ASPEN computer (Ref. 5) with 25 32-bit AT&T DSP32 microprocessors. As can be seen in Tables 2 and 3, the Hypercomputer would excel in the 1-dim FT compared with uniprocessor signal processors and would deliver nearly equivalent performance for the 2-dim FT (for q = 4) compared with the Aspen Processor. This example would easily fit into our machine. Table 3 further shows the effects of serially computing the 1-dim FTs over the two dimensional signal space for "reasonable" values of $t_G$: we assume that the IO and rotation are pipelined so that the rotation is overlapped. Thus, *$t_{rot}$ = 0.*

## CONCLUSIONS

We have considered a complete analysis of the d-dim FT on the hypercomputer architecture. Our main goal has been to highlight implementation and performance and demonstrate that inspite of the machine's proposed simplicity and generality, it delivers competitive response. These results are surprising and may be even counterintuitive yet it underscores what we believe are the delightful possibilities when reconfiguration and massive parallelism are married.

*Table 2. Hypercomputer vs. uniprocessor DSPs.*

| DSP Microprocessor | Clock (ns) | T (ms) |
|---|---|---|
| DSP5600 | 50 | 5.0 |
| TMS320C25 | 40 | 7.1 |
| ADSP2100 | 31 | 7.2 |
| TMS3210 | 50 | 30.0 |
| hypercomputer | | |
| q = 1 | | 0.96 |
| q = 2 | 200 | 0.60 |
| q = 4 | | 0.37 |

*Table 3. Hypercomputer vs. Aspen Parallel Processor.*

| Architecture | # PEs | T (ms) |
|---|---|---|
| ASPEN | 25 | 0.80 |
| hypercomputer | | |
| q = 1 | 2,560 | 1.40 |
| q = 2 | 3,072 | 1.09 |
| q = 4 | 4,096 | 0.88 |
| q = 1, $t_G$ = 32 | 2,560 | 9.37 |
| q = 1, $t_G$ = 10 | | 7.68 |
| q = 1, $t_G$ = 1 | | 6.99 |

## ACKNOWLEDGEMENTS

This work has been funded in part by Applied Concepts, Inc.: we appreciate its generosity in supporting our work. We also thank the staff members of Applied Concepts, Inc., Sally Huns and Maria Gonzalez-Coleman for their spirit of encouragement and assistance, and Haldun Hadimioglu and the Polytechnic Parallel Processing Group for providing insightful technical criticisms and invaluable feedback throughout the evolution of many of the ideas embodied in hypercomputers.

## REFERENCES

1. Cooley, P.M., Tikey, J.M., "An Algorithm for the Machine Computation of Complex Fourier Series", *Mathematical Computation*, April 1965
2. Brigham, E.O., *The Fast Fourier Transform And Its Applications*, Prentice Hall, 1988
3. Chowdary, N., et al, "A High Speed Two Dimensional FFT Processor", *Proceedings of the International Conference on Accoustical, Speech and Signal Processing*, 1984
4. Seales, J., Shively, R., "EMSP: A Data Flow Computer For Signal Processing Applications", *VLSI Signal Processing*, IEEE
5. Gorin, A., et al, "Computing the 2D DFT On The Aspen Parallel Computer Architecture", *Proceedings of the International Conference on Parallel Processing*, 1987
6. Coleman, R., Post, M., Waksman, A., "Hypercomputers: Design And Architecture", *Second Symposium on the Frontiers of Massively Parallel Processing*, Oct 1988 (companion paper)
7. "Digital Signal Processors", *IEEE Micro*, Dec 1986
8. Andrews, W., "DSP Applications Ride the Wave of Floating Point Processing", *Computer Design*, Sept 1988
9. Gertner, I., Shamash, M., "VLSI Architectures for Multidimensional Fourier Transform Processing", *IEEE Transactions On Computers*, Nov 1987
10. Agha, G., *Actors, A Model of Concurrent Computation in Distributed Systems*, MIT Press, 1986
11. Lee, C., "Primary Computational Agents for Parallel Arithmetic", *Department of Electrical Engineering and Computer Science Project Report*, May 1988
12. Hwang, K., Briggs, F., *Computer Architecture and Parallel Processing*, McGraw Hill, 1984

# PARALLEL ALGORITHMS FOR INTERACTIVE MANIPULATION OF DIGITAL TERRAIN MODELS*

E. W. Davis, D. F. McAllister, and V. Nagaraj

Department of Computer Science
North Carolina State University
Raleigh, North Carolina 27695-8206

## ABSTRACT

Interactive three dimensional graphics applications, such as terrain data representation and manipulation, require extensive arithmetic processing. Massively parallel machines are attractive for this application since they offer high computational rates, and grid connected architectures provide a natural mapping for grid based terrain models. This paper presents algorithms for data movement on the MPP in support of pan and zoom functions over large data grids. It is an extension of earlier work that demonstrated real-time performance of graphics functions on grids that were equal in size to the physical dimensions of the MPP. When the dimensions of a data grid exceed the processing array size, data is packed in the array memory. Windows of the total data grid are interactively selected for processing. Movement of packed data is needed to distribute items across the array for efficient parallel processing. Execution time for data movement was found to exceed that for arithmetic aspects of graphics functions. Performance figures are given for routines written in MPP Pascal.

Keywords: interactive graphics, parallel algorithms, MPP, terrain models

## INTRODUCTION

Multiprocessor architectures have been used for several years to meet the demanding computational requirements of interactive, 3D graphics. The computing resources may take the form of specialized hardware that exploits the vector and pipeline suitability of graphics problems.(Refs. 4, 5, 7, 9, and 13). However, there are several architectures which were not designed specifically with graphics applications in mind, but are versatile enough to be used advantageously on the vectorizable nature of the computations (Refs. 1, 2, 3, 6, and 11). This paper focuses on the use of one such machine, the MPP, for a specific graphics problem: representation of digital terrain data and interactive manipulation of corresponding terrain images.

Prior work has shown the feasibility of interactive manipulation of stereo pair images of small terrain models on the MPP (Ref. 10). In the prior work grids of terrain data were 128 by 128 points, exactly matching the dimensions of the MPP and leading to a natural mapping of terrain data to the processing grid. In order to increase the possible applications, it is necessary to implement interactive graphics operations on much larger databases of terrain points. Data structures and algorithms reported in this paper are for pan and zoom functions on larger databases. The work is more completely described in Ref. 8.

## PARALLEL ALGORITHMS

### Data Representation

Grid-based digital terrain models contain an m by n rectangular grid of points $(x_i, y_j)$, $1 \le i \le m$, $1 \le j \le n$, which correspond to longitude and latitude values on the earth's surface. Each grid point has an associated value $z_{i,j}$ which is the elevation above sea level at the point $(x_i, y_j)$.

Typically, a 128 by 128 grid of elevation points is considered to match the MPP's architecture. Elevation points are assigned to processing elements (PEs) in a straightforward way with $PE_{i,j}$ containing the grid points $(x_i, y_j, z_{i,j})$. A grid that just matches the PE array size constitutes a small terrain model and a limited display. In order to examine a different part of the terrain, it is necessary to input a new set of coordinates with elevation points from the host or staging memory. We wish to make a large database available within the array unit at the outset, and be able to display arbitrary parts of the terrain in real-time, under interactive selection control.

The methods described in this paper can handle a terrain database up to size 512 by 512 in the limited 1K per PE memory of the MPP. However, for purposes of illustration, we consider a model with a 4 by 4 array of PEs and an 8 by 8 array of terrain data. That is, there are four data points per PE.

In order to exploit the full parallel capabilities of the MPP it is necessary that terrain data points to be processed be spread across the available PEs. This will require some movement of data within the processing array. A particular storage mapping, shown in figures 1 and 2, is chosen because it supports the movements used in pan and zoom functions. The original 8 by 8 data array, I in figure 1, is reformatted into four 4 by 4 subarrays, A, B, C, and D in figure 2. Data elements are mapped as follows:

$A = \{a_{i,j}\}$ where $a_{i,j} = I_{2i,\ 2j}$;
$B = \{b_{i,j}\}$ where $b_{i,j} = I_{2i,\ 2j+1}$;
$C = \{c_{i,j}\}$ where $c_{i,j} = I_{2i+1,\ 2j}$;
$D = \{d_{i,j}\}$ where $d_{i,j} = I_{2i+1,\ 2j+1}$;
where $0 \le i \le 3$ and $0 \le j \le 3$.

* This work was partially supported by NASA Goddard Space Flight Center through the MPP Working Group.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 |
| 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 |
| 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 |
| 60 | 61 | 62 | 63 | 64 | 65 | 66 | 67 |
| 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 |

Figure 1. Terrain data in scan line order. Array I.

After reformatting, $PE_{i,j}$ will contain data points from the same position in each of the four data arrays. That is, the terrain data points shown in figure 2 as $a_{i,j}$, $b_{i,j}$, $c_{i,j}$, and $d_{i,j}$, collectively called $t_{i,j}$, are mapped onto $PE_{i,j}$.

| | | | |
|---|---|---|---|
| 00 | 02 | 04 | 06 |
| 20 | 22 | 24 | 26 |
| 40 | 42 | 44 | 46 |
| 60 | 62 | 64 | 66 |

Array A

| | | | |
|---|---|---|---|
| 01 | 03 | 05 | 07 |
| 21 | 23 | 25 | 27 |
| 41 | 43 | 45 | 47 |
| 61 | 63 | 65 | 67 |

Array B

| | | | |
|---|---|---|---|
| 10 | 12 | 14 | 16 |
| 30 | 32 | 34 | 36 |
| 50 | 52 | 54 | 56 |
| 70 | 72 | 74 | 76 |

Array C

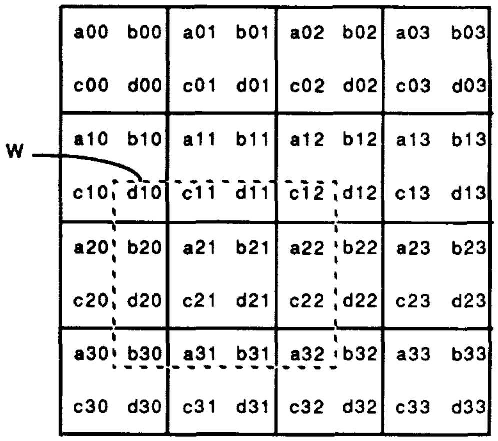
| | | | |
|---|---|---|---|
| 11 | 13 | 15 | 17 |
| 31 | 33 | 35 | 37 |
| 51 | 53 | 55 | 57 |
| 71 | 73 | 75 | 77 |

Array D

Figure 2. Reformatted data, as stored in a 4 by 4 PE array.

## Windows

A subset of terrain points that exactly conforms to the dimensions of the MPP is called a window. Figure 3, where a square corresponds to a single PE, shows that a window only includes data points that are localized to part of the PE array. To exploit the parallelism of the machine, it is necessary to spread the subset of points over the entire PE array such that each PE has one data point from the window. Figure 4 shows the distributed data produced by the "spread" function, described below.



| | | | |
|---|---|---|---|
| a00 b00<br>c00 d00 | a01 b01<br>c01 d01 | a02 b02<br>c02 d02 | a03 b03<br>c03 d03 |
| a10 b10<br>c10 d10 | a11 b11<br>c11 d11 | a12 b12<br>c12 d12 | a13 b13<br>c13 d13 |
| a20 b20<br>c20 d20 | a21 b21<br>c21 d21 | a22 b22<br>c22 d22 | a23 b23<br>c23 d23 |
| a30 b30<br>c30 d30 | a31 b31<br>c31 d31 | a32 b32<br>c32 d32 | a33 b33<br>c33 d33 |

Figure 3. Packed data with a window selected.

| | | | |
|---|---|---|---|
| d10 | c11 | d11 | c12 |
| b20 | a21 | b21 | a22 |
| d20 | c21 | d21 | c22 |
| b30 | a31 | b31 | a32 |

Figure 4. Distribution of the selected window over the array.

## The "Spread" Function

This function consists of a series of bit plane data movement operations. All statements in this routine are executed simultaneously on all PEs. Two data movement masks NSMASK and EWMASK are created. MPP Pascal (Ref. 12) primitives such as "rotate" and "any" are used in conjunction with these data planes to route the selected data to the target PE. The "spread" routine is invoked four times so that elements of A, B, C, and D enclosed by the window can be moved to their target positions, one array at a time. Parameter SOURCE is the particular source array; either A, B, C, or D. Parameter INDEX is the position of the upper left corner of the window when it is packed in the array.

```
routine SPREAD
        MAKEMASK(A, B, C, D, SOURCE, INDEX, NSMASK,
        EWMASK);
        where (NSMASK ≠ 0)
                rotate EWMASK and SOURCE to positions
                indicated by NSMASK;
        end where;
        where (EWMASK ≠ 0)
                rotate SOURCE to position indicated by
                EWMASK;
        end where;
end routine.
```

Routine SPREAD calls the following routine, MAKEMASK.

```
routine MAKEMASK (A, B, C, D, SOURCE, INDEX, NSMASK,
EWMASK);
    for PEs outside the selected window
            NSMASK = 0;
            EWMASK = 0;
            SOURCE = 0;
    end for;
    for PEs inside the selected window
            depending on INDEX
                    SOURCE = A or B or C or D;
            COMPUTEMASK (NSMASK, EWMASK);
    end for;
end routine.
```

Routine MAKEMASK calls routine COMPUTEMASK.

```
routine COMPUTEMASK (NSMASK, EWMASK);
    If $t_{i,j}$ in $PE_{i,j}$ needs to be moved to $PE_{x,y}$
            NSMASK = x - i;
            EWMASK = y - j;
    end if;
end routine.
```

Once data has been spread over the entire array, further functions such as intensity calculations, hidden surface removal, and rendering, can be executed in parallel with each PE handling one data point as in Ref. 10.

### Pan and Zoom Functions

We are interested in the ability to move the window about in the array of terrain point sets. This process of moving a window in object space is called "panning". The routine that follows provides the pan function as simply a selection of the window to be spread. This routine is used prior to computation of intensities and image rendering.

```
routine PAN ( ORIGIN);
    choose window based on user-defined ORIGIN;
    SPREAD;
end routine.
```

Another means of examining the entire database is to sacrifice resolution for extent of coverage. By choosing representative data points from the database it is possible to zoom-in or zoom-out using the routine below. The spacing between adjacent chosen points determines the resolution or extent of zoom. For our small example there are only two zoom settings. Maximum resolution is achieved by choosing a window and carrying out a SPREAD. For minimum resolution, it is possible to simply select one of the four arrays A, B, C, or D. With greater terrain data packing factors, intermediate levels of resolution,, involving different extents of data movement, are possible.

We note again that once data has been spread over the entire array, further functions such as intensity calculations, hidden surface removal, and rendering, can be executed in parallel.

```
routine ZOOM (ORIGIN, RESOLUTION, INDEX);
    if RESOLUTION = MAX
            choose window based on user-defined ORIGIN;
            SPREAD;
    end if;
    if RESOLUTION = MIN
            depending on INDEX
            FINAL = A, or B, or C, or D;
    end if;
end routine.
```

## TIMING ANALYSIS

Graphics programs were written in MPP Pascal (Ref. 11) for a database of 256 by 256 terrain points. Execution time can be determined using system provided timing routines. The structure of a typical graphics program loop with pan and zoom operations is to distribute the data points from the selected window over the entire array, then compute intensities of all pixels in parallel, then render the image. Table 1 gives actual timing measurements

| | |
|---|---|
| distribute data | 2283 milliseconds |
| compute intensities | 5 milliseconds |
| render image | 819 milliseconds |

Table 1. Measured timing for a 256 by 256 database.

The time taken to distribute data is almost entirely accounted for by 28 calls to the SPREAD routine. It is invoked seven times for each of the four arrays A, B, C, and D discussed earlier. Moreover, the time taken to execute SPREAD once is almost entirely accounted for by an inner loop which uses the MPP Pascal "rotate" function extensively. The measured time for one "rotate" is 204 microseconds. The time taken just for executing "rotate" functions while distributing data is 2172 milliseconds. SPREAD is also used in rendering an image and contributes greatly to its execution time.

An equivalent to the "rotate" function can be achieved in lower level languages of the MPP in 3.3 microseconds, rather than the 204. Table 2 is derived from the measured timing by substituting the much lower rotate time.

| | |
|---|---|
| distribute data | 142 milliseconds |
| compute intensities | 5 milliseconds |
| render image | 53 milliseconds |

Table 2. Expected timing with efficient "rotate".

A necessary condition for real-time graphics image generation is that one pass through the .loop of the program must take no more than 33 milliseconds. Even with the expected time table, each pass takes 200 milliseconds, yielding only five frames per second.

An alternative approach is to bypass data distribution in favor of iteratively using a smaller portion of the processing array.

Image generation time using this approach, for the same database as above, was measured at 73 milliseconds. Execution time is reduced but full parallelism of the array is not used. As the number of data points in the database is increased by a factor of K, the number of active PEs in the array unit is decreased by K. This will result in a factor of K increase in time for the intensity computation alone.

A third approach is to maintain the database in the staging memory and bring in only the data points needed for each computation. Image generation time reduces to 59 milliseconds. However, data in the staging memory is only accessible along certain predefined boundaries. This complicates pan and zoom functions.

## CONCLUSION

Prior work has shown massive parallelism to be suitable for graphics applications on data arrays which fit the processing array size. When larger arrays must be handled, the time involved in moving data becomes the dominant part of the problem and can take the performance out of the real-time realm.

## REFERENCES

1. Batcher, K. E., "Design of a Massively Parallel Processor", *IEEE Trans. on Computers*, Vol. C-29, No. 9, Sept. 1980, pp. 836-840.

2. Davis, E. W., and J. H. Reif, "Architecture and Operation of the BLITZEN Processing Element", Proc. of the Third Int. Conf. on Supercomputing, Boston, MA, May 1988.

3. Blevins, D. W., E. W. Davis, R. A. Heaton, and J. H. Reif, "BLITZEN: A Highly Integrated Massively Parallel Machine", Proc. of the Second Symposium. on the Frontiers of Massively Parallel Computing (this proceedings), Fairfax, VA, October 1988.

4. Fuchs, H., et al, "Fast Spheres, Shadows, Textures, Transparencies and Image Enhancements in Pixel-Planes", SIGGRAPH, Vol. 19, No. 3, July 1985, pp. 111-120.

5. Glassner, H., and H. Fuchs, "Hardware Enhancements for Computer Graphics", *Fundamental Algorithms in Computer Graphics,* R. A. Earnshaw (Ed.), 1985, pp. 631-658.

6. Hillis, D. W., *The Connection Machine,* The MIT Press, Cambridge, MA, 1986.

7. Levinthal, A., and T. Porter, "CHAP - A SIMD Graphics Processor", *ACM Computer Graphics*, Vol. 18, No. 3, July 1984.

8. Nagaraj, V., *Graphics Algorithms for Parallel Architectures,* M.S.Thesis, Dept. of Electrical and Computer Engineering, North Carolina State University, 1988.

9. Niimi, H., Y. Imai, M. Murakami, S. Tomita, and H. Hagiwara, "A Parallel Processor System for Three Dimensional Color Graphics", *ACM Computer Graphics,* Vol. 18, No. 3, July 1984.

10. Pol, S., D. F. McAllister, and E. W. Davis, "An Application of the MPP to the Interactive Manipulation of Stereo Images of Digital Terrain Models", Proc. of the First Symposium on the Frontiers of Massively Parallel Scientific Computing, NASA/GSFC, October 1986.

11. Potter, J., Editor, *The Massively Parallel Processor*, The MIT Press, 1985.

12. Science Applications Research, MPP Pascal Programmer's Guide, March 1988.

13. Stellar Computer, Inc., *Stellar Graphics Supercomputer Model GS1000 System Overview,* 1987.

# SORT COMPUTATION

John E. Dorband

NASA/Goddard Space Flight Center/635
Greenbelt, MD 20771

## ABSTRACT

Sorting has long been used to organize data in preparation for further computation, but sort computation allows some types of computation to be performed during the sort. *Sort aggregation* and *sort distribution* are the two basic forms of sort computation. Sort aggregation generates an accumulative or aggregate result for each group of records and places this result in one of the records. An aggregate operation can be any operation that is both associative and commutative, i.e. any operation whose result does not depend on the order of the operands or the order in which the operations are performed. Sort distribution copies the value from a field of a specific record in a group into that field in every record of that group.

Keywords: Sorting, Aggregation, Distribution, SIMD, Massively Parallel, Data Parallel, MPP, Routing.

## INTRODUCTION

Sort computation uses sorting as a control mechanism to support interspersed routing and data manipulation. Sort computation is performed on sets of records, grouped according to a key contained in each record. Groups of records contain only records that have been determined to be equal by some function. The sort computation technique which has been developed here is simple. View a sorting algorithm as having two parts — the comparison of records and the routing of records. The comparison determines if the two records are in the correct order. Routing takes this result and determines where each of the records is to go next. Thus, the sort contains a routing and a comparison routine, where the routing routine calls the comparison routine when necessary. All sort algorithms, such as merge sort, bubble sort, and bitonic sort, consist of these two parts. Sort computation can use the routing part of any sort algorithm. The routing routine only determines the order in which the records finally line up after the sort is through — not how they are modified. The comparison routine, on the other hand, is replaced with a comparison routine whose nature depends on the type of sort computation it is to perform. The comparison routine contains the code that determines how the contents of the records are changed. The comparison routine has two functions. One function is to determine if the two records being compared are in the same group (generally whether or not their keys are equal), whether a record from one group will come before or after a record from another group, and in some cases if the sort is complete. The other function is to modify the records if they both belong to the same group.

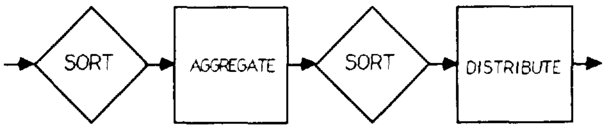


Figure 1. Conventional use of sorting to organize data in preparation for computation.

Sorting has long been used to organize data in preparation for further computation (Figure 1), but sort computation allows some types of computation to be performed during the sort (Figure 2). *Sort aggregation* and *sort distribution* are the two basic forms of sort computation. Sort aggregation generates an accumulative or aggregate result for each group of records and places this result in one of the records. Usually, it is placed in the last record or the one with the largest key value. An aggregate operation can be any operation that is both associative and commutative, i.e. any operation whose result does not depend on the order of the operands or the order in which the operations are performed. Addition, multiplication, AND, OR, and EXCLUSIVE-OR are examples of valid operations. Sort distribution copies the value from a field of a specific record in a group into that field in every record of that group. The record that contains the value to be distributed contains a flag that is set to true. Note

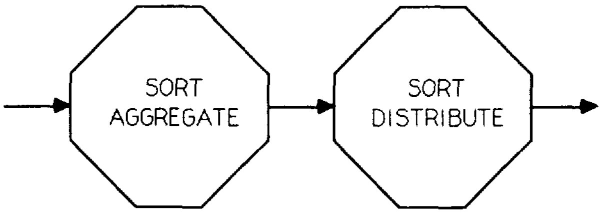


Figure 2. Sort computation allows some types of computation to be performed during the sort.

that there may be more than one record in a group in which the flag is set, as long as all such records contain the same value.

## SORT AGGREGATION

Sort aggregation is described here with pseudo code and a proof is given to show that sort aggregation computes the aggregate result for each group of records within the set of records being sorted. The expression "A[5].(B,C,D)" defines an array of 5 records, where each record has 3 fields, B, C, and D. The terms, sum or summing, are used as the generic terms for finding the result of a valid aggregation operation. Thus, the command "SORT(SUM,A)" performs the sort sum over the array A defined by "A[n].(K,V)". Note that the sum operation can be replaced by any other valid aggregate operation.

SUM (Figure 3) is the comparison routine that will, when used in conjunction with a sort routine, sum all the values in field V of the records for which the K fields are equal. SUM returns a value of true if the records A1 and A2 are in the correct order, and false if they are not. SUM puts the sum of all the V fields of records of the same group in the last (or largest) record in the group.

```
boolean function SUM(A1,A2)
        given A1.(K,V)
        given A2.(K,V)
        if  A1.K = A2.K then
                A1.V = 0
                A2.V = A1.V + A2.V
                return(true)
                end if
        if A1.K < A2.K then
                return(true)
                end if
        if A1.K > A2.K then
                return(false)
                end if
end function
```

Figure 3. SUM routine.

The proof that aggregation works as described goes as follows. Even though the keys of the records being compared may be equal, SUM can affect their ordering by returning the response to the routing routine that the records are in the correct order (true) or not (false). This in effect gives order within a group. SUM always designates the record that contains the result of the sum as the larger of the two records, the larger contains a value of zero. This means that the sum of the value fields of the group's records will be contained in the record that was designated larger than all others. Assume, however, that not all values of records in a specific group were summed into the same record. This means that at least two records contain only part of the result for that group. Each one of these records would have been designated greater than all records of that group. Yet, the records that contained partial results must not have been compared to any others or the partial results would have been summed into it. Thus, each record would have been designated the largest in the group. Because only one record is the largest of a group, there can only be one record that contains the result for any group.

A comparison routine such as SUM can be written for any operation that is both associative and commutative, as described previously.

## SORT DISTRIBUTION

Sort distribution is slightly more complex than sort aggregation and is constrained somewhat compared to sort aggregation. The constraint stems from the fact the result of a sort distribution must be migrated to all the members of a group of records while the result of a sort aggregation only needs to migrate to one record of a group of records. This constraint will be clarified further after the proof.

```
boolean function COPY(A1,A2)
        given A1.(K,F,V)
        given A2.(K,F,V)
        if  A1.K = A2.K then
                if A2.F then
                        A1.V =  A2.V
                        A1.V = true
                        return(true)
                        end if
                if A1.F then
                        A2.V =  A1.V
                        A2.V = true
                        return(true)
                        end if
        else
                return(true)
                end if
        if A1.K < A2.K then
                return(true)
                end if
        if A1.K > A2.K then
                return(false)
                end if
end function
```

Figure 3. COPY routine.

The idea in sort distribution is to copy the value of a record in a group of records, which has been flagged as having a valid value for that group, to all records that do not already have that value. The command to perform this is "SORT(COPY,A)", where SORT is a routing routine, COPY is a comparison routine, and A is an array of records. This array of *n* records is of the form "A[n].(K,F,V)", where K is the key, F is the valid value flag, and V is the value field. COPY used in conjunction with SORT distributes the flagged value in each group to all members of the group (see Figure 4). Like SUM, COPY returns a value of true if the records A1 and A2 are in the correct order, and false if they are not. COPY puts the same value in all records of the same group, or no value at all if no record of the group had its valid value flag set prior to performing the distribution.

The proof that distribution can be accomplished during a sort is similar to that of aggregation. Note that when two records are determined to be in the same group, and one of the records contains a valid value, it is copied to the other record and its valid value flag is set. This, in effect, causes the record with a valid value to be considered both larger and smaller than a record that *does not have a valid value*. Thus, at the completion of the sort computation, at least the largest and smallest record of each group that had a record with a valid value will contain a valid value. Assume that a record without a valid value remained after the sort was completed. If it was either the largest or the smallest record of the group, then no other record in the group had a valid value. If it was not the smallest or the largest value of the group, either there was no record in the group with a valid value, or it was not compared to a record in the group with a valid value. If there is a record without a valid value and one with a valid value in the same group, such a pair exists logically next to each other and has never been compared. If such a pair exists, there is no way of knowing which one is larger, since they have never been compared. Thus, the sort must not have been completed. Therefore, a record can only be left without a valid value if there are no records in its group with a valid value when the sort is complete.

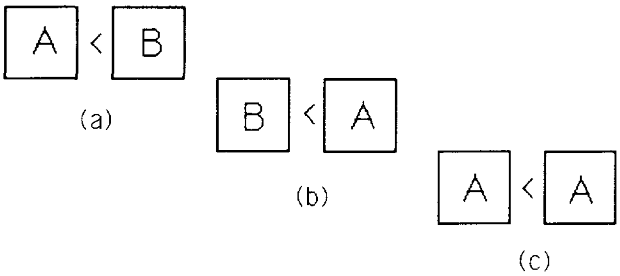


Figure 5. (a) Record A is smaller than B.
(b) Record A is larger than B.
(c) Record A is both larger and smaller than

The statement in the proof written in italics is the key to whether a sort algorithm can be used to perform sort distribution. The distribution record must be seen as being both larger and smaller than the replaced record. Figure 5(a) shows the order of records A and B if A is smaller than B, 5(b) shows the order if A is larger than B, but in 5(c) A appears to be both larger and smaller than B by replacing B with A. If the solution to the fact that two records are out of order is simply that they need to be swapped, then A may be made to appear to be both larger and smaller than B by simply replacing B with A. This is the case with *merge*, *bubble*, and *bitonic sorts*, for example. But this is not the case however with *insertion sorts* that use a log *n* time insertion. In an insertion sort, one of the records being compared has already found its position in the list. Therefore, it is not the case that if the records being compared are out of order, they are simply swapped. Such a sort may be extensively modified to support sort distribution, but it might be more effective to just use a sort that needs no modification.

## GENERALIZATION OF SORT COMPUTATION

Sort computation requires that records of data be grouped according to some criterion. Order merely forces this grouping to occur. Thus, any function that causes the desired grouping may be used to perform the comparison part of the sort. The function used for comparison must evaluate to one of three results, *less than*, *greater than*, or *equal to*, depending on the two records that are being compared. The *data values* need not literally be less than, greater than, or equal to, as long as the end result is an unambiguous ordering that causes the desired grouping of records that are designated as equal.

Records can be grouped, for instance, as a set of non-overlapping ranges. In this case, the conditions of the comparison function would be lower than the minimum of the range, higher than the maximum of the range, or within the range. Range ordering uses two types of records — records whose keys are ranges and records whose keys are single values. Note that in the case of in-range ordering, once a record is determined to be in-range, not only must the appropriate action be performed on its aggregation or distribution fields, but the key field of the in-range record must be modified so that it becomes a range key rather than a single-value key record.

## AGGREGATE DISTRIBUTION

Aggregate distribution differs from aggregation in that all member records of a group obtain the results of the aggregation instead of just one member. It uses a sort algorithm that is made up of merge steps, because the flag field must be set between each merge step. It is not known if aggregate distribution will work for sorts that are not made of merge steps. The following describes how a merge aggregate distribution is performed.

Start with two sorted lists of records $A_i$ and $B_j$, where i=1...n and j=1...m. Each record contains a 2-bit flag. The flags of records in list A are set to 1 and the flags of records in list B are set to 2. During the merge, if two records are determined to be in the same group and one record's flag is 1 and the other's is 2, then the aggregate function is performed, both records are given the result, and their flags are set to 3. If one record's flag is 3 and the other's is not, then the aggregate result contained in the record with the flag value of 3 is copied to the other record. Otherwise, if both records' flags are the same, nothing is done to either record. When the merge is done, all records within a group have the same aggregate results.

## OPTIMIZATION OF SORT COMPUTATION

Sorting is generally a very time consuming function, particularly on a single processor machine. However, on a multiple processor machine such as the Massively Parallel Processor[1] (MPP), a sort of 65536 records of 32 bits each takes about 29 milliseconds, and a sort of 512K records of 32 bits each takes about 1 second. This is very fast but still time-consuming if it is meant to be used very often, as may be the case with sort computation. The time needed to perform the *necessary* sort computation can be minimized in several ways — in either hardware or software. Hardware can be improved by decreasing the interprocessor communication time or using a more far-reaching or elaborate processor interconnection scheme such as a complete hypercube. Hardware improvements to sorting or sort computation will not be discussed here. However, methods to improve sort computation performance through the use of prudent software design and programming techniques have been developed by the author and are discussed next.

A bitonic sort has been implemented by the author on the MPP. This sort and most other sorts, require log *n* merge steps, each merging two sorted lists into one sorted list. Time can be saved during sort computation by performing a merge only if certain conditions are met: the records in each list must be in an order and have values consistent with the result of a sort computation performed on each list. An example of this is the use of sort computation for table look-up. The table can be sorted once before it is used. Therefore, it need only be *merged* with sorted data records when its values are to be distributed to the data records, rather than having to be *sorted* into the data records.

If it is necessary to extract table records out from among the data records to complete a table look-up, the table records could be sorted using a major key that distinguishes them from data records. This, however, defeats the use of the merge to combine the two record types because the sort takes so much longer than the merge. The records can be unmerged in no more time than it takes to merge them by leaving a "trail of corn", so to speak. During the merge, a set of log *n* bits in each processor is used to record whether or not the pair of records in that processor are exchanged during each of the log *n* comparison steps of the merge. This set of bits is then used during the unmerge operation to route the records back to their original locations.

Another means of reducing the time spent in sort computation is to, at times, perform only partial or local sorting of the data. This has shown to be useful during image registration[2] when records are being generated whose values need to be accumulated. Each original pixel in the image is divided into much smaller subpixels. These subpixels carry a fraction of the original pixel's value and a calculated new position. The subpixel values are then summed into their new pixel's value using sort summing. Since the subpixels are likely to be summed with nearby subpixels, many small local sort sums are performed to accumulate as much as possible locally before sort summing across the entire image. This saves space in the processor memory, as well as saving time.

Partial sorting can also be used when a table look-up needs to be performed and the size of the table is much smaller than the number of data records to which the table information is to be distributed. In this case, multiple copies of the table are distributed across the processors, allowing the use of smaller sort distribution operations confined to local areas of the array of processors.

Sort computation can be made faster simply by using a faster sort algorithm. This is interesting because where the records were before the sort and where the records end up after the sort is irrelevant. This allows sorts to be used that leave the records in unusual orders, such as snake row major or shuffle row major, if they are faster.

In the case of sort distribution, if it is known that all records either contain a value or will obtain a valid value during the sort operation, a check can be performed after every comparison step to see if the sort distribution has been completed. Thus, the sort distribution may be terminated before the sort is actually complete.

To extend this concept one step further, it may not be necessary for any arbitrary record to obtain the value it is looking for in any given invocation of the sort computation. Therefore, many local sort operations may be performed to get some local sort computation done quickly between successive complete sort operations. This brings up an issue for further study: can the keys used in the sort operations be generated for records that are created between sort computation operations, so as to minimize the number of complete sort computation operations that need to be performed.

## AN EXAMPLE OF SORT COMPUTATION

Multiplication of a sparse matrix times a vector is now presented as an example of sort computation. This is presented as an iterative refinement of the vector V ( $V_{i+1} = M*V_i$ ). The form of the record used is "T.(R,C,M,V)", where T has four fields: the row R, the column C, the matrix coefficient at row R and column C, and the vector coefficient at position C of the vector. To perform a matrix multiply, first multiply M times C in each record, giving new record values "T.(R,C,M,V=M*V)". A sort sum operation is performed using R as the key and summing over M*V. This leaves one record for each R which contains the value of the new vector at position R. At this point the matrix multiply is complete, but if further iterations must be perform the new vector coefficients must be distributed so the value of V corresponds to the value of C, not of R. This is done by making another set of records "$T_1$.($R_1$=C,$C_1$=R,$M_1$=M,$V_1$)" which contains a record for every record in "T". $V_1$ has been given no value yet. Then form a set of records that is the union of T and $T_1$. A sort distribution is performed using R as the key and distributing the values of V from T to $T_1$. All record of T are deleted and a new set of records for T are created of the form "T.(R=$C_1$,C=$R_1$,M=$M_1$,V=$V_1$)" from the records of $T_1$. Another matrix multiply may now be performed since the values of V correspond to the columns of M.

## VIRTUAL LOCALITY

*Virtual memory* and *virtual processors* have become common concepts. The concept of virtual memory allows the programmer to imagine that there is as much memory as needed, alleviating the need to account for physical memory constraints in designing a program. It also allows him to imagine that he has complete control of all physical memory. This concept is used in most large computers, minicomputers, and the newest 32-bit microcomputers. The concept of virtual processors allows the programmer to view a problem as though it was executing on as many processors as needed, yet it may be using fewer processors[3].

The key point here is that virtual memory addresses are not physical addresses, but appear to be, and virtual processors are not real processors, but also appear to be. The programmer must still deal with addresses and fixed locations of data and the knowledge that he is using one or several processors. The programmer still has to deal with a hardware view of his computational environment, that of memory and processing units, in spite of the fact that it is a virtual hardware view. The author has developed the concept of *virtual locality* to move the programmer further away from hardware architecture concerns and closer to the perception of a computationally pure environment. This is especially important, as well as particularly feasible, when it comes to massively parallel architectures, such as the MPP.

Virtual locality views data in computational units of records. Computations are carried out on the data of these records according to the groups to which the records belong (their locality). These records contain fields, as in any traditional view of data records. Groups can contain any number of records. Records are grouped according to any number of schemes, based on field values. Because all computation is dependent only on the values within the records and the interrelationship of those values, the computation is independent of the record's location in the computation environment before, during, and after the computation and, it is also independent of the number of processors used to accomplish the computation. This differs from the view of computation in other massively parallel architectures that use more traditional routing schemes, or, for that matter, any computer architecture that depends on pointers or fixed addresses to direct data to and from specific locations in the environment. Virtual locality facilitates position-independent computation. It only matters that the appropriate data comes together sometime during the computational step. Generalized routing schemes and traditional memory addressing schemes require that data is placed where it can later be found. Therefore, it has to be allocated space and can only be moved after all places that refer to it have been changed. This makes dynamic allocation, reallocation, de-allocation and garbage collection difficult, if not impossible in some circumstances. Position dependent computation is used in the implementation of virtual locality, but is not seen by the programmer. With virtual locality, records of data may be created and deleted at will without allocating them to specific locations in the environment. Virtual locality is possible through the use of the sort computation concept[2]. Sort computation defines the types of operations supported under virtual locality and describes how they are implemented.

## APPLICATION OF SORT COMPUTATION

Currently, image rotation, image registration, and computer graphic generation by ray tracing have been implemented by the author on the MPP using sort computation techniques. Three-dimensional rendering of elevation maps has also been implemented on the MPP using these techniques by a NASA summer student, Jennifer Trainer, under the direction of the author. However other applications exist that require the processing of irregular arrangements of data. For example, the implementation of pure LISP, which was designed and implemented by Tim Busse of Science Applications Research and the author, requires this capability.

The pure LISP is implemented by distributing the pointer pairs that make up the LISP data structure across the processors of the MPP. Sort computation is used to bring the pointer pairs together according to the functions that must be performed on them, such as the creation of a new pointer. The basic functions of pure LISP were implemented (i.e., CAR, CDR, CONS, EQ, ATOM, COND, APPLY, EVAL, EVLIST, and LAMBDA). The MPP ray tracing approach[4] is based on an algorithm that finds the intersections of light rays and objects in a 3-dimensional space. It is done by recursively subdividing space. Records are created that keep track of whether a specific ray or object intersects a subdivision of space. If a subdivision of space is not intersected by both a ray and an object, all records associated with it are deleted. Sort computation is used to determine where this condition is true. These two applications have been implemented on the MPP using MPP Parallel FORTH.

## CONCLUSION

Future plans in the area of application of sort computation include the study of its use on data bases and for implementation of a compiler inside the MPP array. Virtual locality is worthy of further study also because it allows the simultaneous development of parallel algorithms and hardware architectures, requiring only a minimal amount of effort to port and test previously developed algorithms on new architectures. Sort computation is a feasible means of facilitating virtual locality. As with other virtual concepts, care must be taken, while knowledge about it's effective use and implementation in both software and hardware develops.

## REFERENCES

(1) *The Massively Parallel Processor*, J.L. Potter, ed., ISBN: 0-262-16100, The MIT Press, Cambridge, MA, 1985.

(2) Dorband, John E., *Sort Computation and Conservative Image Registration*, Ph.D. thesis, Pennsylvania State Univ., December 1985.

(3) Hillis, W. Daniel, *The Connection Machine*, ISBN: 0-262-08157-1, The MIT Press, Cambridge, MA, 1985, p. 135.

(4) Dorband, John E., *3-D Graphic Generation on the MPP*, Proceedings of the 2nd International Conference on Supercomputing, Vol. II, pg 305-309, 1987.

# PARALLEL FREQUENCY DOMAIN ADAPTIVE LINE ENHANCER

Mohamed El-Sharkawy    Maurice Aburdene
Electrical Engineering Department
Bucknell University
Lewisburg, PA 17837

Space Systems Technology Department
Naval Research Lab
Washington D.C. 20375-5000

## ABSTRACT

This paper presents a comparative study of three parallel implementations of frequency domain least mean square adaptive line enhancement algorithms using parallel computers. Two models of the Connection Machine[+] (CM-1) and (CM-2) and one model of the Balance Machine[++] (Balance 8000) were used in this study. Simulation results, speedup factors and estimated spectra for algorithms based on circular and linear convolution are presented.

## INTRODUCTION

Recently considerable interest has focused on adaptive line enhancement applications in communications, radar, sonar and control. The adaptive line enhancer is an algorithm that detects and tracks a moving spectral line in broadband noise while enhancing the signal to noise ratio [7]. The algorithms can be implemented in both time [1-2] and frequency [3-6] domains. Frequency Domain Adaptive Line Enhancers (FDALEs) are used for three important reasons. First, the structure of frequency domain implementation is more suitable for parallel processing than time domain implementation. Second, the amount of computation required to process a fixed amount of data can be greatly reduced compared to time domain implementation. Third, the convergence properties of the frequency domain process can be greatly improved compared to time domain process. Furthermore, with the continuing development of parallel processing architectures and low-cost large scale integrated circuits, the FDALE will become increasingly attractive.

This paper presents parallel implementations of three least mean square FDALE algorithms based on circular [3] and/or linear convolution [5-6] of the filter delayed input and its impulse response. The first algorithm is based on circular convolution only. The second algorithm is a constrained least mean square algorithm that performs strictly linear convolution. The third algorithm is an unconstrained least mean square algorithm that allows either linear or circular convolution, whichever best minimizes the mean square error. The Connection and Balance Machines were used to simulate these FDALEs. The Connection Machine (CM) is an array of processors arranged in a hypercube network [9], [14]. This machine is classified as a single-instruction multiple-data machine (SIMD) using Flynn's classification [12]. A typical CM is composed of 16K to 64K processors. In data parallel applications, each data element is assigned to an individual processor. In cases where the application requires more processors than are available, a virtual processor mechanism is used to simulate additional processors. The CM is used in conjunction with a serial computer such as a Symbolics Lisp Machine or a Digital Equipment Corporation Vax. The serial computer is the front end for the CM. The front end computer sends instructions to the CM. CM programs run on the front end machine in dialects of C, Lisp and Fortran 8X. Typically, a program's data resides on both the CM (parallel data) and on the front end (serial data). All code resides on the front end. The results presented in this paper were obtained using the *LISP. *LISP is a dialect of Common LISP with extensions for parallel data and functions.

The Balance Machine, a multiple-instruction multiple-data machine (MIMD), can be configured with as many as thirty 32-bit Intel 80386 microprocessors and takes advantages of up to 1-M bit dynamic random access memories, several custom VLSI chips, and a particularly caching scheme to execute up to 81 millions instruction per second with a relatively low cost [11]. The user specifies the number of processors used to implement his algorithm. The results presented in this paper were obtained using Balance 8000/21000 Fortran.

In the rest of this paper, the three FDALE algorithms are described and comparative study, simulation results, speedup factors and estimated spectra using the Connection and Balance Machines are presented.

## FDALE ALGORITHMS

### Algorithm 1:

This algorithm performs circular convolution of the delayed version of the input signal and the adaptive filter impulse response [3]. The input signal d(n) and the delayed version of the input

[+] Connection Machine is a trademark of the Thinking Machine Corporation.
[++] Balance Machine is a trademark of the Sequent Computer Systems.

CH2649-2/89/0000/0143$01.00 © 1988 IEEE

signal x(n) are accumulated to form N-point data blocks (Figure 1), where

$$x(n) = d\ (n - \Delta) \qquad (1)$$

where $\Delta$ is chosen so that d(n) and x(n) are uncorrelated. These N-point data blocks of x(n) and d(n) are then transformed by N-point Fast Fourier Transforms (FFTs) to obtain X(k) and D(k) respectively, where

$$X(k) = [X_1(k)\ X_2(k)\ \dots\ X_N(k)]^T$$

$$D(k) = [D_1(k)\ D_2(k)\ \dots\ D_N(k)]^T$$

and T indicates transpose.

The input response transform values $D_i(k)$ are subtracted from the values of the product of the delayed input response transform $X_i(k)$ and the weights at corresponding frequencies $H_i(k)$ to form N error signals $E_i(k)$. Each weight is independently updated once for each data block according to the following complex LMS algorithm introduced by Widrow et al. [4]:

$$H_i(k+1) = H_i(k) + \mu\ E_i(k)\ X_j{}^*(k) \qquad (2)$$

where $\mu$ is a constant that determines the rate of convergence and the stability of this adaptive process and $X_j{}^*(k)$ is the conjugate of $X_j(k)$.

The weighted outputs are fed to an inverse FFT transform to produce the output signal y(n). The first algorithm is the simplest one. Unfortunately, the use of circular convolution transforms a linear time invariant filter into a periodic time varying filter whose output is periodically nonstationary for a stationary input [8].

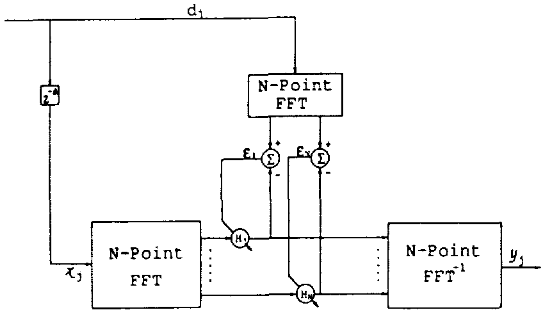


Figure 1. First Algorithm Block Diagram

**Algorithm 2:**

This algorithm produces strictly linear convolution of the delayed input signal x(n) and the adaptive filer impulse response h(n) [5] (Figure 2). This algorithm uses the overlap-save method with 50% overlap [10]. Thus, the weights are padded with N zeroes and both the (k-1)th and kth N-point delayed input blocks are used. These 2N-point data blocks are then transformed by 2N-point FFTs, where

$$H^T(k) = \text{frequency domain weight vector}$$

$$= \text{FFT}\ [h^T(k)\ 0\ \dots\ 0] \qquad (3)$$

$$X(k) = \text{diag}\ \{\text{FFT}[(k-1)\text{th and kth delayed input blocks}]\}$$

$$= \text{diag}\ \{\text{FFT}[x(kN-N)\ \dots\ x(kN-1)\ x(kN)\ \dots\ x(KN+N-1]\} \qquad (4)$$

The frequency domain weight vector update equation is

$$H(k+1) = H(k) + \mu\ \text{FFT}\ [\nabla(k)\ 0\ \dots\ 0]^T \qquad (5)$$

where

$$\nabla(k) = \text{first N terms of FFT}^{-1}\ [X^*(k)\ E(k)] \qquad (6)$$

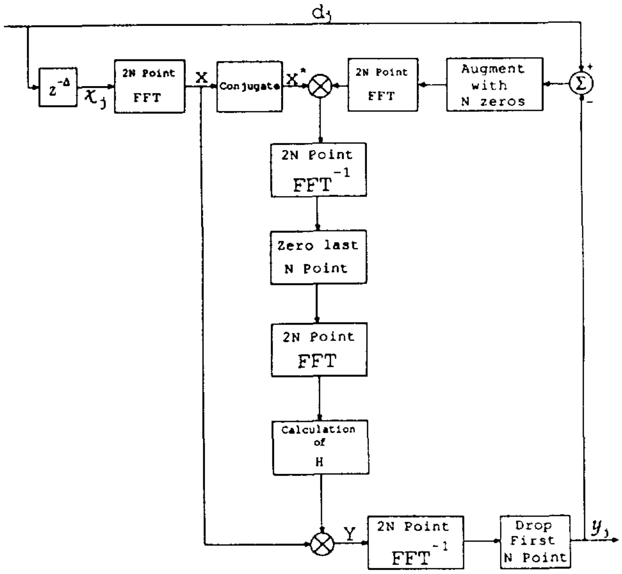


Figure 2. Second Algorithm Block Diagram

Algorithm 3:

This algorithm performs either linear or circular convolution, whichever minimizes the mean-square error [6] (Figure 3). This algorithm reduces the number of FFTs per processed block from five to three. The extra two FFTs are needed for the second algorithm to impose a time-domain constraint in which the last half of the time domain weights are forced to zero to implement strictly linear convolution between the delayed filter input and the filter impulse response. Allowing the filter the freedom to implement circular or linear convolution could reduce the mean square error in cases of large Δ [8]. The frequency-domain weight vector update equation is:

$$H(k+1) = H(k) + \mu X^* (k) E(k) \qquad (9)$$

It has been shown [3-6] that when implementing the time and frequency domain algorithms for single-instruction single-data (SISD) machines, the ratios of the real multipliers of the first, second and third FDLMS algorithms to the real multipliers of the time domain LMS are $(3N\log_2(N/2)+4N)/2N^2$, $(5(\log_2 N)+4)/N$ and $(3(\log_2 N)+4)/N$ respectively. For N=1024, These ratios are 0.015, 0.053 and 0.033 respectively. This indicates that the FDLMS algorithms execute much faster than time domain LMS algorithm. In addition, the independent updates of the weights $H_i(k)$ of the FDALE make it more suitable for parallel processing than the time domain adaptive line enhancer (TDALE). As a result, the FDALE has a better performance than that of the TDALE.

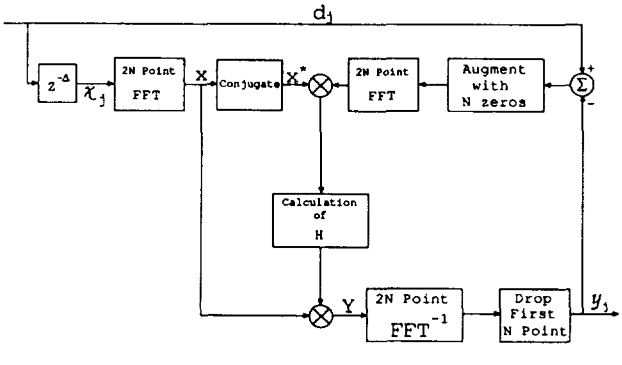


Figure 3. Third Algorithm Block Diagram

SIMULATION RESULTS

The three algorithms were employed in the detection and spectral estimation of a narrowband signal corrputed with white additive noise. The input signal y(k) with normalized frequencies 0.033 and 0.156 HZ is generated by adding a sinusoidal input s(k) to a white Guassian noise n(k). Sample simulation results for the three algorithms using the Connection Machines are presented in Figures 4 and 5. Figure 4 is a plot of the execu-

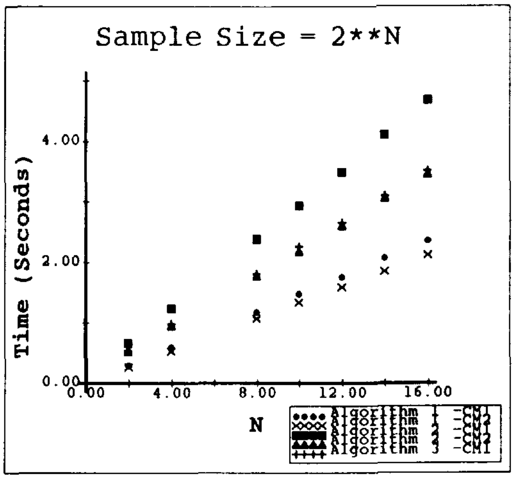
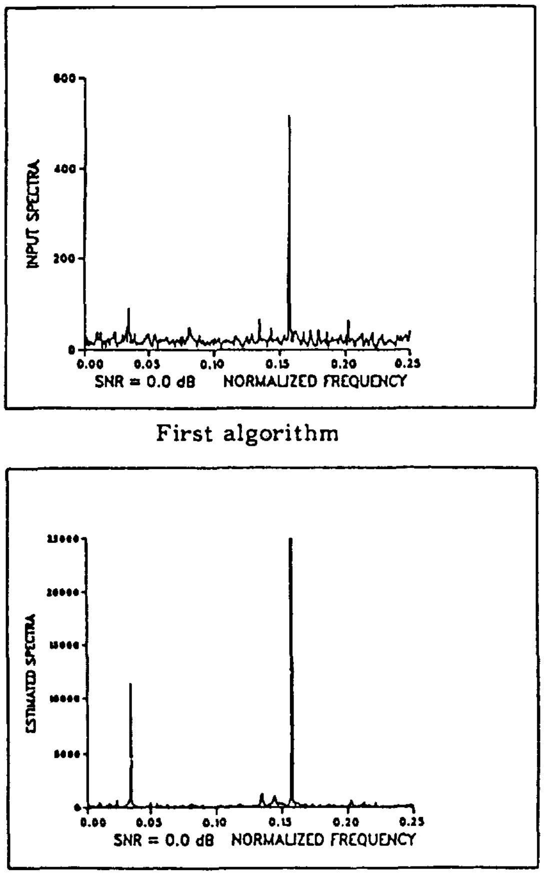


Figure 4. Execution Time of the Connection Machine Versus Input Data Size



First algorithm

Third algorithm

Figure 5. Estimated Spectra for 16K Data Samples and Zero SNR.

tion time of the three algorithms in seconds versus the input data size using the CM-1 and CM-2 machines. Figure 4 indicates that the execution time is a linear function of the input data size when using a CM data parallel computer and the first algorithm is the fastest algorithm followed by the third algorithm and the second algorithm respectively. This linear speed up is due to "scaling up" the problem [13]. Figure 5 shows the estimated spectra for 16K data samples and zero dB signal to noise ratio. Figure 5 demonstrates the close match between the estimated frequencies and the actual frequencies and the third algorithm has the best performance, especially for lower signal to noise ratios, followed by the second and first algorithms respectively.

The speed up factor for the Balance 8000 machine is defined as the ratio between the time needed for one processor to implement the algorithm and the time needed for n-processors to implement the same algorithm. This speed up approaches the limit predicted by Amdahl's [15]. Figure 6 shows the speed-up factors of the first algorithm using the Balance 8000 for different number of processors and N = 512 and 1024. A comparison between the actual performance and the linear one is also shown in Figure 6.

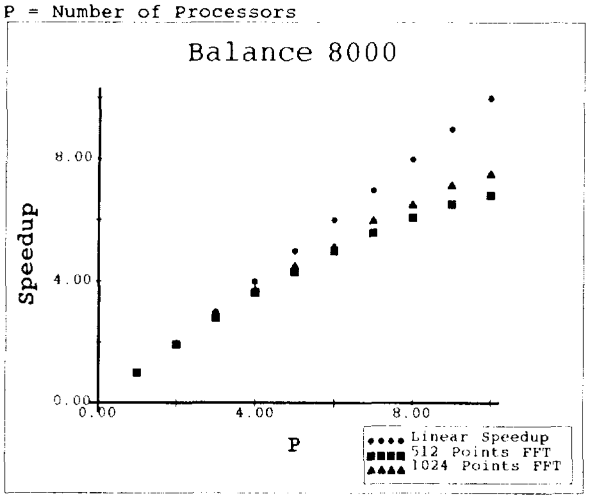


Figure 6. Speed Up Factor For the Balance 8000.

## SUMMARY

In this paper, the fundamental concepts of three parallel frequency domain adaptive line enhancers were presented. Furthermore, the Connection Machines and the Balance Machine were used to simulate these three parallel implementation of the frequency domain adaptive line enhancers. Using the Connection Machines, a linear speed up was obtained. The number of parallel processor are proportional to the size of the data. With the Balance Machine, Amdahl's limit is reached since a fixed size problem is solved using more processors.

## REFERENCES

1. El-Sharkawy M., et. al., "Adaptive Stochastic Filter with no Strict Positive Real Condition", IEEE Transactions on ASSP, Vol. 35, No. 11, pp. 1547-1556, November 1987.
2. El-Sharkawy M., et. al., "Multistage Adaptive Stochastic Filters," IEEE Transactions on Circuits and Systems, Vol. 35, No. 8, August 1988.
3. Dentino, M., et. al., "Adaptive Filtering in the Frequency Domain," Proc. IEEE, Vol. 66, No. 12, December 1978.
4. Widrow, B., et. al., "The Complex LMS Algorithm," Proc. IEEE, Vol. 63, No. 4 April 1975.
5. Ferrara, E. R., "Fast Implementation of LMS Adaptive Filters, "IEEE Trans. ASSP, Vol. ASSP-28, No. 4, August 1980.
6. Mansour, D., et. al., "Unconstrained Frequency Doman Adaptive Filter," IEEE Trans. Vol. ASSP-30, No. 5, pp. 726-734, October 1982.
7. David, R. A., et al., "IIR Algorithms for Adaptive Line Enhancement," Proc. ICASSP-83, April 1983.
8. Cowan, C. F. N., et al., "Adaptive Filter", Prentice Hall Signal Processing Series, 1985.
9. Hillis, W. D., "The Connection Machine," Cambridge, MS, MIT Press, 1986.
10. Oppenheim, A. V., et al., "Digital Signal Processing", Prentice-Hall, 1975.
11. Mauel, T., "How's Sequent's New Model Outruns Most Mainframes", Electronics, pp. 76-79, May 1987.
12. Flynn, M. J., "Very High Speed Computing Systems," Proc. IEEE, Vol. 54, pp. 1901-1909, December 1966.
13. Gustavson, I., et al., "Development of Parallel Methods for a 1024-Processor Hypercube", SIAM J. Sci. Stat. Comp., Vol. 9, No. 9, pp. 1-32, 1988.
14. Tucker, L. W., et al., "Architecture and Applications of the Connection Machine", Computer, Vol. 21, No. 8, pp. 26-38, August 1988.
15. Amdahl, G., "Validity of the Single Processor Approach to Achieving Large Scale Computer, Capabilities," AFIPS Conf. Proc. 30 91967), pp. 483-485.

## ACKNOWLEDGEMENT

The authors wish to thank Paul Anderson, John Church of the Connection Machine Facility at the Naval Research Laboratory, Washington D.C. and Arvind Betrabet of the Electrical Engineering Department at Southern Methodist University for their valuable help.

# Optimal Geometric Algorithms for Digitized Pictures on an Optical Mesh *

Mehrnoosh Mary Eshaghian and V. K. Prasanna Kumar
Department of Electrical Engineering-Systems, SAL 344
University of Southern California
Los Angeles, CA 90089-0781
Tel: (213)-743-5236

## ABSTRACT

In this paper, we present optimal parallel algorithms for finding geometric properties of digitized images on an optical mesh. These include $O(\log N)$ time solutions for identifying and labeling figures, computing convexity properties, determining distances, etc. The proposed optical mesh consists of a two dimensional array of processors which are interconnected through an optical medium. Using free space optical beams, the interconnection topology can be reconfigured dynamically. The computational lower bounds of a generic model reflecting such an organization is shown. Unlike the VLSI models, there are no planarity constraints, since the optical beams can cross each other without any interference. We also study the relationships between the shared memory models and the proposed optical model of computation and its possible physical implementations.

## INTRODUCTION

One of the widely used models of parallel computation is the Parallel Random Access Machine (PRAM). The basic assumption in this model is that in unit time each of $N$ processors can simultaneously access a shared memory (Ref. 20). The unit time simulation of a $N$ processor PRAM using electronic interconnections takes $O(N^4)$ area on a VLSI model in which processors are allowed to have unbounded degree. A lower bound on the time to simulate one step of a PRAM on any bounded degree network of $N$ nodes using electrical interconnects is $\Omega(\log N)$ . The crossbar network is an architecture that simulates one step of PRAM in $O(\log N)$ time, and can be laid out in $O(N^2)$ area. Even though several networks have been designed (Ref. 8), many issues limit their potential application in high performance parallel systems. Finding a switch setting to realize a permutation on a rearrangeable $O(\log N)$ delay network such as the Benes takes as much as $O(\log^4 N)$ time using a cube connected computer or a perfect shuffle computer with $N$ processors (Ref. 14) . Also, the layout area of a Benes network is not superior compared to the area requirement of the $N$ input crossbar network.

In this paper, we study parallel architectures that use free space optics as a means of interprocessor communications. The replacement of the electrical interconnects with optical beams has a significant impact on the performance of a VLSI architecture (Refs. 9,10). This is due to the following two important properties of free space optics. First, free space optical beams can cross each other with out any interference. Also, the connections need not be fixed and can be redirected (Ref. 3). This implies that using optical interconnects, one can design area efficient bounded degree VLSI architectures that can simulate a unit delay interconnection network.

We present efficient parallel algorithms for finding geometric properties of digitized images on an optical mesh. These include $O(\log N)$ algorithms for problems such as finding connected components, determining the convex hull of all figures, and nearest neighboring figure to all figures. Our algorithms require a factor of $O(\log N)$ fewer processors compared to the traditional electronic mesh based architectures (Refs. 13,15) An optical mesh can be looked upon as a two dimensional mesh connected computer which is enhanced with a reconfigurable optical interconnection network.

The rest of the paper is organized as follows. In the next section, we study an optical model of computation, and then show a class of reconfigurable optical interconnection networks in the third section. In the last section, we propose a set of parallel geometric algorithms for digitized pictures using the optical mesh. Our results substantiate the preference of optical interconnections over electronic medium as a means of interprocessor communication.

## AN OPTICAL MODEL OF COMPUTATION

In this section, we introduce an abstract optical model of computation (OMC) to explore speed size relationship in using free space optical beams, as opposed to wires in traditional electronics (Ref. 18), for means of intercommunications. This model closely captures a currently implementable optical network of processors. Hence, the derived lower bounds on its computational efficiency gives us a tool to analyze the optimality of various physical implementations of OMC in solving problems.

*Definition:* An optical model of computation represents a network of $N$ processors each associated with a memory module, and a deflecting unit capable of establishing direct optical connection to another processor. The interprocessor communication is performed through a message passing network satisfying the following rules similar to (Ref. 1):

* This research was supported in part by the National Science Foundation under grant IRI-8710836 and by a grant from TRW.

CH2649-2/89/0000/0147$01.00 © 1988 IEEE

1) At any time a processor can send at most one message. Its destination is another processor.

2) The message will succeed in reaching the processor if it is the only message with that processor as its destination, at that time step.

3) All messages succeed or fail (and thus are discarded) in unit time.

To insure that every processor knows when its message succeeds we assume that the OMC is run in two phases. In the first phase, read/write messages are sent, and in the second, values are returned to successful readers and acknowledgements are returned to successful writers. We assume that the operation mode is synchronous, and that all processors are connected to a central control unit. In (Ref. 5), the above definition is supplemented with a complete set of assumptions for accurate analysis. The following are some of the main ones.

1) The intercommunication is done through free space optical beams.

2) A processor can perform a simple arithmetic/logic operation in one unit of time.

3) A deflector is capable of redirecting an incident beam in one unit of time.

4) An optical beam carries a constant amount of information in one unit of time, independent of the distance to be covered.

In (Ref. 5), using the above assumptions the following relationship can be derived:

$$AT = \Omega(I), \qquad (1)$$

where $T$ is the time required to solve a problem given $I$ as the minimum required information to be transferred, and $A$ is the area occupied by the processing layer.

A related model is VLSIO (Ref. 2), which is a three dimensional generalization of the wire model of the two dimensional VLSI with optical beams replacing the wires as communication channels. Compared to the three dimensional VLSI model of computation (Ref. 16), our model is more computationally resource efficient. The simulation of many parallel organizations using the OMC requires considerably less amount of volume than its layout in a three dimensional VLSI model. For example, the layout volume of a $N$ processor hypercube can be reduced from $O(N^{3/2})$ to $O(N\log N)$ when using OMC with mirrors as deflectors [next section], instead of using a three dimensional VLSI model of computation. Hence, the following result can be stated;

**Proposition:** Any computation performed by a three dimensional VLSI organization having $N$ processors with degree $d$, in time $T$, and volume $V$ can be performed on OMC in volume $v$, and time $t$, where $dT/N \leq t \leq T$, and $Nd \leq v$.

## PARALLEL ARCHITECTURES

In this section, we present a class of optical interconnection networks as a realization of the OMC presented in the previous section. Each of the proposed designs uses a different optical device technology for redirection of the optical beams to establish a new topology at any clock cycle, and represents an upper bound on the volume requirement of OMC.

### Optical Mesh using Mirrors

In this design, there are $N$ processors on the processing layer of area $N$. Similarly, the deflecting layer has area $N$ and holds $N$ mirrors. These layers are aligned so that each of the mirrors is located directly above its associated processor. Each processor has two lasers. One of these is directed up towards the arithmetic unit of the mirror and the other is directed towards the mirror's surface. For a pictorial illustration see (Ref. 6). A connection phase would consist of two cycles. In the first cycle, each processor sends the address of its desired destination processor to the arithmetic unit of its associated mirror using its dedicated laser. The arithmetic unit of the mirror computes a rotation degree such that both the origin and destination processors have equal angle with the line perpendicular to the surface of the mirror in the plane formed by the mirror, the source processor, and the destination processor. Once the angle is computed, the mirror is rotated to point to the desired destination. In the second cycle, connection is established by the laser beam carrying the data from the source to the mirror and from the mirror being reflected towards the destination. Since the connection is done through a mechanical movement of the mirror, with the current technology this leads to an order of millisecond reconfiguration time. Therefore this architecture is suitable for applications where the interconnection topology does not have to be changed frequently. In (Ref. 12), the design of various topologies have been studied to minimize the time complexity of several problems for fixed period of computation.

### Reconfiguration using Acousto Optic Devices

In this organization, $N$ processors are arranged to form a one-dimensional processing layer and the corresponding acousto optics devices are similarly located on a one-dimensional deflecting layer. The size of each of the acousto optic devices is proportional to the size of the processing array, leading to an $O(N^2)$ area deflection layer. Similar to the design using the mirrors, every processor has two lasers, and each connection phase is made up of two cycles. For a pictorial illustration see (Ref. 6). In the first cycle, each processor sends the address of its desired destination processor to the arithmetic unit of its associated acousto optic unit using its dedicated laser beam. The acousto optic cell's arithmetic unit computes the frequency of the wave to be applied to the crystal for redirection of the incoming optical beam to the destination processor. The acousto optic device then redirects the incident beam from the source to the destination processor. One of the advantages of this architecture over the previous design is its order of microseconds reconfiguration time, which is dominated by the speed of sound waves. The other advantage is its broadcasting capability, which is due to the possibility of generating multiple waves through a crystal at a given time. Furthermore, the

** A function $f\ (n)$ is said to be $\Omega(g\ (n))$ if there exist positive constants $c$ and $n_0$ such that for all $n > n_0$, $f\ (n) \geq c.g\ (n)$.

A function $f\ (n)$ is said to be $O(g\ (n))$ if there exist positive constants $c$ and $n_0$ such that $f\ (n) \leq c.g\ (n)$. for all $n \geq n_0$.

above can be extended to interconnect a two dimensional grid of processors as follows.

**Proposition:** Using a $(N^{1/2}\times N^{1/2})$ processing layer, and $(N^{1/2}\times N^{1/2})$ array of acousto optic devices as the deflecting layer, one step of OMC can be realized in $O(\log N)$ time and $O(N^2)$ area.

The area is obtained with similar arguments as in the one dimensional case. The time complexity is due to the movement of data using a standard divide and conquer technique as in (Ref. 19). At the *ith* step a block size $2^i$ is divided into two blocks of half the size. Each subblock only contains the data elements destined to its memory locations.

### Electro Optical Crossbar

This design uses a hybrid reconfiguration technique for interconnecting processors. There are $N$ processors each located in a distinct row and column of the $N\times N$ processing layer. For each processor, there is a hologram module having $N$ units, such that the *ith* unit has a grating plate with a frequency leading to a deflection angle corresponding to the processor located at the grid point $(i,i)$. In addition, each unit has a simple controller, and laser beam. For a pictorial illustration see (Ref. 6). To establish or reconfigure to a new connection pattern, each processor broadcasts the address of the desired destination processor to the controller of each of $N$ unit of its hologram module using an electrical bus. The controller activates a laser (for conversion of the electrical input to optical signal), if its ID matches the broadcast address of the destination processor. The connection is made when the laser beams are passed through the predefined gratings. Therefore, since the grating angles are predefined, the reconfiguration time of this design is bounded by the laser switching time which is in the order of nano-seconds using Gallium Arsenide technology. This architecture is faster than the previous designs and further it compares well with the clock cycle of the current supercomputers. One of the advantages of this simple design is in its implementability in VLSI, using GaAs technology. Unlike the previous designs, this can be fabricated with very low cost and is highly suitable for applications where full connectivity is required. In such applications, the processor layer area can be fully utilized by placing $N$ optical beam receivers in each of the vacant areas to simultaneously interconnect with all the other processors. This design can be easily adopted to implement a neural network of process with optical interconnects (Ref. 4) .

## IMPLEMENTATION OF PARALLEL ALGORITHMS

As described in the previous section, OMC allows unit cost communications but assumes that the shared memory is divided into modules. An OMC with $N$ processors can simulate, in real time, an Exclusive Read Exclusive Write PRAM having $P$ processors and $M$ memory locations, where $N =$ maximum $\{P,M\}$. On the other hand, a $P$ processor EREW PRAM can simulate in real time any $P$ processor OMC. Hence, it is easy to see that using $N$ processor OMC in $O(\log N)$ time, the FFT of $N$ points, and in $O(\log^2 N)$ time the bitonic sort of $N$ elements can be performed.

In (Ref. 7), simple efficient algorithms for simulating an $N$ processor PRAM on OMC using $N/\log N$ processors are presented. Each step of the EREW PRAM is simulated in $O(\log N)$ time with a high probability, and in $O(\log^2 N)$ time deterministically. In this paper, we present optimal algorithms for solving problems in medium level image processing. An optimal parallel algorithm is defined as one which its processor time product is equivalent to the running time of the sequential solution. Due to space limitations only the proof sketchs explaining the main ideas are presented. The details appear in (Ref. 6). Also for an introduction to the basic techniques used in the design of the following algorithms refer to a simple optimal template matching algorithm shown in (Ref. 6).

### Optimal Geometric Algorithms

In this section, we present efficient $O(\log N)$ algorithms for problems such as finding connected components, determining the convex hull of all figures, and nearest neighboring figure to all figures. The input to our algorithms is an $(N\times N)$ image where each of $N^2$ points is called a pixel and can be either a 0 or 1. Two adjacent pixels are connected if they both hold a 1. The connected 1's form a figure. We define the following:

> *Optical Mesh:* An optical mesh of size $N\times N$ has a processor layer with 2-dimensional array of processors which can intercommunicate in unit-time using their corresponding optical device residing on the deflection layer of same size. A simple implementation of this is possible using mirrors which was discussed in the previous section.

Given a 0/1 image a fundamental task is to identify figures in the image. Figures correspond to connected 1's in the image (see (Ref. 17)). The labeling problem 'is to identify and associate an unique ID with the connected 1's in the image.

**Lemma 1:** Given a $N\times N$ 0/1 image, all figures can be labeled in $O(\log N)$ time using an $(N\times N)$-optical mesh.

**Theorem 1:** Given a $N\times N$ 0/1 image, all figures can be labeled in $O(\log N)$ time using an $(N/\log^{1/2}N\times N/\log^{1/2}N)$-optical mesh.

**Proof sketch:** In this, we assign a $\log^{1/2}N\times\log^{1/2}N$ block of image to each processor, and sequentially label the figures within these regions. For each of these blocks, we continue with its boundary information which has $O(\log^{1/2}N)$ pixels. In the next step, we merge these blocks together until the block size becomes $\log N\times\log N$. Since the input size is larger than the processor count by $O(\log^{1/2}N)$, we simulate the $\log\log N$ divide and conquer merging in the following manner. In each region of size $\log N\times\log N$ there are $\log^{3/2}N$ boundary points. At the *ith* iteration of merging, this reduces to $(\log^{3/2}N)/2^i$. Therefore the total boundary points to be merged over the $\log\log N$ iterations is $O(\log^{3/2}N)$. Using Lemma 1, we label $O(\log N)$ boundary points of each iteration at a time, using $\log N$ processors. Hence, the number of iterations to simulate $\log\log N$ merging is $O(\log^{1/2}N)$. This leads to $O(\log^{1/2}N\log\log N)$ time complexity to reduce the image size to match the processor count. Using Lemma 1, the remaining $N\times N$ pixels are labeled with a $N\times N$ optical mesh in $O(\log N)$ time.

Convexity is an important attribute in image processing and vision; many other problems can be solved once the convex hull of figures is obtained. We use the following definition of convexity (Ref. 13) : A set of PEs is said to be convex if and only if the corresponding set of integer lattice points is convex. Given a set S of PEs, the convex hull of S, denoted Hull(S), is the smallest convex set of PEs containing S.

**Theorem 2:** Given a $N \times N$ 0/1 image, the extreme points of all the figures can be enumerated in $O(\log N)$ time using an $(N/\log^{1/2}N \times N/\log^{1/2}N)$-optical mesh.

**Proof sketch:** For each $\log N \times \log N$ portion of the given 0/1 image, it is easy to enumerate the convex hull of all its figures in $O(\log N)$ time, using a $(\log N \times 1)$-optical mesh. After this reduction step, the basic idea of the algorithm is to construct a list for each of the figures simultaneously, and while constructing it, delete those boundary points which are not extreme points. Once the extreme points are found, they can be enumerated using the standard all parallel prefix sum in $O(\log N)$ time.

Another interesting problem is to compute the distances between all figures in a digitized image. In the following we use the $l_1$ metric. However, it can be modified to operate for any $l_k$ metric.

**Theorem 3:** Given a $N \times N$ 0/1 image, The nearest figure to all figures can be enumerated in $O(\log N)$ time using an $(N/\log^{1/2}N \times N/\log^{1/2}N)$-optical mesh.

### Constant Time Geometric Algorithms

One of the most attractive properties of optics is superposition. This property suggests that the resultant disturbance at any point in a medium is the algebraic sum of the separate constituent waves. Hence, it enables many optical signals to pass through the same point in space at the same time without causing mutual interference or crosstalk. Using this property in (Ref. 11), they showed how a single memory element can be read by many processors at the same time. In this paper we employ this characteristic to allow concurrent writes if all the requesting processors want to write a "1". This leads to the constant running time of the following geometric algorithms, under the assumption that broadcasting can be done in constant time:

**Corollary:** Given a $(N^{1/2} \times N^{1/2})$ image, using an $(N \times N)$ optical mesh, in $O(1)$ time,

1. For a single figure, its convex hull and the smallest enclosing box can be found.
2. For each figure, the nearest neighboring figure can be identified.

## CONCLUSION

In this paper, we studied the computational limits in using optical interconnects under a proposed optical model of computation. Motivated by this model, we presented three possible physical architectures to realize the unit time intercommunication delay assumed in PRAM. A direct simulation is possible using the proposed optical mesh using mirrors. A faster architecture is the optical array using acoustic optic devices with broadcasting capability. A considerably less expensive design which is currently implementable with VLSI technology is the electro optical crossbar. This is an optimal design for fully connected networks with a reconfiguration time in the order of nanoseconds. We showed the superiority of using optical interconnects by presenting efficient algorithms for finding the geometric properties of digitized pictures.

## REFERENCES

1. R. Anderson, and G. L. Miller, "Optimal Parallel Algorithms for List Ranking", Technical Report, Dept. of Computer Science, USC, 1987.
2. R. Barakat and J. Reif, "Lower bounds on the computational efficiency of optical computing systems", Journal of Applied Optics, Vol. 26, No. 6, March 15, 1987.
3. L. A. Bergman, W. H. Wu, A. R. Johnson, R. Nixon, S. C. Esener, S. C. Guest, P. Yu, T. J. Drabik, M. Feldman, S. H. Lee, "Holographic optical interconnects for VLSI", Optical Engineering, October 1986.
4. M. M. Eshaghian, and V. K. Prasanna-Kumar, "An Implementation of Neural Networks using Optical Interconnects", IEEE International Conference of Neural Networks, 1988.
5. M. M. Eshaghian, Ho-In Jeon, and V. K. Prasanna-Kumar, "Massively Parallel Architectures with Optical Interconnection Networks", Optical Computing 88, Toulon, France.
6. M. M. Eshaghian, and V. K. Prasanna-Kumar, "Optimal Parallel Algorithms for Image Processing on an Optical Mesh", Technical Report, Dept. of EE-systems,USC.
7. M. M. Eshaghian, "Parallel Computing with Optical Interconnects", Ph.D. thesis, in preparation, USC.
8. T. Y. Feng, "A Survey of Interconnection Networks", IEEE Computer magazine, December 1981.
9. D. H. Hartman, "Digital high speed interconnects: a study of the optical alternative", Optical Engineering, Oct. 1986.
10. P. R. Haugen, S. Rychnovsky, A. Husain, L. D. Hutcheson, "Optical Interconnects for high speed computing", Optical Engineering, October 1986.
11. B. K. Jenkins, and C. L. Giles, "Superposition in Optical Computing", Optical Computing 88, Toulon, France.
12. I. Lee, S. Goldwasser, and D. Smitely, "Synthesis and mapping algorithms for a reconfigurable optical interconnection network", ICPP, 1986.
13. R. Miller and Q. F. Stout, "Geometric Algorithms for Digitized Pictures on a Mesh Connected Computer", IEEE PAMI, March 1985.
14. D. Nassimi and S. Sahni, "Parallel algorithms to set up the Benes permutation network", IEEE Transactions on Computers, Vol. C-31, No. 2, Feb. 1982.
15. V. K. Prasanna Kumar and M. M. Eshaghian, "Parallel Geometric Algorithms for Digitized Pictures on Mesh of Trees", Proc. of ICPP, 1986.
16. A. L. Rosenberg, "Three-dimensional Integrated Circuits", VLSI Systems and Computation, H. Kung, R. Sproul, and G. Steel, Eds., CS Press, 1981, pp. 69-79.
17. A. Rosenfeld, "Parallel Processors for Image Processing: 2-D arrays and extensions", IEEE Computer, January 1983.
18. C. D. Thompson, "A Complexity Theory for VLSI", Ph.D. Thesis, C.M.U., 1980.
19. J. D. Ullman, "Computational Aspects of VLSI", Computer Science Press, 1983.
20. J. C. Wyllie, "The Complexity of Parallel Computations", Ph.D. thesis, Cornell University, Ithaca, NY, 1979.

# AN SIMD PARALLEL $\varepsilon$-APPROXIMATION SCHEME FOR 0/1 KNAPSACK

Thomas E. Gerasch
SPARTA, Inc.
7926 Jones Branch Drive
Suite 1070
McLean, Virginia 22102

## ABSTRACT

A parallel version of a well-known $\varepsilon$-approximation scheme for 0/1 knapsack problems is presented. The model of computation for the parallelization is a shared memory machine in which processors have exclusive read, exclusive write access to memory (an EREW PRAM). The scheme separates the knapsack items into two sets, one of which is used in a dynamic programming-based optimization procedure, and the other of which is used in a greedy selection process. A dominance relation exists for the knapsack problem which is used to limit the growth of feasible solutions during the dynamic programming procedure. The dominance relation permits a simple representation of the feasible solutions which aids in the parallelization of the dynamic programming procedure across all feasible solutions in parallel during the process of considering a new item. The algorithm uses $max\{n,32/\varepsilon^3-8/\varepsilon\}$ processors and takes $O(n)$ time. For moderate values of $\varepsilon$ and values of n which are quite large (tens of thousands of items), the algorithm is realizable on currently available, massively parallel computer systems, such as the Connection Machine System.

Keywords: knapsack, approximation, SIMD, shared memory, dynamic programming.

## 1. INTRODUCTION

The 0/1 integer knapsack problem is defined by a set of $n$ objects having positive integer profits $p_i$ and weights $w_i$, and a positive integer knapsack capacity M. The problem is concerned with maximizing the profit $P=\Sigma_i x_ip_i$ subject to a weight constraint $\Sigma_i x_iw_i \leq M$, where $x_i$ in $\{0,1\}$ for $i=0,1,...,n$. An instance of the knapsack problem consists of an assignment of values for the profits, weights and the capacity. A *feasible solution* to a problem instance is a set of $x_i$ values which satisfy the capacity constraint by indicating whether an item is to be included in the knapsack. This well-known optimization problem is NP-Complete [GJ79], and consequently, considerable interest has been generated in developing algorithms which obtain approximate solutions [LE79; HS78; IK75; SS75].

An $\varepsilon$-approximate solution $(0 < \varepsilon < 1)$ for an instance of the 0/1 knapsack problem is a feasible solution whose profit P satisfies $P^*-P \leq \varepsilon P^*$, where $P^*$ is the maximum profit attainable for the problem instance. An approximation scheme for the knapsack problem is one which takes an $\varepsilon$ as a parameter and for each instance of the knapsack problem finds an $\varepsilon$-approximate solution. Ibarra and Kim [IK75] presented an approximation scheme for the knapsack problem which has both time and space complexities that are polynomial in n and $1/\varepsilon$, i.e. which is a fully-polynomial approximation scheme [HS78]. Lawler [LE79] presented modifications to the ideas of Ibarra and Kim that improved the time and space bounds.

There has been a growing interest in parallel computation and in the parallelization of optimization techniques and algorithms. This interest has extended to the adaptation of knapsack approximation algorithms to parallel models of computation [GRK86; GW88; LSS87]. These parallelizations have been for MIMD [LSS87] as well as SIMD [GRK86; GW88] models of computation

The algorithm to be presented here, like that in [GRK86], is an SIMD parallelization of the $\varepsilon$-approximation scheme of Ibarra and Kim. Although both algorithms are SIMD parallelizations of the same basic strategy, their approaches and actual parallel models of computations differ. The algorithm in [GRK86] is a recursive implementation of a dynamic programming technique applied to a scaled profit version of the entire problem instance. The algorithm presented here is a more direct implementation of the original dominance-based dynamic programming approach of Ibarra and Kim, using scaling of the profit values as well as an item separation technique. The item separation partitions the knapsack items into two disjoint sets of relatively *large* and *small* items, with set membership determined relative to a suitably chosen threshold value. The dominance-based dynamic programming technique is applied to a scaled problem involving the *large* items. Feasible solutions from the *large* item optimization are augmented with *small* items, with the *small* items selected to augment a feasible solution using a greedy approach.

The SIMD model which is used assumes that processors have exclusive-read, exclusive-write access to a common memory (i.e. an EREW PRAM model). (The model used in [GRK86] allows concurrent reading of memory by processors, but exclusive writing, i.e. a CREW PRAM model.) Processors are assigned to manipulate data in a data-parallel fashion using one processor per array index. To insure that data is accessed in an EREW fashion, processors which are actively participating in a computation step will reference the data in a uniform addressing fashion. For example, each active processor $i$ will reference data item $i+k$, where the offset $k$ is the same for all processors.

The analysis of the algorithm will be similar to that typically given for serial algorithms in that arithmetic operations, comparisons, and memory references are counted as unit time operations. However, the same operation performed in parallel by many processors will only be counted as a single (parallel) instruction execution. The assumption that all of the operations mentioned take unit time is not strictly true in actual machines, but has been adopted to simplify the analysis of algorithms. It has recently been argued by Blelloch [1987] that since memory references in the shared memory models are taken as unit time operations, other useful primitive operations should also be allowed as unit time operations. *Scan*, or parallel prefix operations [KRS85; BG87], are among the candidates as useful primitive operations, since they can be implemented with less circuit depth and in comparable circuit area than a general shared memory circuit [FF83; LT84]. A *scan* operator takes an associative operator $\oplus$ and a sequence of elements $[a_1,a_2,...,a_n]$, and returns the sequence $[a_1,(a_1 \oplus a_2),\ldots,((a_1 \oplus a_2 \oplus \ldots \oplus a_{n-1}) \oplus a_n)]$. The *scan* operations used in the algorithms will be *plus_scan*, which uses addition to obtain the sequence of partial sums, *copy_scan* to replicate a data item a specified number of times (a *plus_scan* with all items but the first contributing *0*), and *max_scan* to find the maximum of *n* items. A *scan* operation can also be run in parallel on subsequences of an array, and will be called a *segmented scan* operation.

*Scan* operations are logarithmic time operations in the usual EREW PRAM model. Taking the *scan* operations to be unit time operations yields a parallel model of computation which will be called the *scan* model. The algorithm will be analyzed in both the EREW and *scan* models.

The EREW PRAM, and *scan* model, knapsack approximation scheme to be presented requires $O(n)$ time and $max\{n,32/\varepsilon^3-8/\varepsilon\}$ processors. This time requirement is greater than that of the CREW PRAM algorithm of [GRK86], which is $O(log^3n + log^2nlog(1/\varepsilon))$. However, this $O(n)$ time requirement is not unreasonable. A more interesting measure is the number of processors required by these two algorithms. The CREW model algorithm requires $n^{2.5}/\varepsilon^{1.5}$ processors, in its worst case. For $n=1000$ and $\varepsilon=1/10$, the EREW and *scan* algorithms require 31920 processors, while the CREW algorithm may require up to $10^9$ processors. For $\varepsilon=1/10$ and a number of processors equal to $2^{16}$, the CREW algorithm of [GRK86] can only be guaranteed to handle instances of knapsack problems with fewer than *350* items, while the EREW and scan algorithms could handle problem instances of size $2^{16}$. The processor requirement for the algorithm of [GRK86] can quickly exceed the capability of current technology. However, the processor requirement for the algorithm presented here is within the realm of current technology, and allows for problems of considerable size to be realistically attacked, assuming moderate values of $\varepsilon$. The Connection Machine System of Thinking Machines Corporation has up to $2^{16}$ processors, and supports virtual processors into the millions. Although the memory in the Connection Machine is local to each processor, one can program it in the EREW memory reference paradigm by using interprocessor fetch and send operations to simulate the shared memory read and write operations, and by designing algorithms to not have collisions at any processor during the memory reference operations. It was with the Connection Machine in mind that the EREW PRAM parallelization of the $\varepsilon$-approximation scheme was initially designed.

## 2. ε-APPROXIMATION SCHEME

The approximation algorithm to be parallelized is the ε-approximation scheme of Ibarra and Kim [1975], using the scaling refinements due to Lawler [LE79]. The ε-approximate solution to a knapsack problem instance is obtained by first separating the items into two sets of relatively *large* items and relatively *small* items. The item separation is performed using scaling and threshold factors which ensure that one-half of the relative approximate error is allocated to each of the two calculation stages. These stages correspond to computations involving each of the two sets of items [LE79]. The threshold and scale factors are determined by using an estimate $P_0$ of the optimal profit value for the knapsack instance. This estimate satisfies $P_0 \le P^* \le 2P_0$ where $P^*$ denotes the optimal profit value for the problem instance.

The *large* item computation uses a dynamic programming approach to find an optimal solution to a knapsack instance that consists of the items with scaled *large* profit values and has capacity equal to the original capacity. To restrict the growth of the number of feasible solutions during the *large* item computation, the following dominance relation is used. Let $S_1$ and $S_2$ be two feasible solutions (i.e. sets of elements whose weight sums do not exceed the knapsack capacity). The feasible solution $S_2$ *dominates* $S_1$, written $S_2 > S_1$ if $P(S_1) \le P(S_2)$ and $W(S_1) \ge W(S_2)$, where $P(S)=\Sigma_{i \in S}\, p_i$, and $W(S) = \Sigma_{i \in S}\, w_i$. It should be noted that if $S_j > S_i$ and if $S_k$ is a set of indices disjoint from both $S_i$ and $S_j$, then $S_j \cup S_k > S_i \cup S_k$.

During the *large* item computation, a sequence of feasible solutions is maintained whose profit and weight sums are in increasing order and for which no solution is dominated by any other in the sequence. As each item is considered, the dominance rule is applied to preserve the properties of the sequence of feasible solutions. The last solution in the sequence is actually the optimal solution to the scaled *large* item problem instance. At the end of the *large* item computation, the entire sequence of feasible solutions is retained, not just the last in the sequence.

Each of the feasible solutions from the *large* item computation has its remaining capacity augmented by *small* items. This augmentation of the solutions is done using a greedy approach. The ε-approximate solution to the problem instance is the one with greatest profit value after the *large* and *small* item computation stages.

The algorithm for the ε-approximation scheme is summarized in the following steps:

1.) Find $P_0$ such that $P_0 \le P^* \le 2P_0$, set the scale factor $K=(\varepsilon/2)^2P_0$, and the threshold factor $T=(\varepsilon/2)P0$.

2.) {item separation} Separate the items into *large* and *small* sets, the *large* items being those for which $p_i>T$. Arrange the items so that the *large* item set precedes the *small* item set. For the *large* items set $q_i=\lceil p_i/K \rceil$, the scaled profit values.

3.) {Select a minimal set of *large* items to consider} Arrange the *large* item set in nonincreasing order of $q_i$, and within each $q_i$ group in nondecreasing $w_i$ order. Select the first $(8/\varepsilon^2)/q_i$ items of each $q_i$ group as candidates.

4.) {*large* item optimization} Perform the dominance-based optimization of Ibarra and Kim on the minimal set of *large* item candidates, producing a sequence of feasible solutions whose profit and weight sums are nondecreasing (and consequently for which no solution dominates another).

5.) {Augment feasible solutions with *small* items} Arrange the *small* items in nonincreasing order of $p_i/w_i$ ratios; add a *small* item to a feasible solution if there is sufficient capacity remaining the feasible solution. Select the feasible solution with the greatest profit value.

**Theorem 1.** The algorithm outlined is an $\varepsilon$-approximation scheme for the 0/1 knapsack problem.

That the process outlined above is an $\varepsilon$-approximation scheme was shown by Ibarra and Kim [IK75]. The particular scaling and threshold values used in step 1) are due to Lawler [LE79]. Additional details concerning the steps of the algorithm will be presented in the following section, where the parallelization strategy is presented.

## 3. PARALLELIZATION OF THE ε-APPROXIMATION SCHEME

The first step in the parallel algorithm can be accomplished by sorting the items into nonincreasing profit/weight ratios so that $p_1/w_1 \ge p_2/w_2 \ge \ldots \ge p_n/w_n$, and next determining the largest $m$ such that $p_1 + \ldots + p_m \le C$ but $p_1+\ldots+p_m+p_{m+1} > C$. $P_0$ is then taken to be $max\{p_1+\ldots+p_m,\ max\{p_i:\ m<i \le n\}\}$ [LE79]. An EREW sorting algorithm using $n$ processors, such as an adaptation of the bitonic merge sort [QM87], will be assumed for this and for subsequent sorting steps. The time to sort $n$ items using $n$ processors in the EREW model is taken to be $O(log^2n)$. The CREW model algorithm from [GRK86] uses a time estimate of $O(log\ n)$ for sorting n items using n processors, and is based on the algorithm in [AKS83]. As pointed out by Leighton [LT84], the limit obtained in [AKS83] is an asymptotic limit and the constant of proportionality is so large

that this sorting strategy becomes infeasible from a technological standpoint. For this reason, the more realistic time estimate for sorting of $O(log^2 n)$ using $n$ processors is used here.

In the scan model of computation, a sorting algorithm can be used that is a parallel version of the radix exchange sort [KD73]. If the sort keys are $m$ bits long and $m$ is of the same relative order of magnitude as $log\ n$, then a radix exchange sort can be implemented in the scan model which uses $O(m) = O(log\ n)$ operations [BG87]. In this case, the constant of proportionality is reasonably small. Further, this sorting strategy is also applicable when floating point representations of numbers is used. This parallel radix sort is the sorting algorithm assumed in the *scan* model analysis.

The rearrangement of the items in step 2, the item separation phase, can be accomplished by first using $n$ processors to enumerate the items in each set. This requires parallel prefix summation (*plus_scan*) operations in which each of the items in the set being enumerated contributes a *1* while the other set's items each contribute a *0*. The rearrangement can then be easily accomplished by adding the number of elements in one set to the other's enumeration values and using these values to indicate the new positions for the items. The enumeration steps require $O(log\ n)$ time in the EREW model and $O(1)$ time in the *scan* model. The rearrangement step and its preliminary calculations take $O(1)$ time in both models of computation.

The *large* items are those whose profit values satisfy $p_i > T$. Consequently, the minimum scaled profit value, and the minimum scaled profit of any feasible solution in step 4, satisfies

$$q_i = \lceil p_i/K \rceil > \lceil T/K \rceil = \lceil 2/\varepsilon \rceil.$$

The largest possible scaled profit obtainable is determined by $P^*/K \leq 8/\varepsilon^2$. Hence, no more than $(8/\varepsilon^2)/q_i$ items with scaled profit value $q_i$ can fit in any feasible, *large* item solution. This justifies the selection of the *large* item candidates in step 3.

The rearrangement of the *large* items for the purpose of selecting a minimal set of *large* item candidates can be performed by first sorting the *large* items using the values $((8/\varepsilon^2)-q_i)W+w_i$ as the sort keys. Here, $W$ is the maximum of the weights $w_i$ of the *large* items. This will arrange the *large* items into decreasing order of $q_i$, and within each segment of items having the same scaled profit $q_i$, the items will be in nondecreasing order of $w_i$. A *segmented plus_scan* of *1's* is used to obtain an enumeration of the numbers of elements in the $q_i$ segments in parallel. Next, a simple comparison of each item's segment index with the number of items allowed to be in the segment is performed to determine the items in each segment which should remain active for the *large* item optimization. The time requirements for step 3 are $O(log^2 n + log\ n)$ in the EREW model, owing to the sort and the *segmented plus_scan* operation, and $O(log\ n)$ in the *scan* model.

The feasible solutions in the dominance-based optimization on the *large* items are indexed by the possible profit values in the scaled problem. These index values are, in addition to $0$, between $2/\varepsilon$ and $8/\varepsilon^2$. It can be shown that the maximum number of items in any of the scaled *large* item solutions is bounded above by $4/\varepsilon$. Consequently, the feasible solutions in the *large* item computation can be maintained in $8/\varepsilon^2-2/\varepsilon$ sections of arrays with each section being of length $4/\varepsilon$. A group of similarly indexed array sections is used to maintain the list of indices of the *large* items placed into the feasible solution. In addition to the indices of *large* items placed into a solution, there must also be a flag to indicate which is the next available position for an index, and the capacity remaining for a particular feasible solution. Each feasible solution will be managed by $4/\varepsilon$ processors. To insure EREW memory access, each processor assigned to manage an array section (i.e. a scaled feasible solution) will have its own copy of the capacity remaining in the knapsack for its feasible solution, as well as its own copy of the actual, non-scaled profit for its feasible solution.

The general form of the *large* item computation is as follows:

```
for each large item candidate do
let i' denote the item's index
for all s such that $0 \leq s \leq (8/\varepsilon^2)-q_{i'}$ and $F(s) \neq \varnothing$
        and $W(s)+w_{i'} \leq M$ do
    if $F(s+q_{i'})=\varnothing$ or $W(s+q_{i'})>W(s)+w_{i'}$ then
        replace $F(s+q_{i'})$ by $F(s)$, add index i' to $F(s+q_{i'})$,
        add $w_{i'}$ to $W(s+q_{i'})$
```

This algorithm is a direct parallelization of the dominance-based optimization originally given by Ibarra and Kim [IK75]. For convenience, a feasible solution is denoted by $F(s)$ and its weight by $W(s)$, where the scaled profit of the feasible solution $s$ is its index. Any test involving a feasible solution indexed by s is performed by all $4/\varepsilon$ processors which manage the feasible solution, and is done in parallel. This is possible since the necessary flags and weight sums are replicated so that each processor has exclusive access to its own copy. The key to the SIMD parallelization is this data replication together with the fact that the feasible solutions can be checked for the addition of the next *large* item independently; this is the conditional in the if statement. When an item can be added to a feasible solution and the resulting new solution dominates another in the list, the dominated solution is replaced in a single step by using $4/\varepsilon$ processors. The dominance relation guarantees that all of the necessary updates can also be done in parallel and in an EREW memory reference fashion.

The number of processors needed for the *large* item computation is $(4/\varepsilon)(8/\varepsilon^2-2/\varepsilon)$. There are $max(n,\ (8/\varepsilon^2)\ log(4/\varepsilon))$ *large* item candidates, and hence this determines the time requirement for step 4.

The implementation of the *small* item augmentation of the feasible solutions in step 5 of the approximation scheme is carried out in two stages. In the first stage, the process for obtaining the final collection of feasible solutions does not attempt to remember the indices of any *small* items which would have fit into the remaining capacities of the solutions. In this way, all of the feasible solutions can test the same *small* item simultaneously for inclusion, by having the profit and weight of the item broadcast to all feasible solutions in parallel. After the feasible solution with the largest profit over both the *large* and *small* items is selected, a greedy algorithm is again performed on the *small* items. However, on this second execution of the greedy algorithm, the indices of the *small* items which augment the *large* item solution are saved.

Step 5 iterates over the *small* items. It can be seen that steps 4 and 5 together can iterate over all of the items, and hence the time requirement for these two steps together is $O(n)$. The following theorem summarizes the time and processor requirements for the parallel $\varepsilon$-approximation scheme.

**Theorem 2.** The parallel $\varepsilon$-approximation scheme takes $O(n+log^2 n+log\ n)$ time in the EREW model and $O(n+log\ n)$ time in the scan model. The scheme requires $max(n,\ (4/\varepsilon)(8/\varepsilon^2-2/\varepsilon))$ processors in both parallel models of computation.

## 4. CONCLUSIONS

An SIMD algorithm has been presented which is a parallel $\varepsilon$-approximaiton scheme for the 0/1 knapsack problem. The algorithm implements a dominance-based dynamic programming technique in an exclusive read, exclusive write shared memory model of SIMD programming. The number of processors needed by the algorithm is within current technological bounds for moderate values of $\varepsilon$ and for problem instances into the tens of thousamds of items.

## REFERENCES

[AKS83] Atjai, M., Komlos, J., and Szemeredi, E. (1983), "An O(NlogN) Sorting Network," In *Proceedings of the 15th ACM Symposium on the Theory of Computing* (Boston, Mass., Apr.), ACM, N.Y.C., pp. 1-9.

[BG87] Blelloch, G. (1987), "Scans as Primitive Parallel Operations," In *Proceedings of the 1987 International Conference on Parallel Processing* (St. Charles, Ill., Aug. 17-21), Penn State Press, University Park, Penn., pp. 355-362.

[FF83] Fich, F. (1983), "New Bounds for Parallel Prefix Circuits," In *Proceedings of the 15th ACM Symposium on the Theory of Computing* (Boston, Mass., Apr.), ACM, N.Y.C., pp. 100-109.

[GJ79] Garey, M., and Johnson, D. (1979), *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, Calif.

[GW88] Gerasch, T., and Weidman, S. (1988), "Massively Parallel Computing Applied to 0/1 Knapsack Problems," In *Proceedings of the Third International Conference on Supercomputing*, Volume II (Boston, Mass., May), International Supercomputing Institute, St. Petersburg, Fla., pp. 448-451.

[GRK86] Gopalkrishnan, P., Ramakrishnan, I., and Kanal, L (1986), "Parallel Approximate Algorithms for the 0/1 Knapsack Problem," In *Proceedings of the 1986 International Conference on Parallel Processing* (St. Charles, Ill., Aug. 19-22), IEEE, Washington, D.C., pp. 444-451.

[HS78] Horowitz, E., and Sahni, S. (1978), *Fundamentals of Computer Algorithms*, Computer Science Press, Potomac, Md.

[IK75] Ibarra, O., and Kim, C. (1975), "Fast Approximation Algorithms for the Knapsack and Sum of Subsets Problems," *Journal of the ACM* 22, 4 (Oct.), pp. 463-468.

[KD73] Knuth, D. (1973), *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley, Reading, Mass.

[KRS85] Kruskal, C., Rudolf, L., and Snir, M. (1985), "The Power of Parallel Prefix," In *Proceedings of the 1985 International Conference on Parallel Processing* (St. Charles, Ill., Aug. 20-23), IEEE, Washington, D.C., pp. 180-185.

[LE79] Lawler, E. (1979), "Fast Approximation Algorithms for Knapsack Problems," *Mathematics of Operations Research* 4, 4 (Nov.), pp. 339-356.

[LSS87] Lee, J., Shragowitz, E., and Sahni, S. (1987), "A Hypercube Algorithm for the 0/1 Knapsack Problem," In *Proceedings of the 1987 International Conference on Parallel Processing* (ST. Charles, Ill., Aug. 17-21), Penn State Press, University Park, Penn., pp. 699-706.

[LT84] Leighton, T. (1984), "Tight Bounds on the Complexity of Parallel Sorting," In *Proceedings of the 16th Annual Symposium on the Theory of Computing* (Washington, D.C., April 30 - May 2), ACM, N.Y.C., pp. 71-80.

[SS75] Sahni, S. (1975), "Approximate Algorithms for the 0/1 Knapsack Problem," *Journal of the ACM* 22, 1 (Jan.), pp. 115-124.

[QM87] Quinn, M. (1987), *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, Inc., N.Y.C.

# Fault Tolerant Message Routing on Large Parallel Systems

Jesse M. Gordon* and Quentin F. Stout†

Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI 48109-2122 USA

## Abstract

We consider the problem of designing massively fault tolerant message routing schemes for large parallel systems. Our notion of faults is extremely flexible and applies to all situations where a component is unavailable to participate in message communications. We focus our attention on the performance of schemes which use only local information to make local decisions. A framework for the analysis of fault tolerant routing schemes is presented and is used first to analyze the efficacy of minimal path routing methods.

We derive fault tolerant routing schemes by application of a technique called *sidetracking*. Viewed as making local decisions, a sidetracking scheme attempts to decrease the distance to the destination; if this is not possible, then the packet is routed randomly so as to increase the distance as little as possible. For single message routing on a hypercube, we show that the performance of a sidetracking scheme is near optimal, successfully routing with high probability and low average excess delay. We also present applications of the sidetracking technique to single message routing on a two dimensional mesh, and to multiple message permutation routing on a hypercube.

**Keywords:** Fault tolerant routing, distributed memory multiprocessors, hypercube computer, mesh computer, oblivious routing

As the size of parallel computer systems grows larger, so does the probability of component failure. Since the corresponding mean time to failure is likely to be short, off-line fault diagnosis and the subsequent replacement of failed units is not an attractive alternative for dealing with the problem. Rather, we would like to be able to continue system operation in the presence of such failures. To do this, we must identify the failed components and then avoid their usage. In this paper, we examine the fundamental problem of message communications, studying it in the context of faulty parallel systems.

A basic choice in a (message) routing algorithm is the amount of information about the system that each processor must possess. In a large parallel system, methods which require each processor to have global knowledge of the system incur massive overhead, both in the space required to store such information and in the time required to update the information to reflect changes in system status. Hence, we examine local routing methods, which require processors to have knowledge only of the status of neighboring processors and communication links. We are motivated by the belief that simple algorithms, operating under simple assumptions, can ensure high probabilities of successful message routing in the presence of faults.

## 1 Preliminaries

We study two specific parallel systems in this paper: *hypercubes* and *two-dimensional meshes*. A hypercube contains $2^n$ processors addressed by $n$-bit strings, and any two processors are connected by a (bi-directional) communication link if their addresses differ in exactly one bit position. A two-dimensional mesh contains $n^2$ processors addressed by integer pairs $(x, y)$ with $0 \leq x, y \leq n-1$, arranged so that the processor with address $(x_1, y_1)$ is connected to the processor labeled $(x_2, y_2)$ if either $x_1 = x_2$ and $|y_1 - y_2| = 1$ or if $|x_1 - x_2| = 1$ and $y_1 = y_2$. Both systems are regular (fixed degree) networks and can be viewed as special cases of a product graph known as the *k-ary n-cube*. Viewed in this manner the hypercube is a 2-ary (or binary) $n$-cube and the mesh is an $n$-ary 2-cube. Both of these systems are distributed memory, packet switched systems in which communication time is assumed to predominate, and local processing time can be ignored. In a packet switched system, messages are transmitted in units called *packets*; packets being routed from one processor to another are temporarily stored in the memory of intermediate processors and are later forwarded to their destinations. In this paper, we will concern ourselves solely with short messages, i.e. those which are no longer than one packet in length.

To keep the routing decisions simple, we concentrate our analysis on oblivious routing schemes. In an *oblivious* scheme, the path taken by a message depends only upon its source and destination. Such routing schemes are also called non-adaptive as the path taken by a given message is not responsive to other message traffic in the system. One such routing scheme is *deterministic* routing in which obliviousness means that for each source-destination pair there is a unique route which any message with that source-destination specification must take. Another routing scheme is *random* routing and is based on Phase B of Valiant's probabilistic routing scheme [3]. In this case, obliviousness means that for each source-destination pair there is a fixed probability distribution (independent of any other pair) that specifies for each path from the source to the destination the probability that the path will be taken. For these routing schemes, we consider initially the case of minimal length paths.

*Supported in part by NSF Grant No. DCR-85077851 and a Fellowship from the Unisys Corporation.

†Supported in part by NSF Grant No. DCR-85077851 and a DEC Incentives in Excellence Award.

In both hypercubes and meshes, local routing decisions are made based on the status of all neighbors which lie on minimal paths between the current location and the message destination. On the hypercube, deterministic routing chooses to traverse the lowest numbered available dimension while random routing selects uniformly among the available dimensions. On a mesh, there are two basic classes of routing algorithms: *direction uniform* and *path uniform.* Using a direction uniform approach, deterministic routing always favors the x-direction (if it is available), while random routing chooses uniformly between the x-direction and the y-direction. Using a path uniform approach, deterministic routing favors the direction with the largest distance to go, breaking ties in favor of the x-direction, while random routing chooses between the directions in proportion to the distance remaining to route in each.

We consider two basic routing problems. The first problem is routing a single message from an arbitrary source to an arbitrary destination. At the other extreme, we consider the problem of full permutation routing. Initially, there is a message at each processor, each with a distinct destination address; the task is to route every message to its destination simultaneously.

We model the amount of fault information available to processors in two ways. Model 0 (respectively, Model 1) assumes that individual processors do not know (respectively, do know) which, if any, of their immediate neighbors (processors and/or communication links) are faulty. In our study, a fault is assumed to render the processor or link non-functional for purposes of communicating messages. As such, we may also consider communications hot spots as processor faults, and our results also apply to routing in congested parallel systems. In this paper, we consider only processor faults, but our results may be extended easily to apply to systems with communication link faults.

The final component of our framework for the analysis of fault tolerant routing schemes is the modeling of fault distributions. In this model, faults are distributed independently and binomially, with each processor having a fixed probability, $p$, of failure. We assume that the distribution of faults is chosen before any routing occurs and that this fault pattern remains fixed for the duration of the routing attempt. While this assumption is useful in our analyses, it is not required by the routing algorithms themselves. We start by analyzing the a priori probabilities of successful routing of a single, indivisible message under each of our possible sets of assumptions. We obtain results which depend upon the size of the system (number of processors) as a parameter. We then derive asymptotic results by allowing system size to grow arbitrarily large. In most of these cases, the asymptotic bounds are achieved by the time system size is one thousand processors. Following this, we introduce our technique called sidetracking and analyze the behavior of routing schemes obtained by its application.

## 2 Single Message Routing

The single message routing problem is to deliver a message sent from a non-faulty source to a non-faulty destination. Under the assumptions of Model 0, our first routing information model, individual processors have no knowledge of the fault status of any other processors in the system. Hence, minimal path message routing succeeds if and only if all processors on the chosen path from the source to the destination are non-faulty. If the source and destination are $d$ steps away, then the probability of successful message routing is $(1-p)^{d-1}$. This result holds for both deterministic and random routing. On the hypercube, for antipodal source and destination processor, we denote this probability as

$$p^0(n,p) \;=\; (1-p)^{n-1}\,.$$

Since $\lim_{n\to\infty} p^0(n,p) = 0$, we see that the no-information model is not a useful one for deriving massively fault tolerant routing schemes.

The second routing information model, Model 1, assumes that processors know the fault status of their immediate neighbors. Routing schemes using this assumption make their local decisions from the set of all *non-faulty* neighbors of the processor. On the hypercube, routing using one-step local information yields a great improvement over routing methods which have no fault knowledge. The corresponding probability of successful message routing for antipodal source and destination processors is

$$p^1(n,p) \;=\; \prod_{k=2}^{n}(1-p^k)\,.$$

Using real analysis, we can show that $\lim_{n\to\infty} p^1(n,p)$ converges. By computing the values (which are shown in Table 1) we see that the probability of successful message delivery is high even for an exceedingly large number of faults. In particular, with 50% of the processors faulty ($p = 0.5$), a single message successfully reaches its antipodal destination 57.76% of the time. We note that these asymptotic values are attained in all cases before $n = 10$. In other words, for hypercubes with more than one thousand processors, the probability of successful single message minimal path routing is essentially independent of system size. A derivation of these results and a more detailed discussion of fault tolerant single message routing on a hypercube is contained in [2].

For single message routing on the mesh, we have presented four basic minimal path routing strategies: deterministic direction uniform (denoted DD), random direction uniform (RD), deterministic path uniform (DP) and random path uniform (RP). As was the case for the hypercube, we can derive closed form solutions for the probability of successful message routing using each of these schemes. For maximally distant source and destination processors, using Model 1, the direction uniform solutions are:

$$\mathrm{DD}^1(n,p) \;=\; (1-p)^{2n-3}\sum_{i=0}^{n-1}\binom{n-2+i}{i}p^i$$

and

$$\mathrm{RD}^1(n,p) \;=\; 2\sum_{i=0}^{n-2}\binom{n-2+i}{i}\left(\frac{1-p^2}{2}\right)^{n-1+i}(1-p)^{n-2-i}\,.$$

Unfortunately, because of the fixed degree of the processors, we are able to show that all four routing algorithms have an asymptotic success probability of zero. In fact, most of the probabilities are nearly zero by the time $n = 32$, or by time mesh size is about one thousand processors.

We next analyze the behavior of *sidetracking*, an oblivious routing method which combines the concepts of local information and randomization. Using sidetracking, and in the one-step local information routing model, a message will be routed forward using random routing. If the message reaches a blocked processor (no non-faulty neighbors along a minimal path to the destination) it will be sent to a non-faulty neighbor, chosen uniformly at random from the set of non-faulty neighbors. We contrast the sidetracking approach with the scheme known as *deflection routing*. While both schemes are non-minimal path schemes, deflection routing is an *adaptive* routing method whose goal is to ensure that messages are not delayed by waiting in queues. To this end, a message will be routed along a forward edge if one is available; if not, the packet will be sent along a backward edge. We use simulation experiments to determine the performance of sidetracking schemes, analyzing the probability of successful routing and the expected path length of a routed message.

The empirical performance of the sidetracking algorithm demonstrates convincingly its efficacy as a fault-tolerant routing scheme. On the hypercube, for the case of single message message routing, we find that, in the limit as the cube dimension grows larger and for a fixed probability of processor failure, the probability of successful message routing, $S(n,p)$, tends to 100%. In addition, the delay incurred by routing along non-minimal paths is small; with one-half of the processors faulty ($p = 0.5$), the average excess path length for a cube of dimension $n = 20$ is less than 4 steps. On the mesh, for single message routing, sidetracking is unable to provide any asymptotic performance improvement as, in the limit, its success probability also tends to zero. However, in meshes of small size, where successful routing is possible, sidetracking results in a substantial performance improvement. The results discussed in this section are summarized in two tables: hypercube results in Table 1 and mesh results in Table 2. For sidetracking, the maximum path length is the number of routing steps allowed before a message will be declared not routed. The respective minimum path lengths are 20 steps for the 20-dimensional hypercube, and 62 steps for the 32 x 32 mesh. Note that the cube results are for a hypercube of over a million processors and the mesh results are for meshes of over a thousand processors (using random path uniform routing).

## 3 Multiple Message Routing

For multiple messages, the worst case delay incurred by using oblivious routing schemes is high. For both hypercubes and meshes, the worst case time required to route any permutation is on the order of the square root of the number of processors in the network [1]. This worst case occurs since bad bottlenecking can be forced at a single processor. However, the use of randomization helps us to achieve better average case routing time. It does this by spreading out the message congestion. In particular, on a hypercube, routing a full permutation in a system with no faults, empirical studies (see [4]) show that minimal path random routing achieves near optimal results.

On the hypercube, we extended our simulation program to handle the case of multiple message routing. We studied the problem of routing a full permutation of the non-faulty processors of the system. Note here that, even though we have assured that all of our sources and destinations are non-faulty, they are no longer necessarily antipodal. The results of our experiments, this time for hypercube of about one thousand processors, are presented in Table 3. The values presented are averages over 100 iterations of each experiment. A routable message is one whose source and destination are in the same connected component. The values for the percentages of routable and routed messages are out of the percent of non-faulty processors (for the particular value of $p$). The results obtained show that the strong fault tolerant properties observed in the single message case have largely been carried over to the multiple message case. In particular, the average message delay for individual messages due to either time spent waiting in message queues or the use of non-minimal paths are impressively small. For instance, when $p = 0.4$, 99.9% of the messages are routed and the average message waits only 0.3 steps in queues and is routed only 1.8 extra steps. The only drawback is that time to complete the routing of all of the messages has increased greatly to 81.2 steps. Overall, these results show the viability of the sidetracking scheme as a massively fault tolerant multiple message routing scheme.

## 4 Summary

The basic conclusion to be drawn from our study is that the combination of randomization and local information is an effective one when it is applied to routing. For routing a single message on a hypercube in the presence of faults, we have shown the following asymptotic behavior. Using minimal path random routing with no local information, the probability of successful message routing ($p^0(n,p)$) is 0%. Using minimal path random routing with one-step local information, the probability of successful message routing ($p^1(n,p)$) is quite good, but tails off rapidly for a high probability of processor failure. Finally, using sidetracking (or *non*-minimal path random routing with one-step local information), the probability of successful message routing ($S(n,p)$) is 100%. For single message routing on a two-dimensional mesh, because of the fixed degree of the processors, the asymptotic routing success probabilities all tend to 0%. Lastly, we presented results that showed the efficacy and viability of sidetracking as a massively fault tolerant scheme on a hypercube.

## References

[1] A. Borodin and J. E. Hopcroft, "Routing, merging and sorting on parallel models of computation," *Journal of Computer and System Sciences*, vol. 30, pp. 130–145, 1985.

[2] J. M. Gordon and Q. F. Stout, "Hypercube message routing in the presence of faults," in *Proceedings of the 3rd Conference on Hypercube Concurrent Computers and Applications*, pp. 318–327, January 1988.

[3] L. G. Valiant, "A scheme for fast parallel communication," *SIAM Journal of Computing*, vol. 11, no. 2, pp. 350–361, May 1982.

[4] L. G. Valiant, "Optimality of a two-phase strategy for routing in interconnection networks," *IEEE Transactions on Computers*, vol. C-32, pp. 861–863, September 1983.

| Dim.<br>$n$ | Failure<br>$p$ | No Info.<br>$p^0(n,p)$ % | 1-Step Info.<br>$p^1(n,p)$ % | Sidetracking<br>$S(n,p)$ % | Avg. Path Length<br>(Maximum = 400) |
|---|---|---|---|---|---|
| 20 | .10 | 13.5 | 98.9 | 100.0 | 20.04 |
| 20 | .20 | 1.4 | 94.9 | 100.0 | 20.20 |
| 20 | .30 | 0.1 | 87.3 | 100.0 | 20.65 |
| 20 | .40 | 0.0 | 75.6 | 100.0 | 21.62 |
| 20 | .50 | 0.0 | 58.0 | 100.0 | 23.99 |
| 20 | .60 | 0.0 | 35.8 | 99.8 | 30.48 |
| 20 | .70 | 0.0 | 14.0 | 95.0 | 51.29 |
| 20 | .80 | 0.0 | 1.8 | 50.0 | 82.99 |
| 20 | .90 | 0.0 | 0.01 | 0.3 | 86.44 |

Table 1: Hypercube Single Message Routing Success Probabilities

| $n$ | Failure<br>$p$ | No Info.<br>$RP^0(n,p)$ % | 1-Step Info.<br>$RP^1(n,p)$ % | Sidetracking<br>$RP(n,p)$ % | Avg. Path Length<br>(Maximum = 186) |
|---|---|---|---|---|---|
| 32 | .10 | 0.0 | 48.1 | 89.8 | 66.81 |
| 32 | .20 | 0.0 | 6.1 | 44.2 | 78.45 |
| 32 | .30 | 0.0 | 0.3 | 5.8 | 89.96 |
| 32 | .40 | 0.0 | 0.0 | 0.1 | 87.78 |
| 32 | .50 | 0.0 | 0.0 | 0.0 | 0.00 |

Table 2: $n \times n$ Mesh Random Path Uniform Routing Success Probabilities

| | Processor Failure Probability, $p$ | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 0.0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 |
| % Routable Messages | 100.0 | 100.0 | 100.0 | 100.0 | 99.9 | 99.8 | 98.8 | 92.9 | 60.9 | 5.5 |
| % Routed Messages | 100.0 | 100.0 | 100.0 | 99.9 | 99.9 | 98.6 | 90.7 | 61.7 | 22.1 | 4.7 |
| Total Time Steps | 9.6 | 12.0 | 20.9 | 42.2 | 81.2 | 102.6 | 114.0 | 123.2 | 108.9 | 32.3 |
| Excess Path Length: | | | | | | | | | | |
| Average | 0.0 | 0.1 | 0.2 | 0.7 | 1.8 | 4.3 | 8.1 | 11.8 | 9.6 | 1.5 |
| Standard Deviation | 0.0 | 0.5 | 1.1 | 2.7 | 5.6 | 10.2 | 15.0 | 18.2 | 16.4 | 6.0 |
| Message Queue Delay: | | | | | | | | | | |
| Average | 0.1 | 0.1 | 0.2 | 0.2 | 0.3 | 0.4 | 0.7 | 1.9 | 3.0 | 0.9 |
| Standard Deviation | 0.4 | 0.4 | 0.4 | 0.5 | 0.5 | 1.4 | 3.1 | 7.6 | 10.3 | 3.7 |

Table 3: Hypercube Multiple Message Simulation Results ($n$ = 10)

# LOAD BALANCING FOR MASSIVELY-PARALLEL SOFT-REAL-TIME SYSTEMS

Max Hailperin*

Knowledge Systems Laboratory, Computer Science Department
Stanford University, Stanford, CA 94305

## ABSTRACT

Global load balancing, if practical, would allow the effective use of massively-parallel ensemble architectures for large soft-real-time problems. The challenge is to replace quick global communications, which is impractical in a massively-parallel system, with statistical techniques. In this vein, we propose a novel approach to decentralized load balancing based on statistical time-series analysis. Each site estimates the system-wide average load using information about past loads of individual sites and attempts to equal that average. This estimation process is practical because the soft-real-time systems we are interested in naturally exhibit loads that are periodic, in a statistical sense akin to seasonality in econometrics. We show how this load-characterization technique can be the foundation for a load-balancing system in an architecture employing cut-through routing and an efficient multicast protocol.

*Keywords: load balancing, real-time, time-series analysis.*

## INTRODUCTION

Our research group, the Stanford Knowledge Systems Laboratory Advanced Architectures Project, is exploring the construction of massively-parallel, object-oriented, knowledge-based, soft-real-time signal-interpretation systems. It seemed clear early on that some sort of adaptive load-distribution scheme would be necessary to allocate resources to such dynamic systems. Otherwise, in order to assure acceptable real-time performance, the system could only be lightly loaded, and the large-scale signal-interpretation problems the massive parallelism was intended to allow would not be possible. The remainder of this section explains why we desire a scheme which globally balances loads by migrating objects, and how we can exploit the somewhat periodic nature of our systems' loads to do global balancing in a manner appropriate to thousands of processing elements.

Much discussion in the load-distribution literature recently has centered on the choice of load balancing *vs.* load sharing (Ref. 12). While load balancing strives to keep all sites equally loaded, load sharing merely tries to prevent unnecessary idleness. Load balancing is appropriate to object-oriented real-time systems because

- real-time systems need to prevent long waits for processing—load balancing, by reducing the variance as well as the average of waiting times better achieves this; also,
- migrating objects to balance current load tends to also balance the future arrival of additional work at sites.

Traditionally, decentralized adaptive load-balancing systems have been local: they balance loads in small neighborhoods (the neighborhoods may be logical, rather than physical), and rely on repeated local adjustments to achieve global balance. (For example, see the *diffusion* scheme in Ref. 11.) We find this inappropriate to our circumstances because

- modern interconnection networks employing cut-through or wormhole routing reduce the importance of locality (Ref. 7),
- local techniques can fall prey to oscillation and wave-front-like propagation in the face of non-ideal conditions, and
- local techniques have difficulty responding quickly enough for dynamic and time-critical systems.

A global load-balancing system must somehow allow each site to estimate the current (or near-future) system-wide total load, in order that it may acquire or jettison sufficient work to bring its own load to the system-wide average. This seems incompatible with the constraints of a massively-parallel system: a site in a massively-parallel system must wait a considerable time to acquire global knowledge.

This apparent contradiction can be reconciled by using a stochastic time-series model to use prior load information to predict current loads. However, this approach is useless in most computer systems, as their loads are not very predictable.

Luckily, the real-time systems we are interested in (and many others) exhibit a different behavior. Their loads are periodic—not rigidly so, but rather in the same loose, statistical sense as many economic variables are seasonal. This periodicity is induced by sampled or scanned inputs and by sample-to-sample or scan-to-scan consistency in the outside world. Periodicity makes the loads more predictable, at least for lead times not greater than the period. As the period is generally relatively long, each site can have complete knowledge of loads at least through one period ago. This allows reasonably accurate prediction of current (or near-future) system-wide loads.

*An expanded version of this paper was published as Stanford Computer Science Department Technical Report STAN-CS-88-1222. This material is based upon work supported under a National Science Foundation Graduate Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation. This work was also supported by DARPA Contracts F30602-85-C-0012 and MDA903-83-C-0335, NASA Ames Contract NCC 2-220-S1, Boeing Contract W266875, and Digital Equipment Corporation. I thank Anoop Gupta, Bruce Delagi, Harold Brown, John Hennessy, and the entire Advanced Architectures Project for their assistance.

CH2649-2/89/0000/0159$01.00 © 1988 IEEE

Notice that the statistical nature of this approach makes it appropriate to massively-parallel systems with thousands of processing elements:

- The large number of sites makes more straightforward methods employing global communications impractical.
- On the other hand, the large number of sites is necessary to make the statistical methods valid.

We are not suggesting this approach for real-time systems which are rigidly periodic; more direct use can be made of their periodicity.

## AN EXAMPLE TIME SERIES

In this section we examine the evolution over time of the system-wide load in one of our real-time systems—an aircraft tracking and classification system (Ref. 14). We show that a simple stochastic model reasonably approximates this time series, that it is consistent with a common-sense understanding of the system, and that it allows moderately accurate prediction without recent complete information.

The solid line in Figure 1 shows the load over a ten-period interval out of the larger time series which was analyzed. Each period is ten time quanta long, and the load value for each quantum is an average total of task queue lengths over that quantum.

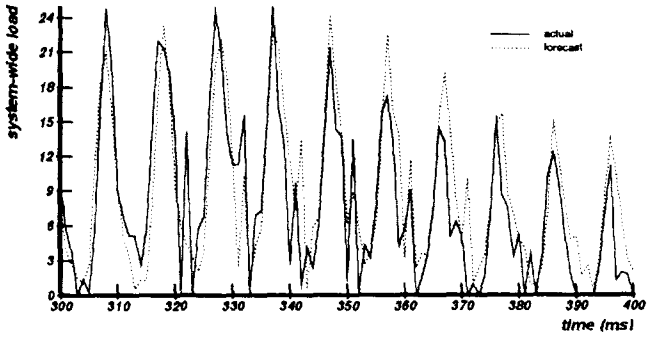


*Figure 1. Example Time Series. A sample of a load time series, with forecast from data through one period earlier.*

Notice that the pattern gradually shifts from period to period. Also, notice that as the observed activity diminishes, the system's performance varies from not quite keeping up with the input to having a relatively long period of quiescence between cycles. It is characteristic of real-time systems that they are sized so as to perform acceptably during peak periods, even if this means idleness at other times; this allows the periodicity of the input to show through as a periodicity of the load.

### Stochastic Model

We analyzed this series using the methods of Box and Jenkins (Ref. 3), and identified as a suitable first-cut model for it a multiplicative integrated moving average (IMA) process of orders (0,1,1) x $(0,1,1)_{10}$. This model has the form:

$$z_t = z_{t-1} + z_{t-10} - z_{t-11} + a_t - \theta a_{t-1} - \Theta a_{t-10} + \theta\Theta a_{t-11}, \qquad (1)$$

where $z_t$ is the system-wide load, $a_t$ is a white-noise series, and $\theta$ and $\Theta$ are parameters. The structure of this process is more evident when written using the backwards shift operator $B$:

$$(1 - B)(1 - B^{10})z_t = (1 - \theta B)(1 - \Theta B^{10})a_t. \qquad (2)$$

Adding the constraint that loads must be non-negative improves this basic model.

This model, while suggested by statistical evidence, is also plausible in terms of the mechanism of the system. The non-periodic component of the model essentially states that the load persists, except that it is subject to random perturbations. Some fraction ($\theta$) of each random perturbation is of short-term effect only, while the remainder lasts until counteracted; this fits well with a birth-death view of processes. The periodic component of the model is identical in form, and can be similarly justified: the aircraft under observation (and thus the load pattern) remain constant except for random perturbations, some fraction ($1-\Theta$) of which are long-lasting entries or departures from the field of observation.

This model belongs to the broad class of stochastic processes known as ARMA (autoregressive—moving average) processes. It is interesting to ask why this particular ARMA process should be chosen—might others not fit as well? The answer is partially that this is the simplest periodic ARMA process whose periodic and non-periodic components are both:

- non-stationary (i.e., they have no fixed level),
- stable (i.e., they don't grow explosively), and
- homogeneous (i.e., everywhere self-similar except for level).

### Forecasting

The non-periodic component of the model is that which is conventionally used for aperiodic computer systems; it gives rise to the familiar exponentially-weighted average forecast function. The periodic component in effect adds an exponentially-weighted average of corrections to this forecast, derived from the experience at corresponding points in earlier periods.

Depending on the relationship between $\theta$ and $\Theta$, the heaviest weight in the forecast may either be on the most recent value, or on the one a period ago. In the aircraft tracking case (and many others, we speculate), there is more consistency from period to period than from instant to instant (as aircraft are more inertial than processes).

Forecasts can also be computed directly from the difference equation we used to define the model. In either case, forecasts for greater lead times can be calculated by repeated use of the step-ahead formula. (By lead time we mean the time from when the total load is last known to when the forecast is for.)

Since the period (in this case, the scan time of a radar) is long relative to the communication latencies of the system, it is reasonable to suppose that each site can have complete knowledge of all other sites' loads at least up until one period earlier, with diminishing knowledge thereafter. It should be possible in principle to make some use of the more recent, incomplete, information to improve the forecast, given a model of the load distribution with load balancing. In the next section we address this problem and show a heuristic solution. However, Figure 1 shows that even forecasts made using only data up through one period in advance are usually moderately accurate.

### How Typical Is This Example?

Though this section presented a case study of a single time series taken from a single application, we believe the basic features are common to other systems as well. Preliminary results from experimentation with a passive radar interpretation system (Ref. 4) confirm this belief. The IMA (0,1,1) x $(0,1,1)_p$ model used here may well suit many such systems, though its suitability should of course be tested in each case. As well as testing the suitability of the model to a particular application, it is necessary to tune the parameters using sample time series. Systems with more than one period, for example from heterogeneous sensors, would necessitate a straightforward extension of the model.

One potential stumbling block in generalizing this technique to more realistic systems is that higher-level processing tends to be triggered by significant changes in the input (or by the lack of expected changes), rather than by the input itself. For example, a system that not merely tracks aircraft, but also attempts to deduce possible objectives, would reconsider the objective of an aircraft that sharply turned, or that failed to turn when it was expected to. This reduces the scan-to-scan consistency of the load. It remains to be seen how troublesome this is; clearly this depends on how much of the processing is special-case. When this issue came up in a discussion with a group familiar with actual systems, the consensus was that the load on present-day systems is indeed quite periodic (Ref. 13).

## INCOMPLETE INFORMATION

The simple stochastic model presented in the preceding section only allows load information old enough to be complete (i.e. available from all sites) to be used. In this section we refine our model to allow incomplete information (i.e., more recent loads from some sites) to be employed. We formulate the problem, show an exact but impractical solution, and then present provably good practical heuristic approximations.

### The Problem

In order to understand what use a site can make of recent but incomplete information, we must refine our model to include how the system-wide total load is divided among the $N$ sites. A simple, plausible version of this is to assume that the sites are independent instantaneously, but in the longer-term are successfully balanced. Formally, the model we have in mind is

$$z_{i,t} = a_{i,t} + \frac{z_{t-1} + z_{t-10} - z_{t-11} - \theta a_{t-1} - \Theta a_{t-10} + \theta\Theta a_{t-11}}{N}, \quad (3)$$

where we use $z_{i,t}$ for the load of site $i$ at time $t$ (with $z_t = \sum_i z_{i,t}$) and similarly for $a_{i,t}$ and $a_t$ (the $a_{i,t}$ are independently normally distributed, with variance $\sigma_a^2$).

As long as all $z_{i,t}$ are known, the $a_{i,t}$ can be calculated, and thus used for forecasting. When the information is incomplete, the deviation of the known $z_{i,t}$ from the step-ahead forecasts can no longer be attributed solely to their corresponding $a_{i,t}$, but rather will also include the persistent fraction of earlier unknown perturbations. The problem is to find the expected division between these two sources of perturbation, as the expected value of each $a_{i,t}$ should be incorporated into the forecast in its own way.

### Exact Solution

This problem can be solved by applying Bayes's theorem:

- We are given as a prior distribution for the $a_{i,t}$ that they are independently normally distributed with some variance $\sigma_a^2$.
- We make observations which imply a joint likelihood for the $a_{i,t}$ that is uniform where certain linear combinations of them (given below) equal the known $z_{i,t}$ and zero elsewhere.
- We would like to find the posterior joint distribution of the $a_{i,t}$, specifically its expected value, for use in forecasting.

The non-zero regions of the likelihood function can be found by rewriting the equation for $z_{i,t}$ in terms of the $a_{i,t}$ alone, using the summation operators $S = (1 + SB)$ and $S_{10} = (1 + S_{10}B^{10})$:

$$z_{i,t} = a_{i,t} + \frac{((1-\theta)SB + (1-\Theta)S_{10}B^{10} + (1-\theta)(1-\Theta)SS_{10}B^{11})a_t}{N}. \quad (4)$$

The posterior distribution can readily be written using Bayes's theorem, provided one is willing to leave some messy integrals in it. Unfortunately, this leaves numerical integration as the only way to find the needed expected value. This seems to be too much work to expect a load-balancing system to perform each time interval. What is needed is a pre-posterior analysis—a general analysis done in advance, into which specific numbers can be plugged at run time. Unfortunately, we know of no such approach to this problem in the general case. In the next subsection we consider heuristic approximations appropriate to our intended implementation. The analysis above serves as the standard by which the heuristics are judged, as well as suggesting them.

### Heuristic Approximations

The simplest heuristic is to simply assume that the full deviation of each known load $z_{i,t}$ from its step-ahead forecast is purely its corresponding $a_{i,t}$. This heuristic is actually the truth (given our model) for the first time-quantum with incomplete information, and can be shown to be a conservative approximation provided there is less than a period of incomplete information. By a conservative approximation, we mean that this heuristic is guaranteed to be more accurate than simply ignoring the incomplete information. This is because mistaking the retained portion of prior perturbations for current perturbation leads to it's being erroneously re-multiplied by $(1 - \theta)$, i.e. underestimated.

We can improve this approximation by taking advantage of one feature of our intended implementation. We suggest a randomized style of information spreading known as "rumor mongering" which spreads each site's load information to an exponentially widening fraction of the other sites. Thus the amount of load information a site has drops off exponentially with recency, and only the earliest incomplete load information is of any real significance.

In particular, for realistic parameters (e.g. a spreading factor of eight) the only significant improvement that could be made in the above simple heuristic would be to better account for the deviations observed in the second incomplete-information time-quantum. Moreover, this division between the first two incomplete-information time-quanta need not make use of information from later time-quanta, as such information would be very weak under these assumptions. This leaves a tractable two-quanta version of the general problem of the preceding subsection.

The $a_{i,t}$ from the $N_n$ non-reporting sites of the first quantum can be lumped together, as can those from the $N_r$ reporting sites of the second quantum. This is because of the symmetry amongst them. We will call the contribution of the former to the second-quanta deviations $X$ and that of the latter $Y$. Our prior distributions for them are independent, normal, both have mean zero, and (by elementary probability theory) have the variances

$$\sigma_x^2 = \frac{N_r^2}{N^2}(1-\theta)^2 N_n \sigma_a^2 \tag{5}$$

$$\sigma_y^2 = N_r \sigma_a^2. \tag{6}$$

We know that $X$ and $Y$ sum to the observed deviation, $\delta$, of the second-quanta loads from their step-ahead forecasts. Therefore, the posterior distribution from Bayes's theorem gives us the following posterior expected values:

$$E(X) = \frac{\int_{-\infty}^{\infty} x e^{-x^2/2\sigma_x^2-(\delta-x)^2/2\sigma_y^2} dx}{\int_{-\infty}^{\infty} e^{-x^2/2\sigma_x^2-(\delta-x)^2/2\sigma_y^2} dx} \tag{7}$$

$$= \delta \frac{\sigma_x^2}{\sigma_x^2+\sigma_y^2} \tag{8}$$

$$E(Y) = \delta \frac{\sigma_y^2}{\sigma_x^2+\sigma_y^2}. \tag{9}$$

Thus we can readily at run time use the observed values of $\delta$, $N_n$ , and $N_r$ to calculate a very good approximation to the best forecast possible with the available information.

## PRECISION OF FORECASTS

In this section we analyze the potential for practical utility of our load-characterization scheme. We show that for the large numbers of sites characteristic of massively-parallel architectures, our scheme provides load estimates which are accurate enough to be useful for load balancing.

We can use the model of Equation 1 to calculate probability limits of forecasts—that is, the region around the forecast in which the actual system-wide load will lie some specified fraction of the time. Additionally, the more detailed model of Equation 3 specifies how the individual sites' loads can be expected to be distributed about the system-wide average load. What is most interesting is combining these two, in order to determine

- what fraction of the sites can be expected to be over- or under-loaded at some significance level, and
- how much relative error can be expected in the amount of work transferred between sites, due to erroneous forecasts.

Happily, we show that the accuracy of the forecasts relative to the standard-deviation of the site loads goes up with the square-root of the number of sites, so that for massively-parallel systems the uncertainty in the forecasts is unproblematic (assuming the validity of the model).

### Probability Limits of Forecasts

The conditional probability distribution of the system-wide load about its forecast value is simply the sum of those of the $a_t$ not included in the forecast. The error in the forecast will thus be normally distributed with mean zero and variance increasing with lead-time. For the IMA (0,1,1) x $(0,1,1)_p$ model, if the forecast is made using complete information only, with lead time $l < p$, the variance is

$$V(l) = (1+(l-1)(1-\theta)^2)N\sigma_a^2. \tag{10}$$

We can use the above formula to calculate approximate probability limits for the forecasts by substituting an estimate for $\sigma_a$. If the system-wide load sample standard deviation is $s$, then we can estimate that with probability $\varepsilon$ the actual load differs from the lead $l$ forecast by more than $u_{\varepsilon/2} s\sqrt{1+(l-1)(1-\theta)^2}$, where $u_{\varepsilon/2}$ is the $\varepsilon/2$-tail-area point of the unit normal distribution. Notice that these bounds are for the total load---the standard deviation, and hence probability limits, for the average load are smaller by a factor of $N$.

### Comparison with the Distribution of Site Loads

Our model asserts that the loads of the individual sites at any time are normally distributed about the system-wide average load with standard deviation $\sigma_a$. We can compare this with the standard deviation of the lead $l$ conditional probability distribution of the average load, which we derived in the previous subsection. The latter is larger by a factor of $\sqrt{1+(l-1)(1-\theta)^2}/\sqrt{N}$; the factor of $\sqrt{N}$ results from averaging $N$ independent deviates.

This implies that for large systems the forecasts will be accurate enough to be useful. For example, our example system could be spread among 1024 sites, even one-period-ahead forecasts would have a factor of 27 lower standard deviation than the site loads. Thus virtually all apparent over- or under-loads would be statistically significant, and the relative error in the amount of work transferred would be small (roughly 1/27).

## LOAD-BALANCING MECHANISM

In this section we outline a load-balancing scheme employing the load-characterization methodology of the preceding sections. Our scheme relies on a "rumor mongering" style of information spreading (Ref. 9), which is appropriate to our architecture. We show that the mechanism not only allows sites to assess their load with respect to the system-wide average, but also allows overloaded sites to reliably find sufficiently underloaded sites to which objects can be migrated.

If each site stores its knowledge of all sites' load histories, then they can spread their information around by a process of "rumor mongering"—that is, by randomly sharing information (Refs. 10, 1, 2, 9). Naturally, the histories can be compressed by discarding information old enough to be scarcely relevant and by combining together loads from all sites where they all are known. Some information may be young enough to relevant to forecasting, but old enough to be well-known. This information can be retained but not passed on; (Ref. 9) has a good discussion of such issues.

Our CARE ensemble architecture (Ref. 8) uses a cut-through interconnection network, so latency is not proportional to distance (in the absence of contention). Additionally, it supports an efficient multicast protocol (Ref. 5). Therefore, we suggest that the information spreading be achieved by each site periodically multicasting its information to a random sample of the other sites. While the number of sites that each site will hear from in any given period varies, it can be shown that the distribution (a binomial distribution, rapidly approaching a Poisson distribution) is such that a paucity of information will be rare, even with a quite moderate sample size, e.g. eight.

C-7

Upon receiving a load-information message, a site should integrate the information into its own knowledge, and then use the time-series model (provided *a priori* based on experiments with the particular system) to estimate the current system-wide average load with probability limits. It should then compare this predicted average with its own current load, and with the load of the sender at the time of the sending. If the recipient appears significantly underloaded and the sender appears significantly overloaded, a request for work should be sent back.

This is a combination of random gossiping to distribute the information needed to decide whether and how much work to transfer, together with polling/bidding to match up the participating sites. As with all bidding schemes, some precautions are needed to avoid races. The underloaded site should not place any other requests for work until it receives work or an apology from the overloaded site. As the inter-arrival time for messages from overloaded sites should be high relative to the round-trip message time, few conflicts should occur.

It should be rare that an overloaded site cannot find enough total underload among the sites it samples to match its own overload. For example, suppose that the loads are normally distributed (as they are in the model of Equation 3), and that the sample size is eight. Of the eight sites sampled, it can be expected that four will be underloaded. The expected value of the absolute value of a normal deviate is $2/\sqrt{2\pi}$, or about .8 standard deviations, so the four underloaded sites will on the average have approximately 3.2 standard deviations worth of underload. But the originating site must really be far out on the tail of the distribution to have more than 3.2 standard deviations worth of overload.

The only aspect of load balancing not addressed by this mechanism is the choice of which objects to migrate. Here again the real-time nature of the system must be addressed. In general neither the highest- nor lowest-priority objects are best migrated, so as to neither unfairly advance a low-priority object nor hold up (due to migration time) a high-priority object. Chang addresses these issues in (Ref. 6).

## REFERENCES

1. Yeshayahu Artsy, Hung-Yang Chang, and Raphael Finkel. Processes migrate in Charlotte. Technical Report 655, Computer Sciences Department, University of Wisconsin–Madison, August 1986.
2. Amnon Barak and Amnon Shiloh. A distributed load-balancing policy for a multicomputer. *Software—Practice and Experience,* 15(9):901–913, September 1985.
3. George E. P. Box and Gwilym M. Jenkins. *Time Series Analysis: Forecasting and Control.* Holden-Day Inc., 1976.
4. Harold D. Brown, Eric Schoen, and Bruce A. Delagi. An experiment in knowledge-based signal understanding using parallel architectures. Technical Report STAN-CS-86-1136, Department of Computer Science, Stanford University, October 1986.
5. Gregory T. Byrd, Russell Nakano, and Bruce A. Delagi. A dynamic, cut-through communications protocol with multicast. Technical Report STAN-CS-87-1178, Department of Computer Science, Stanford University, September 1987.
6. Hung-Yang Chang. Dynamic scheduling algorithms for distributed soft real-time systems. Technical Report 728, Computer Sciences Department, University of Wisconsin–Madison, 1987.
7. William J. Dally. Wire-efficient VLSI multiprocessor communications networks. In *Advanced Research in VLSI: Proceedings of the 1987 Stanford Conference,* pages 391–415. The MIT Press, 1987.
8. Bruce A. Delagi, Nakul Saraiya, Sayuri Nishimura, and Greg Byrd. An instrumented architectural simulation system. In *Artificial Intelligence and Simulation: The Diversity of Applications.* The Society for Computer Simulation International, February 1988.
9. Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing,* pages 1–12, August 1987.
10. Zvi Drezner and Amnon Barak. A probabilistic algorithm for scattering information in a multicomputer system. Technical Report CRL-TR-15-84, Computing Research Laboratory, University of Michigan, March 1984.
11. Robert H. Halstead, Jr. and Stephen A. Ward. The MuNet: A scalable decentralized architecture for parallel computation. In *Proc. 7th Annual Symposium on Computer Architecture,* pages 139–145, May 1980.
12. Phillip Krueger and Miron Livny. Load balancing, load sharing and performance in distributed systems. Technical Report 700, Computer Sciences Department, University of Wisconsin–Madison, August 1987.
13. Personal communication, September 10, 1987. Discussion with members of MIT Lincoln Laboratories Machine Intelligence Group.
14. Russell Nakano and Masafumi Minami. Experiments with a knowledge-based system on a multiprocessor. Technical Report STAN-CS-87-1188, Department of Computer Science, Stanford University, October 1987.

# EFFICIENT ALGORITHMS FOR MASSIVELY PARALLEL COMPUTERS
## I. DESIGN OF STABLE COMPUTATIONAL SYSTEMS USING LINEAR SYSTEMS MODELS

Harold M. Hastings
Department of Mathematics
Hofstra University
Hempstead, NY 11550

Ivan Kadar
Grumman Corporation
Mail Stop B35–35
Bethpage, NY 11714

## ABSTRACT

The authors investigate the stability of massively parallel computations using a linear systems approach. Stability is important for several reasons. These include bounding the response of the algorithm to numerical noise so that the typically small amount of local memory can be used efficiently, as well as designing algorithms and even hardware to be fault tolerant. Both Lyapunov stability (insensitivity to small changes in data and to noise) and structural stability (fault tolerance in hardware and software) are studied. The methodology is motivated by neural network modeling but may have larger applications.

*Keywords: Lyapunov stability, structural stability, massively parallel computing.*

## INTRODUCTION

### Approach

We chose to study the stability of massively parallel computation using linear systems for several reasons. Here and below, a linear system is a vector equation of the form

$$x(t+1) = Mx(t), \qquad (1)$$

where $x(t)$ and $x(t+1)$ are column vectors of length n and M is a square matrix of size n x n. An iterated linear system or nonautonomous linear system is specified by a sequence of matrices M(t), and vector equations of the form

$$x(t+1) = M(t)x(t). \qquad (2)$$

Our model for the stability of computation consists of random linear systems of the form (1) or random iterated linear systems of the form (2). Such systems are described by the following parameters of the matrices M or M(t):

| | |
|---|---|
| size | n |
| connectance | C of the associated digraph of interactions—this digraph contains an edge from j to i whenever M(i,j) is nonzero, and |
| distribution of nonzero entries of M, frequently in particular | |
| its mean | $\mu$ |
| and variance | $\alpha^2$. |

See References 1–6. The role of connectance is frequently to specify that there is one system and not two or more disjoint systems (Ref. 7).

Foremost among our reasons for this approach is that the stability of large systems has historically been studied with linear systems (Refs. 1–5). Many systems can be approximated by linear systems near equilibria. In particular, many algorithms (Gauss-Siedel iteration, finite element techniques, etc.) involve iterated linear maps. Much of neural network algorithms also involves linear maps (Refs. 8–9).

The response of an algorithm to a small amount of noise involves a small perturbation to the algorithm. Such perturbations are frequently studied via linear approximations.

The limits of cellular automaton models are partial differential equations, and frequently diffusion equations—which are linear. Thus, linear systems arguments may also be useful in studying massively parallel hardware. The particular relevance of the linear systems approach may easily be seen in the following representation of the graph associated with a linear system (1) and an iterated linear system (2), Figures 1 and 2, respectively. Figure 2 accurately represents data flow in a parallel computation with processing elements (PE's) 1, 2, ..., n using the vertex (j,t) to represent PE j at time t and an edge from (j,t) to (i,t+1) for each nonzero entry in M(t) representing the flow of data from PE j at time t to PE i at time t+1.

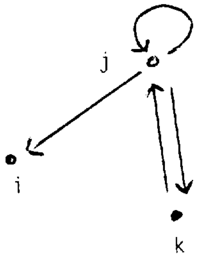


FIGURE 1. Part of the digraph of a linear system with M(i,j), M(j,j), M(j,k), and (M(k,j) nonzero. Also, M(j,i) = M(i,k) = M(k,i) = 0.

CH2649-2/89/0000/0165$01.00 © 1988 IEEE

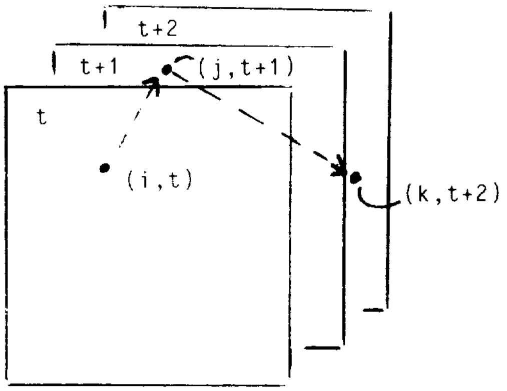


FIGURE 2. Part of the digraph of an iterated linear system, illustrating sheets for times t, t+1, and t+2. Here M(t) (j,i) and M(t+1) (k,j) are nonzero.

Theoretical results for linear systems are available, and may suggest the local behavior of nonlinear systems.

We shall investigate Lyapunov and structural stability of large, random linear systems, as neutral starting points for this type of modeling.

### Stability Definitions

We informally review the definitions of Lyapunov and structural stability and refer the reader to Reference 10 or a similar text for precise definitions.

An equilibrium of a system is (Lyapunov) stable if the system returns to that equilibrium when started from nearby points.

A system is structurally stable if nearby systems display the same dynamics (have similar stable equilibria, etc.). Bifurcations are the opposite of structural stability.

## STABILITY OF RANDOM LINEAR SYSTEMS: A REVIEW

Historically, the first result was due to Gardner and Ashby (Ref. 1) who studied the behavior of fully connected systems with fixed distributions as a function of size. They found that Lyapunov stability decreases with increasing size. Their results are in qualitative agreement with a standard stability bound, which follows from the Gerschgorin Circle Theorem that states that the size of the eigenvalues of a matrix M is bounded by its maximum row sum, where a row sum is the sum of absolute values of entries in a row. Consequently, systems (1) and (2) are Lyapunov stable if all row sums are less than 1.

The maximal row sum of M is also its L1 matrix norm. A similar result holds for other matrix norms, but these are much harder to compute or estimate. However, the Gerschgorin bound overestimates eigenvalues of typical random matrices wherein the nonzero entries have mean 0. The Gerschgorin bound does yield a useful asymptotic bound for suitably connected systems with nonzero mean (Ref. 4); the criterion is

$$\mu nC < 1. \tag{3}$$

May (Ref. 2) conjectured a more useful criterion for random linear systems of mean 0, following results of Wigner (Ref. 6). In this case, the conjectured asymptotic stability criterion is

$$\alpha^2 nC < 1. \tag{4}$$

This bound is much more delicate, and requires suitable technical hypotheses (Refs. 4–5) due to Cohen and Newman and Geman. Cohen and Newman showed that bound (4) also holds for iterated linear systems. Their argument also extends to bound (3) for iterated linear systems.

These results may be summarized loosely in the statement that with fixed interaction rules, Lyapunov stability decreases with increasing complexity (nC).

However, if interaction strength decreases sufficiently fast with increasing complexity, for example in the mean 0 case, if

$$\alpha = \sigma (\sqrt{nC}), \tag{5}$$

then Lyapunov stability increases with increasing complexity (Ref. 3, see also Ref. 11 for background).

Structural stability behaves in a more complex manner. In particular, one needs the mean interaction to decrease faster than linearly with increasing complexity:

$$\mu = o(nC). \tag{6}$$

Criterion (5) holds for many highly distributed representations (neural networks) and interactions. Therefore, Lyapunov stability typically increases with complexity in such systems. Structural stability, however, requires careful design considerations because for random systems one usually only obtains

$$\mu = O(nC), \tag{7}$$

a weaker condition than condition (6) faster than linearly with increasing complexity.

It is thus a key requirement that design translate fault tolerance questions into Lyapunov stability questions by designing large parallelism and redundancy.

## EXAMPLES

We describe several examples and interpretations (possible applications) of the above results.

### Large Parallel Computers

In this case, Lyapunov stability may be interpreted as response to noise, which may typically be increased by a highly distributed representation and many weak interactions. Structural stability (including fault tolerance) is more difficult to handle, and requires careful limitations of complexity or built-in redundancy, which implies scaling rule (6).

### Neural Networks (highly distributed representations)

As explained above, Lyapunov stability may be interpreted in terms of stable responses to small changes in inputs, and the above calculations provide a theoretical basis for the empirical fact that highly distributed' systems provide such stable responses. In addition, adaptive programming as in neural nets can provide structural stability with respect to hardware and software problems.

### Stable Algorithms

We simply remark that many computation schemes such as Gauss-Seidel iteration and finite element methods for diffusion equations are probably both Lyapunov and structurally stable.

## CONCLUSIONS

One can begin to usefully investigate stability of large computational systems using approaches from the study of other large systems. This idea begins with Gardner and Ashby (Ref. 1). Highly distributed interactions and representations can increase stability.

## REFERENCES

1. Gardner, M.R. and W.R. Ashby. Connectance of large dynamic (cybernetic) systems: critical values for stability. *Nature* 228, p. 784.

2. May, R.M. Will a large complex system ever be stable? *Nature* 238, pp. 413–414.

3. Hastings, H.M. Stability of large systems. In *Proceedings, International Conference of Qualitative Theory of Differential Equations*. Eds. W. Allegro and G.J. Butler, University of Alberta, Canada, 1985, pp. 203–209.

4. Cohen, J.E. and C. Newman. The stability of random matrices and their products. *Annals of Probability,* 1985.

5. Geman, S. The spectral radius of large random matrices. Preprint, Brown University, 1984.

6. Wigner, B. Statistical properties of real symmetric matrices with many dimensions. In *Proceedings, Fourth Canadian Mathematical Congress*. Ed. M.S. MacPhail, University of Toronto Press, 1959, pp. 174–184.

7. Bollobas, B. Graph Theory. *Graduate Texts in Mathematics* 63, New York: Springer-Verlag, 1979.

8. Rumelhart, D.E. and J.L. McClelland (eds.). *Parallel Distributed Processing,* Cambridge, MA: MIT Press, 1986.

9. Kadar, I. Robust tracking novelty filters based on linear models. *Proceedings, IEEE First International Conference on Neural Networks* 1987, pp. IV-611–IV-617.

10. Hirsch, M. and S. Smale. *Differential Equations, Dynamical Systems, and Linear Algebra.* New York: Academic Press, 1974.

11. Harrison, G. Stability under environmental stress: resistance, resilience, persistence and variability. *Amer. Natur.* 113, pp. 659–669.

# APPLIED GEOMETRIC ALGORITHMS ON BOOLEAN N_CUBE COMPUTERS

*Wen-Jing Hsu†, Xiaola Lin†, and Kuan-Tsae Huang‡*

†
*Department of Computer Science   Michigan State University   East Lansing, MI 48824   hsu@cpswh.msu.edu*
‡ IBM T. J. Watson Research Center Yorktown Heights, NY 10598 (914)789-7742 kth@ibm.com

**(Extended Abstract)**

Due to the polymorphism of the N-Cube interconnections, the N-Cube Computers have applications in numerous applications. The computer architecture is also one of the most promising for massively parallel processing in the future. Both algorithms and software tools for these parallel computers are in demand. To investigate techniques of designing parallel algorithms for this type of machines, we studied solutions for a class of geometric problems on the N-Cube parallel computers. The geometric problems are the *Convex Hull Problem*, the *Line Intersection Problem*, and the *Nearest Neighbors Problem*. These problems have found applications in VLSI design, Computer Graphics, Image Processing/Pattern Recognition, and Robotics. The selected problems are also known to be related to many other problems of theoretical as well as practical importance. Efficient solutions of these key problems have immediate applications and can lead to solutions of other problems.

**Key words:** *Parallel Algorithms, Hypercube, Geometric Problems, Divide-and-Conquer.*

## 1. Introduction

The Boolean N-Cubes, (or, N-Cubes, Hypercubes, for short) are parallel computers with the hypercube interconnection between the processors (Ref. 3). These parallel computers are becoming popular due to their relatively low costs (compared to other supercomputers like Crays) and vast potential. It is believed that this potential is derived from the following two structural advantages:

i) The computer is polymorphic:

It has been demonstrated that the Boolean N-Cubes can be programmed to simulate many other parallel architectures: linear arrays (for systolic, pipelined operations), meshes, trees, Pyramids, etc., where each of the architectures has identified application areas. Thus, the cube architecture is open to many applications, which is a great advantage over many other parallel architectures.

ii) The Boolean N-Cubes have a recursive structure:

An N-cube can be recursively divided into two isomorphic subcubes. This property matches very well with the recursive programming techniques ( e.g. *Divide-and-Conquer* (Ref. 1) ). Thus the N-Cube computers are a natural choice for designing recursive algorithms. Furthermore, the experience gained with the N-Cube computers should be of value to future system architects as well as programmers of current parallel computers.

## 2. The Geometric Problems

The problems studied fall in the domain of *Computational Geometry* (Refs. 4-5), a recent branch of *Analysis of Algorithms* (Ref. 3). Each of the problems (Convex Hull, Line Intersection, and Nearest Neighbor) actually has a cluster of related problems.

(1) the *Intersection Problem* is to determine whether two geometric objects (points, lines, polygons, etc.) share a common point. For example, the *Line-Intersection Problem* is to determine intersections of a set of lines segments. This problem has many variations, by specializing on the type of queries (e.g. *reporting* all instances or *detecting* one instance of intersections), or the type of objects involved (e.g. vertical/horizontal lines, half-planes, polygons, etc.).

CH2649-2/89/0000/0169$01.00 © 1988 IEEE

The problem of detecting intersections in a set of N line segments arises naturally in many applications. The solution can be applied to an important problem in graphics known as *hidden line elimination* when using a 2-D device to display 3-D objects. Also, an algorithm for finding intersection of vertical and horizontal line segments obviously can be used in design rule checking of VLSI layout. The problem also has application in pattern recognition and robotics where geometry is involved.

(2) the *Convexity Problem* is to determine properties related to convexity. For example, the *Convex Hull Problem* is to determine the convex hull for a set of points. There are many variations to this particular problem, where either the points may be dynamically or statically specified, or the point set may have some special properties ( e.g. the points lie on a simply connected polygon). The Convexity Problems, especially the problems related to computing convex hulls, have applications in Pattern Recognition and Statistics (Ref. 5). Thus, efficient algorithms for these abstract problems have immediate uses in the applications mentioned above.

(3) the *Proximity Problem* is to determine the neighborhood of given objects, based on the specified metrics. For example, the *Nearest-Neighbor Problem*, the *Euclidean Minimum Spanning Tree Problem*, and the *Triangulation Problem*, and the computation of the *Voronoi Diagrams* are in this category. The Proximity problems can be applied to Pattern Recognition, Finite Element Analysis, and path planning of robots (Ref. 5).

## 3. Existing Results

The geometric problems have received considerable attention, and many results have appeared recently. (For instance, there have been several conferences devoted entirely to Computational Geometry.) However, most of the results are geared toward sequential processing only. Comparatively, still very few upper bounds and virtually no lower bounds exist for parallel algorithms. We confine our survey to the parallel algorithms which are available in the more recent literature. Preparata and Lee (Ref. 4) give an excellent survey on the sequential algorithms.

(1) Convex Hull Algorithms

Chow (Ref. 13) was among the first to develop parallel algorithms for the geometric problems. Using a shared memory model, she showed that the convex hull of a planar point set can be determined in $O(log^2N)$ time. The number of processors is bounded by N. In (Ref. 19), Miller and Stout showed that the convex hull of a digitized picture input can be found in $O(N^{1/2})$ time on an $N$-node mesh. In (Ref. 20), Miller and Miller presented a Hypercube algorithm for computing Convex Hulls for an $M$ x $M$ digitized picture input. Using $N$ processors, the algorithm has a worst-case complexity of $O(M^2/Nt_c+P^2+M^{2/3}t_{s/r}+M^{2/3}t_c)$, where P denotes the number of "candidates" for the vertices on the convex hull; $t_c$ and $t_{s/r}$ denote, respectively, the time required for a computation step on the individual processor and the time required for sending (or receiving) a unit-length message. Note that, in both cases, the input points are represented by finite integer coordinates, which is a valid assumption in the applications they are concerned. However, it is a relatively strong assumption, and must be considered when doing comparisons.

(2) Intersection Algorithms

In (Ref. 18), Miller and Stout give asymptotically optimal ($O(N^{1/2})$-time) algorithms for using mesh computers to determine intersections among line segments and among polygons. Using a different approach, Jeong and Lee (Ref. 15) also give optimal results on mesh computers for the same problem. No N-Cube algorithm is currently available.

(3) Parallel Algorithms for Proximity Problems

Using the shared-memory model, Chow (Ref. 13) showed that the Voronoi diagram can be computed in $O(log^3NloglogN)$ time using N processors, where N denotes the number of input points. Aggrawal et al. (Ref. 11) presented an $O(log^3N)$ time parallel algorithm to solve the same problem with $O(NlogN)$ total space. Chow also presented an $O(log^4N)$ algorithm on the Cube-Connected-Cycles using $O(logN)$ storage space per processor. Lu (Ref. 16) showed that $O(N^{1/2}logN)$ time is sufficient to compute Voronoi diagrams on an $N^{1/2}$ x $N^{1/2}$ mesh computer, with constant space requirement on each processor. Recently, in (Ref. 15), Jeong and Lee improved the time bound to $O(N^{1/2})$, which is already optimal for the mesh computers. Currently no N-Cube algorithm for the Proximity problems is published.

The development of Pyramid computers, originally intended for image processing, pattern recognition, and computer vision applications, also led to the discovery of a cluster of geometric algorithms (Refs. 21-23). A pyramid of size N is defined to have an $N^{1/2}$ x $N^{1/2}$ mesh connected computer as its base, and $\log_4N$ levels of mesh connected computers above (Ref. 21). Assuming the input to be an

$N^{1/2}$ x $N^{1/2}$ digitized picture, Miller and Stout (Ref. 22) showed that the nearest neighbor problem can be solved in $O(N^{1/2})$ steps.

#### 4. New Results

We will present new algorithms for the following three problems on the hypercubes:

(1). Convex Hull Problem: specify the convex hull for a set of N points.

(2). Nearest Neighbor Problem: Find a nearest neighbor for each of the N planar points.

(3). Line Intersection Problem: Detect intersection for a set of N input lines.

For input of our algorithms, it is assumed that the N data points have been evenly distributed on the M processors, where $M = 2^k$ for some integer $k$, and the output are also represented in the distributive manner. Our algorithms are based on the *divide-and-conquer* approach (Ref. 1). Specifically, a problem is solved recursively by subdividing the input data into two subsets which are allocated on two subcubes until a primitive case is encountered, then combining (again recursively) the two partial results by using the communication links between the two subcubes.

(1). new *Convex Hull Algorithm*

The algorithm is based on a new set of decision rules which enable us to determine, in a distributive manner, the common tangents between two convex hulls. The total time complexity of the new algorithm is $O(log^2N)$ in the worst case, where N, the number of of input data points, is equal to M, the number of processors available. In the case M<N, we also present a generalized algorithm which achieves $O(Nlog(N/M))$ time complexity.

(2). new *Nearest Neighbors Algorithm*

We present a new $O(log^2N)$ time algorithm for finding the nearest neighbor for each of the N points on the plane. In the k-dimensional case, the algorithm has a time complexity of $O(log^kN)$ for k > 2. Since it takes only $O(logN)$ time to find a minimum on the hypercube, the *Closest-Pair Problem* can also be solved in the same time complexity.

(3). new *Line Intersection Algorithm*

Assuming there are N planar line segments, the worst-case time complexity of our new algorithm for vertical/horizontal lines is $O(log^3N)$. For input lines of general orientations, it takes $O(\log^4N)$ amount of time in the worst case.

Compared to the previous results, the improvements achieved in the new algorithms are significant. The techniques developed here may also be applied to solving other problems of similar character. We will also discuss possible extensions to these results. In particular, two key issues will be addressed:

i). How to distributively represent and manipulate different data types on the hypercube architecture, and

ii). How to synchronize the processors so that maximal parallelism can be achieved.

## II. REFERENCES AND BIBLIOGRAPHY

#### 1. General References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, "The Design and Analysis of Computer Algorithms," Addision-Wesley, Reading, MA, 1974.

2. K. E. Batcher, Design of Massively Parallel Processor, IEEE Trans. on Computers, C-29, pp. 836-840, 1980.

3. C. Seitz, The Cosmic Cube, Comm. ACM 29, 8, Aug., 1978, pp. 666-677.

4. F. P. Preparata and D. T. Lee, Computational geometry - A Survey, IEEE Trans. on Computers, Vol. c-33, no. 12, pp 1072-1100, Dec. 1984.

5. F. P. Preparata and M. I. Shamos, Computational Geometry: An Introduction, Springer-Verlag, New York, NY, 1985.

6. S. L. Johnsson, Combining Parallel and Sequential Sorting on a Boolean N-Cube, Int'l Conf. on Parallel Processing, 1984.

7. D. Nassimi and S. Sahni, Data Broadcasting in SIMD Computers, IEEE Trans. Computers, Vol. c-30, no. 2, pp. 101-106, Dec. 1981.

8. F. P. Preparata, and J. Vuillemin, The Cube-Connected-Cycles: A Versatile Network for Parallel Computation, Comm. ACM, Vol. 24, pp. 300-309, 1981.

9. X. Lin and W. J. Hsu, A Parallel Algorithm for Constructing Convex Hulls on Hypercube Computers, Submitted.

10. W. J. Hsu, K. T. Huang, and J. S. Liu, Parallel Algorithms for Detecting Line Intersections on Hypercube Computers, Submitted to HICSS-22, March 1988.

i). Shared Memory model:

11. A. Aggarwal, et al., Parallel Computational Geometry, Proc. IEEE FOCS, 1985, pp.468-477.

12. M. J. Atallah and M. T. Goodrich, Efficient Plane Sweeping in Parallel, Purdue Univ. CSD-TR-563, 1986.

13. A. L. Chow, Parallel Algorithms for Geometric Problems, Ph.D. Dissertation, Dept. of Computer Sci., Univ. of Illinois, Urbana, IL, 1980.

ii). Linear Arrays (Systolic Algorithms)

14. B. Chazelle, Computational Geometry on a Systolic Chip, IEEE Trans. on Computers, vol. c-33, no. 9, Sept, 1984, pp. 774-785.

iii). Mesh:

15. C. S. Jeong and D. T. Lee, Parallel Geometric Algorithms on Mesh-Connected Computers, ACM/IEEE FJCC, 1987, pp. 311-318.

16. M. Lu, Constructing the Voronoi Diagram on a Mesh-Connected Computer Proc. Int'l. Conf. on Parallel Processing, pp. 806-810, 1986.

17. M. Lu and P. Varman, Solving Geometric Proximity Problems on Mesh-Connected Computers, Proc. of IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management, 1985, pp. 248-255.

18. R. Miller and Q. F. Stout, Mesh Computer Algorithms for Line Segments and Simple Polygons, Proc. of Int'l Conf. on Parallel Processing, 1987, pp. 282-285.

19. R. Miller and Q. F. Stout, Computational Geometry on a Mesh-Connected Computer, Proc. Int'l. Conf. on Parallel Processing, pp. 66-73, Aug. 1984.

iv). Cubes:

20. R. Miller and S. E. Miller, Using Hypercube Multiprocessors to Determine Geometric Properties of Digitized Pictures, Proc. Int'l. Conf. on Parallel Processing, pp. 638-640, 1987.

v). Pyramids

21. R. Miller and Q. F. Stout, Convexity Algorithms for Pyramid Computers, Proc. Int'l Conf. on Parallel Processing, 1984.

22. R. Miller and Q. F. Stout, Pyramid Computer Algorithms for Determining Geometric Properties, Proc. ACM Symp. on Computational Geometry, 1985, pp. 263-277.

# A FAST ALGORITHM FOR VORONOI DIAGRAM CALCULATION BASED ON DISTANCE DOUBLING

David Izraelevitz

The Analytic Sciences Corporation (TASC)
Reading, MA 01867

## ABSTRACT

Many problems involving distances between points have natural solutions in terms of the Voronoi diagram. In this paper, we describe an efficient algorithm for calculation of the Voronoi diagram over one and two-dimensional lattices. The algorithm proceeds by propagating the location of Voronoi points through the lattice using a distance doubling strategy. This algorithm is designed for implementation on a fine-grain parallel computer such as the Connection Machine. We show how the algorithm is extended to calculation of distance-from-region computations and the use of non-standard metrics.

Keywords: Voronoi Diagram, Distance Doubling, Parallel Algorithms, Computational Geometry.

## BACKGROUND

Consider a finite subset of the points on a lattice, to be called the Voronoi point set. We are interested in dividing the lattice into sets of points which are closer to a common Voronoi point than to any other. Each set is called a Voronoi polygon and the corresponding Voronoi point is its polygon center. The tesselation of the lattice into Voronoi polygons the Voronoi diagram. Our task is to associate with each point on the lattice a label which denotes the point as a member of a specific Voronoi polygon. The Voronoi diagram plays a central role in many aspects of computational geometry (Ref. 1), and two-dimensional distance problems (Ref. 2). In this paper we consider the calculation of the Voronoi diagram for one and two-dimensional lattices only.

The computational model we follow is based on the Connection Machine (CM). We assume a fine-grained SIMD parallel machine implementation with a hypercube connectivity architecture incorporating very efficient communication among adjacent processors. By fine-grained parallelism we assume that a single processor (physical or virtual) is associated with each point in the lattice.

Define the processor address of a point as the address of the processor associated with this point. The lattice address of a point is its cartesian coordinate representation on this lattice. A natural processor to lattice mapping for one-dimensional lattices is to use a gray-code mapping so that adjacent points in the lattice correspond to adjacent processors. For the two-dimensional lattice case, we use a coordinate-wise gray-code. The gray-code representations just described are standard representations on the CM, and address conversion between representations is supported in hardware.

The traditional strategy for computation of the Voronoi diagram over a two-dimensional lattice on the CM is based on a "brush-fire" algorithm (Ref. 3) where the distance of each point on the lattice to its nearest Voronoi point is iteratively improved in a layer by layer manner. The algorithm begins by setting the distance estimate to infinity except at the location of the Voronoi points where it is zero. Processors retrieve the distance of their left neighbor D(left), and compare D(left) + 1 with their present distance estimate. If lower, then D(left)+1 becomes the new distance estimate. The right, top, and bottom neighbor is queried and the distance similarly updated. A more accurate estimation procedure queries in addition diagonal neighbors and compares D(diag) + $\sqrt{2}$ to the present distance estimate. This query sequence is continued until the distance estimates for the whole lattice do not change over a full cycle. We will denote this procedure as the classical brush-fire algorithm.

## VORONOI POINT ADDRESS PROPAGATION

The first improvement we propose is based on propagating the processor address of Voronoi points rather than the distance to this point. Suppose that each processor contains a parallel variable (pvar in CM nomenclature) called *addr* . The contents of this pvar at each iteration is the processor address of what is estimated as being the closest Voronoi point. The variable *addr* is initialized to the processor address for all Voronoi points and some special value such as "nil" everywhere else. At the end of processing, the value of *addr* will be the processor address of the nearest Voronoi point at each point in the lattice.

The propagation sequence is the same as in the previous algorithm except that neighbors are queried as to their value of *addr* rather than the distance to the Voronoi point. Each processor then converts processor addresses to lattice addresses, compares the distance to this new Voronoi point and decides if this distance is smaller than the distance to the present Voronoi point estimate, and updates *addr* accordingly. In our later discussion we will refer to this algorithm as the updated brush-fire algorithm.

Since this algorithm calculates the Euclidean distance, or in fact any other metric, to a Voronoi point, the resulting tesselation of the plane does not have the distortions due to the distance approximations required by the earlier algorithm. Also, propagation of the Voronoi point location tesselates the lattice directly, since the processor address of each Voronoi point can be used as a unique label for the corresponding Voronoi polygon.

In Figure 1 we show two typical images, which present the lattice tesselation after classical brush-fire propagation and the updated brush-fire algorithm. This example points out the distortions from Euclidean distance generated by the classical algorithm.

The convergence properties of either algorithm can be studied via the analysis of the Voronoi polygon radius under a certain metric. The 4-distance of a point **A** to its polygon center **P** is found by considering all paths connecting **A** and **P** which reside wholly inside the Voronoi polygon. The minimum under the

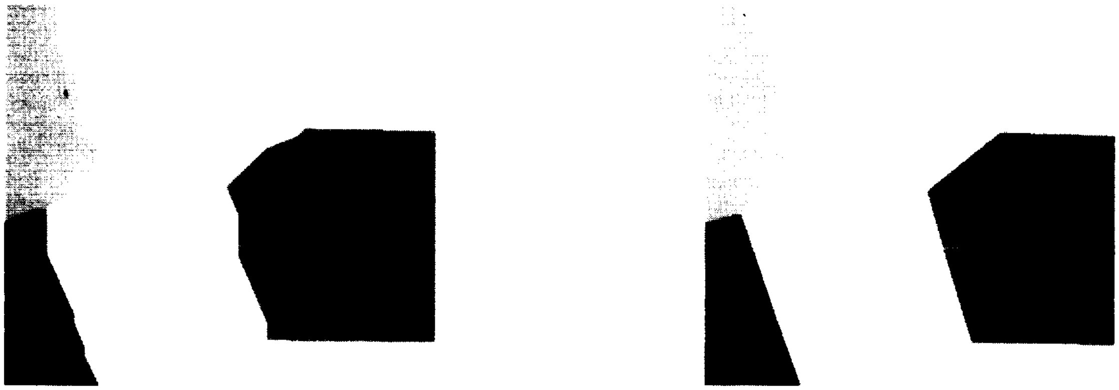


Figure 1 - (a) Voronoi tesselation using classical brush-fire algorithm. (b) Tesselation using new algorithms.

Manhattan metric we define as the 4-distance of **A** to **P**. Finally the 4-radius of a Voronoi polygon is the maximum 4-distance over all points **A** in the polygon to the center **P**.

It is clear that after N full cycles all points at a 4-distance of N will have received the final distance for the classical brush-fire algorithm or the correct Voronoi point processor address in the updated brush-fire algorithm. Therefore the convergence properties of the two algorithms is governed by the Voronoi polygon with the largest 4-radius. In fact, the number of iterations is essentially this radius.

The brush-fire propagation strategy just described will be sufficient in situations where the distribution of Voronoi points yields uniformly small Voronoi polygons. However, the case where the diameter of any one Voronoi polygon is large will result in most processors becoming idle for large periods of time in the later iterations of the algorithm as the propagation reaches the outer boundaries of this largest polygon.

In the next section we discuss a new algorithm which again propagates the processor addresses rather than the distance itself so it maintains the accuracy of the updated brush-fire algorithm. The main advantage of the new algorithm is that it propagates the processor address of Voronoi points using a doubling strategy to speed up the propagation process in situations unfavorable to brush-fire algorithms. The distance doubling strategy is well suited for implementation on the CM.

## DISTANCE DOUBLING FOR VORONOI DIAGRAM COMPUTATIONS

We discuss first the one-dimensional problem since it shows the basic ideas of the algorithm and is actually used as a component of the two-dimensional case.

### One-Dimensional Voronoi Diagram Extraction

The first step of the algorithm is identical to the updated brush-fire algorithm in one-dimension. Initially each processor queries the value of *addr* of its left neighbor and deposits it in pvar *candidate*. Each processor then compares the distance to the point represented by its own value of *addr* vs the point represented by *candidate*. If the distance is lower then *candidate* becomes a better estimate of the nearest Voronoi point. The query is now performed of the right neighbor. The speedup occurs in the subsequent iterations. The algorithm transmits *addr* to processors at a distance of 2, then 4, etc., at each stage doubling the distance of transmission. The algorithm ends after $log_2 nx$ where *nx* is the number of points in the lattice. The reason that we can guarantee convergence of the one-dimensional algorithm after $log_2 nx$ iterations is that the Voronoi "polygons" are line segments, and therefore the distance doubling strategy is bound to "visit" every point on this polygon in less than $log_2 N$ iterations where $N$ is the length of the Voronoi polygon, and $N$ must be less than *nx*.

The strategy of distance doubling is an emerging standard approach to achieving speedups in hypercube-connected massively parallel computers (Ref. 3,5,6) and is closely related to the primitive scan operators introduced by Blelloch (Ref. 4). Since processors at a distance $2^n$ are neighbors on the hypercube, this transmission can occur very quickly. After n iterations, all points at a distance $2^n$ or less from the corresponding Voronoi point have been correctly labeled. The formal algorithm is described below (we use the notation introduced in Ref. 5):

*nx:* number of points in lattice
*k*: address of each processor
*mask[k]*: TRUE if a Voronoi point is located at address *k*, FALSE elsewhere
*addr [k]*: address of nearest Voronoi point to lattice point associated with processor *k*.

```
for all k in parallel do
  addr [k] = nil
  if mask[k] = TRUE then
    addr [k] = k
  fi
od
for j=0 to $log_2 nx$ do
   for all k in parallel do
      candidate [k] = addr [k - $2^j$ ]
      if distance to candidate [k] < distance to addr [k] then
         addr [k] = candidate [k]
      fi
      candidate [k] = addr [k + $2^j$ ]
      if distance to candidate [k] < distance to addr [k] then
         addr [k] = candidate [k]
      fi
   od
od
```

### Two-Dimensional Voronoi Diagram Extraction

In the case of two-dimensional lattices, the associated Voronoi polygons are no longer line segments; informing each point in the polygon of the location of the polygon center is more involved. The issue is how to perfom this information transmittal efficiently. The ease or difficulty of this transmittal depends on the shape of the Voronoi polygons involved. We consider first the case of scattered Voronoi points where the associated polygons are 4-connected.

If a polygon is 4-connected then there must be a 4-connected path between the polygon center and every point in the polygon. Again, we use a pvar *addr* which at convergence contains the address of the nearest Voronoi point. After applying the one-dimensional algorithm along each row which contains a Voronoi point, all points in each Voronoi polygon which are connected horizontally with the polygon center will have the correct center location in ***addr***. The second step is to apply the one-dimensional algorithm on each column. After this point, all points which are connected to the polygon center by a path in the polygon consisting of one horizontal leg followed by a vertical leg will have the correct value of *addr*. We see that alternating horizontal and vertical iterations will update those points with successively more complicated paths leading to the Voronoi polygon center.

In the previous section we used the 4-distance metric to analyze the convergence property of the brush-fire algorithms. We can define a step distance which similarly controls the convergence of the distance-doubling algorithm. Define the step length of a 4-connected path as the number of horizontal plus vertical runs in a path. The step distance of a point **A** in the Voronoi polygon centered at **P** is the minimum step length over all paths connecting **A** and **P** which reside wholly in the Voronoi polygon. Thus in Figure 2, the step distance from **A**, **B**, and **C** to **P** is 1, 2, and 4 respectively. Finally the step radius of a Voronoi polygon is the maximum step distance over all points in the polygon to the center **P**.

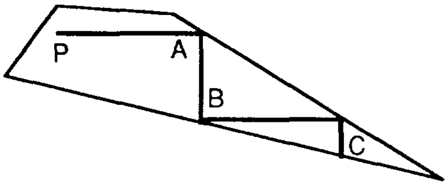


Figure 2 - Voronoi Polygon with Center at P.

It is clear from our earlier discussion that after N iterations, all Voronoi polygons of step radius less than N will have been correctly updated. The reason for the expectation of faster convergence is that for large Voronoi polygons, the step distance between two points is much smaller than the 4-distance, while the distance doubling paradigm updates points along each row or column quicker than brush-fire propagation.

## PERFORMANCE COMPARISONS

We considered several test cases in order to evaluate the performance of the new algorithm via comparisons to the classical and updated brush-fire algorithms. The results are summarized below. Images were generated with Voronoi points distributed randomly over a 512 by 512 lattice. Five trials were generated for each Voronoi point density and the computation time averaged. All experiments were performed on a CM-2 with 8k processors. Similar results were observed for 128 by 128 and 256 by 256 lattices.

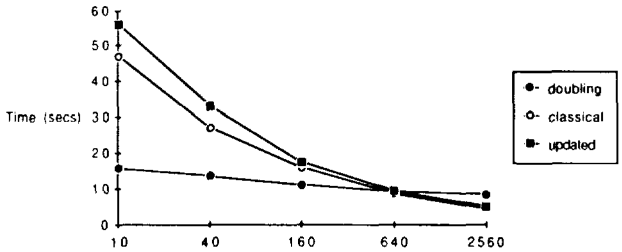


Figure 3 - Computation time for Voronoi diagram calculation as a function of the number of Voronoi points.

From the above figure, we can see that the distance doubling algorithm is substantially faster than either brush-fire technique as the number of Voronoi points decreases. As the number of points increases, the size of the largest Voronoi polygon decreases, so that the brush-fire techniques require fewer iterations. Therefore, for dense Voronoi diagrams, the brush-fire techniques are preferable. Note that the updated brush-fire algorithm is comparable to the classical algorithm even though it provides better accuracy.

## GENERALIZATIONS

### Distance-from-Region Computations

In our earlier discussion, we considered the Voronoi diagram problem of finding distances to scattered points. In many cases of interest, it is necessary to calculate the minimum distance not to a set of points but to a set of regions or more specifically the minimum distance to the boundary of a set of regions. Such a computation can be performed in our formulation by considering each point on the boundary of each region as an Voronoi point. Thus after convergence each lattice point will know the location of the nearest boundary point.

In some situations, it is also necessary to know to which region a given point is closest. Given that a point knows the processor address of the nearest boundary point, a common label needs to be provided to all boundary points of the same region and that label propagated along with the address location.

For this application, we encounter the situation that a Voronoi polygon may not be 4-connected. This occurs if three Voronoi points are aligned diagonally, resulting in a Voronoi polygon consisting of a diagonal line. Such a configuration is likely to occur in the distance-from-region case. In this situation, we have found it adequate to first perform a propagation be performed in each diagonal direction to accommodate those polygons and then continue with alternating horizontal and vertical propagations.

In Figure 4, we show a distance-to-nearest-region computation using the new algorithms.

### Voronoi Diagram Computation Under Non-Standard Metrics

The fact that the location of the Voronoi point rather than the distance to this point is the information propagated allows for the application of either the updated brush-fire or distance-doubling algorithms to a variety of distance functions. In fact we can associate a different distance function with each Voronoi point. For example, there could be associated with each Voronoi point a weighing factor such that the "distance" to this Voronoi point is first weighed prior to comparison with the distance to another Voronoi point. We have generated some experiments with non-standard metrics in order to understand the possible

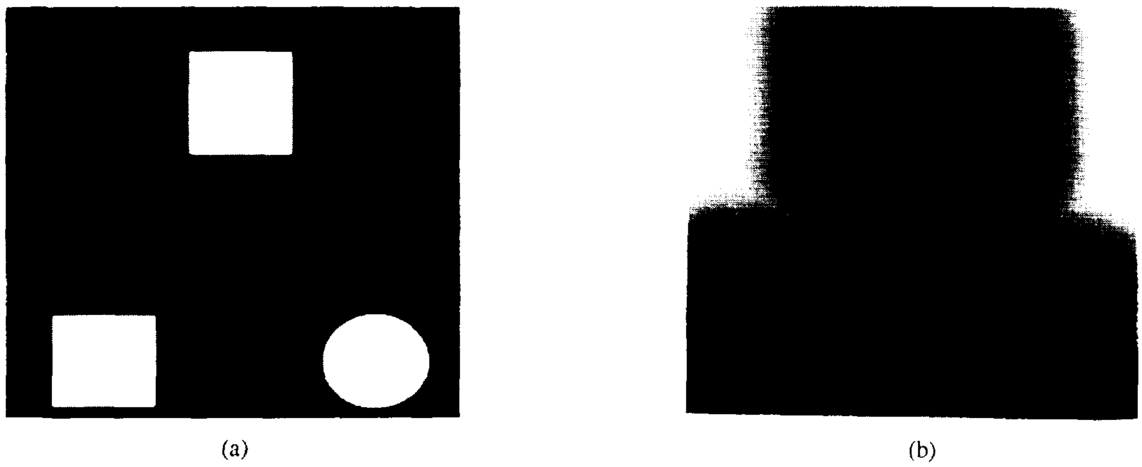


Figure 4 - (a) Three regions for distance computation.b) Euclidean distance to nearest region.

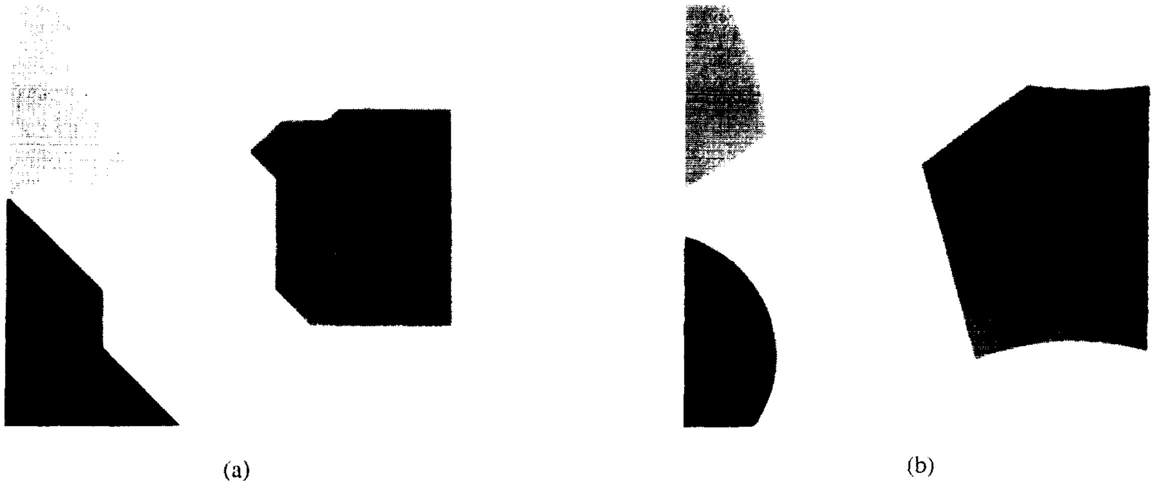


Figure 5 - Voronoi tesselations of the same point set under different metrics

generalizations of the procedure developed here. As an example of the possible generalizations, we generated the Voronoi diagram under two different distance functions for the same Voronoi point set as in Figure 1. Figure 5a shows the use the distance metric D = max(|dx|,|dy|). Figure 5b shows the use of the weighed Euclidean metric.

## SUMMARY

In this paper we introduce two new algorithms for the calculation of the Voronoi diagram over one and two-dimensional lattices. Both new algorithms propagate the explicit location of the nearest Voronoi point rather than the distance to the point and so are more accurate than traditional methods. The first algorithm is based on brush-fire propagation while the second algorithm relies on distance-doubling. We find that the distance-doubling algorithm is more efficient than the classical algorithm for the case when at least one Voronoi polygon is large. It was found that there is little penalty in using the the updated brush-fire algorithm over the traditional algorithm. Generalizations of the algorithm to distance from region computations and non-standard metrics are also presented.

## REFERENCES

[1] D. T. Lee and F. P. Preparata, Computational Geometry - A Survey, *IEEE Transactions on Computers*, Vol. C-33, no. 12, December 1984.

[2] Robert Sedgewick, *Algorithms*, Addison-Wesley, Reading, MA.

[3] J. J. Little, *Parallel Algorithms for Computer Vision on the Connection Machine*, A. I. Memo 928, MIT Artificial Intelligence Laboratory, November 1986.

[4] G. E. Blelloch, Scans as Primitive Parallel Operations, *Proceedings Int. Conf. on Parallel Processing*, August 1987.

[5] W. Daniel Hillis and Guy L. Steele, Jr., Data Parallel Algorithms, *Communications of the ACM*, Vol. 29, no. 12, December 1986.

[6] W. Lim, A. Agrawal, and L. Nekludova, *A Fast Parallel Algorithm for Labeling Connected Components in Image Arrays*. Technical Report NA86-2, Thinking Machines Corporation, 1986.

# Provably Good Parallel Algorithms For Channel Routing of Multi-terminal Nets[1]

Sridhar Krishnamurthy
Department of Electrical Engineering
Systems Research Center
University of Maryland
College Park, MD. 20742

Joseph JáJá
Department of Electrical Engineering
Institute for Advanced Computer Studies
Systems Research Center
University of Maryland
College Park, MD. 20742

**Abstract**

We consider the channel routing problem of a set of multi-terminal nets in the knock-knee model. We develop a new approach to route all the nets within $d+\alpha$ tracks, where $d$ is the channel density, and $0 \leq \alpha \leq d$, such that the corresponding layout can be realized with three layers. Both the routing and the layer assignment algorithms have linear time sequential implementations. In addition both can be implemented on the CREW-PRAM model in $O(\frac{n}{p}+\log n)$ time, with $p$ processors, $1 \leq p \leq n$, where $n$ is the size of the input.

## 1. Introduction

Routing plays a central role in automated VLSI layout systems. This problem has been intensively studied in literature (e.g. [CJ],[MP],[P],[PL],[O],[RF]). Because of the combinatorial nature of routing, most of the corresponding optimization problems turn out to be NP-complete (for example see [S]). However good heuristics have been used effectively to generate good layouts. In this paper, we continue our research efforts in developing *efficient* parallel programming techniques to handle various routing problems. Our goal is to develop routing strategies that will result in parallel algorithms whose running time is $O(\frac{t(n)}{p}+f(p))$, where $t(n)$ is the best known sequential time, $p$ is the number of processors, and $f(p)$ is a non decreasing function that reflects the routing cost on the given parallel model. The routing produced by these algorithms is expected to be as good as the best known sequential algorithms.

We consider the channel routing of multi-terminal nets in the knock-knee model. Provably good approximation algorithms (sequential) have been reported in [MPS],[SP] and more recently in [GK]. The basic strategy used is the well known greedy strategy applied either one column or one row at a time. However, it has been shown recently ([delaT]) that the routing produced by several variations of this strategy are P-complete, and hence there is little hope for parallelizing these strategies efficiently. We provide a new strategy which obtains provably good routing (which is in general different from those obtained in [SP],[MPS] and [GK] methods), such that the routing algorithms has a linear time sequential implementation. Moreover, the algorithm is fully parallelizable in the sense that it can be implemented on a Concurrent Read, Exclusive Write (CREW) PRAM model in $O(\frac{n}{p}+\log n)$ time with $p$ processors, $1 \leq p \leq n$, where $n$ is the size of the input. We are assuming that all terminals lie in the range $[1, N]$, where $N = O(n)$. A modified version of the algorithm will guarantee that the number of tracks is $d+a$, $0 \leq a \leq d$, where $d$ is the density of the channel. In particular, for two terminal nets the modified version provides an optimal solution.

All our results are stated for the shared memory model. However, our algorithms have fast implementations on fixed-interconnection networks such as the mesh or the hypercube. For example, all the algorithms stated in this paper can be implemented on a $\sqrt{n} \times \sqrt{n}$ mesh in time $O(\sqrt{n})$ where $n$ is the input length.

## 2. Definitions

We borrow some of the basic definitions of channel routing from [SP],[PL]. A net N is an ordered pair of integer sequences $((p_1, p_2, \cdots p_k), (q_1, q_2, \cdots q_h))$ where the $p_i$'s are the *lower* terminals and the $q_i$'s are the *upper* terminals. Without loss of generality, we assume that $k+h \geq 2$. If $k+h > 2$, then the net N is said to be a *multiterminal net*, otherwise N is a two-terminal net. An instance of a general channel routing problem (GCRP) is a channel consisting of rectangular grid, and a set of nets, each of which specifies a subset of terminals which lie on the grid points of the (horizontal) parallel boundaries. The goal is to route the wires such that the channel width is as small as possible.

Let $N_i = ((p^i_1, \cdots, p^i_{k_i}), (q^i_1, \cdots, q^i_{h_i}))$ be a set of nets. Let $l_i = min(p^i_1, q^i_1)$ and $r_i = max(p^i_{k_i}, q^i_{h_i})$. The interval $[l_i, r_i]$ is a lower bound to the horizontal track demand of $N_i$. We can transform the GCRP into a fictitious two-terminal net channel routing problem, where each net $N_i$ is replaced by $N^*_i$ and $l_i$ and $r_i$ are referred to as the *left* and *right* terminals of this fictitious net respectively. The *local density* $d_x$ at $x$ is defined to be the number of nets $[l_i, r_i]$ such that $l_i \leq x < r_i$. The density $d$ is given by $d = max_x(d_x)$. It is clear that $d$ is a lower bound for the minimum number of horizontal tracks and we call $d$ the *essential density* of the GCRP. The type of terminals within a column (entry, exit or continuing terminals) define the *state* of

[1] Supported in part by NSA Contract No. MDA-904-85H-0015, NSF Grant No. DCR-86-00378 and by the Systems Research Center Contract No. OIR-85-00108

CH2649-2/89/0000/0177$01.00 © 1988 IEEE

the column. All the possible states of a column are shown in Figure 1.

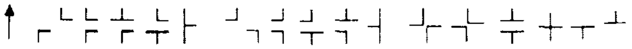


Figure 1: States of a column

## 3. Channel Routing

Given an instance of a GCRP of essential density $d$, our goal is to determine a wiring of all the nets within $2d$ tracks. In addition, the resulting layout should be realizable in three layers.

The algorithm developed in [SP] produces the layout column-by-column. The overall strategy is similar to the approach of the 'greedy router' of [RF]. Unfortunately this approach seems to be inherently sequential. Our method is quite different and consists of the following main steps:

1. Create two sets of chains $S_u$ and $S_l$. Each set consists of a partition of the nets into $d$ chains satisfying certain properties to be outlined later. In particular, the nets in each chain define a set of nonoverlapping intervals. Initially a net has two symmetric segments above and below the track $y = 0$.
2. Assign a track number from the upper $d$ tracks in the channel to each chain in $S_u$ and a track number from the lower $d$ tracks to each chain in $S_l$. Then wire all the nets for all the columns simultaneously.

The algorithm produces a layout which maintains the following property:

**Property 1.** Any net $N_i$ which is active in column $c$ has two strands $y = t_1(i) > 0$ and $y = t_2(i) < 0$.

We shall summarize the algorithm *Create Chains* developed in [CJ] which partitions a set of nets into $d$ chains, where $d$ is the density of the corresponding CRP. This algorithm will be used later on to obtain the initial sets of chains.

*Algorithm Create Chains*

*Input:* terminals $l_i$'s and $r_i$'s of all the nets $N_1, N_2, \cdots, N_n$.
*Output:* $d$ chains of nets, where $d$ is the density of the CRP.

1. Mark all the terminals. For each left terminal $l_i$ of a net $N_i$, set $p(l_i) = r_j$ such that $r_j$ is the nearest right terminal of some other net to the right of $N_i$. If two such terminals exist, then pick the one whose corresponding net is of the same type as $N_i$. However if no such $r_j$ exists, then set $p(l_i) = \emptyset$. Similarly define $p(r_i)$ for each right terminal.
2. If $p(l_i) = r_j$ and $p(r_j) = l_i$ then set $succ(N_j) = N_i$ and unmark $r_j$ and $l_i$. Create a reference point $k$ between $r_j$ and $l_i$.
3. Let $R_1, R_2, \cdots, R_m$ be the intervals determined by the reference points. For each $R_i$ create lists $L(R_i)$ and $R(R_i)$ consisting of all the marked left and right terminals in $R_i$.
4. Create links between corresponding terminal pairs in $R(R_i)$ and $L(R_{i+1})$. Unmark all those terminals thus linked and merge intervals $R_{2i-1}$ and $R_{2i}$. Repeat this step until only one interval is left.

**Lemma 1 [CJ].** The number of chains created by the above algorithm is exactly $d$, where $d$ is the channel density. This algorithm can be implemented on a CREW-PRAM in time $O(\frac{n}{p} + \log n)$ with $p$ processors, $1 \leq p \leq n$. □

The above chains are then modified in Algorithm 'Modify Chains' [CJ] so that they have the following property. Let $c$ be any column. Then either

1. $c$ is empty, or
2. $c$ contains only one entry terminal, or
3. $c$ contains two entry terminals of nets $N_i$ and $N_j$. Let $N_i =< c, b_i >$and $N_j =< t_j, c >$. The following two cases can then arise:
   - If $N_i$ has an exit terminal and $N_j$ has an entry terminal in $c$, then they both belong to the same chains and one is a successor of the other.
   - Suppose both $N_i$ and $N_j$ exit at $c$. The other case is dealt with similarly. Let $N_i' = succ(N_i)$ and $N_j' = succ(N_j)$. Then they either have their entry terminals on the same column or the column of $N_i'$ or $N_j'$ which is closer to $c$ has only one entry terminal.

We will now outline each of the main steps of our algorithm. The algorithm below creates initial chains of nets in sets $S_u$ and $S_l$ which will be modified later to satisfy certain desired properties.

*Algorithm Initial Chaining*

*Input:* terminals $l_i$'s and $r_i$'s of all the fictitious nets $N_1^*$, $N_2^*$, $\cdots, N_n^*$.

*Output:* two sets $S_u$ and $S_l$ with each set containing a partition of the fictitious nets into $d$ chains, where $d$ is the essential density.

1. Using algorithm Create-Chains outlined above, obtain $d$ chains where $d$ is the essential channel density.
2. Duplicate the $d$ chains thus obtained into two sets $S_u$ and $S_l$ each of which correspond to the upper and lower halfs of the channel. The chains in the two sets will be modified independently later on to satisfy certain properties.

As an example consider the general channel routing instance shown in Figure 2. The chains produced are shown in Figure 3.

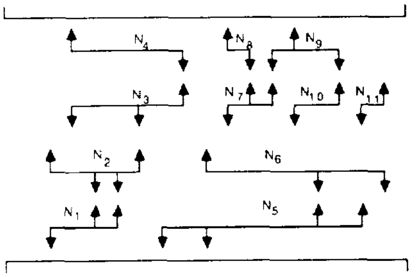


Figure 2: An instance of a GCRP

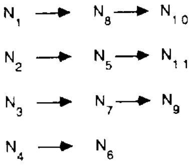

$N_1 \rightarrow N_8 \rightarrow N_{10}$
$N_2 \rightarrow N_5 \rightarrow N_{11}$
$N_3 \rightarrow N_7 \rightarrow N_9$
$N_4 \rightarrow N_6$

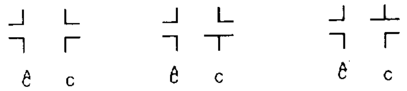
Figure 3: Chains created by Algorithm Initial Chaining

**Lemma 2.** If there exists a column $c$ containing an exit-terminal $N_i$ and an entry-terminal $N_j$, then $N_j$ is the successor of $N_i$ in one of the chains. Property 1 will also be satisfied. □

The above chains can be used to wire all the nets in $2d$ tracks but the corresponding layout may not be realizable in three layers. So we have to modify the chains in both the sets $S_u$ and $S_l$.

A column $c$, is said to be a *terminating* column, if

1. $c$ has an exit terminal of $N_i$, or
2. $c$ is the closest successor entry column of net $N_i$, and $N_i$ has exit terminals in a column $\hat{c}$ whose *state* = $\begin{array}{c}\lrcorner\\ \urcorner\end{array}$ (see figure below).



All other columns are said to be *non-terminating*. We associate the pair $< c, N_i >$ to each terminating column, and refer to it as a *terminating pair*. A terminating pair $< c, N_i >$, is said to be an upper (lower) terminating pair if

- Net $N_i$ has an exit terminal on the upper (lower) boundary in $c$ or
- $N_i$ terminates in $c$, and the successor of $N_i$ is $S_u$ ($S_l$) has its first upper (lower) terminal in $c$.

To satisfy the three layer wirability, the chains in the sets $S_u$ and $S_l$ are modified so as to satisfy the following property:

**Property 2.** Let $c_i$ be any column. Then either

1. $c_i$ is non-terminating, or
2. if $c_i$ is associated with a lower (upper) terminating pair $< c_i, N_i >$, then either
   a. the column $c_i'$ containing the first upper (lower) terminal of $N_i' = succ(N_i)$, in $S_u$ ($S_l$) is non-terminating, or
   b. column $c_i'$ = column $c_i$.

The following algorithm outlines how to modify the chains $S_u$ so that the above property holds. For modifying the set $S_l$ we can essentially use the same algorithm with the obvious modifications.

*Algorithm Chain Modification*

*Input:* state of columns and initial chain set $S_u$.

*Output:* new set of chains $S_{unew}$ satisfying Property 2.

1. If a column has an upper exit terminal of a net $N_i$, and is also the first upper terminal of the net, then delete $N_i$, from its chain in $S_u$, unless the column is of *state* = $\begin{array}{c}\lrcorner^{N_i}\\ \urcorner\end{array}$.
2. Mark all columns of $state(c)$ = $\begin{array}{c}\perp\\ \urcorner\end{array}$.
3. Now consider all columns of $state(c)$ = $\begin{array}{c}\lrcorner^{N_i}\\ \urcorner_{N_j}\end{array}$. Two cases arise for such columns.
   a. Column contains an upper terminal labeled $N_i$, which is both the first as well as the last upper terminal. Let $\hat{N}_i = pred(N_i)$ in $S_u$. Modify $S_u$ by setting the $succ(N_j) = N_i$ and $succ(\hat{N}_i) = oldsucc(N_j)$. Let $c_i$ be the column to the right of $c$ such that $c_i$ contains the closest entry terminal of $succ(N_i)$ among its two successors. Mark $c_i$ only if the entry terminal is a lower terminal.
   b. For the remaining columns of $state(c)$ = $\begin{array}{c}\lrcorner^{N_i}\\ \urcorner_{N_j}\end{array}$ not considered in the previous step, we process as follows. Let $c_i$ and $c_j$ be the nearest columns to the right of $c$, such that they contain the entry terminals of the successors of $N_i$ and $N_j$. Mark these columns if the entry terminals are the lower terminals.
4. For each marked column create an ordered pair $< c, c' >$ where $c'$ contains the first upper terminal of $N_i' = succ(N_i)$ and $N_i$ is the net which terminates in column $c$.
5. Group the pairs $< c, c' >$ into maximal groups $< c_0, c_1 >$, $< c_1, c_2 >, \cdots, < c_k, c_{k+1} >$. Let $N_i$ denote the net which terminates in column $c_i$. Update the successors of these nets by setting the new successor of $N_i$ to be the previous successor of $N_{i-1}$ for all $0 < i \leq k$. In addition, set the new successor of $N_0$ to be the previous successor of $N_k$.

As an example consider the chains in Figure 3. The new set of chains created by the above algorithm are shown in Figure 4.

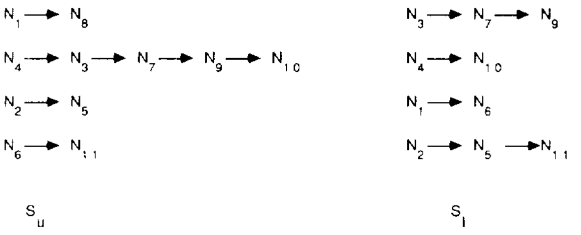

$N_1 \rightarrow N_8$
$N_4 \rightarrow N_3 \rightarrow N_7 \rightarrow N_9 \rightarrow N_{10}$
$N_2 \rightarrow N_5$
$N_6 \rightarrow N_{11}$
$S_u$
$N_3 \rightarrow N_7 \rightarrow N_9$
$N_4 \rightarrow N_{10}$
$N_1 \rightarrow N_6$
$N_2 \rightarrow N_5 \rightarrow N_{11}$
$S_l$

Figure 4: Modified sets of chains

**Lemma 3.** Algorithm 'Chain Modification' modifies the chains such that the new set of chains satisfies Property 2. Moreover the algorithm runs in $O(\frac{n}{p}+\log n)$ time with $p$ processors, $1 \leq p \leq n$, on the CREW-PRAM model. □

After having obtained the modified sets of chains we proceed to do the wire layout details of which are left for the full paper. The wire layout for our example is shown in Figure 5.

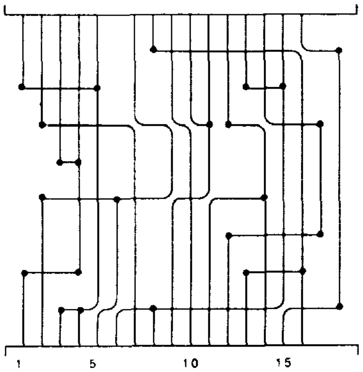


Figure 5: Wire layout for the example

**Theorem 1.** Given an instance of a GCRP of essential density $d$, it is possible to wire all the nets using $2d$ tracks in time $O(\frac{n}{p}+\log n)$ on a CREW-PRAM model with $p$ processors, $1 \leq p \leq n$, where $n$ is the size of the input. □

The wire layout produced by the above algorithm can be laid out in three layers. [PL] provides necessary and sufficient conditions for wiring a layout using three layers. The routing produced by our algorithm satisfies the following property:

**Property 3.** Any column irrespective of its type will have at most two knock-knees, with a diagonal \ on the bottom and a diagonal / above it.

Using the wiring algorithm given in [CJ], we can realize the wire layout produced by our algorithm in three layers.

**Theorem 2.** Given an instance of a GCRP, it is possible to determine a three-layer assignment of the routing, in time $O(\frac{n}{p}+\log n)$ on a CREW-PRAM model using $p$ processors, $1 \leq p \leq n$, where $n$ is the size of the input. □

## 3. References

[CJ] Chang, S. and J. JáJá, " Parallel Algorithms for Channel Routing in the Knock-Knee Model," Proceedings of the International Conference on Parallel Processing, 1988, pp.18-25.

[delaT] de la Torre, P., "On parallelism and some generalisations of the line packing problem," Unpublished manuscript 1988.

[GK] Gao, S. and M.Kaufmann, "Channel Routing of Multiterminal Nets," Proc. of the 19th Ann. ACM Symposium on Theory of Computing, pp.316-325, 1987.

[MP] Melhorn, K. and F. Preparata, "Routing Through a Rectangle," JACM, vol. 33(1), Jan. 1986, pp. 60-85.

[MPS] Melhorn, K., Preparata, F., and M. Sarrafzadeh, " Channel Routing in Knock-Knee Mode: Simplified Algorithms and Proofs," Algorithmica, 1986, pp. 213-221.

[O] Ohtsuki, T., "Layout Design and Verification," Advances in CAD for VLSI, vol. 4, North-Holland, 1986.

[P] R. Pinter, "River routing: methodology and analysis," Proceedings of the third CALTECH Conference on Very Large Scale Integration, March 1983, pp. 141-163.

[PL] Preparata, F. and W. Lipski, "Optimal Three-Layer Channel Routing," IEEE Trans. on Computers, C-33, 1984, pp. 427-437.

[RF] Rivest, R., and C. Fiduccia, " A Greedy Channel Router," Proceedings of the 19th Design Automation Conference, 1982, pp. 418-424.

[S] Sarrafzadeh, M., "Channel-Routing Problem in the Knock-Knee Model is NP-Complete," IEEE Transactions on CAD, vol. 6, 1987, pp. 503-506.

[SP] Sarrafzadeh, M., and F. Preparata, "Compact Channel Routing of Multiterminal Nets," Annals of Discrete Math., no. 25, April 1987, pp. 255-279.

# SPARSE MATRIX VECTOR MULTIPLICATION ON POLYMORPHIC-TORUS

Hungwen Li
IBM Research
Almaden Research Center
650 Harry Road, CA 95120

Ming-Cheng Sheng
IBM Research
Thomas J. Watson Research Center
Yorktown Heights, NY 10598

## ABSTRACT

A sparse matrix is a two-dimensional irregular static data structure representing a wide class of physical and engineering problems. Its solution for the massively parallel fine-grained SIMD computers has not been as satisfactory as desired because matching an irregular data structure to a fixed architecture topology usually results a high percentage of idle processors hence low system utilization. This paper describes a two-stage algorithm for sparse matrix vector multiplication on the Polymorphic-torus, a reconfigurable massively parallel fine-grain architecture, to demonstrate how reconfigurability helps to alleviate the matching difficulty. The first stage of the algorithm is the structured condensation which converts the irregular sparse matrix into a more uniform and much denser data structure, while the second stage demands the architecture to reconfigure itself to fit the condensed data structure. The algorithm highly increases the system utilization of the SIMD machines and has a lower bound in arithmetic operation count.

## 1. INTRODUCTION

Sparse matrix solving is an important problem for parallel processing because it represents a very large class of problems in engineering and physical simulation applications that need tremendous computing power.

As the name implies, the sparse matrix is of irregular connectivity and therefore is not well-suited for SIMD parallel processors with very regular topology (e.g. mesh and tree) because many processors can be idle due to the mismatch of the matrix connectivity and the hardware network topology. It is also felt that sparse matrix solving is not suited for fine-grained SIMD parallel processors with very simple (e.g. bit-serial) ALUs because floating point operations are usually needed to maintain the numerical stability and floating-point operations incur high cost for bit-serial processors. Computers in this category includes DAP [1], MPP [2], the Connection Machine [3] and the Polymrophic-torus [4-6].

There are, however, two major reasons to study the sparse matrix solving on massively parallel fine-grained SIMD processors: first, the size of the matrix is ever increasing, which demands parallel processing even more, and, second, the advance of the VLSI technology allows the integration of several hundreds of bit-serial processors in a single chip, which makes fine-grained parallel processing more cost-effective. Although the problem size and the integration size tend to push toward a direction in favor of the fine-grained parallel processors, they, unfortunately, also push the degree of the mismatch between the sparse connectivity and the interconnection network at the same proportion. The mismatch problem remains a hurdle.

In a related prior work on DAP [7, 8], the worst-case complexity of multiplying an nxn matrix with a dense vector of length n on a NxN processor array is $O((n/N)^3)$. This result can be improved by the reconfigurability provided by the Polymorphic-torus architecture in two approaches: (1) the connectivity of the sparse matrix can be converted to fit the architecture, and (2) the architecture can reconfigure itself to fit the sparse connectivity. This paper deals with both approaches and advocates a combination of both to deliver the best result.

## 2. POLYMORPHIC-TORUS ARCHITECTURE

The polymorphic-torus [4, 5] is a VLSI-oriented massively parallel fine-grained SIMD architecture with a two-level interconnection network. In a $N^2$-processor Polymrophic-torus system, processors are physically arranged as an NxN two-dimensional torus with each processor located in the coordinate [pidx, pidy] where $1 \leq pidx, pidy \leq N$. Usually, N is taken as a power of 2 for the sake of control simplicity. A second level of switches is woven intimately with the torus at each node (Figure 1) to facilitate efficient graph matching with very low wiring complexity.

Such a two-level approach for connecting allows the K physical ports of each processor be wired statically by one network (e.g. the torus in this case) and dynamically by programming a second network (e.g. a crossbar switch in this case). By systematically selecting the active processors and arranging the switches of the second network, efficient graph matching can be derived. We called this methodology the polymorphic concept.

The polymorphic methodology is a form of reconfiguration; however, it emphasizes that the switch setting of the second network is a function of the local condition of each processor. A conditional short-circuit function is uniformly provided for the switches of the second network in each processor. This function is represented as

IF (condition) THEN SHORTPORT { ports}.

When the condition is met, any arbitrary ports of a processor can be logically wired together by the switch as if they are a single entity at the same logic level. For example, the processor selection and the switch setting shown below

IF (pidx = 0) THEN SHORTPORT {E, W};

creates a bus along the 0-th row of processors where pidx is the row coordinate of a processor. Thus, a variable in any processor, say processor [0, 0], can be broadcast to all others in the row along the bus.

Each processor of the Polymorphic-torus is equipped with a large amount of memory not only for data storage but also for extending the connectivity of the architecture. The memory is segmented into layers, each of which is in fact a replication of the same processor connectivity as the most front layer. When the problem size (e.g. matrix size) is larger than the processor array size, more layers are used to represent the problem. Using the memory, we can create more virtual processors than $N^2$ and each virtual processor carries a virtual coordinate [layer number, pidx, pidy].

## 3. STRUCTURED CONDENSATION

This section describes a sparse matrix representation in Section 3.1 and a condensation algorithm in Section 3.2 to convert the representation into a format better suited for the Polymorphic-torus.

### 3.1 Sparse Matrix Representation

Traditionally, a sparse matrix is represented by three arrays: (1) an Element array, E, which holds the values of the nonzero entries of the matrix; (2) a Row Pointer array, RP, and (3) a Column Index array, CI; the latter two arrays store the connectivity of the sparse matrix.

The construction of the three arrays can be illustrated in Figure 2. The element array E is built by scanning in a row-major order but only the nonzero entries are recorded. For a matrix with n rows, the RP array contains n + 1 items in which the i-th item points to the position of the first nonzero entry of the i-th row in the E array. The n + 1-th item which is one greater than the number of nonzero entries in the matrix is regarded as a delimiter. For example, the third item in RP (i.e 4) points to the 4-th item in array E (i.e. 4.0) which is the first nonzero entry of the third row. The CI array has as many items as the E array and each item contains the column position of the corresponding element in the array E. For example, the column position of 4.0 is 3.

### 3.2 Condensation Algorithm

The purpose of the Condensation algorithm is to convert the representation of the sparse matrix to a format better suited for the Polymorphic-torus. In specific, the goals are

(1) to assign only the nonzero entires of the matrix to processors so that the converted format is denser and the utilization of the SIMD system can be increased;

(2) to preserve the connectivity of the sparse matrix; and

(3) to related the RP and CI arrays to the virtual coordinate of the processors (i.e. [layer, pidx, pidy]) in the Polymorphic-torus.

The above algorithm consists of two phases. In the first phase, the RP array is checked to determine the length of a logical row (i.e. row_size). Due to the sparsity, the number of nonzero entries in a row of the sparse matrix is usually small. Therefore, multiple matrix rows can be bound to a single processor row, to obtain a denser data structure. Furthermore, the length of a logical row is always chosen to be a power of 2 to simplify the control. Figure 3 depicts the multiple-row binding.

```
CONDENSATION(RP, E, N)
int RP[],N; /* N: side size of processor array */
float E[];
{
   int i=0; /* dummy variable */
   int row=0; /* total no. of rows in the matrix */
   int max=0; /* maximum length of row in the matrix */
   int row_size=1; /* size of a logical row */
   int pointer=1; /* pointer to the beginning of next row */
   int counter=1; /* counter of elements in array E */
   int layer=0, pidx=0, pidy=0; /* virtual coordinate */
   struct quadruple { /* data structure of mapping form */
      float element; /* nonzero value and virtual coordinate */
      int  layer, pidx, pidy; }
   quadruple mapping[];
   boolean start; /* flag used to indicate the starting of a logical row */
   /* Phase 1 */ /* check RP array to count the no. of rows and
   determine the maximum length of row */
   while (RP[i]!=NULL){
      if ((RP[i]–RP[i–1]) > max)
         { max=RP[i]–RP[i–1]; }
      i=i+1; }
   row=i;
   if (max > N) exit(); /* row length larger than array size, exception */
   /* Determine the size of a logical row */
   /*     for matrix side size greater than processor array side size: */
   /*     Try to map multiple matrix row into a processor row. */
   if (row > N){
      while ( row_size < max ) { row_size=row_size*2;} }
   /* for matrix side size smaller than and equal to processor array side
   size */
   else { row_size= N; }
   /* Phase 2 */ /* read Element array and determine the coordinate
   of binding processor */
   i=0;
   While (E[i]!=NULL){
      /* binding element */
      mapping[i].element=E[i];
      mapping[i].layer=layer;
      mapping[i].pidx=pidx;
      mapping[i+ +].pidy=pidy;
      if (+ +counter= =RP[pointer]){ /* beginning of a new logical row */
         start=TRUE; /* binding a new logical row below */
         while ( start || (RP[pointer]= =RP[pointer+1])){
         /* while loop also detects empty rows */
            pidy=((pidy/row_size)+1)*row_size;
            if (pidy > = N) { /* jump to a new physical row */
               pidx=pidx+1;
               /* folding to next layer while fills up a array plane */
               if (pidx= =N) { pidx=0; layer+ + ; }
               pidy=0; }
            start=FALSE; pointer+ + ; }}
      else{ pidy+ + ; }
}
```

The second phase of the algorithm calculates the virtual processor coordinate for each element in the E array. An example of this calculation is shown in Figure 4, which uses the matrix in Figure 2 as input. For example, the first element is bound to processor [0, 0, 0] (layer = 0, pidx = 0, pidy = 0). A "counter" is increased to check whether the next element belongs to a new matrix row. If not (i.e. the case of both 2nd and 3rd elements), "pidy" is increased while "layer" and "pidx" remain unchanged. When a new matrix row begins (i.e. 4th element), decision is made whether a new processor row needs to be started. If yes, "pidx" is increased to reflect the binding. This process continues until one layer is completely filled then "layer" is increased so that a second layer of memory is needed as a virtual processor array. For simplicity, the algorithm listing handles only the single-layer situation.

Although simple, the condensation algorithm is powerful and delivers denser result than the method in DAP [7]. In comparison, the condensation algorithm "squeezes" the matrix in the "pidy" direction while the method in DAP [7] squeezes the matrix in the direction of "layer". The difference in the condensation philosophy lies in the difference in the architectural support. For DAP, to fully utilize the row/column highway capability, the placement of a non-zero element is confined to both row and column directions. For the Polymorphic-torus, the "short-circuit" capability allows us to move the non-zero elements more freely. With more freedom in moving the non-zero elements, the condensation algorithm yields a denser matrix, which leads to a higher utilization of the SIMD system.

## 4. CONDENSED MATRIX VECTOR MULTIPLICATION ALGORITHM

One advantage of the condensation algorithm is that the nonzero entries in the same row of the matrix are bound into the same row of the polymorphic-torus. By embedding a sum tree for each matrix row in the Polymorphic-torus [4, 5], the summation of the product terms from row/vector multiplication can be performed in an optimal logarithmic time. The other advantage is that a global bus can be formed among all processors so that variables (e.g., the vector elements) can be broadcasted in a unit time.

The purpose of this algorithm is to compute X′ = ΛX, where A is a condensed nxn sparse matrix with nonzero entries $a_{ij}$ contained in the E array and column index j in the CI array; X is a` nx1 vector containing $x_j$ and X′ is the updated X. Iterative methods are used for solving and the inputs to this algorithm are
(1) the size of the logical row,
(2) initial guess of $x_j$s with $x_j$s located at the first locations of each logical row;
(3) the element array, E[], bound to processor [layer, pidx, pidy] according to the condensation algorithm; and
(4) the column index array, CI[], distributed in the same way as the E array.

```
Condensed_Matrix_Vector_Multiplication(n,a,ci,row_size,x)
int row_size, n; /* size of logical row and side size of matrix */
int ci; /* column index of $a_{ij}$ distributed as described */
float a, x; /* $a_{ij}$ and $x_j$ */
{
    int i; /* dummy variable */
    int x_index=0, y_index=0; /* index of processor which broadcasts $x_j$ */
    int distance=1;  /* relative distance used in logarithm tree sum */
    boolean btemp=TRUE; /* TRUE while holds a partial sum */
    float ftemp;

    /* Phase 1: Distribute $x_j$ */
    for(i=1; i < =n; i+ +){
        SYNC { /* global synchronization */
        if (ci= =i) { /* check column index to determine whether to receive
        x */
            /* get x from the first processor in the i-th logical row */
            ftemp=x of [x_index, y_index]; }
        {
        /* up to beginning of next logical row */
        y_index=y_index + row_size;
        if (y_index> = N){y_index=0; x_index=x_index + 1; }
    }
    /* Phase 2: Multiplication */
    x=a*ftemp;
    /* Phase 3: Summation by tree sum */
    for (i=0; i <log row_size; i+ +){
        SYNC { /* global synchronization */
        /* accumulate partial sum by left sibling */
        if ( btemp && (pidy[i]= =0)){ /* ccheck the i-th bit of pidy */
            x=x+x of [pidx, pidy + distance]; }
        }
        else{
            btemp=FALSE;}
        distance=distance*2; } /* increase distance for higher level */
}
```

The output of the algorithm is the updated $x_j$s for the next iteration in a complete matrix solving. Furthermore, the new $x_j$s are in the same place as the old $x_j$s.

The complexity of the algorithm can be analyzed in accordance with the three phases of the algorithm as follows. In the first phase, the distribution phase, the variables $x_j$ in the dense vector, are broadcast in sequence to processors that hold $a_{ij}$. This phase needs O(n) steps for an nxn matrix. Note that a global bus is configured for the broadcasting, and the sender to and the receivers from the bus are determined simultaneously and locally in each processor by using the Column Index (CI) bound to each processor at the condensation stage. In the second phase, a floating-point multiplication is performed locally in every processor to produce a product; this takes one floating-point multiplication step. Finally, in the third phase, new $x_j$s are obtained by accumulating the corresponding partial products. This third phase takes O(log(logical_row_size) ) floating-point addition steps because all corresponding partial products for each $x_j$ are allocated in the same processor row by the condensation algorithm and furthermore n sum trees (one for each condensed matrix row) can be embedded simultaneously in the Polymorphic-torus. For simplicity, only the single-layer case is handled by the above algorithm. When multiple layers are presented, the complexity is in proportion to the number of layers.

The speedup of the algorithm is contributed by the capability of reconfiguring a global bus and many sum trees adaptive to the size of the matrix row after the condensation. In fact, the reconfiguration required in the sparse matrix vector multiplication is dynamic and dependent heavily on the local conditions. The polymorphic feature strongly emphasizes such a local and data-dependent reconfiguration. It is therefore understood that the speedup is attributed partially to the dynamic reconfigurability and partially to the condensation. The merit of this two-stage approach is discussed further in the next section.

## 5. DISCUSSIONS

### 5.1 Higher Density

The condensation process converts the sparse matrix into a dense data structure whose density implies low number of idle processors and high system utilization. Furthermore, it allows a larger matrix to be solved in a small processor array.

Compared with the layer-oriented condensation used in DAP [7], our condensation method packs matrix along pidy-direction (or row direction) and gives higher density. In fact, it is extremely difficult, if not impossible, for the layer-oriented condensation to yield a full matrix. The layer-oriented method used in DAP can be used in the Polymrophic-torus because the column/row broadcasting capability is supported both in DAP and the Polymorphic-torus as discussed. Vice versa, our condensation algorithm is applicable to the DAP architecture, however the efficiency will be lower because the DAP lacks the capability of embedding simultaneous sum trees. In summary, the condensation results in higher density but requires a corresponding reconfigurability for the best speedup.

### 5.2 Polymorphic

The choice of condensing a sparse matrix along the row direction is driven by the polymorphic feature of the Polymrophic-torus, namely, multiple sum trees can be formed adaptively to the condensed matrix rows for a logarithmic-time summation and a global bus can be formed with a unit broadcast time. Contrasted with MPP, the polymorphic feature is superior in the following aspects:

(1) the complexity of the broadcasting phase for MPP is O(n*N) if the new $x_j$s are circulated within the array and is O(n) + T if the $x_j$s are circulated via the central controller, where T is the time to move the entire matrix through the staging memory. In the latter case, T becomes the dominating factor;

(2) the complexity of the summation phase of MPP is O(logical_row_size) in comparison with O(log(logical_row_size)) of the Polymorphic-torus.

Aspects (1) and (2) also reveal the value of the polymorphic feature added to the plain mesh network.

The algorithm for the Polymrophic-torus is also applicable to MPP and can increase its its system utilization due to the high density achieved by the condensation. Compared with the worst-case arithmetic operation count $O((n/N)^3)$ of DAP, the arithmetic operation count for the polymorphic-torus ( O(1) for multiplication and O(log(logical_row_size)) for addition) produce a faster solver. However, our O(n) cost in the broadcasting phase may be inferior to that of DAP $(O((n/N)^3))$ for a large N. Nevertheless, considering the relative low broadcasting cost to the cost of the floating-point operation (e.g. 1:9 in DAP), our algorithm is more suitable for bit-serial SIMD machines that incur a high arithmetic cost.

### 5.3 Ideal Architecture

It is appropriate to ask what is the ideal SIMD architecture and algorithm for the sparse matrix vector multiplication. We expect the algorithm/architecture mapping to bear the following features:
(1) a matrix row is mapped onto a group of processors that can be structured as a tree;
(2) a matrix column is mapped onto in a group of processors that can be structured as a bus; and
(3) the condensed structure is full.
Such a mapping then allows for a O(1) broadcast, a O(1) multiplication and a O(log(row_size)) summation, which is the lowest complexity of the sparse matrix vector multiplication. Moreover, there is no idle processor in the SIMD system. Such an ideal mapping remains as an open research topic.

## 6. CONCLUSION

We have presented a sparse matrix vector multiplication algorithm on the Polymorphic-torus, a reconfigurable massively parallel fine-grained SIMD architecture. We emphasized in the paper how reconfigurability, or the polymorphic feature, can help to match the sparse connectivity onto a regular network topology. We advocate a two-stage approach.

The first stage, the condensation process, is an algorithm that converts the sparse connectivity into a more uniform data structure suitable for the topology of the polymrophic-torus. More specifically, many matrix rows can be packed into a processor row such that there are fewer idle processors and the system utilization is higher.

The second stage performs the condensed matrix vector multiplication. This second stage requires the Polymrophic-torus to reconfigure itself to match the condensed data structure resulted from the first stage. The polymorphic feature allows all multiplications be done in unit time while the summation is done in logarithmic time, which is optimal.

The condensation algorithm is applicable to other SIMD machines such as DAP or MPP. When applied to MPP, the utilization of MPP is higher due to the higher density achieved by the condensation. For its application to DAP, there can be no improvement because the DAP lacks the required companion reconfigurability and the condensation algorithm does not effectively use DAP's capability in row/column broadcast.

In a more general view, what has been exploited in this paper is a methodology of mapping a static irregular task graph into a massively parallel fine-grained SIMD architecture with regular network topology. We feel that it is difficult either to demand the algorithm designer and/or compiler to translate the sparse connectivity directly down to the network or to demand the architecture to reconfigure itself to match the irregular task graph. Rather, we advocate that the mapping process be done in a compromised way, i.e., the irregular task graph is converted to a more uniform one while the architecture is given more reconfigurability to meet the converted task graph.

## Bibliography

1. S. F. Reddaway. DAP - A flexible number cruncher. *Proc. LASL Workshop on Vector and Parallel Processors*, pages 233-234, 1978.

2. K. E. Batcher. Design of a Massively Parallel Processor. *IEEE Trans. on Computer*, C-29(9):836-840, September 1980.

3. W. D. Hillis. *The Connection Machine*. The MIT Press, Cambridge (MA), 1985.

4. H. Li and M. Maresca. Polymorphic-Torus Network. *Proc. Int. Conference on Parallel Processing*, pages 411-414, 1987.

5. H. Li and M. Maresca. Polymorphic-Torus : A New Architecture for Vision Computation. *Proc. IEEE Workshop on Computer Architectures for Pattern Analysis and Machine Intelligence*, pages 176-183, Seattle, 1987.

6. M. Maresca and H. Li. Toward Connection Autonomy of Fine Grain SIMD Parallel Architecture. *Int. Conf. on Parallel Processing for Computer Vision and Display*, Leeds (UK), 1988.

7. M. Morjaria and G. Makinson. Unstructured Sparse Matrix Vector Multiplication on the DAP. in D. Paddon, editor, *Supercomputers and Parallel Computation*, pages 157-166, Claredon Press, 1984.

8. R. Barlow, D. Evans, and J. Shanehchi. Sparse Matrix Vector Multiplication on the DAP. in D. Paddon, editor, *Supercomputers and Parallel Computation*, pages 147-155, Claredon Press, 1984.

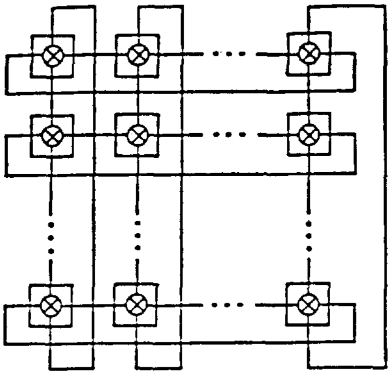


Figure 1. Two-level Interconnection Network of the Polymrophic-Torus

$$A = \begin{bmatrix} 0.0 & 1.0 & 0.0 & 0.0 & 2.0 & 0.0 & 0.0 & 0.0 \\ 3.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 4.0 & 0.0 & 0.0 & 5.0 & 0.0 & 0.0 \\ 0.0 & 6.0 & 0.0 & 7.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 8.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 9.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 2.0 & 0.0 & 3.0 \end{bmatrix}$$

**Row pointer/ Column Index representation:**

Element array: E[ ] = 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 9.0, 1.0, 2.0, 3.0;
Row pointer array: RP[ ] = 1, 3, 4, 6, 8, 9, 10, 11, 13;
Coloumn index array: CI[ ] = 2, 5, 1, 3, 6, 2, 4, 4, 2, 7, 6, 8;

Figure 2. Saprse Matrix Representation.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1.0 | 0 | 0 | 2.0 | 0 | 0 | 0 |
| 3.0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 4.0 | 0 | 0 | 5.0 | 0 | 0 |
| 0 | 6.0 | 0 | 7.0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 8.0 | 0 | 0 | 0 | 0 |
| 0 | 9.0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1.0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 2.0 | 0 | 3.0 |

(a) a 8x8 sparse matrix with maximum row length equal to 2 (max = 2).

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1.0 | 2.0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3.0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4.0 | 5.0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6.0 | 7.0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8.0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 9.0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1.0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2.0 | 3.0 | 0 | 0 | 0 | 0 | 0 | 0 |

(b) matrix is adjusted in row direction and row_size = 2 is determined.

| | | | |
|---|---|---|---|
| 1.0 | 2.0 | 3.0 | - |
| 4.0 | 5.0 | 6.0 | 7.0 |
| 8.0 | - | 9.0 | - |
| 1.0 | - | 2.0 | 3.0 |

(c) mapped into a 4x4 array, two logical rows are bound to a array row.

Figure 3. Binding of Multiple Logical Rows.

```
N = 4;      /* 4x4 array */
max = 2;    /* maximum length of row */
row = 8;    /* no. of row */
row_size = 2; /* size of logical row */
```

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| E[i] | 1.0 | 2.0 | 3.0 | 4.0 | 5.0 | 6.0 | 7.0 | 8.0 | 9.0 | 1.0 | 2.0 | 3.0 |
| pointer | 1 | 1 | 2 | 3 | 3 | 4 | 4 | 5 | 6 | 7 | 8 | 8 |
| RP[pointer] | 3 | 3 | 4 | 6 | 6 | 8 | 8 | 9 | 10 | 11 | 13 | 13 |
| counter | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| layer | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| pidy | 0 | 1 | 2 | 0 | 1 | 2 | 3 | 0 | 2 | 0 | 2 | 3 |
| pidx | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 2 | 2 | 3 | 3 | 3 |

Figure 4. Execution Snapshot of Matrix in Figure 2.

# ALMOST LINEAR SPEED-UP OF DISTRIBUTED DISCRETE EVENT SIMULATIONS

*Boris D. Lubachevsky*

Bell Laboratories
Murray Hill, New Jersey 07974

## ABSTRACT

As yet the problem of distributed discrete event simulation has found no commonly accepted solution. The paper presents a new distributed simulation algorithm which explores the topology of the simulated system using precomputed minimum propagation delays between subsystems. The algorithm also uses opaque periods which are the delays caused by the non-preemptive states these subsystems can enter. The algorithm provably achieves $\Theta(N/\log N)$ speed-up when run on an appropriate physically realizable $N$-processor parallel computer. No other algorithm for distributed discrete event simulation has been theoretically shown to achieve this level of efficiency. Experiments show speed-ups of greater than 20 on 25 processors of a shared memory MIMD computer and greater than 1900 on $2^{14}$ processors of a SIMD computer.

## *1.* INTRODUCTION

The goal of simulation is to reconstruct the time history of a system under investigation. Efficiency in a *serial* discrete event simulation means an efficient implementation of a priority queue, which is a well studied problem. In contrast, it is not known how to design and validate an efficient *distributed* discrete event simulation algorithm in the general case.

Several approaches have been proposed for the problem including Time Warp [5], Deadlock Detection/Resolution [1], Deadlock Avoidance [4] and other techniques [11,10]. However not much is proven about these algorithms. The few empirical investigations on today's small parallel processors [2, 3, 12] do not unquestionably confirm their efficiency and scalability.

We argue that in order for a distributed discrete event simulation to be efficient, the *topology* of the simulated system should be explored. A new algorithm is presented which employs *precomputed minimum propagation delays* between subsystems and the *opaque periods* which are the delays caused by the non-preemptive states these subsystems can enter. It is proven that on an appropriate physically realizable parallel processor the algorithm achieves almost linear speed-up for a large class of simulations. Specifically, it is shown that if $N$ processing elements (PEs) execute the proposed simulation algorithm in parallel, and if the simulated system exhibits sufficiently many events, then, on average, processing one event requires the $O(\log N)$ instructions of one PE. This implies a $\Theta(N/\log N)$ speed-up for a stationary event stream.

The practical efficiency of the proposed algorithm is shown by simulating several examples, in particular, asynchronous multiple-loop queuing networks, a task which is usually considered difficult for parallel programming. Experiments on a shared memory MIMD bus computer (Sequent's Balance) and on a SIMD computer (Connection Machine) show speed-ups of greater than 16 on 25 PEs of a Balance, and greater than 1900 on $2^{14}$ PEs of a Connection Machine.

In [6, 7, 8] examples of application of an earlier version of the algorithm were described. This paper presents new examples, an improved formulation of the algorithm, and its efficiency proof.

## 2. EXAMPLES

Four examples are considered: 1) a queuing system, 2) an Ising model, 3) a timed logic simulation, and 4) billiards. In the first three examples, the discrete events occur in a fixed network, which is an $n \times n$ grid on a torus in examples 1 and 2. In example 4, events occur in random space locations.

**A queuing system.** A node of the network is a server with an attached input queue buffer of infinite capacity. An *idling* server is constantly trying to change its status to *serving* by removing a job from the input buffer and starting service. The service durations can be deterministic or random. Their distributions are arbitrary except that the durations are bounded from below by $\sigma$ time units, $\sigma > 0$. When the service is completed, the job is either deleted from the system or sent into the input queue of one of the four neighboring servers. After a job is served, the server assumes an *idling* status if no more jobs remain in its input queue or, if at least one job is left in the input queue, the server removes one job from the queue and resumes a *serving* status.

**Asynchronous Ising model.** Each node hosts a physical atom. The atom at node $i$ has a magnetic spin $s(i)$: $s(i) = +1$ or $s(i) = -1$. The spins attempt to change at asynchronous, discrete, and random times. Attempted spin change arrivals for a particular atom form a Poisson point process. Arrivals for different atoms are independent, the arrival rate is the same for each atom.

When an attempt arrives, the new spin is determined, using the spin values of the given atom and the neighboring atoms just before the update time. A random "coin tossing" may also be involved in the determination.

Well-known among physicists is the Monte-Carlo algorithm [9] for Ising simulations. In this algorithm, spins are chosen for update in a serial order. Algorithm [9] is traditionally believed to be inherently serial. Contrary to this belief, our algorithm is an exact and efficient parallel counterpart to [9].

**Timed logic simulation.** A logic element has several inputs and several outputs. A network of such elements is considered in which, except for several external inputs and outputs, each element input is an output of another element. At each time each input or output has a signal value of 0 or 1, the signals of all outputs of a given element are the same and may only change simultaneously. The changes, if they occur, are instantaneous and are identified with the discrete events. If the value of an input to an element changes at time $t$, then the values of all outputs at time $t$ + *delay* may change according to a specified law, e.g. according to the logical function the element represents. Here *delay* is a positive value specified for each element before the simulation. A complication of the model is that the delays can be different not only for different elements, but also for different changes, e.g. the "rise" delay, when output changes from 0 to 1 may be smaller than the "fall" delay when the output changes from 1 to 0. As a result, a change in an output scheduled by a change in an input may be canceled later when another input change arrives.

**Billiards.** Identical balls are moving on a surface of a torus. Ball collisions are the events. The two velocities after a collision are easily computed given the two velocities before the collision, assuming no change in total momentum and energy. The billiards simulation is an easily understandable prototype for many important simulation problems, e.g. simulation of hard-sphere molecular dynamics in statistical physics.

## 3. ALGORITHM

In a simple form of the algorithm, a server, atom, logic element, or square on the billiards space is represented by a separate PE. PE$i$ maintains the *pool of tentative local events* $\Pi_i$. Each event $e \in \Pi_i$ is a pair $e =$ (contents, time). PE$i$ processes its events one at a time, during which it modifies its own pool $\Pi_i$, and/or the pools $\Pi_j$ of its neighbors (in the simulated network).

For the queuing network, it can be shown that the positive lower bound $\sigma$ on the service times induces the positive minimum propagation delays and that opaque periods correspond to service periods which can not be preempted by other activities.

In the Ising model example, the minimum propagation delays are zero. However, it has been shown [7] that substantial opaque periods do exist and these make possible efficient parallel simulations.

In the timed logic simulations, the minimum propagation delays are well defined, but determining the opaque periods is somewhat different from the previous two cases. For example, an *OR*-gate with the signal 1 at one input is guaranteed to remain opaque until this input changes.

In the billiards simulation, opaque periods do not exist while the minimum propagation delay requires the assumption that there is a speed limit for the balls. If $v$ is the average ball speed, and $n$ is the number of balls, then the exact upper speed limit is $vn^{1/2}$. If the number of balls is not specified no fixed speed limit can be guaranteed. But because the tail probability of large speeds is very small, the algorithm which relies on the assumption that speed limit is a constant, say $Cv$ with a suitably large $C$, would be broken rarely. A simple check-point technique works when it breaks: the speed limit is adjusted, and the simulation is repeated starting with the latest check point.

**Figure 3.1.** An algorithm for distributed discrete event simulation

```
1. while $floor < end\_time$ do {
2.    compute estimate $\alpha(i)$ of the earliest time, when the history
                at node $i$ can be affected by the other nodes ;
3.    synchronize ;
4.    while the minimum of event times at node $i$, $T_i$, satisfy
                $T_i \le floor + B$ and $T_i < \alpha(i)$ do {
5.       process events $e$ with locally minimal time, $T_i$ ;
         if required, schedule new events for $\Pi_i$ or other $\Pi_j$
            and/or delete some events from $\Pi_i$ or other $\Pi_j$;
6.       delete the processed events from $\Pi_i$
                and compute new $T_i$ ;
      };
7.    synchronize ;
8.    $floor := \min_{1 \le i \le N} T_i$ ; broadcast $floor$ to all nodes ;
9.    synchronize ;
   }
```

A form of a general algorithm which uses both the minimum propagation delays and the opaque periods is shown in Fig.3.1. Observe that the algorithm is synchronous while the simulated system may exhibit events asynchronously. The algorithm uses the minimum propagation delays in tandem with the *bounded lag restriction*. The latter means that the difference in the simulated time between the events processed concurrently must not exceed a predetermined positive constant, say, $B$. Initially $floor = 0$. Each PE$i$ is supposed to execute the program in Fig.3.1 on behalf of its node, $i$.

If the number of available PEs is less than the number of nodes, one PE may be assigned several nodes. In this case, whenever the algorithm prescribes independent actions by different nodes, a PE takes the nodes it carries in turn and performs the actions in turn. For example, to process events (Steps 4, 5, and 6), a PE takes an $i$, processes events in $\Pi_i$ and modifies $T_i$ as long as conditions in Step 4 are met; then the PE takes next $i$, etc. When the PE exhausts all $i$ in its subnetwork, it invokes synchronization, Step 7. Synchronization, minimization in Step 8 and some other actions require cooperation among the PEs. A way to implement such cooperative actions is described in Sec.4.

We suggest two methods for computing estimates $\alpha(i)$. One method uses formula

$$(3.1) \qquad \alpha(i) = \min_{\substack{j \in S\downarrow(i,B)) \\ j \ne i}} \{d(j,i) + \min\{T_j,\ d(i,j) + T_i\}\},$$

where $d(i,j)$ is the minimum propagation delay from node $i$ to node $j$, and $S\downarrow(i,B)$ is the incoming reachability sphere of radius $B$, i.e. the set of nodes $j$ such that $d(j,i) \le B$. Note that delay $d(i,j)$ must be precomputed so that the triangle inequality

$$(3.2) \qquad d(i,j) + d(j,k) \ge d(i,k)$$

holds for any three nodes $i$, $j$, and $k$.

According to (3.1), each node $i$ computes its $\alpha(i)$ independently of the other nodes. A different method to compute $\alpha(i)$, wherein different nodes and the PEs carrying them cooperate, is presented in Fig.3.2. Here $\alpha(i)$ are updated in several iterations cooperatively by all PEs.

**Figure 3.2.** Dynamic algorithm for computing $\alpha(i)$

```
2.1.  $\alpha(i) := +\infty$ ; $\beta(i) := T_i$ ;
      $MIN\_\beta := floor$ ; $\alpha\_CHANGED := 1$ ;
2.2.  synchronize ;
2.3.  while $MIN\_\beta \le floor + B$ and $\alpha\_CHANGED = 1$ do {
2.4.     $new\_\beta(i) := \min_{\substack{j \in neighbors(i) \\ j \ne i}} \{d(j,i) + \beta(j)\}$ ;
2.5.     if($new\_\beta(i) < \alpha(i)$) {
            $changed\_\alpha(i) := 1$;
            $\alpha(i) := new\_\beta(i)$;
         } else $changed\_\alpha(i) := 0$ ;
2.6.     synchronize ;
2.7.     $\beta(i) := new\_\beta(i)$ ;
2.8.     $MIN\_\beta := \min_{1 \le i \le N} \beta(i)$ ;
         $\alpha\_CHANGED := \max_{1 \le i \le N} changed\_\alpha(i)$ ;
         broadcast $\alpha\_CHANGED$ and $MIN\_\beta$ to all nodes ;
2.9.     synchronize ;
      }
```

The convention for interpreting the code in Fig.3.2 is the same as for the code in Fig.3.1: For each step between synchronizations, a PE executes a loop over all nodes $i$ within its subnetwork. Auxiliary variables $\beta(i)$, $changed\_\alpha(i)$, and $new\_\beta(i)$ are employed by the algorithm. After $k$ iterations $\beta(i)$ becomes an estimate for the earliest time when existing events can affect node $i$ after traversing exactly $k$ links in the graph.

## 4. O(LOG K)-TIME, O(K)-SPACE TREE ALGORITHMS

For **synchronization** barriers, [6] suggests the use of a binary tree implemented in hardware. On a shared-memory parallel computer we can implement the barrier using a pointer-tree in the shared memory. Fig.4.1 presents the *C*-language code of the barrier routine.

The idea of the algorithm is simple. Let $K$ be the number of PEs. Each PE must have its individual identification $pe\_id$ in the range 0 to $K-1$ and must invoke the routine with this identification. Let us first assume that $K$ is a natural power of 2. The algorithm works in tours. At the first tour the $K$ PEs form $K/2$ pairs, each pair electing a winner for the second tour. At the second tour the $K/2$ winners form $K/4$ pairs and so on until after $\log_2 K$ tours one winner remains. The way of pairing

PEs is predetermined, and so is the winner in each pair. The final winner signals the end of algorithm by changing *flag*[*ROOT*][*turn*]. All the PEs can see this flag changed. If $K$ is not a natural power of 2, then the tree has $\lceil K/2 \rceil + \lceil \lceil K/2 \rceil /2 \rceil + \cdots + 2 + 1$ nodes and some PEs at some tours are without a companion. Pointer arrays *companion*[ ] and *next*[ ] reflect the structure of the tree and are initialized in the shared memory before the simulation starts (the code of initialization routine is omitted). The case $K = 9$ is represented in Fig.4.2.

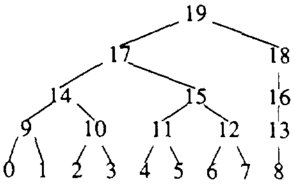
**Figure 4.1.** Barrier tree-synchronization

```
# define ROOT 19   /* for 9 PEs */
shared int flag[ROOT+ 1][2], companion[ROOT+ 1],
       next[ROOT+ 1];
static int turn = 0;

void sync(pe_id)
int pe_id;        /* pe_id is in the range 0 to 8 for 9 PEs */
{
  int index;
  index = pe_id;
  do {
       flag[index][turn] = 1 - flag[index][turn] ;
       while
         (flag[companion[index]][turn] != flag[index][turn]);
         /* wait_until
            (flag[companion[index]][turn] = = flag[index][turn]); */
       index = next[index];
     } while(index);
         /* index = 0 means the pe looses the tour */

  while (flag[ROOT][turn] != flag[pe_id][turn]);
   /* wait_until
       (flag[ROOT][turn] = = flag[pe_id][turn]);  */
  turn = 1 - turn;
}
```

**Figure 4.2.** The tree and the pointer arrays for 9 PEs



| *i* | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| *next*[*i*] | 9 | 0 | 10 | 0 | 11 | 0 | 12 | 0 | 13 | 14 |
| *companion*[*i*] | 1 | 0 | 3 | 2 | 5 | 4 | 7 | 6 | 8 | 10 |
| *i* | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| *next*[*i*] | 0 | 15 | 0 | 16 | 17 | 0 | 18 | 19 | 0 | 0 |
| *companion*[*i*] | 9 | 12 | 11 | 13 | 15 | 14 | 16 | 18 | 17 | 19 |

Array *flag* is initially zero. It then alternatively takes on values 1 and 0 to signal arrival of PEs to synchronization point, in the same time reinitializing itself before next invocation. Variable *turn* also takes on values 0 and 1 pointing alternatively to different copies of the array *flag*. Two copies of this array are used instead of one to prevent a deadlock.

Note that the presented solution works efficiently for a CREW (concurrent-read-exclusive-write) shared memory parallel computer. It is easy to modify the code into an efficient EREW version.

Clearly, this procedure takes time $O(\log K)$ and requires memory space of $O(K)$. These are asymptotically minimal values for both parameters.

For **minimization or maximization and broadcasting**, [6] suggests the use of the same hardware binary tree as for the barrier. As for the barrier, the same software pointer-tree in the shared memory can be used for these purposes.

## 5. EFFICIENCY PROOFS

It is assumed below that one node is carried by one PE, so $N = K$. Similar argument applies for the cases where there are fewer PEs than nodes $N$, provided that the maximal number of nodes per PE is $O(1)$ when $N \rightarrow \infty$. A key property which assures the efficiency of the algorithm is:

(#) sphere $\bar{S}{\downarrow}(i,B)$ contains only a finite number of nodes, finitely bounded from above independently of $N$

Thus, we can not efficiently simulate, say, a fully connected network, with small propagation delays between any pair of nodes. On the other hand, the simulations in which events occur is a space of a low dimension with substantial propagation delays between distant nodes, are feasible. Using (#) and other mild assumptions, we show that each PE spends $O(1)$ at each iteration, not counting synchronizations, computing, and broadcasting the minimum in Step 8. The latter requires $O(\log N)$ instructions per each PE. Enough "distributability" of the activity in the simulated system is required for the simulation to be efficient. We impose a mild assumption on the event density which requires that on average at an iteration

(##) $\Theta(N)$ nodes have events within the processing range $[floor, floor+B]$

To violate (##) the system should exhibit a low activity, in which case the available parallelism should be small, and no parallel algorithm can be efficient.

Using (##) and some other mild assumptions, it is proven that $\Theta(N)$ nodes are processed at each iteration on average. Hence, on average, $O(\log N)$ instructions of one PE is spent for processing one event.

A disadvantage of the computation by formula (3.1) is that it does not involve opaque periods. Another more efficient algorithm for computing $\alpha(i)$ which uses both the minimum propagation delays and the opaque periods is presented in [8].

The latter method is used in the Ising model simulations. In this example, since the minimum propagation delays are zero, the efficiency proof outlined above is not valid. The framework for the efficiency evaluation in the Ising model can be explained in the form of the following easily understandable "salary upgrade" scheme:

Once a year the employees of a company are asked to sit around a table, each year in the same locations along the table. An employee receives a salary raise, if his/her previous salary does not exceed the minimum of the previous salaries of his/her neighbors at the table. Otherwise, the salary remains the same. The amounts of the raises are independent identically distributed positive random variables. How many raises will there be per year on average?

The salary represents the simulated time, a raise corresponds to an event processing and advancement of simulated time, and the form of the table represents the network topology. (A round table corresponds to a cyclic network.) Thus, more salary raises mean a more efficient algorithm. Specifically, if for $N$ employees, there are $> cN$ raises on average, where $c > 0$ is independent of $N$, then ignoring the $\log N$ degradation caused by synchronization, the algorithm exhibits a linear speed-up with efficiency of at least $100c\%$.

To prove that such a positive $c$ exists appears to be a difficult task. Empirical evaluation suggests that $c \cong .25$ for exponentially distributed raises (the case in the Ising model) if the topology is circular and $N \rightarrow +\infty$. It is also known empirically that when the dimension of the graph increases, $c$ decreases, roughly in inverse proportion to the dimension.

Proofs of efficiency for the considered algorithms pose several purely mathematical questions, which are as yet without answers.

## 6. EXPERIMENTS

We try to keep constant the event density by keeping the same density of "sources", and the same parameters of probabilistic rules for job movements and eliminations. Specifically, 1 source is maintained in each 4× 4 square, and, after the service, a job is equiprobably sent in 4 possible directions, unless it disappears which happens with probability $p = 1/10$. This results in that the fraction of nodes with events is in the range 0.51 to 0.59. The service duration has the exponential distribution of mean 1 which is shifted by 1 in the positive direction (so its mean value becomes 2 and $\sigma$, the lower bound on the service times, is 1). The lag bound $B$ is 4. We run the simulation from 0 until the *floor* reaches 200 simulated time units. The system reaches an equilibrium during the first 100 units, and all measurements are taken during the following interval from 100 to 200.

We simulate a $PM \times PM$ queuing networks on $K = P^2$ PEs. Fig.6.1 presents relative completion times for two series of runs, with $M = 4$ in one series, and $M = 16$ in the other. The variable in a series was $P$, $P = 1,2,3,4$, and 5. It is seen, that while both curves rise with the problem size, most of the increase occurs at smaller sizes. Fig.6.2 presents number of iterations of the algorithm in Fig.3.1. At small sizes, the number of iterations increases noticeably (see Fig.6.2), which is the largest contributor to the completion time increase. For larger sizes, the number of iterations stabilizes. In theory, the number of iterations is $O(1)$, i.e. it should remain roughly constant for large sizes, which is exactly what the experiments show. After this constant number of iterations is reached, the main contributor in the completion time increase becomes the execution times of synchronizations. This agrees with the fact that the increase in both curves slows down at 16 PEs. 16, being a natural power of 2, is a "good" number of participants for the synchronization algorithm. One should expect similar slow-downs for 64 PEs, 256 PEs etc. 4 is also a "good" number, but its "good" influence is seemingly overshot by "bad" influence of the small size.

**Figure 6.1.** Relative completion times for different simulation sizes

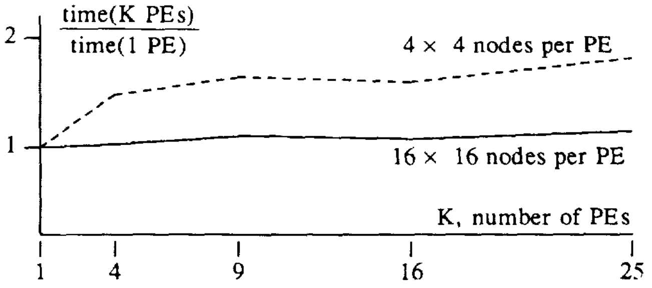


**Figure 6.2.** Numbers of iterations for different simulation sizes

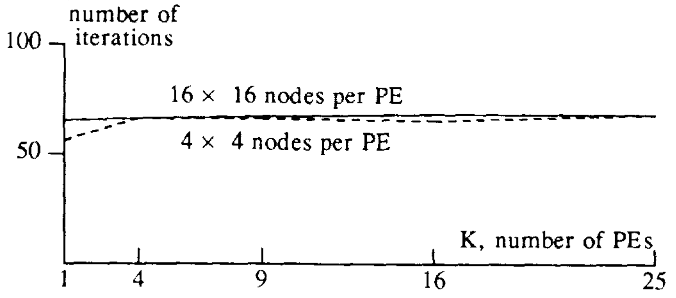
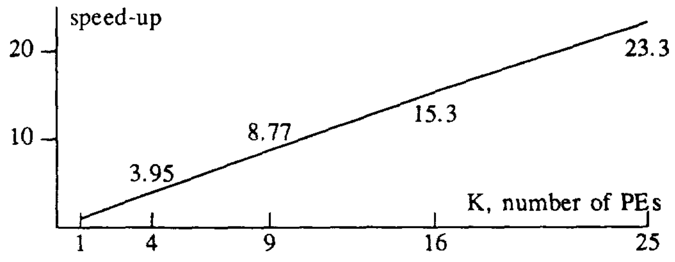


Fig.6.3 presents the traditional self-speed-up computed as

$$\text{speed-up } (K \text{ PEs}) = \frac{\text{execution time for 1 PE}}{\text{execution time for } K \text{ PEs}},$$

where the same code executed with different numbers of PEs supplies the numerator and the denominator for the fraction. The side of the simulated network square is 60, which is the minimal number divisible by 2,3,4, and 5.

**Figure 6.3.** Speed-ups in simulating a 60× 60 network



The shape of the graph is almost a straight line.

## REFERENCES

[1] Chandy, K.M. and Misra, J. Asynchronous distributed simulation via a sequence of parallel computations, *Communications of the ACM*, **24**, 3 (April 1981), 198-206.

[2] Fujimoto, R.M. Performance measurements of distributed simulation strategies, in *Distributed Simulation*, B.Unger, and D.Jefferson eds., SCS, 1988, *Simulation Series*, **19**, 3, 14-20.

[3] Gafni, A., Berry, O., Jefferson, D. Optimized virtual synchronization, *Proc. 2nd Int. Workshop on Applied Mathematics and Performance/Reliability Models*, Univ. of Rome II, 1987, 229-244.

[4] Holmes, V. *Parallel algorithms on multiple processor architectures*, Ph.D. dissertation, Comp. Science Dept., Univ. Texas at Austin, 1978.

[5] Jefferson, D.R. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7, 3 (July 1985), 404-425.

[6] Lubachevsky, B.D. Bounded lag distributed discrete event simulation (Extended Abstract), in *Distributed Simulation*, B.Unger, and D.Jefferson eds., SCS, 1988, *Simulation Series*, **19**, 3, 183-192.

[7] Lubachevsky, B.D. Efficient parallel simulations of dynamic Ising spin systems, *Journal of computational physics*, **75**, 1 (March 1988), pp. 103-122.

[8] Lubachevsky, B.D. Efficient distributed event driven simulations of multiple-loop networks, *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May (1988), Santa Fe, New Mexico, pp. 12-21.

[9] Metropolis, N., Rosenbluth, A.W., Rosenbluth, M.N., Teller, A.H., and Teller, E. Equation of state calculations by fast computing machines, *Journal of Chemical Physics*, **21**, no.6 (1953), pp. 1087-1092.

[10] Misra, J. Distributed discrete-event simulation. *Computing Surveys*, **18**, 1 (March 1986).

[11] Peacock, J.K., Wong, J.W., and Manning, E.G. Distributed simulation using a network of processors, *Computer Networks*, **3**, 1 (1979).

[12] Reed, D.A, Malony, A.D., and McCredie, B.D. Parallel discrete event simulation using shared memory, *IEEE Trans. Software Eng.*, **14**, 4 (1988), 541-553.

# Dynamically allocating sets of fine-grained processors to running computations

David Middleton

ICASE, NASA Langley Research Center
Hampton, VA 23665.

## Abstract

*We explore an approach to using general purpose parallel computers which involves mapping hardware resources onto computations instead of mapping computations onto hardware. Problems such as processor allocation, task scheduling and load balancing, which have traditionally proven to be challenging, change significantly under this approach and may become amenable to new attacks. We describe the implementation of this approach used by the FFP Machine whose computation and communication resources are repeatedly partitioned into disjoint groups that match the needs of available tasks from moment to moment. Several consequences of this system are examined.*

Keywords: *reconfigurable computers, partitionable computers, variable granularity, fine granularity.*

## Mapping hardware resources onto computation structures

A standard part of parallel computation is mapping the computations onto the given structure of the hardware resources. The FFP Machine supports an alternative approach of mapping hardware resources onto the structure of running computations. We describe the method it uses and discuss some of the consequences of this approach.

The motivation for this inversion is that, although computations are more mutable than hardware, they are also highly dynamic in their structure. For example, an inner product computation begins with many small operations, the individual multiplications, which might be performed simultaneously given a fine-grained hardware structure. This is followed by a single, potentially large, summation (which may have a finer structure) better suited to more coarsely grained hardware. As a result of computations' highly variable structure, a fixed hardware structure will likely encounter difficulties implementing some of their stages efficiently. The philosophy of mapping hardware onto computations leads to a hardware design with a more flexible structure, which may reduce mismatches between the structures of the hardware and of the various computation stages.

Operations such as program decomposition, task scheduling and load balancing, which have traditionally been both essential for good performance and highly sensitive to several disparate parameters, heavily reflect the "mapping computations onto hardware" philosophy. The "mapping hardware onto computation" view ought to recast these problems drastically, opening the way to new methods for solving them.

As part of this approach, the FFP Machine implements the concept of a *virtual machine*, defined as an abstract entity created to perform a single task and consisting of many processing elements connected by a tightly-coupled message-passing combining network. The FFP Machine *partitions* its hardware elements into disjoint *resource groups* such that each virtual machine is provided with one entire resource group dedicated to its progress.

## Properties of Partitioning

The FFP Machine is a reconfigurable fine-grained MIMD computer consisting of a linearly ordered set of processors which communicate through a tree-structured network of communication nodes. Each resource groups created by partitioning consists of a contiguous set of the processors connected by a tree of message processors embedded in the physical communication network. The following properties of the partitioning process are noteworthy.

> Partitioning is very fast, yet still provides the flexibility usually associated with late binding.
>
> From the innermost reduction rule of Backus's FP language [1], computations are defined as parenthesized expressions, with innermost parentheses delimiting computations that can proceed immediately. The expressions reside in the processors and resource groups are constructed by creating "breaks" where parentheses occur. Partitioning involves a single wave of messages that passes up through the tree network. The messages contain three bits; one indicates the presence of parentheses in the subtree

CH2649-2/89/0000/0191$01.00 © 1988 IEEE

and is sufficient for partitioning; the other two are used to determine which resource groups are delimited by a balanced pair of parentheses and so contain a virtual machine that can immediately proceed with its task. Each tree node calculates a logical sum, two logical products and sets its three communication channel switches. Figure 1 shows the internal structure of a communication node with the channel switches and the message processor which is allocated to one of the resource groups. Figure 2, Color Plate I (p. 693), demonstrates the result of partitioning, emphasizing the distinction between the physical tree structure and the tree structures of the resource groups. Partitioning takes logarithmic time, although with additional interlocks, constant time can be achieved through pipelining because the lower portions of the resource groups can be used while their upper portions are still being configured. Because of its simplicity, partitioning should add little overhead to machine operation and so may be performed frequently.

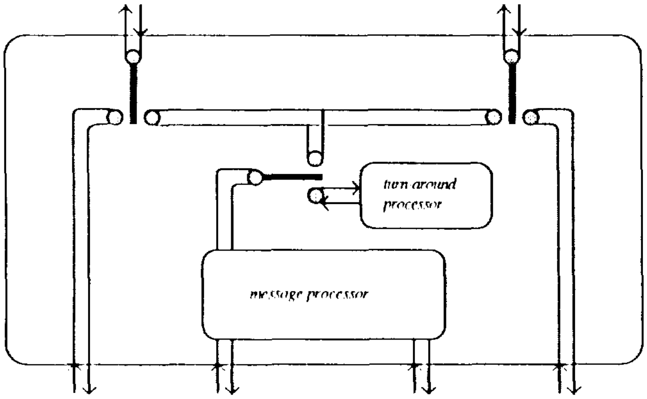


**Figure 1.** Communication node

- Apart from a virtual machine's determination of the pattern of parentheses it leaves in its result, no planning is required by either compiler or programmer to control the creation, activation or deletion of virtual machines. The size, placement and lifetime of the virtual machines is completely determined by the partitioning process, according to the positions of parentheses. These arise as the results of immediately preceding virtual machines without any prior calculation or storage of information.
- There are no size or alignment constraints placed on the virtual machines. Most notably, this avoids wasting resources through fragmentation, so, for example, a subtree of the physical machine with a thousand processors can support without help two virtual machines, one needing six hundred processors and the other, four hundred. (As a consequence of the non-alignment, the average depth of a virtual machine with $n$ processors is $(lg(n) + 2.0)$ rather than $\lceil lg(n) \rceil$.) Virtual machines are *variably-grained*: their size can be tailored to reflect closely their individual needs without regard for other virtual machines. In particular, the size of a virtual machine relates closely to the size of its operands; identical functions applied to different data will generally be performed by different sized virtual machines.
- Resource groups are constrained to contain non-overlapping groups of processors. This imposed locality provides an upper bound on the number of resource groups (three) that a communication node may be required to support, allowing the design of the communication node to provide dedicated hardware.

No contention or interference arises between communication operations occurring in different virtual machines; however, no communication can occur either. There does remain a communication bottleneck local to each resource group due to its tree structure. A richer interconnection in each resource group could be provided were the interconnection structure of the FFP Machine similarly enhanced [3, 5].

- Virtual machines can grow during their operation, with the communication network acquiring additional processors in a consistent fashion. This growth is achieved by shifting the contents of the processors so that more of them separate the pair holding the delimiting parentheses. In the current design, this *storage management* takes linear time and is the primary situation where one virtual machine can affect the operation of others, by potentially requiring that the contents of neighboring machines' processors also be shifted to make room.
- Multiple levels of parallelism are exploited. Concurrent virtual machines execute simultaneously, each internally exploiting fine-grained parallelism. (The term MSIMD has been used to describe parallel computers in which multiple SIMD machines execute simultaneously; by comparison, this might be described as an MMIMD machine).
- The message processor networks in the resource groups support *combining operations without* requiring costly associative memories in the switch nodes [2]. Each resource group has a tightly-coupled circuit-switched network of ALUs which can perform such cumulative operations (possibly within *groups* [7]) at hardware speeds rather than at the processors' instruction speeds. Cumulative operations provide a powerful mechanism capable of performing data permutations and parsing operations useful to the FFP Machine, without suffering from the bottleneck in a resource group. Other implementations

of combining networks in general purpose machines have resulted in switch nodes that are too costly, probably due to the range of possibilities that the nodes must handle. By assuming that any given task involves closely coordinated actions by the processors, the complexity of the communication nodes is greatly reduced.

## The potentials of flexible virtual machines

These virtual machines as supported by the resource groups defined above have a flexibility that provides opportunities for accomplishing tasks in new ways. Because partitioning creates virtual machines so cheaply, a task can profitably use many of them. In each of its stages, the resources already allocated to a task can be restructured into a different set of groups, so that, for example, stages that exhibit fine-grained parallelism can use many disjoint machines operating simultaneously. The following list demonstrates some of the ways that this flexibility can be used.

- It is possible to alternate between virtual machines that allow long-distance communication within the task, with the attendant communication bottleneck, and isolated virtual machines performing localized operations that communicate with no or greatly reduced contention.

- The TRAC machine could avoid explicit communication by reconnecting memory banks to different processors and so transferring data implicitly [4]. In an analogous fashion, some explicit communication in the FFP Machine can be avoided by reconfiguring the processors holding data into different resource groups, so that they belong to different virtual machines at different times. This is the standard method for passing results between functions when executing FFP programs.

- Computations structured as pipelines, or more generally, data-flow graphs (possibly with complex computations at the nodes) can be implemented by alternating between a set of virtual machines specialized to the individual nodes, and a set of virtual machines that perform the communication along the arcs of the graphs.

These uses, together with others, can be combined freely depending on the particulars of the task. We present one abbreviated example to show the possibilities [8].

OPS5 is a Production System language. When specified patterns can be found among subsets of known facts, corresponding actions are performed. Finding such patterns consumes a large majority of the processing time in OPS programs. The RETE algorithm, the best current technique for matching facts to the rule patterns, uses a discrimination network in which the nodes store partial matches found so far and compare them with new partial matches that arrive along their input arcs.

The discrimination network can be naturally implemented using virtual machines, as shown in Figure 2. A node has four parts including the local memory for storing partial matches and input and output buffer areas, each occupying as many processors as necessary. Pattern matching in each node uses a three stage cycle. With the node organized as a single virtual machine, a new pattern is broadcast from the input buffer to the processors holding partial matches. In the second stage, these processors are divided into many small machines each of which compares the new pattern with one previous partial match. In the third stage, successful comparisons cause a combined match to be placed in the output buffer.

Interleaved with the operation of virtual machines corresponding to nodes in the discrimination network is a set of machines corresponding the network arcs, which transmit successful matches from output areas of some nodes to the input areas of their descendants.

Figure 2 shows a discrimination network and its mapping as a hierarchy onto the linear array of processors. The graph is laid out as a series of nodes, each having four parts; individual processors within those parts are too small to be seen. Beneath the linear array of processors, bars show the groupings of processors into virtual machines for different phases in the matching operation. The first three rows correspond to the stages of the network node virtual machines and the last row corresponds to those for the arcs.

The arcs being able to send simultaneously relies on two facts: the discrimination network was created as a skew tree, and the input and output buffers could be placed within each node so as not to interfere. These choices display an important part of efficiently using virtual machines. Given the "logic in memory" and associative processing style of operation [6], it is less important that data be organized with regard to access methods that reflect sequential styles of algorithms. Instead, the data need to be organized so as to provide locality, in some sense, at appropriate stages in the tasks.

The ability of virtual machines to have different sizes depending on the amount of data is particularly important since the memories in the discrimination network nodes display a high variance with different input values.

## Conclusions

The approach of mapping hardware resources onto computation structures, rather than vice versa, provides many novel opportunities for performing tasks. The implementation of this philosophy embodied in the FFP Machine has a significant affect on the way in which computations are viewed and organized.

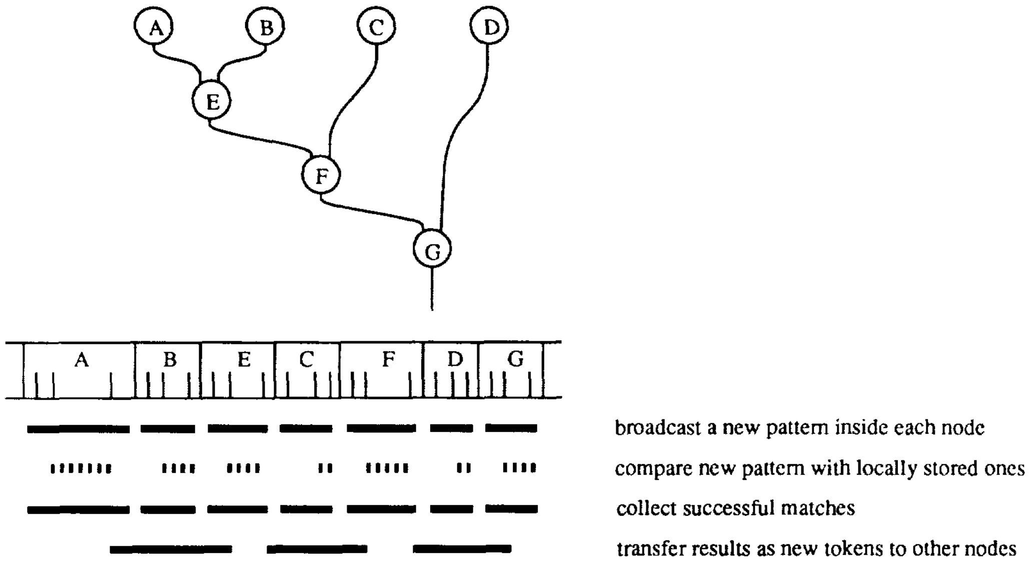

A
B
C
D
E
F
G
A
B
E
C
F
D
G
broadcast a new pattern inside each node
compare new pattern with locally stored ones
collect successful matches
transfer results as new tokens to other nodes

**Figure 2.** Virtual machine implementation of OPS5 discrimination network

Programming effort concentrates on arranging that data are organized to be clustered when they are combined or otherwise manipulated in a task. The "logic in memory" character by which the data reside in processors removes the requirement that the data be sorted and stored in structures that reflect the physical problem. "Associative programming" techniques allow data to be stored "out of order", in some sense, but with descriptors that determine when and how the data participate in operations [6].

Tasks are defined syntactically. The programmer's control over scheduling concentrates on creating and deleting the parentheses that delimit virtual machines. Barrier synchronization derives naturally from the partitioning mechanism; virtual machines delimited by non-innermost parentheses do not begin operation until those inner computations have completed and the parentheses are removed. Other synchronization and scheduling mechanisms can be created with little additional effort.

## References

[1] J. Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs", Communications of the ACM, Volume 21 No. 8, pp. 613-641, August 1978.

[2] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph and M. Snir, "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer", IEEE Transactions on Computers, Volume C32 No. 2, pp. 175-189, February 1983.

[3] J.N. Kellman "Parallel Execution of Functional Programs" Master's Thesis. University of California at Los Angeles. 1983.

[4] G.J. Lipovski and A. Tripathi, "A reconfigurable varistructure array processor", Proceedings of the 1977 International Conference on Parallel Processing, pp. 165-174, August 1977.

[5] D.A. Plaisted (a) "An Architecture for Fast Data Movement in the FFP Machine" Proceedings of the 1985 Conference on Functional Programming Languages and Computer Architecture. Springer-Verlag, LNCS 201. pp. 147-163. September 1985. Nancy, France.

[6] J.L. Potter, "Programming the MPP", pp. 218-229, in "The Massively Parallel Processor", edited by J.L. Potter, MIT Press, 1985.

[7] J.T. Schwartz, "Ultracomputers", ACM Transactions on Programming Languages and Systems, Volume 2 No. 4, pp. 484-521, October 1980.

[8] B.T. Smith and D. Middleton, "Exploiting fine-grained parallelism in Production Systems", Proceedings of the 7th Biennial Conference of the Canadian Society for Computational Studies of Intelligence, pp. 262-270, edited by R. Goebel, Edmonton, Alberta, Canada, June 6-10 1988.

# Portable Parallel Algorithms for Geometric Problems

(Preliminary Version)

Russ Miller*
Department of Computer Science
State University of New York at Buffalo
Buffalo, NY 14260 USA

Quentin F. Stout†
Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122 USA

### Abstract

Because the interconnection scheme among processors (or between processors and memory) significantly affects the running time, efficient parallel algorithms must take the interconnection scheme into account. This in turn entails tradeoffs between efficiency and portability among different architectures. Our goal is to develop algorithms that are portable among massively parallel fine grain architectures such as hypercubes, meshes, and pyramids, while yielding a fairly efficient implementation on each. Our approach is to utilize standardized operations such as prefix, broadcast, sort, compression, and crossproduct calculations. This paper describes an approach for designing efficient, portable algorithms and ~~gives~~ sample algorithms to solve some fundamental geometric problems. The difficulties of portability and efficiency for these geometric problems have been redirected into similar difficulties for the standardized operations. However, the cost of developing efficient implementations of them on the various target architectures can be amortized over numerous algorithms.

**Keywords** Portable parallel algorithms, computational geometry, data movement operations, distributed memory parallel computers.

## 1 Introduction

Massively parallel computers consisting of perhaps millions of processors are now becoming available. While such machines offer significantly faster solutions to many problems, they also impose severe programming requirements to utilize their potential. Old "dusty decks" do not typically work on such machines, and hence new algorithms and programs need to be developed. Since each processor contains only a small fraction of the total data, for most problems there must be extensive communication among processors. This communication often dominates the total running time of the program, and efficient programs must be developed with this in mind.

If the introduction of massive parallelism only brought about a one-time need to reprogram, then the reprogramming costs would at least be fairly well understood and for a variety of applications would be affordable. However, extensive additional costs are introduced due to the significant differences among massively parallel architectures. Different massively parallel architectures have significantly different communication characteristics, and hence have significantly different running times on the same programs. For example, on a square two-dimensional mesh with $n$ processors, it takes $\Theta(\sqrt{n})$ time on average for two processors to exchange information, while on a hypercube or pyramid it takes $\Theta(\log n)$. For $n$ processors to exchange data takes $\Theta(\sqrt{n})$ time on the mesh and pyramid, or $O(\log^2 n)$ time (worst-case) on the hypercube. Notice that in one case the pyramid is similar to the hypercube, while in the other case it is similar to the mesh. Due to such differences, for a single problem one may have two programs $A$ and $B$ with the property that on one massively parallel machine $A$ is significantly faster than $B$, while on another massively parallel machine $B$ is significantly faster than $A$.

This paper is concerned with developing algorithms which can be ported among different fine grain, massively parallel architectures and yield reasonably good implementations on each. Our approach is to write algorithms in terms of general data movement operations, and then implement the data movement operations on the target architecture. Efficient implementation of the data movement operations requires careful programming, but since the data movement operations form the foundation of many programs the cost of implementing them can be amortized. The use of data movement operations also helps programmers think in terms of higher-level programming units, in the same way the use of standard data structures helps programmers of serial computers.

In Section 2 we give several data movement operations, and in Section 3 we illustrate our approach by giving some geometric algorithms written in terms of these data movement operations. Many data movement operations have been proposed, and the list is still growing as programmers acquire experience in parallel programming. Our list is intended as an illustrative sample, not an exhaustive collection. Further, the types of problems for which this approach is useful is quite large, and in this short paper we make no attempt to even survey such problems.

## 2 Data Movement Operations

A variety of data movement operations have been proposed for parallel computers. Often they originated as steps in the midst of some algorithm, and then later it was realized that they might have widespread utility. More recently there have been attempts to promote specific data movement operations as a programming

*Partially supported by NSF grants DCR-8608640 and IRI-8800514.

†Partially supported by NSF grant DCR-8507851 and an Incentives for Excellence Award from Digital Equipment Corporation.

aid [2,3], or to develop a collection of data movement operations particularly useful for a specific architecture [5].

Several of the operations are defined in terms of some semigroup operation $\otimes$ over a set $B$, and our analyses of running time will assume that $\otimes$ can be computed in constant time. Sorting is a central operation, with several operations assuming that the data is already in sorted order. For such operations we assume that there is a linear ordering of the processors and a linear ordering of the set from which the items are chosen. Some operations are performed in parallel on disjoint consecutive sequences of items in sorted order, which are called *(ordered) intervals*.

Due to space limitations, we can give only a few of the proposed data movement operations. Two of the operations given below, namely, reducing a function and searching, originated with geometric problems, while the others have had somewhat wider usage. Interested readers might consult [2,3,5,6,7] for additional operations and extensive uses of the operations discussed here. Implementations of these operations for a variety of architectures appear in [7].

1. *Sort:* Given data distributed arbitrarily one per processor, order the data with respect to the processors.

2. *Merge:* Suppose that a set of data $D$ is chosen from a linearly ordered set. Further, suppose $D_1$ is ordered one item per processor with respect to one subset of the processors, and $D_2$ is ordered one item per processor with respect to a disjoint subset of the processors, where $D = D_1 \cup D_2$. The merge operation combines $D_1$ and $D_2$ to yield $D$ ordered with respect to the entire set of processors.

3. *Semigroup Computation:* Suppose each processor has a record with data from $B$ and a label, and that the records form ordered intervals with respect to their label. Each processor ends up with the result of applying $\otimes$ to all data items with its label.

4. *Broadcast/Report:* Broadcast and report are often viewed and implemented as inverse operations. Both operations involve moving data within disjoint ordered intervals. They also both require a distinct processor, called the *leader*, of each interval. In broadcasting, the leader of each ordered interval delivers a piece of data to all other processors in its interval. In reporting, all processors within each interval have data from $B$, and $\otimes$ is applied to these items, with the result ending up at the leader. Often broadcast and report involve only a single interval. Some computer architects have proposed special hardware to implement "op-and-broadcast," which is our broadcast with a single interval and "op" equal to $\otimes$.

5. *Concurrent Read/Write:* In concurrent read and concurrent write we assume that there are master records indexed by unique keys. In the concurrent read each processor specifies a key and ends up with the data in the master record indexed by that key, if such a record exists, or else a flag indicating that there is no such record. In the concurrent write each processor specifies a key and a value from $B$, and each master record is updated by applying $\otimes$ to all values sent to it. (Master records are generated for all keys written). These concurrent read and concurrent write operations are extensions of the operations of concurrent read and concurrent operations normally associated with parallel random access machines (PRAMs). They model a PRAM with associative memory and a powerful combining operation for concurrent writes. On most distributed memory machines the time to perform these more powerful operations is within a multiplicative constant of the time needed to simulate the usual concurrent read and concurrent write, and the use of the more powerful operations can result in significant algorithmic simplifications and speedups.

6. *Compression:* Compression moves data into a region of the machine where optimal interprocessor communication is possible. For example, compressing $k$ items in a square mesh will move them to a $\sqrt{k} \times \sqrt{k}$ subsquare, while compressing them in a mesh-of-trees with at least $k^2$ base processors moves them to the diagonal of a $k \times k$ subsquare.

7. *Searching:* Given a set of $n$ processors, suppose every processor $P_i$ contains *searching item* $s_i \in S$ and *target item* $t_i \in T$. Further, suppose there exists a Boolean relation $R(s,t)$, $s \in S$, $t \in T$. The *searching operation* requires each processor $P_i$ to find the largest $t_j$ such that $R(s_i, t_j)$ is true. This really should be viewed as a class of data movement operations since for any machine there are significant differences in the times searching takes, based on the properties of $R$. For our purposes we can make the strong assumption that the items and $R$ are such that $R(s,t)$ is monotone in each variable, and that $S$ and $T$ are stored in sorted order. In this case the searching operation can be accomplished through merging and broadcasting within intervals (see [7]).

8. *Parallel Prefix:* If processor $P_i$ initially contains value $a_i$ from $B$, then the *parallel prefix* computation results in $P_i$ containing $a_1 \otimes a_2 \otimes \cdots \otimes a_i$. In [2] this operation is called a *scan*. Note that the hardware feature known as "fetch-and-op" implements a variant of parallel prefix, where "op" is $\otimes$ and the ordering of the processors is not required to be deterministic.

9. *Reducing a Function:* Given sets $Q$ and $R$, let $g$ be a function mapping $Q \times R$ into $B$. The map $f$ from $Q$ into $B$ defined by $f(q) = \otimes\{g(q,r) \mid r \in R\}$ is the *reduction of* $g$, and in the reducing a function operation each processor starting with an element $q$ of $Q$ ends up with $f(q)$. For example, if $Q$ and $R$ are sets of planar points, $g$ is distance, $B$ is the reals, and $\otimes$ is minimum, then $f(q)$ is the minimum distance from $q$ to any point in $R$.

The reader might note that several of these operations can be easily obtained from others, sometimes as special cases. However, each of these has proven useful, and sometimes the special cases can be implemented significantly faster than the general operation.

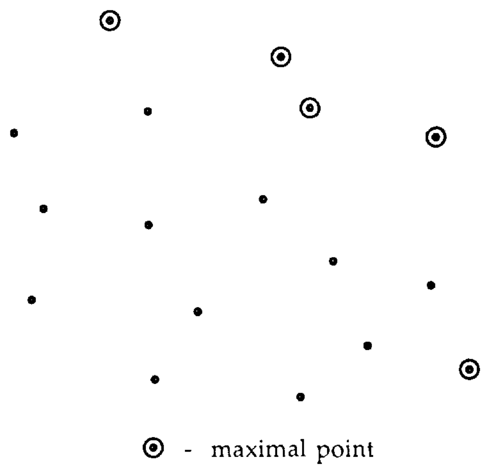


Figure 1. Maximal Points

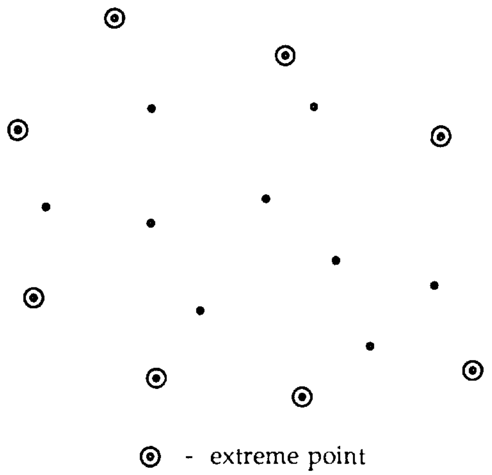


Figure 2. Extreme Points

## 3 Sample Algorithms

Our illustrative algorithms involve finding special points from a collection of planar points. Given a finite set $S$ of planar points, a point $p = (p_x, p_y)$ in $S$ is a *maximal point* of $S$ if $p_x > q_x$ or $p_y > q_y$ for every point $q \neq p$ in $S$. The *maximal point problem* is to determine all maximal points of a given set. See Figure 1. A point $p \in S$ is an *extreme point* of $S$ if it is not in the convex hull of $S - \{p\}$, or, equivalently, if it is a corner (vertex) of the smallest convex polygon containing $S$. The *extreme point problem* is to determine all extreme points of a given set. See Figure 2. Readers interested in serial algorithms for these problems, and in the numerous applications of maximal points and extreme points, might consult [9].

In the following algorithms, $n$ will denote the number of points. To simplify discussion, we will assume that the number of processors is also $n$. Extensions to cases where there are a few points per processor, rather than a single point per processor, are quite straightforward. In particular, we note that Thinking Machine's Connection Machine can be programmed using more virtual processors than real processors, and one is encouraged to write algorithms assuming a single point per virtual processor.

### 3.1 Maximal Points

Our first sample algorithm determines all maximal points, and was apparently first noted by Atallah and Goodrich [1].

**Maximal Point Algorithm**

1. Sort the $n$ planar points so as to order them in reverse order by $x$-coordinate, with ties broken by reverse order by $y$-coordinate. That is, after sorting the points, they will be ordered so that if $i < j$ then either the $x$-coordinate of the point in processor $P_i$ is greater than the $x$-coordinate of the point in processor $P_j$, or else the $x$-coordinates are the same and the $y$-coordinate of the point in processor $P_i$ is greater than the $y$-coordinate of the point in processor $P_j$. Let $(x_i, y_i)$ denote the coordinates of the point ending up in processor $P_i$.
2. Use parallel prefix, with $\otimes$ representing maximum and $y_i$ as the data, to have each processor determine the largest $y$-coordinate stored in any processor of smaller index. Let $L_i$ denote the value determined by processor $P_i$.
3. The point $(x_i, y_i)$ is an extreme point if and only if $y_i > L_i$.

The running time of this algorithm, $T(n)$, is given by

$$T(n) = Sort(n) + Prefix(n) + O(1) ,$$

where $Sort(n)$ is the time to sort $n$ items and $Prefix(n)$ is the time to perform parallel prefix. On all massively parallel architectures known to the authors, $Prefix(n) = O(Sort(n))$, and hence on such machines the time of the algorithm is $\Theta(Sort(n))$. Further, it is known that, at least for serial algorithms, determining maximal points is as hard as sorting [4]. Thus it appears that this portable algorithm is within a multiplicative factor of being optimal for all known massively parallel architectures.

### 3.2 Extreme Points

The following algorithm is based on the well-known tactic of using divide-and-conquer. To simplify exposition we assume that no two points have the same $x$-coordinate. This assumption can easily be removed by including a few extra special cases in the algorithm.

**Extreme Point Algorithm**

1. *Preprocessing:* Sort the $n$ planar points of the set $S$ so as to order them by $x$-coordinate.
2. If $n \leq 2$ then all points are extreme points. Otherwise, note that if $S_1$ denotes the points in processors $0 \ldots (n/2) - 1$, and $S_2$ denotes the points in processors $(n/2) \ldots (n-1)$, then all points in $S_1$ have $x$-coordinates less than those of $S_2$. (We assume that processors $0 \ldots (n/2) - 1$, and processors $(n/2) \ldots (n-1)$, form subsystems similar to the original machine. For example, in a hypercube we want the subsystems to be subcubes. On machines such as two-

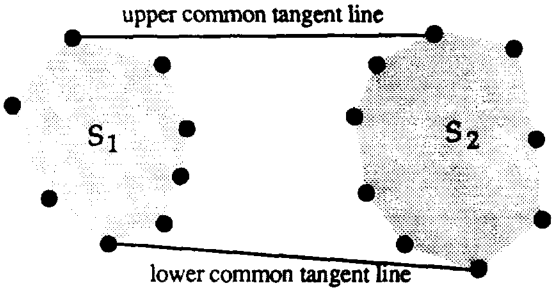


Figure 3. Common Tangent Lines

dimensional meshes or pyramids, one would subdivide into 4 pieces to achieve the proper subsystems.)

3. Recursively identify the extreme points of $S_1$ and the extreme points of $S_2$, enumerating them in counterclockwise fashion. This is a recursive call to step 2, not to step 1.

4. Identify the upper and lower common tangent lines between the extreme points of $S_1$ and the exteme points of $S_2$ by performing a searching operation. See Figure 3. This operation is performed by comparing the slopes of hull edges. Specifically, suppose $\overline{p_iq_j}$, $p_i \in S_1$, $q_j \in S_2$, is the upper tangent line between convex sets $S_1$ and $S_2$, as in Figure 3. Then it can be shown [8] that the slope of $\overline{p_iq_j}$ is between

   (a) the slope of $\overline{p_ip_{i-1}}$ and the slope of $\overline{p_{i+1}p_i}$, and

   (b) the slope of $\overline{q_{j-1}q_j}$ and the slope of $\overline{q_jq_{j+1}}$.

   Therefore, each extreme point simply needs to find the edges of the other set with slopes just above and just below the slopes of the edges it is incident on. Since the extreme points are kept in sorted order, this can be accomplished by merging with respect to the slopes of the edges and then performing broadcasts within intervals.

5. Eliminate all extreme points between the common tangent lines (i.e., all extreme points of $S_1$ and $S_2$ that are inside the quadrilateral formed by the four endpoints representing the common tangent lines) and renumber the remaining extreme points. This is accomplished by broadcasting the information pertaining to the four endpoints to all processors maintaining a point of $S$, and then having each processor make a constant time decision as to whether or not it remains an extreme point, and if so, what its new number is.

The running time of the algorithm is given by

$$T(n) = T'(n) + Sort(n) \ ,$$

where $T'(n)$ is the time to perform all but the first step. $T'(n)$ satisfies the recurrence

$$T'(n) = T'(n/2) + Search(n) + Broad(n) + Elim(1) \ ,$$

where $T'(n/2)$ is the time for the recursive call, $Search(n)$ is the time to perform the grouping operation to determine the upper and lower common tangent lines, $Broad(n)$ is the time to perform a broadcast operation on a machine of size $n$, and $Elim(1)$ is the time required for each processor to make the final extreme point decision.

On a $d$-dimensional mesh or a $d$-dimensional pyramid, this gives a total running time of $\Theta(n^{1/d})$, which is easily seen to be optimal. On a hypercube the running time is $\Theta(\log^2 n)$, since the time for $T'(n)$ is $\Theta(\log^2 n)$ and sorting can be completed in the same time by using bitonic sort. It is not known if this is worst-case optimal, since it is an open question as to whether a hypercube can sort in $o(\log^2 n)$ worst-case time. While we do not have space to explain the details, we note that one can modify the above algorithm so that it subdivides the original set into $n^c$ pieces at each stage [6], with $0 < c < 1$. On the hypercube the modified algorithm achieves $T' = \Theta(\log n)$, which gives a total worst-case running time of $\Theta(Sort(n))$. This modified version also runs in $\Theta(\log n)$ time on an EREW PRAM.

## 4 Final Remarks

Data movement operations should be thought of as the parallel computing analogue of data structures in serial computers. Both provide higher level constructs which help programmers organize their thoughts and programs, and both allow programmers to reuse carefully optimized implementations. Initial users of parallel computers were often willing to spend considerable programming time to achieve the performance available through parallel processing, but as parallel computers move from research into practice there will be resistance to significant reprogramming for each new massively parallel architecture. Systematic use of data movement operations seems to provide a means of achieving high performance on future architectures without unending reprogramming.

## References

[1] M. Atallah and M. Goodrich, "Efficient plane sweeping in parallel", *ACM Symp. Comp. Geo.*, 1986, pp. 216-225.

[2] G. Blelloch, "Scans as primitive parallel operations", *Proc. 1987 Int'l. Conf. Parallel Proc.*, pp. 355-362.

[3] C. Kruskal, L. Rudolph, and M. Snir, "The power of parallel prefix", *Proc. 1985 Int'l. Conf. Parallel Proc.*, pp. 180-185.

[4] H.T. Kung, F. Luccio and F.P. Preparata, "On finding the maxima of a set of vectors", *JACM* **22**, pp. 469-476.

[5] R. Miller and Q.F. Stout, "Data movement techniques for the pyramid computer", *SIAM J. Computing* **16**, pp. 38-60.

[6] R. Miller and Q.F. Stout, "Efficient parallel convex hull algorithms", *IEEE Trans. Computers* **37**, 1988, to appear.

[7] R. Miller and Q.F. Stout, *Parallel Algorithms for Regular Architectures*, The MIT Press, 1988.

[8] F.P. Preparata and S.J. Hong, "Convex hulls of finite sets of points in two and three dimensions", *Comm.ACM* **2**, pp. 87-93.

[9] F.P. Preparata and M.I. Shamos, *Computational Geometry*, Springer-Verlag, 1985.

# ALGORITHMS FOR LONG FAST FOURIER TRANSFORMS ON A CONNECTION MACHINE

J. P. Norris, P. Hertz, K. S. Wood
E. O. Hulburt Center For Space Research
Naval Research Laboratory, Washington, DC

P. Anderson
Planning Research Corporation, McClean, Virginia

## ABSTRACT

We have developed several fast algorithms for very long 1-D and 2-D fast Fourier transforms (FFT) on a Connection Machine CM-2. These codes maximally exploit the parallelism afforded by single-instruction multiple-data (SIMD) machines like the CM-2. We take explicit advantage of the CM-2 hardware (32-bit FPA, 8k byte memory per processor, programmable hypercube connectivity) and software (Paris – machine level language) to simulate the butterfly connectivity required for the FFT computation. A CPU benchmark of 4.6 s for a complex $2^{21}$-point FFT has been established and an execution time of about 9.5 s is predicted for $2^{22}$-point FFTs on a 16k processor CM-2. Longer FFTs, of length at least $2^{24}$ points, are possible on a full 64k processor CM with no increase in CPU time. We also describe algorithms for use on Connection Machines for achieving coherence recovery of frequency-broadened signals.

Keywords: Fast Fourier Transforms, Connection Machine, Signal Coherence Recovery

## INTRODUCTION

The two most attractive features of a Connection Machine are its very high degree of parallelism, 64k simultaneous physical processors (pp) on a full CM, and its capability for programmable connectivity between these processors. However, with bit serial processors and a relatively modest clock speed, the current generation of CMs would appear to perform rather slowly on classes of problems where solutions require a large ratio of inter- to intra- processor operations. When the memory associated with each pp is apportioned to virtual processors (vp's) in order to tackle problems of larger dimensionality or scale, the pp performs as a multiplexer, serving the vp's, and the execution time begins to scale with problem size as in an ordinary von Neuman machine. The decrease in performance is worse if communication between vp's dominates the design of the program. For long FFTs (length > pp number) this is the case unless interprocessor communication time can be significantly reduced.

We discuss two kinds of algorithms for long FFTs which address the problem of communications overhead and which we have successfully run on a CM-2 configured with 16k processors (Ref. 1). Our first approach attempted to minimize inter-(physical) processor communication by transforming a long 1-D FFT into short 2-D FFTs each of which fit inside one pp's memory (Optical Analog FFT). The ability to configure processor connectivity and upgrades which commissioned the new CM-2, including more efficient communication between vp's and a 32-bit floating point accelerator, led to the development of a more efficient and elegant algorithm, the Hypercube Connectivity FFT.

## FFT WITH HYPERCUBE CONNECTIVITY

The central concept of a hypercube FFT on a CM-2 is straightforward: at each stage of the butterfly pattern, arithmetic operations required between two associated points (vp's) are performed in one time step along an axis of a hypercube of rank **m** with side length 2. This optimum communication pattern is realized on a CM-2 by instructing the machine to configure its communication grid (NEWS) into a hypercube geometry. The CM hardware configuration is constructed as two nested hypercubes each with side length equal 2; hence radix 2 FFTs are most natural. Nevertheless, higher and mixed radix FFTs can be performed on a CM-2 by defining "hyper-rectangles" with longer and unequal sides (the execution speed may then be slightly less than optimal since the prescribed geometry is not homologous to the hardware).

The bit reversal operation is also performed using the NEWS grid with the specified hypercube geometry. The algorithm is similar to a Danielson-Lanczos FFT in that the trigonometric factors are computed for each point in parallel rather than propagating phase offset factors point-to-point as in a Cooley-Tukey FFT.

A documented version of the Paris code for the 1-D hypercube FFT is presented in the Appendix. Memory overhead per pp has been optimized for the case of 64-bit complex data; 32 bytes/processor (if radix 2 is used) are required to represent the data (during processing), auxiliary variables, and buffers. Thus a quarter CM-2 is limited to $2^{14}$ pp x (8k bytes/pp) / (32 bytes/point) or $2^{22}$-point FFTs. A factor of two in FFT length may be gained by processing the real and imaginary vectors in series rather than in parallel. An integer representation affords no gain in FFT length over floating point because of increased demands which would be made on memory by additional auxiliary variables.

CH2649-2/89/0000/0199$01.00 © 1988 IEEE

If it is necessary to perform several FFTs on the same data interval (performing an additional transformation before the FFT - see COHERENCE RECOVERY ALGORITHMS below), it is desirable to maintain a copy of the original data in CM memory and thus eliminate costly I/O calls. In the case of sparse data (e.g., byte-sized) this additional demand on memory does not affect the size of a radix 2 FFT which may be performed.

On a full 64k processor CM-2, for FFTs with lengths exceeding a vp-to-pp ratio of one, the factor of 4 increase in CPU power means that FFTs 4 times as long (up to $2^{24}$ points, perhaps $2^{25}$, with current CM-2 memory) can be performed with no increase in execution time. This is illustrated in the comparison in Table 1 where the benchmarks for 8k and 16k processor runs are listed.

**Hypercube Geometry for N-D FFT** - FFTs of higher dimension may be accommodated by specifying hyperplanes and changing the axes for subsequent dimensions, thus configuring a CM for N-dimensional FFTs. A communication efficiency similar to the 1-D FFT case may be obtained.

## OPTICAL ANALOG FFT

As is well known from optical Fourier analysis (Ref. 2) a long 1-D FFT can be recast into (1) a set of short row FFTs, (2) multiplication by phase factors, row and column dependent, and (3) a set of short column FFTs. For a discrete time series f(n) where n = 0, 1, 2 ··· N-1, the Fourier transform is

$$F(k) = \sum_{n} f(n)\exp\{i2\pi nk/N\} \quad k = 0, 1, 2 \cdots N\text{-}1 \qquad (1)$$

Using auxiliary variables, the 1-D Fourier transform, F(k), may be expressed in 2-D format as

$$F(k_x,k_y) = \sum_{n_x} [ \sum_{n_y} f \exp\{i2\pi n_y k_y/M\}] \cdot\exp\{i2\pi n_x k_y/N\}\exp\{i2\pi n_x k_x/L\} \qquad (2)$$

$$n = Ln_y + n_x \qquad n_y = 0, 1, 2 \cdots M\text{-}1$$
$$n_x = 0, 1, 2 \cdots L\text{-}1$$
$$k = Mk_x + k_y \qquad k_y = 0, 1, 2 \cdots M\text{-}1$$
$$k_x = 0, 1, 2 \cdots L\text{-}1$$

The salient feature with regard to SIMD machines like a CM-1 is that short FFTs can be done inside pp's, whereas long FFTs require extensive communication between a much larger number of vp's. The optical analog algorithm performs the row FFTs within pp's, transposes the square array (L = M) containing the intermediate result in order to place each column within a processor, applies the phase multiplication factors and then performs the column FFTs. A final transpose is necessary if the frequency components need to be ordered correctly. The transposes require inter-processor communication; thus for most efficient execution a connectivity optimized for this purpose should be specified. Current CM-2 memory limits square FFTs to length $512^2$. In principle one could string 4 pp's together (at the cost of additional inter-processor communication) to form longer rows and columns, and thus reach $2^{22}$-point FFTs.

## COHERENCE RECOVERY ALGORITHMS

Briefly, the problem of coherence recovery (CR) involves searching for an intrinsically weak temporal signal which is frequency broadened (and hence the Fourier spectral peak decreased), for example, by virtue of motion of the source. In such problems several long FFTs must be performed on the same time series, with an accompanying transformation either in the time or frequency domain, in order to achieve optimum CR, i.e., approximate coalescence of the smeared power into one spectral bin. CR is preferred over incoherent summation of the smeared signal both for enhancing the chance of detection and improving confidence of detection.

**Coherence Recovery in the Time Domain** - Applying a family of quadratic time transformations to effect CR of a frequency-broadened signal is described in detail in Ref. 3. A time series is rebinned for $N_{trans}$ trials according to the formula

$$t' = t + \alpha t^2 , \quad \delta_\alpha = P_{crit}/(2T^2) \qquad (3)$$

where $P_{crit}$ is usually the Nyquist period $(1/\nu_{Nyq})$ and T is the integration time. This algorithm is shown (Ref. 3) to be the optimal one-parameter transformation for CR of an unknown modulating function for sufficiently short integration times (e.g., for orbital motion, $T \lesssim 4\pi P_{orb}$, where $P_{orb}$ is the orbital period). A copy of the original data may be maintained in memory and transformed according to Equation (3). The new time array acts as a pointer to the vp where the data will be sent along the communications grid. This realization of CR applied in the temporal domain requires a time transformation and an FFT for each trial.

**Coherence Recovery in the Frequency Domain** - Alternatively, CR may be achieved by processing in the frequency domain, in which case only one FFT of the data is required. This method is similar to an analog CR technique in which a modified Fourier kernel, containing a quadratic phase, is utilized in an acousto-optic Fourier processor (Ref. 4). The algorithm implemented digitally is known as a Weiner filter, and requires building a table array,

$$N(\nu_s,m) = \sum_{n} N(\nu_n)\ M^*(\nu_n,\nu_s,m) \qquad (4)$$

where $m = \alpha N_o^2$, $N(\nu_n)$ is the FFT of the input data and $M^*$ is the FFT of a quadratic chirp signal,

$$M^*(\nu_n) = \sum_{j} A\ \exp\{i2\pi\nu_s t(1+\alpha t)\}\exp\{-i2\pi jn/N_o\} \ . \qquad (5)$$

For each CR trial, a subarrary of the product of the table and the FFT of the input is summed over a varying range of non-negligible frequency contributions to yield the result. Because the time to create the table array is approximately equal to the time required to process all of the required time transformations and FFTs as described above, a computational advantage is realized only if more than one input data set is processed using the same CR parameters − the table is then computed just once. A disadvantage is that for very long FFTs the entire table may not fit into CM memory, in which case the relevant subarray must be read from external data storage for each CR trial.

## ACKNOWLEDGEMENTS

We appreciate the advice of the Connection Machine staff at NRL. Especially, we thank Robert Whaley for programming assistance on numerous occasions. This work was sponsored by the Office of Naval Research.

## REFERENCES

1. Hertz, P., and J.P. Norris, "Fast Fourier Transforms on the NRL Connection Machine," 1988, NRL Report, in preparation.

2. Turpin, T.M., "Spectrum Analysis Using Optical Processing," 1981, **Proc. IEEE, 69**, pp. 79-92.

3. Norris, J.P., and K.S. Wood, "Discovery of 5 Hz Quasi-Periodic Oscillations in Cygnus X-2," 1987, **Astrophys. J., 312**, pp. 732-738.

4. Norris, J.P., K.S. Wood, and H.W. Smathers, "Signal Coherence Recovery Algorithm for Acousto-optic Fourier Transform Architectures with High Bandwidth," 1988, **Proc. SPIE, 936**, pp. 213-220.

TABLE 1

CM-2 Execution Time in Seconds
$2^m$-point Complex (Hypercube Connectivity) FFT

| m | (no patch code)* | | (patch code)$ | |
|---|---|---|---|---|
| | 8k | 16k | 8k | 16k |
| 13 | .018 | | .026 | |
| 14 | .039 | .019 | .052 | .029 |
| 15 | .108 | .048 | .106 | .056 |
| 16 | .289 | .118 | .230 | .121 |
| 17 | | | .479 | .248 |
| 18 | | | .997 | .513 |
| 19 | | | 2.081 | 1.068 |
| 20 | | | 4.336 | 2.220 |
| 21 | | | ˜9.1 | 4.615 |
| 22 | | | | ˜9.5 |

*Provisional times without optimized bit reversal.

$Patch required to circumvent maximum dimension (16) hypercube currently possible on CM-2 (maximum dimension of 32 is expected).

## APPENDIX

The following code for the Hypercube FFT is written in the C/Paris language and operates under Paris version 5.0B.

```
/* HYPERCUBE-FFT: Implements a 1 dimensional complex FFT.  The real and
 *       imaginary parts are presented separately in two contiguous member
 *       floats.  The length of the data array (which is equal to the number
 *       of virtual processors) is n, and n is 2^p.  */
#include <cm/paris.h>
void hypercube-fft (real_part, imag_part, p, dir)
     unsigned real_part, imag_part, p;  int dir;
/* real_part length 64, imag_part subfield length 32 */
{ static unsigned last_value_of_p = 0;
  static CM_geometry_id_t hypercube_geometry, default_geometry;
  static unsigned dimension_array[32] = {
    2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2 };
  unsigned n, p1, p2, p_phys, axis, spacing;
  unsigned name, coordinate, real_assoc, imag_assoc, temp, trig_temp;
  unsigned i, j, i_coord, j_coord, news_coord_i, news_coord_j, active_flag;
  float pi_signed, theta_0, scale_factor;
  n = 1 << p;
  if (dir < 0) pi_signed = -M_PI; else pi_signed = M_PI;
  /* store current geometry */
  default_geometry = CM_vp_set_geometry(CM_current_vp_set);
  /* skip defining geometry if p = last_value_of_p */
  if (p != last_value_of_p) { last_value_of_p = p;
  /* create first geometry to be 2^p hypercube using NEWS order by default */
    hypercube_geometry = CM_create_geometry(dimension_array, p);  }
  /* select vp set and make appropriate processors active */
CM_set_vp_set_geometry(CM_current_vp_set, hypercube_geometry);
CM_set_context();
/* bit reversal using hypercube */
temp = CM_allocate_stack_field(64);
i_coord = CM_allocate_stack_field(1);
j_coord = CM_allocate_stack_field(1);
active_flag = CM_allocate_stack_field(1);
for (i = 0; i < p/2; i++) {
  j = (p-1) - i;
  news_coord_i = (i + p + CM_purely_virtual_address_length) % p;
  news_coord_j = (j + p + CM_purely_virtual_address_length) % p;
  CM_my_news_coordinate_1L(i_coord, news_coord_i, 1);
  CM_my_news_coordinate_1L(j_coord, news_coord_j, 1);
  CM_logeqv_3_1L(active_flag, i_coord, j_coord, 1);
  CM_logand_context(active_flag);
  CM_get_from_news_1L(temp, real_part, news_coord_j, CM_upward, 64);
  CM_send_to_news_1L(real_part, temp, news_coord_i, CM_upward, 64);
  CM_set_context(); }
CM_deallocate_stack_through(temp);
/* allocate space on stack */
p_phys = 1 + p;
name = CM_allocate_stack_field(p_phys);
coordinate = CM_allocate_stack_field(1);
CM_my_send_address_1L(name);
real_assoc = CM_allocate_stack_field(64);
imag_assoc = CM_add_offset_to_field_id(real_assoc, 32);
temp = CM_allocate_stack_field(32);
trig_temp = CM_allocate_stack_field(32);
/* butterfly pattern mapped onto hypercube */
for (axis = 0, spacing = 0; axis < p;
     axis++, spacing = spacing + spacing) {
  /* send data to associate processor */
  CM_get_from_news_always_1L(real_assoc, real_part, axis, CM_upward, 64);
  /* determine secondary processors and swap data with associate data */
  CM_my_news_coordinate_1L(coordinate, axis, 1);
  CM_load_context(coordinate);
  CM_swap_2_1L( real_part, real_assoc, 64);
  CM_f_negate_1_1L(real_assoc, 23, 8);
  CM_f_negate_1_1L(imag_assoc, 23, 8);
  CM_set_context();
  /* calculate theta and store sign in test_flag */
  theta_0 = pi_signed / (1 + spacing);
  CM_logand_constant_3_1L(temp, name, spacing, p_phys);
  CM_f_u_float_2_2L(trig_temp, temp, p_phys, 23, 8);
  CM_f_multiply_const_always_2_1L(trig_temp, theta_0, 23, 8);
  CM_f_le_zero_1L(trig_temp, 23, 8);
  /* calculate cosine(theta) and use as needed */
  CM_f_cos_1_1L(trig_temp, 23, 8);
  CM_f_mult_add_1L(real_part, trig_temp, real_assoc, real_part, 23, 8);
  CM_f_mult_add_1L(imag_part, trig_temp, imag_assoc, imag_part, 23, 8);
  /* calculate sine(theta) from cosine(theta) and sign(theta) */
  CM_f_mult_subfrom_constant_1L(trig_temp, trig_temp, trig_temp, 1.0, 23, 8);
  CM_f_sqrt_1_1L(trig_temp, 23, 8);
  CM_logand_context_with_test();
  CM_f_negate_1_1L(trig_temp, 23, 8);
  CM_set_context();
  /* use sine as needed */
  CM_f_mult_subfrom_1L(real_part, real_part, trig_temp, imag_assoc, 23, 8);
  CM_f_mult_add_1L(imag_part, trig_temp, real_assoc, imag_part, 23, 8);  }
/* renormalize for inverse transform */
if (dir < 0) {
  scale_factor = 1.0 / power_of_two[p];
  CM_f_multiply_const_always_2_1L(real_part, scale_factor, 23, 8);
  CM_f_multiply_const_always_2_1L(imag_part, scale_factor, 23, 8);  }
/* restore geometry, release stack space */
CM_deallocate_stack_through(name);
CM_set_vp_set_geometry(CM_current_vp_set, default_geometry); }
```

# SIMULATING NEURAL NETWORKS USING C*

Mark J. Norton
GE Aerospace, Advanced Technology Laboratories,
Artificial Intelligence Laboratory,
Moorestown, NJ 08057

## ABSTRACT

Simulation of two neural network models are used to illustrate the benefits of using C* as a programming language to create applications which utilize massive, fine-grained parallel processing architectures such as the connection machine. An overview of the C* language is made and general data structures to be used are discussed. Two neural network models are presented and contrasted: a relaxation energy model (Hopfield), and a forward propagation model (Rumelhart and McClelland). A discussion is made of using C* for simulation purposes and advantages of simulating neural networks on large parallel processors.

## INTRODUCTION

The simulation of large neural networks is a memory and computationally intensive task. Fully connected networks require $n^2$ memory locations to store connection weights. Evaluation typically includes a dot product, and some nonlinearity yielding calculations which also growns linearly.

Fortunately, neural network simulation can be structured to take advantage of data parallelism provided by system such as The Connection Machine (®Thinking Machines, Inc.). Using parallel processing techniques, much of the computational load can be distributed increasing system throughput.

The Connection Machine (CM) was initially configured around a Symbolics Lisp Machine as the front end and was programmed in *Lisp, a parallel version of Lisp which utilizes the capabilities of fine-grained parallelism in a SIMD architecture. Initial CM applications were developed in *Lisp or in Lisp calling PARIS (PARallel Instruction Set).

Last year (1987), a new language was introduced by Thinking Machines which capitalizes on the broad knowledge of the C programming language in the Software Engineering profession. This language, C* [1], is a set of extensions to C which supports parallel programming on massive, fine-grained machines. Extensions made are also compatible with the C++ object oriented programming language.

The use of C* to create applications for the Connection Machines is examined here using Neural Network simulation as an example. Neural networks are well-suited to implementation on the CM, and have the advantage of existing on other architectures for comparison [2].

## AN OVERVIEW OF C*

The C* programming language was developed as a set of extensions to the C programming language developed by Bell Laboratories. C has been called an intermediate programming language since it is close the hardware platform while retaining some of the structure of higher-level languages.

Several extensions to C have been developed over the years to augment the basic language. One of these, C++, was used as a model for C*. C++ is an object oriented programming language which provided support for defining generic objects and manipulating instances of them. C* has been designed to be compatible with C++ and borrows several language concepts from it..

The underlying model for the language is an array of processors with data memory each executing the same instruction at a given time. Data organization is the same in each processor's memory. The only thing that differs is the values contained in these data structures. This model allows developers familiar with C to easily picture the parallel operations being performed, since array operations are common in C. This visualization ease is a strong feature of the language.

Four basic extensions were made to support parallelism: poly data types, the domain concept, processor selection, and a few new operations. An effort to stay close to C was made by retaining the existing syntax as much as possible while interpreting them in a parallel domain. For example, assignment works just like it does in C, but also allows parallel assignments to be made. One design goal of C* is to allow compilation of pure C code in C* (this would run on the front end only).

## RELAXATION MODEL

The first neural network model simulated on the Connection Machine was a relaxation model based on the work of J. Hopfield [3]. As shown in figure 1, each of the net nodes are fully connected to all others in the system, although self-connections are suppressed. The output of the jth element is:

$$X_j = f\left\{\sum_{i,j}(W_{ij} * X_i) - T_j\right\} \qquad (1)$$

Where $W_{ij}$ is the connection weight from element i to j, and $X_i$ is the output of element i. $T_j$ is a threshold associated with the element, and f is a non-linear function such as Sign or Sigmoid.

Because this type of network is fully connected, a connection weight matrix must be stored with each network element. The size of this matrix increases as the square of the network size. This weight matrix dominates all other data structures and makes large networks difficult, since the amount of memory per processor element in the CM-2 is limited to 64K bits.

One way around this problem, which was chosen for this simulation, is to represent a single network node as an array of processors in the Connection Machine. Associated with each of these processors is a single connection weight. Inner products are summed over the array of processors to produce a new $X_j$ in the zero$^{th}$ element. C* provides all the numerical manipulations needed to calculate these quantities in an efficient manner.

This technique was used to create a relaxation network of 256 nodes which were fully connected. A variety of graphic pattern recognition problems were run, including compensation for positional, and rotational shifts in images to be recalled.

## FORWARD PROPAGATION MODEL

The Forward Propagation model differs from the Relaxation model by having distinct layers which propagate information forward to a single output representing the recognized pattern. This type of network has been used extensively for Neural Network experimentation by researchers such as Rumelhart and McLelland [4], Hinton [5], Sejnowski [6], etc.

This model has an advantage over the relaxation model in that substantially fewer connections are required. In general, connections grow linearly with the size of the network. A connection weight matrix must be saved in each processor element, but the size of matrix is not as large. Furthermore, excess processors can be used as additional layers, since the forward propagation model does not require a symmetrical network.

In a fully configured CM-2 with 64K processors, 65536 network nodes could be simulated arranged as 64 layers of up to 1024 nodes each assuming 16 bit weight values, as shown in figure 2. If learning is added to the system using a rule such as back propagation, the number of nodes drops to 32768, since two copies of the network are needed to implement the rule.

Figure 3 shows how information flows through the network in a Connection Machine implementation of the Forward Propagation model. At each sample iteration, an input vector, $I_i$ is initialized in the left most column of a two dimensional array of processors. This array can be up to 1024 by 64 in size. Each value is multiplied by a connection weight

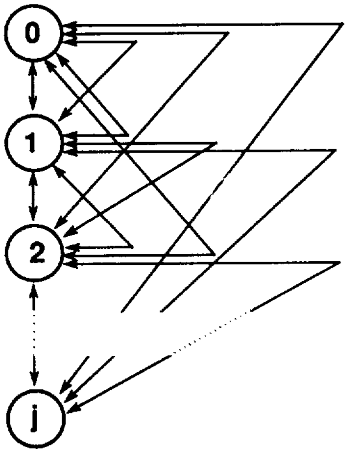


**Figure 1: Relaxation Model**

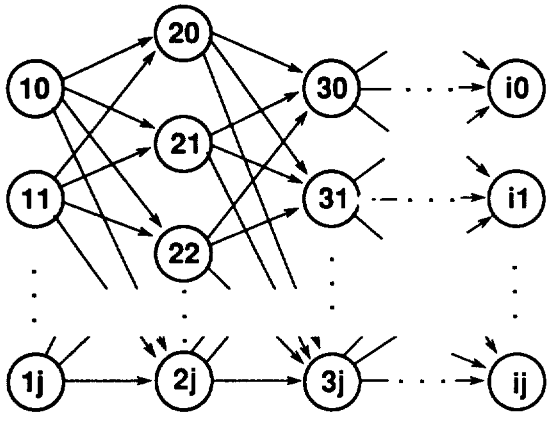


**Figure 2: Forward Propagation Model**

associated with a processor in the next layer, and summed. The sum is then thresholded and passed through a non-linearity as in the Relaxation model. This value is then deposited in the appropriate processor the next layer.

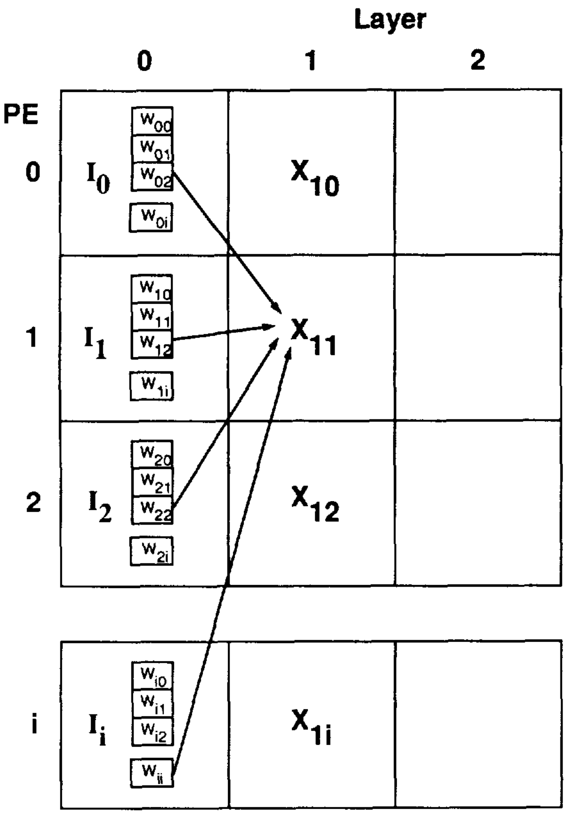


**Figure 3: Data Flow**

## ANALYSIS

The C* language provides all of the capabilities needed to build these simulations. In both cases, values must be communicated between processors to evaluate a new nodal output. Parallel assignment and looping provide the means to accomplish this. Evaluation of the network occurs in parallel for the most part, although looping does introduce some serial operations.

The overriding limit on the size of the network which can be simulated is dependent on the amount of memory available to each processor element. While 64K bits is substantially better than the 1K bits available in the CM-1, it is still confining. This limit also forces data precisions down from an acceptable level of 32 bits to 16 bits (or less). This can limit the learning and performance capabilities of the network.

In general C* provides a powerful, clean syntax and eases the visualization of parallel calculations. The domain concept allows data to be organized and manipulated in a parallel fashion. By extrapolating existing C syntax to parallel behavior, transition from C to C* is a much less painful process than learning the structure of a new language.

## CURRENT AND FUTURE WORK

The Relaxation model described has been implemented and tested on a series of 8 by 8 images. The system is able to recall simple patterns encoded in the network. It has been tested up to 25% noise levels. Networks that correct for translational and rotational shifts have also been tested. The Forward Propagation model exists as a crude prototype. Lack of good test cases has made validation difficult.

The Connection Machine provides a powerful execution environment for neural network simulations. Future work with it will include development of learning rules, such as the Generalized Delta Rule and other neural network models, such as Kosko's Bidirectional Associative Memory and Hecht-Nielsen's Spatio-temporal Pattern Recognition model.

## REFERENCES

(1) Thinking Machines, Corp.; *The C* Reference Manual: Version 4.0A Field Test*, 1987.

(2) Norton, Mark J.; "Three Simulations of a Neural Network," *AAAI Spring Symposium on Parallel Models of Intelligence* , March 1988.

(3) Hopfield, John J.; "Neurons with Graded Response have Collective Computational Properties like those of Two-State Neurons," *Proc. of Natl. Acad. of Sci., Vol 81*, May 1982.

(4) Rumelhart, David E. and McLelland, James L.; "Parallel Distributed Processing," *MIT Press*, 1986.

(5) Hinton, Geoffrey F.; "Connectionist Learning Procedures," *CMU-CS-87-115*, June 1987.

(6) Gorman, R. Paul and Sejnowski, Terrenance J.; "Analysis of Hidden Units in a Layered Network Trained to Classify Sonar Targets," *Neural Network J., Vol. 1, No. 1*, 1988.

# $\mathcal{D}$-Trees: A Class of Dense Regular Interconnection Topologies

B. Ramkumar and L.V. Kalé

Dept. of Computer Science, University Of Illinois at Urbana-Champaign
Urbana, Illinois 61801

## Abstract

In this paper we propose a class of dense regular hierarchical interconnection topologies called $\mathcal{D}$-trees. These topologies are denser than the interconnection networks in use today such as the ring and the n-dimensional boolean hypercube, and compare favourably with other proposed interconnection schemes like the *star graph* and the *pancake graph*. In addition, the class of topologies proposed by us is more flexible in that both the degree and the diameter can be varied in the construction of the required topology. These topologies are also incrementally scalable in the number of nodes that can be connected. We derive expressions for the number of nodes that may be connected in this manner and the corresponding diameters of such topologies. We also compare them with the boolean hypercube and the *star graph*.

**Keywords:** interconnection topologies, dense graphs, regular graphs.

## 1 Introduction

The prospect of solving large problems in parallel using a massive number of processors to attain speedup has always been extremely attractive. Recent advances in device technology have now made this feasible by reducing the cost of individual processors. As a consequence, machines comprising several thousands or processors can be built inexpensively and have the potential to outperform the supercomputers of today both in cost and execution time.

To achieve this, however, it is necessary to be able to connect a large number of processors together in a manner that keeps the diameter of the interconnection topology small, i.e. the topology must be *dense*. In addition, for a uniform treatment of processors, the interconnection problem must be restricted to *regular* graphs, i.e. graphs in which every vertex has the same degree.

The density of a graph G with degree $d$ and diameter $k$ is measured by how close $|G|$ (no. of vertices in G) is to the Moore bound (see later) for the given degree and diameter. If the Moore bound is attained, the graph is called optimally dense. Using this criterion, topologies like the hypercube are not particularly dense. Density is a very desirable property for a topology since the average distance between any two vertices in the topology is small. This means that, in routing, the average time spent by messages at intermediate nodes in a path is small. This also suggests an increase in the computation to communication ratio for the topology. Also, if we define a *neighbourhood* of radius r of a vertex $v$ as the set of all vertices that are $\leq r$ hops away from $v$, it is easy to see that the denser the topology, the larger the neighbourhood (of given radius) of a vertex. This in turn leads to better dynamic load distribution because a large neighbourhood leads to a more uniform distribution of work and hence a better utilisation of processors.

In this paper we propose a class of regular hierarchical graphs which are denser than the interconnection networks in use today, and compare favourably with other interconnection topologies proposed thus far. In addition, they are highly flexible in that they allow both the degree and the diameter of the graph to be varied, and they are incrementally scalable (i.e. allow the number of nodes connected in this manner to be varied). We derive expressions for the number of nodes that can be connected using this class of topologies and their corresponding diameters, and compare it with the n-dimensional Boolean hypercube and the star-graph.

## 2 Related Work

The problem of constructing dense regular graphs of small diameter is well-known, and has been extensively studied. However, computing the maximum number of vertices that can be connected using a regular graph of degree $d$ and diameter $k$ is still an open problem [5, page 213, problems 7,8]. The Moore bound has been proven [9] to be attainable only for degrees 2, 3 (Peterson graph) and 7 (Singleton graph), and possibly 57 since it is not known whether there is a Moore graph of order 57. However, several relatively dense graphs have been found over the years and catalogued by Leland [15], Bermond [4], and others. Bermond has subsequently released updates containing more graphs as technical reports. Uhr also gives a list of known dense topologies in his book [23, pages 136-7]. De Bruijn networks [7] are another well known class of dense topologies that have found several applications in VLSI implementations and communication networks. In 1981, de Bruijn networks were proven to be denser than any other previously discovered graphs by Imase and Itoh [10].

Other (not so dense) graphs have been proposed as interconnection topologies, notably the n-Cube [19], the Cosmic cube [22], the *star graph* and the *pancake graph* [1, 2]. The n-Cube and the Cosmic cube are now available on several commercial machines. However, these machines connect a relatively small number of processors and are designed to support global communication. The *star graph* is an attractive alternative; it is a relatively dense regular topology that is also symmetric. However, the choice of the degree $d$ of each node determines both the diameter and the number of processors $N$ that can be connected. *Star graphs* are also not incrementally scalable in that fewer than N processors cannot be connected for a given $d$. Thus, the price paid for the symmetry in this topology is inflexibility.

The idea of using *trees* to design network topologies is not new either. Arden and Lee [3] proposed a multi-tree network in 1978. They, however, restrict their discussion to graphs with degree 3. Also in 1978, Despain and Patterson [6] proposed an augmented binary tree based architecture called the X-tree. In their paper they make a strong case for tree based architectures. The Sneptree, proposed by Li and Martin [16], is another augmented binary tree network like the X-tree which has been shown to be suitable for VLSI implementations. However, tree-based networks have not been proposed as dense topologies although trees have been known to be among the densest known graphs.

## 3 $\mathcal{D}$-trees -Trees connected with Dense Graphs

In this section, we describe the basic construction of the proposed class of topologies and their properties. We call these graphs $\mathcal{D}_k$-trees. Following that, in section 3.2, orthogonal extensions to the basic construction are proposed; we call these graphs $\mathcal{D}^m$-trees.

The number of nodes that can be connected together with a given degree $d$ and a given diameter $k$ is bounded above by

$$\frac{d(d-1)^k - 2}{d-2} \qquad (d > 2) \qquad (1)$$

This bound is called the Moore bound and can be visualised by considering a tree of height $k$ with its root having $d$ children, all nodes except the leaves having $(d-1)$ children, and all leaves equidistant from the root. The distance from the root to a leaf is clearly the smallest diameter that can be achieved for a $d$-regular graph.

CH2649-2/89/0000/0207$01.00 © 1988 IEEE

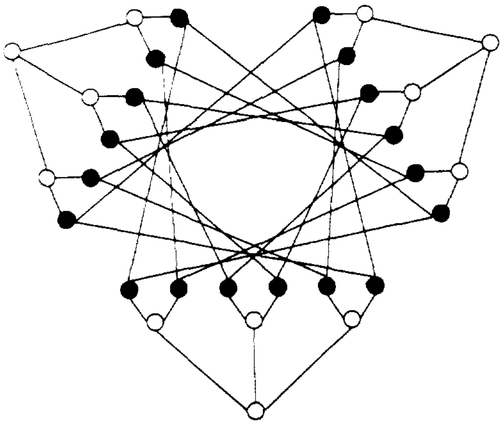


Figure 1: A $\mathcal{D}_1$-tree with degree 3 and diameter 5.

### 3.1 Construction of $\mathcal{D}_k$-trees

A few definitions are necessary to make the constructions easier to follow.

**Definition 1** *A* single-rooted maximal *tree with degree* d *and height* h *is a tree in which the root has* d *children, and all other nodes except the leaves have* (d − 1) *children. All leaves are at distance* h *from the root and have degree 1.*

**Definition 2** *A* two-rooted maximal *tree with degree* d *and height* h *is a construction where two identical trees connected are by an edge at their roots. All nodes except the leaves have* (d − 1) *children; all leaves are at a distance* h *from their respective roots and have degree 1.*

In the construction of $\mathcal{D}_k$-trees, we use *maximal* trees as building blocks to construct dense graphs. These graphs are based on constructions by Korn [14] except that the leaves are not shared in our construction. We construct a $\mathcal{D}_1$-tree with degree d and height h as follows:

**Construction 1** *Take* d *copies of a* maximal *tree T having degree* d *and height* h*. For each tree, label the leaves from left to right as* 1 . . . *L where L is the number of leaves in the* maximal *tree. For each* i*, connect all leaves labelled* i *in the* d *copies of T with a* $K_d$ *graph (the complete graph on* d *vertices).*

This construction produces the same graph as one proposed by Memmi and Raillard in [17] if *single-rooted maximal* trees are used. If *single-rooted maximal* trees are used, this construction will give us $\mathcal{D}$-trees with odd diameter. Graphs with even diameter can be constructed using *two-rooted maximal* trees in Construction 1, but are not as dense. Fig. 1 shows a $\mathcal{D}_1$-tree with degree 3 and diameter 5.

The diameter of the dense regular graph used to connect the *maximal* trees is used to classify the topology. Thus, if a dense graph with diameter $k$ is used, we get a $\mathcal{D}_k$-tree. It is easy to see that every leaf in the *maximal* trees used will have degree $d$ in the $\mathcal{D}_k$-tree. To obtain the densest $\mathcal{D}$-trees possible, we are interested in optimal, or close to optimal, dense regular graphs with small diameter.

It is possible to extend this idea by using regular graphs having diameter 2 to connect *maximal* trees at their leaves to get $\mathcal{D}_2$-trees. However, unlike in Construction 1, there is no known expression for the densest regular graph having degree $(d-1)$ and diameter 2. Therefore, we propose that the densest *known* regular graphs with degree $(d-1)$ and diameter 2 be used to construct $\mathcal{D}_2$-trees having degree $d$. One can construct relatively dense $\mathcal{D}_2$-trees by using, for example, the optimally dense graphs with diameter 2, namely the Peterson graph and the Singleton graph. For $\mathcal{D}_2$-trees, *single-rooted maximal* trees need to be used in the construction to produce graphs with even diameter, whereas *two-rooted maximal* trees yield graphs with odd diameter.

In general, $\mathcal{D}_2$-trees are denser than $\mathcal{D}_1$-trees. A simple example can illustrate this point. If we wish to construct a $\mathcal{D}_1$-tree with degree 8 and diameter 6, we can do so by connecting 8 *two-rooted maximal* trees of height 2 as described in Construction 1. This will give us a graph with 912 nodes (see Theorem 2 for a general expression). However, we can construct a $\mathcal{D}_2$-tree with degree 8 and diameter 6, then by using the Singleton graph, we can connect 50 *single-rooted maximal* trees with degree 8 and height 2. The number of nodes that can be connected in this manner is 3250.

Using the same idea, we can construct $\mathcal{D}_3$-trees, $\mathcal{D}_4$-trees and so on. Singleton graph with diameter 3 are known to exist if $(d-1)$ is a power of a prime number [21]. They have order $\frac{2(d-1)^k-2}{d-2}$ and can be used to construct dense $\mathcal{D}_3$-trees. However, only a small number of such graphs can be connected. De Bruijn graphs are among the densest known graphs for larger values of $k$ and can also be used to construct higher order $\mathcal{D}$-trees. In general, less is known about dense regular graphs with diameter 3, 4 and higher. However, for practical purposes this is not serious, since it can be argued that a very large number of nodes can be connected using $\mathcal{D}_1$-trees and $\mathcal{D}_2$-trees since they permit both the diameter and the degree to be varied.

We now analyse $\mathcal{D}_k$-trees and derive expressions for the diameter and maximum number of nodes that can be connected using this topology. The proofs of the lemmas and theorems that follow are quite simple and can be found in [20].

**Lemma 1** *A $\mathcal{D}_1$-tree of height* h *has diameter* (2h + 1) *if* single-rooted maximal *trees are used in the construction and* (2h + 2) *if* two-rooted maximal *trees are used in the construction.*

Lemma 1 can be generalised for a $\mathcal{D}_k$-tree of height $h$:

**Theorem 1** *A $\mathcal{D}_k$-tree of height* h *has diameter* (2h + *k*) *if* single-rooted maximal *trees are used and and* (2h + *k* + 1) *if* two-rooted maximal *trees are used.*

**Lemma 2** *The maximum number of nodes that can be connected using a $\mathcal{D}_1$-tree topology with degree* d *and height* h *is given by*

$$d\frac{d(d-1)^h-2}{d-2} \qquad (d>2) \qquad (2)$$

*if* single rooted maximal *trees are used in the construction. If* two-rooted maximal *trees are used in the construction of $\mathcal{D}_1$-trees, the expression for the maximum number of nodes is*

$$2d\frac{(d-1)^{h+1}-1}{d-2} \qquad (d>2) \qquad (3)$$

This result too can be generalised for $\mathcal{D}_k$-trees:

**Theorem 2** *Let* p *be the number of nodes in the dense regular graph with degree $(d-1)$ used to connect the* maximal *trees in the $\mathcal{D}$-tree. The maximum number of nodes that can be connected using a $\mathcal{D}_k$-tree topology with degree* d *and height* h *is given by*

$$p\frac{d(d-1)^h-2}{d-2} \qquad (d>2) \qquad (4)$$

*if* single-rooted maximal *trees are used in the construction. If* two-rooted maximal *trees are used, the expression for the maximum number of nodes is*

$$2p\frac{(d-1)^{h+1}-1}{d-2} \qquad (d>2) \qquad (5)$$

The number of nodes for $\mathcal{D}_2$-trees depends upon the dense graph used for the interconnection of the leaves. Since Moore graphs with degrees 2, 3 and 7 are optimally dense, they are the best graphs that can be used for $\mathcal{D}_2$-trees with degrees 3, 4 and 8.

Construction 1 restricts neither the height nor the degree of the *maximal* tree used. It is therefore possible to construct $\mathcal{D}_k$-trees for any given degree and diameter.

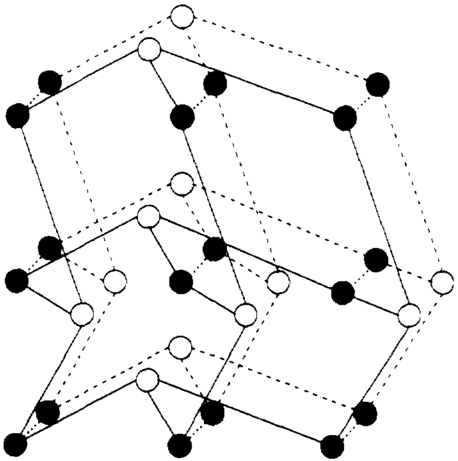


Figure 2: A $\mathcal{D}_1^2$ tree having degree 3 and height 1. Leaf nodes are shown as discs and non-leaf nodes as circles. The two copies of the *tree mesh* are connected together at the leaves.

### 3.2 Construction of $\mathcal{D}^m$-trees

One disadvantage of tree-based topologies in general is traffic congestion that tends to develop around the root(s) of the tree making it a potential bottleneck. This is not a serious problem if a communication within a neighbourhood strategy is used, but for global communication it can prove to be serious. To reduce traffic flowing through the roots, we propose an orthogonal extension to the basic scheme described above. We call these trees $\mathcal{D}^m$-trees. The $\mathcal{D}_k$-trees constructed in section 3.1 can be thought of as $\mathcal{D}_k^1$-trees. Before we describe the $\mathcal{D}^m$-tree topology, we first need to define the *tree mesh* which forms the building block for their construction.

**Definition 3** *Consider a maximal tree with degree* d *and height* h *and let $L$ be the number of leaves in this tree. Lay out $L^2$ "leaves" in a $L$ by $L$ grid. Construct $2L$ maximal trees using each row and each column of the grid as leaves. A construction of this form is defined to be a* tree mesh.

Tree meshes have also been known under different names, for example, *orthogonal trees* in [18].

We now show how $\mathcal{D}^2$-trees can be constructed.

**Construction 2** *Let $D$ be the densest known regular graph with degree* (d − 2), *diameter* k *and order* p. *Take* p *copies of a* tree mesh *having degree* d *and height* h. *Label the leaves on the grid uniquely as $1 \ldots L^2$, where $L$ is the number of leaves in the maximal tree used to construct the* tree mesh. *For each* i, *connect the leaves labelled* i *in the* p *copies of the* tree mesh *using a copy of $D$.*

This construction increases the number of roots in the $\mathcal{D}$-tree at the cost of increasing the diameter. Also, only graphs with either odd or even diameter can be constructed, depending on whether the diameter of graph D in Construction 2 is odd or even. Fig. 2 shows a $\mathcal{D}_1^2$-tree having degree 3 and height 1.

In general, as $m$ increases, the density of the $\mathcal{D}_k^m$- trees decrease. It can be argued that the loss in density is the price that must to be paid to support global communication more efficiently.

In the same manner we can define $\mathcal{D}^3$-trees, $\mathcal{D}^4$-trees and so on. The choice of the $\mathcal{D}^m$-tree will then depend on the level of global communication that needs to be supported. In the limit, the multi-dimensional grid of nodes dominates the construction. Also, with higher order $\mathcal{D}^m$ trees, constructions for all diameters are no longer possible, although the set of diameters for which each $\mathcal{D}^m$ tree can be constructed remains countably infinite.

The construction of $\mathcal{D}^m$-trees results in a rapid increase in the diameter of the topology as $m$ increases. We now derive the expressions for the diameter and the maximum number of nodes that can be connected together using $\mathcal{D}^m$-trees. Once again, the proofs can be found in [20].

**Lemma 3** *The diameter of a $\mathcal{D}_k^2$-tree of height* h *is given by* (4h + k) *if* single-rooted maximal *trees are used in the construction; if* two-rooted maximal *trees are used the diameter is* (4h + k + 2).

This result can be generalised for $\mathcal{D}_k^m$-tree of height $h$:

**Theorem 3** *The diameter of a $\mathcal{D}_k^m$-tree of height* h *is given by* (2mh+k) *if* single-rooted maximal *trees are used in the construction and* ($m$[2h + 1] + k) *if* two-rooted maximal *trees are used.*

**Lemma 4** *Let the order of the dense graph used to connect the* maximal *trees in the $\mathcal{D}$-tree be* p. *The maximum number of nodes that can be connected using a $\mathcal{D}_k^2$-tree with degree* d *and height* h *is given by*

$$p\left\{[d(d-1)^{h-1}]^2 + 2d(d-1)^{h-1}\frac{dd^{h-1}-2}{d-2}\right\} \tag{6}$$

*if* single-rooted maximal *trees are used in the construction. If* two-rooted maximal *trees are used, the maximum number of nodes is given by*

$$p\left\{[2(d-1)^h]^2 + 8(d-1)^h\frac{(d-1)^h-1}{d-2}\right\} \tag{7}$$

The following theorem is a generalisation of this lemma:

**Theorem 4** *Let the order of the dense graph used to connect the leaves of the* maximal *trees in the $\mathcal{D}$-tree be* p. *The maximum number of nodes that can be connected using a $\mathcal{D}_k^m$-tree with degree* d *and height* h *is given by*

$$p\left\{[d(d-1)^{h-1}]^m + (m[d(d-1)^{h-1}]^{m-1})\frac{dd^{h-1}-2}{d-2}\right\} \tag{8}$$

*if* single-rooted maximal *trees are used in the construction. If* two-rooted maximal *trees are used, the maximum number of nodes is given by*

$$p\left\{[2(d-1)^h]^m + (2m[2(d-1)^h]^{m-1})\frac{(d-1)^h-1}{d-2}\right\} \tag{9}$$

The number of roots in $\mathcal{D}_2^m$-trees are greater than the number of roots in $\mathcal{D}_2^1$-trees by a factor of $[m\left\{d(d-1)^{h-1}\right\}^{m-1}]$ (for *single-rooted maximal* trees). This reduces the load per root in the $\mathcal{D}$-tree considerably.

The routing algorithms for $\mathcal{D}_k$-trees and $\mathcal{D}^m$-trees are quite simple. A simple tree traversal algorithm can be adapted to handle "jumps" between multiple copies of *maximal* trees. In the interests of brevity, routing is not discussed in this paper. An outline of simple shortest path routing algorithms for these trees can be found in [20].

### 3.3 Enhancing Incremental Scalability

So far, the constructions described used only *maximal* graphs in the construction of $\mathcal{D}$-trees. However, there is no reason to constrain ourselves to the use of *maximal* graphs. It is possible to use any trees which possess two properties: all non leaf nodes have degree $d$, and all leaf nodes have degree 1. These trees can be used as building blocks for the construction of $\mathcal{D}$-trees instead. These graphs will clearly not be as dense as those constructed in the previous sections, but should there be some constraint requiring this restriction (for example, cost or layout area) they can be used. If $h$ is the height of this "non-maximal" tree, the diameter of these constructions will still be the same as stated in Theorems 1 and 3. However, the average distance between any two nodes in the $\mathcal{D}$-tree may decrease somewhat.

It is also possible to extend the scalability argument to the dense graph used to interconnect the leaves of the trees used in the construction. Any regular graph can be used, for example a ring (although one may fail to see why). From a more practical viewpoint, the densest known regular graph(s), albeit suboptimal, can be used. In Table 1, some of the densest known graphs with diameter 2 have been used to construct the $\mathcal{D}$-trees. The order of these graphs have been stated in parentheses. These values were obtained from [4]. The values marked with an asterisk are those which are provably optimal.

| *d* | *n-Cube* | | *star graph* | | $\mathcal{D}_1^1$*-tree* | | $\mathcal{D}_2^1$*-tree* | | $\mathcal{D}_2^2$*-tree* | |
|---|---|---|---|---|---|---|---|---|---|---|
| | *k* | size | *k* | size | *k* | size | *k* | size | *k* | size |
| 2 | 2 | 4 | 3 | 6 | 3 | 6 | - | - | 4 | 8 |
| 3 | 3 | 8 | 4 | 24 | 5 | 30 | 5 | 30 (5*) | - | - |
| 4 | 4 | 16 | 6 | 120 | 7 | 212 | 6 | 170 (10*) | 6 | 120 (5*) |
| 5 | 5 | 32 | 7 | 720 | 8 | 850 | 8 | 1590 (15*) | 8 | 800 (10*) |
| 6 | 6 | 64 | 9 | 5040 | 9 | 5622 | 9 | 7488 (24*) | 10 | 19800 (15*) |
| 7 | 7 | 128 | 11 | 40320 | 11 | 76202 | 11 | 99520 (32) | 10 | 58464 (24*) |

Table 1: A comparison of some of the topologies. The numbers in parenthesis in columns 9 and 10 give the size of the regular graph used to interconnect the leaves.

## 4 Comparison with Other Topologies

Table 1 tabulates the diameter and the number of nodes that can be interconnected for a few values of the degree for the *n-Cube*, the *star graph* and three different types of $\mathcal{D}$-*trees*. For $\mathcal{D}$-trees, only constructions with comparable diameter are tabulated. As can be seen, $\mathcal{D}$-trees can interconnect a far greater number of nodes than both the *star graph* and the *n-Cube*. In every case, its diameter exceeds that of the *star graph* by no more than one. Asymptotically, the number of nodes that can be connected using a *star graph* with degree *d* and diameter *k* can be shown to have the same order of complexity as a $\mathcal{D}_1^1$-tree with the same degree and diameter. Like the *star-graph*, $\mathcal{D}$-trees are also highly fault tolerant since their connectivity is $(d-1)$ where *d* is the degree of the $\mathcal{D}$-tree.

The $\mathcal{D}$-trees, however, do not have the symmetric and recursively decomposable properties of the *n-Cube* and the star graph. However, the symmetry in these topologies makes them inflexible in that both the diameter and the number of nodes that can be interconnected using these topologies are fixed upon choosing the degree. $\mathcal{D}$-trees provide the freedom to choose both the degree and the diameter of the topology. In addition, $\mathcal{D}$-trees are also incrementally scalable in the number of nodes that can be connected for a given degree and diameter. So, if desired, it is possible to construct sub-optimal topologies as described in the previous section.

## 5 Discussion

In the construction of $\mathcal{D}_k$ trees, the lack of many known dense regular graphs with diameter 2 and greater may appear to be a problem. However, it can be argued that this is not a problem in practice. First, a large number of dense graphs are known for small degrees (i.e. less than 10). Second, it is possible to connect a very large number of nodes with topologies having degree 10 or less, and dense graphs with higher degrees are therefore of less practical interest.

Also, the flexibility offered by the $\mathcal{D}$-tree allows us to constrain ourselves to a known dense configuration, for example, one which uses Singleton graphs to interconnect the leaves of the *maximal* graphs. It is still possible to construct increasing numbers of nodes together with this fixed configuration alone, with a corresponding increase in the diameter. With a degree of 8 and a diameter as small as 12, it is possible to connect over 8 million processors using a $\mathcal{D}_2^1$-tree.

The other issue that needs to be addressed is global communication (or rather, the lack of it). Global communication is counter-productive for massively parallel systems since the cost incurred by every processor for supporting it is very high. We argue that very dense topologies having small diameter are best exploited by restricted communication schemes like *contract-within-neighbourhood* (CWN) [11] proposed by Kalé and the *gradient* model [13] proposed by Lin and Keller. For such schemes, symmetry is not a necessary feature, "local symmetry' i.e. regularity, is sufficient. These schemes are far more effective in keeping the traffic curtailed, thereby reducing the communication overhead on each processor.

## References

[1] S.B. Akers, B. Krishnamurthy, "A group theoretic model for Symmetric Interconnection Networks", *Proceedings of ICPP* 1986.

[2] S.B. Akers, D. Harel, B. Krishnamurthy, "The Star Graph: An Attractive Alternative to the n-Cube", *Proceedings of ICPP* 1987.

[3] B. Arden, H. Lee "A multi-tree structured network", *Proceedings 17th COMPCON*, Sept. 1978, pp.201-210.

[4] J.C. Bermond, C. Delorme, J.J. Quisquater "Tables of large graphs with given degree and diameter" *Info. Proc. Let. 15* , 1982.

[5] B. Bollobas *Extremal Graph Theory*, Academic Press, 1978.

[6] A.M. Despain, D.A. Patterson "X-tree: A Tree Structured Multi-Processor Computer Architecture" *Proceedings of SigARCH* 1978.

[7] N.F. de Bruijn, "A combinatorial problem" *Koninklijke Nederlands, Academe van Wetenschappen, Proceedings Vol. 49 part 20* 1946.

[8] H.D. Friedman, "A design for (d,k) graphs", *IEEE Trans. on Elec. Comp. (Short Notes), Vol. EC-15*, April 1966.

[9] A.J. Hoffman, R.R. Singleton "On Moore Graphs with diameter 2 and 3" *IBM Journal of Research and Development., Vol. 4* 1960.

[10] M. Imase, M. Itoh, "Design to Minimize diameter on Building-Block Network", *IEEE Trans. on Comp. Vol. C-30, No. 6*, June 1981.

[11] L.V. Kale "A comparison of two Dynamic Load Distribution Models" *Proceedings of ICPP* September 1987.

[12] L.V. Kale, W. Shu "The Chare-Kernel Language for Parallel Programming: A Perspective" *Tech. Report no. UIUCDCS-R-88-1451* August 1988.

[13] R. Keller, F.C. Lin "Simulated Performance of a Reduction Based Multiprocessor" *Computer, Vol. 17, No.* 7 July 1984.

[14] I. Korn, "On (d,k) graphs", *IEEE Transactions on Electronic Computers, Vol. EC-16*, Feb. 1967, pp.90.

[15] W.E. Leland *et al*, "High Density graphs for Processor Interconnection", *Information Processing Letters 12*, 1981, pp. 117-120.

[16] P.P. Li, A.J. Martin, "The Sneptree— A versatile interconnection network" *Proceedings of ICPP* 1986, pp. 20-27.

[17] G. Memmi, Y. Raillard "Some new results about the (d,k) graph problem", *IEEE Trans. on Comp., C-31* 1982, pp.784-791.

[18] D. Nath, S.N. Maheshwari and P.C.P. Bhatt, "Efficient VLSI Networks for Parallel Processing based on Orthogonal Trees" *IEEE Trans. on Comp., C-32* 1983.

[19] M.C. Pease "The Indirect Binary n-Cube Microprocessor Array", *IEEE Transactions on Computers*, 1977, pp. 458-473.

[20] B. Ramkumar, L.V. Kale "$\mathcal{D}$-Trees: A Class of Dense Regular Interconnection Topologies" *Tech. Report in prep., UIUC* 1988.

[21] H. Sachs, "On regular graphs with given girth" *Theory of Graphs and its Applications* 1964.

[22] C.L. Seitz, "The Cosmic Cube" *CACM*, 1985.

[23] L.M. Uhr, *Multicomputer Architectures for Artificial Intelligence*, John Wiley and Sons, 1987.

[24] L.D. Wittie, "Communication Structures for Large Networks of Microcomputers" *IEEE Trans. on Comp., C-30, 4*, April 1981.

# A MODEL OF TASK MIGRATION IN PARTITIONABLE PARALLEL PROCESSING SYSTEMS

Thomas Schwederski[1], Howard Jay Siegel[2], and Thomas L. Casavant[3]
Parallel Processing Laboratory, Electrical Engineering School
Purdue University, West Lafayette, IN 47907, USA

## ABSTRACT

Many parallel machines can be partitioned into independent submachines based on rules that depend on the interconnection network used. Here, the effects of task migration (i.e., movement of a task from one partition to another) in systems employing multistage cube or hypercube networks are studied. *Direct overhead* encompasses the cost of moving the task when no other tasks affect the migration, while *indirect overhead* is the additional time needed for migration that is due to influences of other tasks. The sum of direct and indirect overhead is the time needed to migrate a task. In addition, the migration may affect other tasks, and thus can incur a *migration penalty*. The indirect overhead and migration penalty represent the interference that may occur in the interconnection network between the migrating task and other tasks attempting to use the network simultaneously. A model for parameterizing migration costs is given. This research is part of the PASM Parallel Processing Project.

Keywords: task migration, load balancing, partitioning, PASM, multiple-SIMD, MIMD, partitionable SIMD/MIMD, parallel processing.

## 1 INTRODUCTION

Many current commercial and research parallel machines are partitionable; i.e., they can be subdivided into independent machines, and each of these machines can perform a separate task. Examples of such commercial machines are the Intel Hypercube [6], the NCube system [5], the Connection Machine [12], and the BBN Butterfly [1]. Examples of partitionable research machines are RP3 [2] and PASM [11]. In all of these machines, memory is physically divided and a module is connected to each processor (although in some cases the memory may be addressed as one global address space). Also, all of these machines employ cube interconnection networks. The Intel Hypercube, NCube, and Connection Machine systems use single-stage cube (hypercube) networks, while the Butterfly, RP3, and PASM use multistage cube networks. The ways in which single-stage and multistage cube networks can be partitioned into independent subnetworks, creating independent submachines, is described in [10].

Much research has been performed in the area of load balancing and process migration in distributed systems [e.g. 2, 9]. In these studies, a process that executes or awaits execution on single processor is moved to a different processor so that a more balanced load distribution in the system is achieved. However, the area of *task migration* in partitionable systems (i.e., moving a task executing on a group of processors) has received little attention. Here, a model for parameterizing task migration costs is given.

One motivation for task migration is *partition restructuring;* i.e., the movement of small tasks to make larger partitions available. In parallel processing systems with cube interconnection networks, the ways in which processors can be combined into independent submachines obey certain restrictions. As a consequence, tasks that use small partitions can prohibit larger partitions from being formed. Consider a hypercube network with eight nodes as illustrated in Fig. 1. If an independent hypercube network with four nodes is desired, all four nodes must lie on one of the six surfaces of the cube. For example, nodes 1, 3, 5, and 7 can form a partition of size four, because these four nodes can communicate among themselves using a size four hypercube topology without going through any of the other nodes. However, if two small tasks occupy nodes 0 and 7, each of the six surfaces contains one of the small tasks. No size four partition can be formed, even though six nodes are available. That is, without using nodes 0 and 7, no four node hypercube can be formed. By moving one task, e.g., by moving the task in node 0 to node 4, a size four partition becomes possible, consisting of nodes 0, 1, 2, and 3, where these nodes form a size four hypercube. This migration can result in an increase of overall machine performance, or it can be used to meet real-time constraints that apply to incoming tasks.

Fault tolerance is a second motivation for task migration. Assume that node 7 in the hypercube system in Fig. 1 is faulty, and node 0 contains a task. Then no partition of size four is possible. Migrating the task in node 0 to node 1 will make possible the use of nodes 0, 2, 4, and 6 to partition the network to form an independent size four hypercube.

As a third motivation, consider a parallel program that currently occupies a partition, where each processor contains one process. Now assume that each process executes a fork system call, i.e., generates an identical copy of itself in a different processor. In a non-shared memory environment, this implies copying all relevant process information to a new set of processors. If multiple choices for selecting a partition for the newly forked processes exist, the model presented here provides one criterion for making a choice that minimizes the time to copy the required information to the new partition.

A fourth motivation is load balancing. Consider a paral-

[1] Supported by the Air Force Office of Scientific Research under grant number F49620-86-K-0006. Currently with the Institute for Microelectronics Stuttgart, Allmandring 30a, 7000 Stuttgart 80, West Germany.

[2] Part of this work was done while on leave at the Supercomputing Research Center, Lanham, MD, and part at Purdue supported by the above grant.

[3] Supported by the Supercomputing Research Center, Lanham, MD, under contract number 6925.

lel processing system that permits multiprocessing in its processors. For example, if a partition A shares its time between five tasks, while another partition B of the same size has only three tasks, migration of one task from A to B may represent a more favorable distribution of system load.

PASM is a partitionable SIMD/MIMD parallel processing system (where each partition can independently switch between the SIMD and MIMD modes of parallelism at the instruction level), a prototype of which is in operation at Purdue University [11]. Currently, the idea of an Image Understanding System *(IUS)* that automatically configures and reconfigures the system is being studied [4]. The IUS can change the resource allocation to subtasks of a given task, often as a result of data dependent decisions at execution time. This involves dynamic changes in the number of resources (processors) allocated to each subtask (subtask shrinking or expansion). The work here provides a framework for some of the basic knowledge about the PASM system that is required for the IUS. Thus, in general, this research can be used to support task/data dependent run-time system reconfiguration for optimizing resource utilization.

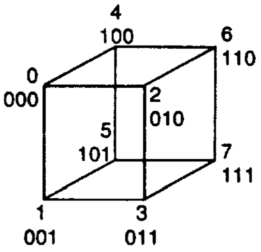


Fig. 1. Three-dimensional cube structure.

The system model that is used to analyze task migration is described in Section 2. Section 3 discusses the parameters of task migration qualitatively. Section 4 examines the task migration parameters in a quantitative way.

## 2 SYSTEM MODEL FOR TASK MIGRATION

The analyses are applicable to multiple-SIMD*, MIMD, and partitionable SIMD/MIMD* parallel machines utilizing hypercube or multistage cube interconnection networks and enforcing partitioning (i.e., each subnetwork of size S must have the properties, including topology, of a hypercube or a multistage cube built to be of size S). The architectures under consideration are physically organized such that each processor is paired with a memory module, forming a Processing Element *(PE)*. Each PE contains programs and data if in MIMD mode, and data only if in SIMD mode. If the task is to be moved to a different partition, all local components must be relocated.

It is assumed that overall system activities are supervised by a ***system controller,*** which can be a dedicated processor (e.g., the System Control Unit in PASM [11]), or a program distributed among the system processors. Among other duties, this system controller is responsible for allocating and deallocating partitions, and for assigning tasks to partitions. The activities in each partition are supervised by a *partition controller*. Similar to the system controller, it could be one or more designated processors (e.g., the Micro Controllers in PASM [11]) which also act as SIMD control units *(CUs),* or it could be a program distributed among the processors of the partition.

In this paper, a ***task*** is the execution of a parallel program on a partition. An *SIMD process* is the execution of an SIMD procedure on a partition. An *MIMD process* is the execution of an MIMD procedure on a PE that is part of the partition. A task can be composed of one or more SIMD and/or MIMD processes. Associated with each process are data, program code, stack, etc. ***Suspension*** of a process implies halting the process execution and saving the state of the process (e.g., values of the CPU registers) such that the process can be resumed as if no interruption had occurred. *Transfer* of a process implies allocation of appropriate resources at the destination, and transfer of all information associated with the process. For example, if a process has b bytes of stack space available but uses only part of it, b bytes of stack must be allocated at the destination, the currently used part of the stack must be moved, and stack pointers must be set appropriately.

## 3 QUALITATIVE VIEW OF MIGRATION

*Direct overhead* encompasses the time to move the task when no other tasks affect the migration, while *indirect overhead* is the additional time needed for migration that is due to influences of other tasks. The sum of direct and indirect overhead is the time needed to migrate a task. In addition, the migration may delay other tasks, and thus can cause a *migration penalty.*

Consider direct overhead. First, the system controller must make a decision whether to migrate a task, and to which destination to migrate [8]. It then instructs the partition controller of the source partition $P_s$ to suspend the currently active task, and allocates the destination partition $P_d$. The partition controller in $P_s$ stops the current task, saves all state information necessary to restart the task at $P_d$ (e.g., CPU registers), and determines all information that must be transmitted. If the partition runs in MIMD mode, only information from the processors and memories in the partition must be moved. If the partition is in SIMD mode, information resident in the CU of $P_s$ must also be moved to the CU of $P_d$. The amount of information to be transferred is dependent on the size of the program, on the size of the data set, and on the number of temporary variables currently in use.

The physical data paths used for data transmission are the interconnection network for the PEs, and an inter-CU communication link (e.g., a shared bus or shared memory) for transfers between CUs. The time to accomplish the data transfer depends on the amount of data to be transmitted, the location of $P_s$ and $P_d$, the tasks influenced by the migration, the type of interconnection network, and system implementation.

*In multiple-SIMD machines, the system can be partitioned into independent SIMD subsystems, each with its own control unit for instruction broadcasting [7]. Similarly, in partitionable SIMD/MIMD machines, each partition must have a control unit available for it for SIMD processing.

After the data transfer has been completed, the system controller can reassign $P_s$ to a new task, usually as part of a larger partition. At $P_d$, the partition controller resumes the migrated task, and the migration process is completed.

Now consider indirect overhead and network *conflicts* (i.e., two PEs wanting to send data over the same network link simultaneously). Each partition of a Cube network (either single-stage or multistage), has a complete subnetwork to itself. In such a subnetwork, conflicts in the subnetwork can only be caused by transfers between the PEs of a partition. Assume $P_s$ and $P_d$ are such that they can be combined to form a single larger partition $P_c$. For example, in a 16-PE system, if $P_s = \{0,4,8,12\}$ and $P_d = \{2,6,10,14\}$, then they can be combined to form a single partition $P_c = \{0,2,4,6,8,10,12,14\}$. Then transfers from any PE in $P_s$ to any PE in $P_d$ are inside $P_c$, and no other partition in the system is affected. However, in general, this combination is not always possible; thus, transfers from PEs in $P_s$ to PEs in $P_d$ can violate partition boundaries. Thus, the transfer of task information from $P_s$ to $P_d$ can cause interconnection network conflicts in other partitions of the system. These conflicts interfere with both the migration of the task and the tasks running in partitions affected by the migration. As an example, consider a 16-PE system, where PEs 0 to 3 form $P_s$, PEs 12 to 15 form $P_d$, and PEs 4 to 7 and 8 to 11 form two more partitions. Then, to transfer information from PEs in $P_s$ to PEs in $P_d$ will require using subnetworks associated with either PEs 4 to 7 and/or PEs 8 to 11.

Assume the migration influences a sequence of partitions $Q_k$, $1 \leqq k \leqq R$. If partition $Q_k$ delays the migration by the indirect overhead time $T_{I,k}$, the total indirect overhead time is given by

$$T_I = \sum_{k=1}^{R} T_{I,k}$$

If partition $Q_k$ utilizes $S_k$ PEs and is delayed by the migration by time $T_{P,k}$, the migration penalty MP is defined as

$$MP = \sum_{k=1}^{R} S_k T_{P,k}$$

MP is a resource-time product. If any influenced partition has a strict time constraint (e.g., real-time process) and the migration would violate the constraint, an infinite migration penalty is incurred ($MP=\infty$), which effectively prohibits the migration from being performed. $T_{I,K}$ and $T_{P,K}$ are functions of the influenced partitions' network usage and the network implementation.

Two migration methods are (a) passing the migrating task through the affected partition without halting the task in the affected partition, and (b) halting all interconnection network use of the task in the affected partition while the migration is performed. If the affected task is running in SIMD mode, only (b) avoids possible erroneous computations. In SIMD mode, PEs send messages into the network, and, generally, in the next step read the messages. Because the PEs are implicitly synchronized via hardware, no check is required whether a message arrived before reading from the network. However, if a migrating task delays some of the messages in the partition, these messages might not have arrived at their destination when a PE tries to read them. Therefore some of the PEs might read old data, and the computations would be incorrect. For MIMD tasks, both (a) and (b) are feasible, and the method resulting in better system performance should be chosen.

## 4 PARAMETERIZATION OF OVERHEAD

Assume that the overhead caused by dattransfers is known, and that a task in $P_s$ is to be transferred to $P_d$. Then the time required to transmit all necessary information from the $k^{th}$ source PE to the $k^{th}$ destination PE is given by $T_{TRPE,k}$ (it is assumed size $P_s$ = size $P_d$).

Assume at t=0, a new task has entered the system, or a need for possible reconfiguration of a task has arisen. The system controller determines which migration is to be performed, if any, using time $T_{DEC}$. Once the decision to migrate has been made, the system controller must allocate a $P_d$, which requires time $T_{PAL}$. It must then instruct the $P_s$ controller to migrate its current task to $P_d$, and instruct the $P_d$ controller to accept the migrated task. This migration initiation takes time $T_{MI}$.

The $P_s$ controller now suspends the task, then the task is transferred and resumed. Because these procedures differ for multiple-SIMD, MIMD, and partitionable SIMD/MIMD machines, the machine types will be discussed separately.

The PEs in a multiple-SIMD machine execute only instructions broadcast to them by a CU. Therefore the instructions that save the PE state (e.g., CPU registers) and move the task must be broadcast by the CU. Thus, the user task in the CU (i.e., the process that broadcasts the user program PE instructions to the PEs) must be stopped first, and then the CU must broadcast instructions to the PEs that save the PE state. The CU also determines which information from both the PEs and CU must be moved in order to migrate the task. The overall time to save the SIMD state of the task is denoted $T_{SSS}$.

Then all necessary information in the CU in $P_s$ must be moved to the CU in $P_d$, and the information in the PEs of $P_s$ must be moved to the PEs of $P_d$. The movement of the CU information takes $T_{TRCU}$. Some machines permit an arbitrary assignment of CUs to PEs (e.g., MAP [7]), so the CU can be reassigned to $P_d$, and $T_{TRCU}$ is zero. Because all PEs execute the same program in SIMD mode, the amount of data to be transferred is the same for all PEs, and thus $T_{TRPE,k}=T_{TRPE}$ $\forall$ k. All SIMD instructions are broadcast to the PEs by the CU. If no overlap between CU and PE operations exists, the CU will be busy broadcasting SIMD instructions for time $T_{TRPE}$, resulting in a maximum task transfer time of $T_{TRCU} + T_{TRPE}$. Some systems, e.g., PASM [11], have the capability to overlap CU and PE operations, and thus a lower bound for the total time to move the CU and PE information is given by $\max(T_{TRCU}, T_{TRPE})$. If the PEs contain a DMA controller, the CU must first initialize the PE DMA controllers (which takes $T_{DMA}$) and can then move its own state (which takes $T_{TRCU}$), while the PEs' DMA controllers move the PE state simultaneously. In this case, the total time to move the PE and CU information is given by $T_{DMA} + \max(T_{TRPE}, T_{TRCU})$.

After the necessary information is moved, the CU in $P_d$

restores the SIMD state of the task, which requires time $T_{RSS}$. Included in this time is the time needed to restore the CU and the PE state.

A worst case estimate of the migration time of a SIMD task in a multiple-SIMD machine is therefore given by

$$T_{SIMD} = T_{DEC} + T_{PAL} + T_{MI} + T_{SSS} + T_{TRCU} + T_{TRPE} + T_{RSS}$$

Assume that in an MIMD partition containing R PEs, each PE executes one MIMD process. Consider the processor that executes process k. After the partition controller informs the PE of the migration, the PE suspends its process (using time $T_{SPE,k}$) and moves all necessary information to the appropriate destination PE (using time $T_{TRPE,k}$). The destination PE resumes the process (using time $T_{RPE,k}$). Therefore the overall time to suspend, transfer, and resume process k is $T_{SPE,k} + T_{TRPE,k} + T_{RPE,k}$. Because all PEs can operate individually, the maximum over all k is the time when all PEs have resumed execution. A migration time in an MIMD machine is therefore given by

$$T_{MIMD} = T_{DEC} + T_{PAL} + T_{MI} + \max_{k=1}^{R}(T_{SPE,k} + T_{TRPE,k} + T_{RPE,k})$$

In a partitionable SIMD/MIMD machine, capable of dynamically switching between MIMD and SIMD modes, three cases have to be distinguished: the task to be migrated is (a) purely MIMD, (b) purely SIMD, and (c) use both modes of parallelism. Case (c) encompasses both (a) and (b), and is discussed first.

The SIMD state may have components in the CU and the PEs, and saving this state requires time $T_{SSS}$. Both PEs and CU will be involved in saving the SIMD state, but they may operate concurrently, and there may be overlap with other migration steps (e.g., saving of CU information might proceed concurrently with saving the PEs' MIMD state). Once the SIMD state has been saved, the CU can transfer its information (using time $T_{TRCU}$), while the PEs suspend their MIMD state, transfer their memory and register contents (SIMD and MIMD), and restore their MIMD state, using time

$$\max_{k=1}^{R}(T_{SPE,k}+T_{TRPE,k}+T_{RPE,k}) .$$

Finally, the SIMD state in PEs and CU can be restored, using time $T_{RSS}$. Therefore an upper bound for the migration time of an SIMD/MIMD task is:

$$T_{SMIMD} = T_{DEC} + T_{PAL} + T_{MI} + T_{SSS} + \max(\max_{k=1}^{R}(T_{SPE,k} + T_{TRPE,k} + T_{RPE,k}), T_{TRCU}) + T_{RSS}$$

In all cases above, $T_I$ is part of $T_{TRPE,k}$. MP is an additional factor that must be considered.

## 5 SUMMARY

One important aspect of system reconfiguration, task migration, was studied. Several motivations for task migration were discussed. The task migration cost was classified into direct and indirect overhead, and migration penalty. The parameters for task migration cost were examined for multiple-SIMD, MIMD, and partitionable SIMD/MIMD machines. The results here can be extended to the cases where size $P_s \neq$ size $P_d$ and where processors are not paired with memories. For these extensions and further information see [8].

**Acknowledgment--** We thank J. Armstrong for his comments and help in preparing the final version of this paper.

## REFERENCES

1. BBN, *Inside the Butterfly Plus,* BBN Inc., 1987.
2. W. C. Brantley, K. P. McAuliffe, J. Weiss,. "RP3 processor-memory element," *1985 Int'l Conf. Parallel Processing,* Aug. 1985, pp. 782-789.
3. T. L. Casavant, J. G. Kuhl, "A taxonomy of scheduling in general-purpose distributed computing systems," *IEEE Trans. Software Engg.,* Vol. SE-14, Feb. 1988, pp. 141-154.
4. E. J. Delp, H. J. Siegel, A. Whinston, L. H. Jamieson, "An intelligent operating system for executing image understanding tasks on a reconfigurable parallel architecture," *IEEE Computer Society Workshop on Computer Arch. for Pattern Analysis and Image Database Management,* Nov. 1985, pp. 217-224.
5. J. P. Hayes, T. N. Mudge, Q. F. Stout, S. Colley, "Architecture of a hypercube supercomputer," *1986 Int'l Conf. Parallel Processing,* Aug. 1986, pp. 653-660.
6. Intel Corp., *A New Direction in Scientific Computing,* Intel Corp., 1985.
7. G. J. Nutt, "Microprocessor implementation of a parallel processor," *4th Ann. Symp. Computer Arch.,* Mar. 1977, pp. 147-152.
8. T. Schwederski, H. J. Siegel, *The PASM Parallel Processing System: Hardware Design and Operating System Concepts,* Purdue Univ. Tech. Rep. in prep.
9. E. Shamir, E. Upfal, "A probabilistic approach to the load-sharing problem in distributed systems," *J. Parallel and Distrbuted Computing,* Vol. 4, Oct. 1987, pp. 521-530.
10. H. J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies,* Lexington Books, D. C. Heath and Co., Lexington, MA, 1985.
11. H. J. Siegel, T. Schwederski, J. T. Kuehn, N. J. Davis IV, "An overview of the PASM parallel processing system," in *Computer Arch.,* D. D. Gajski, V. M. Milutinovic, H. J. Siegel, B. P. Furht, eds., IEEE Computer Society Press, Washington, D.C., 1987, pp. 387-407.
12. L. W. Tucker, G. G. Robertson, "Architecture and application of the Connection Machine," *Computer,* Vol. 21, Aug. 1988, pp. 26-38.

# Sparse Matrix Computations on an FFP Machine* (Preliminary Version)

B.T. Smith, R.K. Singh and G.A. Magó
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175

## Abstract

We describe and analyze an algorithm for performing Gaussian elimination on sparse linear systems with an FFP Machine, a small-grain parallel computer. Given an equation $Ax = b$, where A is an $n \times n$ matrix, our algorithm yields a permuted upper-triangular system, from which we obtain $x$ by back-substitution. If $A$ has $e$ non-zero entries and if $f$ fill-ins are created during elimination, then our algorithm solves the system in $O(h \times (e + f))$ time, using $O(e + f)$ processing elements. (The parameter $h$ is the height of the FFP Machine's connection network, which is $O(\log(e + f))$.) The algorithm makes no assumptions about the structure of $A$ and requires no pre-processing. The pivot order may be given in advance, or it may be chosen at run-time by the Markowitz heuristic with only a linear increase in cost. We also present results of simulations on sample problems, both randomly generated and from the Boeing-Harwell set. The results of the simulations, in operation counts, are used to estimate the performance of an FFP Machine hardware prototype.

## The Problem

Matrix problems are encountered in such disciplines as physics, engineering, econometrics and operations research. Common to many of these problems is the occurrence of matrices that are sparse, i.e., many elements of the matrices are zero.

The matrices associated with problems from physical sciences and engineering, in addition to being large and sparse, are frequently structured. For example, they may be symmetric, diagonally dominant, positive definite or banded. Hence, they lend themselves to an efficient solution by a variety of special techniques.

In contrast, problems in such areas as operations research, nonlinear optimization or management can yield unstructured sparse matrices. As a consequence, more general sparse matrix techniques have been developed for less structured problems [3]. Such techniques are characterized by relatively few operations per data element and an unpredictable, dynamic growth of data structures. These issues, as summarized in [2], necessitate dynamic storage management and efficient data structure handling methods.

In the prevailing paradigm of parallel computation, we address these issues by considering a parallel form of the direct Gaussian elimination method, augmented by the Markowitz heuristic to establish pivot ordering, for the solution of systems of linear equations. The parallel approach we take is based on a computational model of a fine-grain, distributed-memory, network-based MIMD computer called the *FFP Machine* (*FFPM*). In keeping with our interest in less structured problems, we make no assumptions about the structure of the coefficient matrices. We give a brief description of the FFPM architecture in the next section. The algorithm and results of its theoretical and experimental analyses are presented in the following sections.

## FFP Machines

*FFP Machines* are a family of small-grain, parallel computers [7] designed to execute the FFP languages of Backus [1]. An FFPM, as shown in Figure 1, consists of a linear array of PEs, called the *L-array* of *L-cells*, connected to each other and to an interconnection network of *T-cells*. Each L-cell is a small, programmable computer with an ALU and a very small memory. There is also a *front-end machine* that handles I/O, but for the most part FFP execution takes place in the L-cells and T-cells. In a simple FFPM, as shown in Figure 2, the T-cells are organized as a binary tree with an L-cell at each leaf and the front-end machine above the root. A useful FFPM would contain at least a few thousand L-cells.

FFP's primary data structure is the sequence, and the FFPM treats sequences as ***dynamic arrays*** [6]. That is, it is possible to randomly access the elements of a sequence, and at the same time easy to add or delete elements at arbitrary positions. Moreover, elements of a sequence may be accessed either by relative position in the sequence or by content, as in associative memory[5]. Many of these operations correspond to FFP's primitive functions, but an FFPM can support functions not in Backus's original language [8,9]. This paper shows how an FFPM can provide operations on sequences that are well suited to sparse matrix computation. Such operations may be added as new FFP primitive functions to be used in Gaussian elimination and other computations.

An FFP expression is placed in the L-array, each symbol in a different L-cell, and the FFPM evaluates the expression by rewriting innermost function applications, known as *reducible applications* or *RA*s, until no more applications remain. The FFPM operates in machine cycles

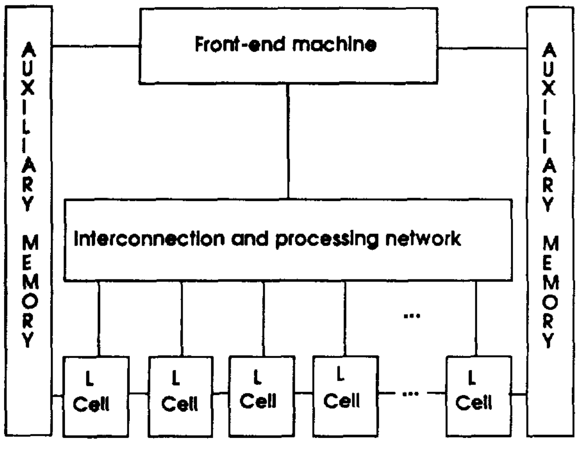


Figure 1: The components of an FFP Machine.

*This work was supported in part by NSF grant MIP-8702277, and by the Office of Naval Research, Contract N00014-86-K-0680.

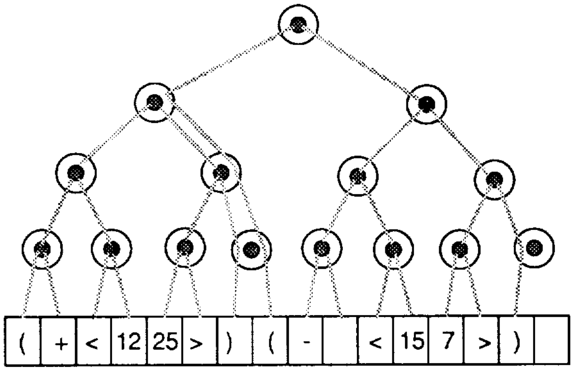


Figure 2: Partitioning creates a submachine for each RA.

of partitioning, execution and storage management. ***Partitioning*** creates an independent sub-machine for each RA, consisting of the L-cells holding the RA and a binary tree of T-cells, as shown in Figure 2. During ***execution***, the sub-machine's L-cells rewrite their RA by performing local computations and exchanging messages. Message packets are sent from the L-cells and contain instructions on how the T-cells shall treat them. The sub-machine's T-cell network can select or sort messages, broadcasting the result to all L-cells in the sub-machine. The T-cell network can also perform parallel prefix operations. A sub-machine might request extra space and suspend execution, as occurs when an FFP expression grows during evaluation. During ***storage management***, the contents of the L-cells, ***the L-cell images***, are shifted through the L-array, retaining their left-to-right order, to make empty L-cells available where needed, as shown in Figure 3. The shifted L-cell images obtain a new T-cell network in the next partitioning stage, and execution continues.

## The Algorithm

The system of equations $Ax = b$ is given row-wise, as an FFP sequence

$$<row_1, \ldots, row_n>$$

and each $row_i$ is of the form

$$<a_{i,j(i,1)}, \ldots, a_{i,j(i,k_i)}, b_i>$$

where the $i^{\text{th}}$ row of $A$ contains $k_i$ nonzero entries in columns $j(i,1)$, ..., $j(i,k_i)$. The entries $b_1$, ..., $b_n$ are from the vector $b$. Each $a_{i,j(i,l)}$ contains the corresponding coefficient in $A$ along with its row and column numbers, and it has space for some additional values used during the computation. (For example, values used in choosing pivot elements by the Markowitz heuristic.) The number of values required per entry is independent of the matrix size, and for this reason, an entry might be an FFP atom and reside in one L-cell. In that case, new primitive FFP functions would be required to operate on its components. Or, it could be a small FFP sequence, manipulated by the standard FFP functions. The choice would influence time and space performance on a particular FFPM, but it has no effect on the algorithm or its analysis.

We will describe the algorithm in three parts. First, and in the greatest detail, we present the basic Gaussian elimination algorithm. Second, we describe how the solution is obtained via back-substitution. Finally, we discuss how to modify the basic algorithm to choose pivots at runtime, using the Markowitz heuristic.

***Gaussian elimination***: Initially all rows are "active". We choose a (non-zero) pivot element in an active row and subtract the appropriate multiple of that row from the other active rows, so that afterwards they have 0s in the pivot element's column. Then we mark the pivot element's row as inactive and repeat the process. When there are no non-zero elements in active rows – after $n$ steps, if $A$ is non-singular – we have the system in (permuted) upper triangular form, and we can find the solution easily by back-substitution.

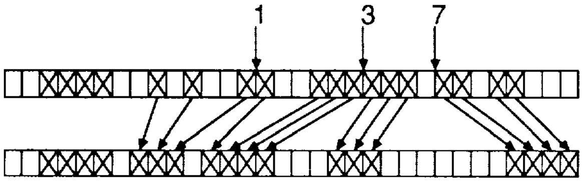


Figure 3: During storage management, L-cell images are shifted to provide empty L-cells where requested.

The algorithm proceeds by two types of operations: ***global operations***, where the whole matrix is contained in one RA, and ***row operations***, where each row is contained in its own RA. The global operations choose the pivot, broadcast the pivot row and update the values for the Markowitz heuristic. The row operations subtract the appropriate multiple of the pivot row, and create fillins. Since Gaussian elimination is so familiar, it suffices to describe in detail the "inner loop", as it applies to one active, non-pivot row.

Suppose we have a 5 x 5 system of equations (neither large nor sparse) that contains the following:

$$\begin{array}{llllll} 3x_1 & & +x_3 & -4x_5 & = 3 & \text{(row 2 = pivot row)} \\ & x_2 & +2x_3 & & = 5 & \text{(row 3)} \end{array}$$

and that $a_{2,3}$ has been chosen as the pivot. We will trace the effect of a single step on the third row. Figure 4 (a) shows row 3 at the beginning of the step. The column numbers of the entries in the pivot row are broadcast to the entire matrix in a global operation, and each active element counts those less than or equal to its own, shown as ***temp*** in Figure 4 (b). Each element also records whether one of these matches its own column number, shown as ***hit***. Next, each entry computes the number of elements in the pivot row between itself and its left neighbor. (This is done separately in each active row, by a parallel prefix computation.) This value, minus 1 if ***hit*** is ***true*** (***temp*** in Figure 4 (c)) gives the number of fillins to be created to the left of the entry. The fillin entries are created after an FFPM storage management cycle. The elements who are going to be "hit" on this step compute their relative order within the row, ***temp*** in Figure 4 (d). (This is another parallel prefix computation, done separately in each active row.) Now, in another global operation, the values in the pivot row are broadcast once again, sorted by column number in the T-cell network, and each entry to be hit receives the column number and pivot row coefficient indicated by its index from the previous step. This is shown in Figure 4 (e), where ***temp*** now is the value from the pivot row. Finally, in another row operation, the element in the pivot column broadcasts the quotient ***value ÷ temp*** and every "hit" entry performs the multiplication and subtraction.

A subtle point of the algorithm is worth mentioning. The newly created fillins do not have column numbers until the pivot row is broadcast the second time. (Shown by "-" in Figure 4 (d).) Before the pivot row is broadcast the first time, there is no way, in general, for another row to know how many fillins it will need. Nor is there any way to know where they will be created. As we remarked earlier, an L-cell is fairly small. It is reasonable to assume that it can store several small integers, but it is not reasonable to assume that it can store arbitrarily many column numbers. So, the reason for the second broadcast of the pivot row is simply that we want to create all fillins for one step of the Gaussian elimination in one FFPM storage management cycle, rather than deciding as each pivot row entry arrives whether it requires a fillin.

We analyze the Gaussian elimination algorithm as follows. An L-cell does a bounded number of arithmetic operations for each value

(a)

| | | | |
|---|---|---|---|
| *value* | 1 | 2 | 5 |
| *row* | 3 | 3 | 3 |
| *col* | 2 | 3 | b |
| *active* | T | T | T |
| *p-row* | F | F | F |
| *p-col* | F | T | F |
| *hit* | - | - | - |
| *temp* | - | - | - |

(b)

| | | | |
|---|---|---|---|
| *value* | 1 | 2 | 5 |
| *row* | 3 | 3 | 3 |
| *col* | 2 | 3 | b |
| *active* | T | T | T |
| *p-row* | F | F | F |
| *p-col* | F | T | F |
| *hit* | F | T | T |
| *temp* | 1 | 2 | 4 |

(c)

| | | | |
|---|---|---|---|
| *value* | 1 | 2 | 5 |
| *row* | 3 | 3 | 3 |
| *col* | 2 | 3 | b |
| *active* | T | T | T |
| *p-row* | F | F | F |
| *p-col* | F | T | F |
| *hit* | F | T | T |
| *temp* | 1 | 0 | 1 |

(d)

| | | | | | |
|---|---|---|---|---|---|
| *value* | 0 | 1 | 2 | 0 | 5 |
| *row* | 3 | 3 | 3 | 3 | 3 |
| *col* | - | 2 | 3 | - | b |
| *active* | T | T | T | T | T |
| *p-row* | F | F | F | F | F |
| *p-col* | F | F | T | F | F |
| *hit* | T | F | T | T | T |
| *temp* | 1 | - | 2 | 3 | 4 |

(e)

| | | | | | |
|---|---|---|---|---|---|
| *value* | 0 | 1 | 2 | 0 | 5 |
| *row* | 3 | 3 | 3 | 3 | 3 |
| *col* | 1 | 2 | 3 | 5 | b |
| *active* | T | T | T | T | T |
| *p-row* | F | F | F | F | F |
| *p-col* | F | F | T | F | F |
| *hit* | T | F | T | T | T |
| *temp* | 3 | - | 1 | -4 | 3 |

(f)

| | | | | | |
|---|---|---|---|---|---|
| *value* | -6 | 1 | 0 | 8 | -1 |
| *row* | 3 | 3 | 3 | 3 | 3 |
| *col* | 1 | 2 | 3 | 5 | b |
| *active* | T | T | T | T | T |

Figure 4:
Snapshots of one row of a system of equations during one step of Gaussian elimination. Each column shows the contents of one entry (one L-cell). The coefficient, row and column number are shown as *value*, *row* and *col*, respectively. ("*col* = *b*" identifies elements of vector *b*.) Booleans *active*, *p-row* and *p-col* show that this is an active row, not the pivot row, with an entry in the pivot column. The Boolean *hit* shows which entries correspond to (non-zero) entries in the pivot row. The values in *temp* are used for creating fillins in (b)-(d), and are the pivot row entries in (e).

it receives, so we may restrict our attention to the time required for communication. Suppose there are $e$ entries in the original system and that, for the sequence of pivots we choose, $f$ fillins are created. (For convenience we assume $A$ is non-singular, so that $e \geq n$ and so that Gaussian elimination will take $n$ steps.) Clearly, we need $O(e + f)$ L-cells, and the height $h$ of the binary tree of T-cells needed to connect them is $O(\log(e + f))$. The cost of shifting in the original problem and of making room for fillins (over all $n$ steps) is $O(e+f)$, since the amount of space requested bounds the cost of storage movement. Each entry or fillin is in a pivot row exactly once, so it is broadcast twice for that purpose. Thus $O(e + f)$ messages go through the root of a submachine of height $O(h)$ in $n$ message waves. That requires $O(n \times h)$ time to fill the T-cell pipeline $n$ times, plus $O(e+f)$ time for the messages to arrive in sequence. Thus there is a total of $O(h \times (e+f))$ time for broadcasting the pivot rows. There are a fixed number of row operations for each step, and these only broadcast single values and perform parallel prefix computations. Each of these requires time $O(h)$, so over $n$ steps this is also $O(h \times (e + f))$. We conclude that the entire Gaussian elimination, then, requires $O(h \times (e + f))$ time.

*Back-substitution*: Once we have the system in permuted upper-triangular form, it is easy to finish solving the system by back-substitution. Each row (in an independent row operation) determines if it has the value for a variable by seeing if it has exactly one non-zero $a_{i,j}$ entry. (One parallel prefix operation can count the number of entries.) If so, it computes the value of $x_j$ (by one division) and broadcasts it on the next global operation and becomes inactive. Rows that remain active receive the $x$ values and eliminate those variables from their equation. Eventually, assuming the system has a unique solution, every row will have found the value of one variable. The time for this is $O(h \times n)$, which is again $O(h \times (e + f))$.

*Choosing pivots*: We modify the Gaussian elimination algorithm to choose pivots at runtime by the Markowitz heuristic, as follows. Let each entry keep two additional values, *nr* and *nc*. The number of other non-zero elements in an entry's row will be *nr*, and the number of other non-zero elements in its column will be *nc*. We can initialize the *nr* and *nc* values by broadcasting the *row* and *column* numbers of the original matrix entries, and letting each entry count the number of matches. Every time an entry becomes inactive (or zero), it globally broadcasts its row and column numbers, and other (active) entries decrement their *nr* and *nc* values, respectively, if the row or column numbers match. Every time a fillin is created, it globally broadcasts its row and column numbers, and other (active) entries increment their *nr* and *nc* values, respectively. To choose a pivot, each active entry computes the product of its *nr* and *nc* values, and the T-cell network chooses a minimal one. The additional time required for this is $O(h \times (e+f))$, with the analysis much like that for the basic algorithm.

## The Simulations

Our simulations supplement the analysis presented in the previous section. The set of sample problems contains both randomly generated matrices and matrices from various real-life applications (obtained from the Boeing-Harwell sparse matrix collection[4]). The simulator was written in C and run on a Convex C-220 system (two processors), with an implicit vectorizing/parallelizing compiler.

The simulator is in two parts. The first part is used to study fillin behavior and to count the number of floating point operations in triangularizing a coefficient matrix. It omits the back-substitution computation, but it does count the work required to choose pivots by the Markowitz heuristic. The output of the first part of the simulator is input to the second part, which uses parameters for an FFPM hardware prototype to provide performance estimates. In the remainder of this section, we describe the two parts of the simulator and summarize its results.

The coefficient matrices are transformed into a 0/1 representation, i.e., non-zero entries are replaced by 1s and zero entries by 0s. Due to the relatively small size of our sample problems and the simplicity of programming, we use dense matrix representations for storing the 0/1 matrices. We note, however, that the sparse nature of the problem is maintained in that only operations for non-zero entries are counted. The simulator follows the basic Gaussian elimination technique. The Markowitz heuristic, used to choose pivots, tends to minimize the number of fillins, retaining the sparsity of a matrix. Numerical stability issues were not taken into account, but one can add a threshold based criterion for selecting a numerically stable pivot without changing the complexity of the algorithm.

The results of the first part of our simulations on randomly generated matrices and matrices from the Boeing-Harwell collection are presented in Tables 1 and 2, respectively. The number "entries" is the

| order | entries | fillins | mean size of pivot row | sequential operations |
|---|---|---|---|---|
| 100 | 516 | 746 | 7 | 9955 |
| 200 | 1995 | 9380 | 29 | 548937 |
| 300 | 4466 | 31332 | 60 | 3560740 |
| 400 | 7922 | 68512 | 96 | 11749105 |
| 500 | 12421 | 125662 | 139 | 29876633 |

Table 1: Randomly generated matrices.

| order | discipline | entries | fillins | mean size of pivot row | sequential operations |
|---|---|---|---|---|---|
| 180 | astrophysics | 2659 | 145 | 7 | 30628 |
| 199 | stress analysis | 701 | 703 | 4 | 5003 |
| 541 | chemical kinetics | 4285 | 11442 | 14 | 301068 |
| 600 | oil recovery | 13760 | 30304 | 37 | 2060332 |
| 822 | linear programming | 4790 | 1701 | 5 | 15132 |

Table 2: Matrices from the Boeing-Harwell collection.

number of non-zero entries in the initial coefficient matrix, "fillins" is the number of entries created during Gaussian elimination, and "sequential operations" is the sum of divisions, multiplications, and subtractions performed. The results presented in Table 1 were obtained by averaging the results of 5 separate trials on random matrices with an average density of 5%.

Next we present the performance estimates produced by the second part of the simulator. Due to the pipelined nature of the FFPM architecture, there is a considerable overlap between useful computation and communication. Therefore, it is most meaningful to account for the net solution time for a problem. Messages are typically made up of several packets, based on data size and the format required by the T-cells. The speed of operation is directly governed by the parameters of hardware employed. The parameters used in this model were derived from the specifications of a transputer (20 MIPS) operating at the clock rate of 20 MHz. The communication channels in the tree-network were modeled to be bit-serial with peak throughput of 10 Mbps. The communication channels among cells in the L-array were considered to be byte wide with a peak data rate of 10 MBps.

The results are shown in Tables 3 and 4, corresponding to the entries in Tables 1 and 2. Here "parallel operations" is the sum of all divisions, multiplications, and subtraction operations performed in each disjoint sub-machine, and corresponds to the operations performed on the last message received by an L-cell. We have presented the results of total solution time for the problems under two distinct situations, the first where the pivoting sequence was defined before elimination, and the second where Markowitz heuristic was employed to determine the pivoting sequence during the computation. (Both use the same ordering for each matrix.) The cost of choosing pivots at runtime is surprisingly high, but this is due, in part, to the size of the message packets required in the design being simulated.

| order | parallel operations | time (msec) | |
|---|---|---|---|
| | | given order | runtime pivoting |
| 100 | 1338 | 18 | 61 |
| 200 | 7084 | 102 | 486 |
| 300 | 19989 | 293 | 1497 |
| 400 | 41042 | 601 | 3172 |
| 500 | 72579 | 1064 | 5708 |

Table 3: Performance estimates (random matrices).

| order | parallel operations | time (msec) | |
|---|---|---|---|
| | | given order | runtime pivoting |
| 180 | 2675 | 36 | 128 |
| 199 | 2108 | 27 | 77 |
| 541 | 12078 | 161 | 693 |
| 600 | 25682 | 394 | 1861 |
| 822 | 8151 | 128 | 352 |

Table 4: Performance estimates (Boeing-Harwell matrices).

## Conclusions

We have described and analyzed an algorithm for performing Gaussian elimination on sparse matrices with an FFPM. This work is relevant to FFPM development in two ways. First, it demonstrates how the basic FFP language can be extended with new primitive operations, to better support operations on sparse matrices. Secondly, it provides performance estimates for an FFPM hardware prototype on a real-life problem.

## Acknowledgements

We would like to thank Dr. A.M. Erisman and Dr. R.G. Grimes of Boeing Computer Services, Seattle, Washington, for providing us with Boeing-Harwell collection of sparse matrices. Mr. Mike Padrick and Mr. Larry Mason, of Academic Computing Services at UNC, provided assistance with systems related issues on the Convex computer. In the Department of Computer Science, Vernon Chi, William Partain and Donald Stanat provided encouragement and valuable suggestions, and Deborah Stogner produced the figures.

## References

[1] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613-641, 1978.

[2] I.S. Duff. The use of vector and parallel computers in the solution of large sparse linear equations. In *Large Scale Scientific Computing*, Birkhäuser, 1986.

[3] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices*. Clarendon Press, 1986.

[4] I.S. Duff, R.G. Grimes, and J.G. Lewis. *Sparse Matrix Test Problems*. Technical Report, Computer Science and Systems Division, Harwell Laboratory, 1987.

[5] G.A. Magó. Data sharing in an FFP Machine. In *Conference Record of the 1982 ACM Symposium on LISP and Functional Programming*, pages 201-207, 1982.

[6] G.A. Magó and W. Partain. Implementing dynamic arrays: a challenge for high-performance machines. In *Proceedings of the Second International Conference on Supercomputing*, pages 491-493, 1987.

[7] G.A. Magó and D.F. Stanat. The FFP Machine. In *High-Level Language Computer Architecture*, Computer Science Press, 1988.

[8] D. Middleton and B.T. Smith. FFP Machine support for language extensions. In *Proceedings of the 19th Hawaiian International Conference on Systems Sciences*, pages 59-65, 1986.

[9] B.T. Smith and D. Middleton. Exploiting fine-grained parallelism in production systems. In *Proceedings of the Seventh Biennial Conference of the Canadian Society for Computational Studies of Intelligence*, pages 262-270, 1988.

# A PARALLEL ALGORITHM FOR FINITE ELEMENT COMPUTATION

P. Subramaniam

Picker International
595 Miner rd.
Highland Hights, OH. 44143

N. Ida

Electrical Engineering Department
The University of Akron
Akron, OH. 44325

## ABSTRACT

The work presented here deals with the parallel implementation of finite element analysis algorithms for computation of electromagnetic fields. The methods apply equally well to other areas. The choice of a parallel implementation is based on the fact that many of the operations and algorithms used for finite element analysis (FEM) are essentially parallel or can be parallelized with a moderate level of effort. The solution of electromagnetic field problems is particularly appropriate into the context of parallel machines because of the open boundary nature of the problem and the size of the finite element matrices.

## INTRODUCTION

The basic stages of the finite element method have been parallelized and a working implementation has been tested. The first stage in the solution is the definition of elemental matrices. The sizes of these matrices range from as little as 3x3 for a simple 2-D element to 60x60 or more for 3-D elements. The solution of the system of equations is handled by a parallelized Gaussian elimination algorithm. Postprocessing consists essentially of calculating field intensities and flux densities as well as coil impedances.

The essential steps in FEM analysis are:

1. Discretization of the solution domain
2. Calculation of elemental matrices
3. Assembly of a global matrix
4. Solution of the system of equations
5. Post-processing of the results

This work concentrates on steps 2 and 3. Step 1 is a geometrical problem that includes definition of a geometry, decision on a discretization level, input of a variety of geometrical data, material properties etc. This is usually handled through a graphics preprocessor and is not suited for parallel machines. The interaction of the designer with the computer at this stage is essential and therefore, the best approach is to use a graphics workstation. The input for the FEM program is a geometric and problem dependent data file. This is assumed to have been generated for the purpose of this work.

The solution of the system of equations generated in the FEM process has been treated elsewhere [2]. The parallel solution routines developed are used in conjunction with the programs described here.

Finally, the postprocessing step has been left out because of its highly specialized nature. This may include calculation of electromagnetic fields everywhere or, perhaps a single scalar value like the total energy in the system or the impedance of a coil.

## THE FINITE ELEMENT METHOD

A brief outline of the FEM is given below with special reference to the following boundary-value problem:

$$\nu_x\frac{\partial^2 A}{\partial x^2} + \nu_y\frac{\partial^2 A}{\partial y^2} + J = 0 \qquad (x,y) \epsilon G \qquad (1a)$$

$$A=A_0 \qquad \text{on boundaries of } G \qquad (1b)$$

where $\nu_x$, $\nu_y$ are material properties associated with the solution domain and J is the current in the domain. In the solution domain, G, the magnetic vector potential (MVP) A satisfies Poisson's equation, while on the boundary B of the solution region, the MVP or its first derivatives are known. Here, and throughout this work, reference is made to the magnetic vector potential. Equation 1 however, applies to a variety of physical quantities, vector or scalar. Time dependent problems may also be considered but for simplicity, these are not discussed here. Also, a 2-D equation is used. 3-D equations are treated similarly with a somewhat lengthier process and with larger arrays.

The boundary-value problem in Eq.(1) can be stated by the following variational problem:

$$F(A)=\int_G [\frac{1}{2}[\nu_x(\frac{\partial A}{\partial x})^2+\nu_y(\frac{\partial A}{\partial y})^2]+J.A]dxdy \quad (x,y) \epsilon G \quad (2a)$$

$$A=A_0 \qquad \text{on boundaries of } G \quad (2b)$$

Minimization of F(A) yields the "best solution" to the original equation.

The case of known normal derivatives is not included in Eq. (2). This is a natural boundary conditions for the functional and need not be specified.

The solution region is divided into a number of "finite elements". Here, each element is a rectangle or a triangle. The elements are assumed to be interconnected at a number of nodal points situated on their boundaries. The MVPs at these nodal points are the unknowns.

The MVP within each element can be defined using various approximations. There are two basic methods of defining these approximations. One is based on products of polynomials in a local system of coordinates while the other is based on a polynomial over the element, defined in the system of coordinates in which the problem is solved [1].

The two methods are quite different and will be outlined briefly below as this is necessary for their implementation. To illustrate the process we use the two simple elements in Fig. 1.

### ISOPARAMETRIC FINITE ELEMENTS

The approximation used for the function A in the interior of the finite element in Fig. 1a is:

$$A(x,y)=\sum_{i=1}^{4} N_i A_i \qquad (3)$$

where $N_i$ are a set of shape functions defined at the four nodes of the elemet and $A_i$ are the values of the unknown function at the same nodes. Similarly, the derivatives of A can be taken as:

$$\frac{\partial A(x,y)}{\partial x}=\sum_{i=1}^{4}\frac{\partial N_i}{\partial x}A_i \qquad \frac{\partial A(x,y)}{\partial y}=\sum_{i=1}^{4}\frac{\partial N_i}{\partial y}A_i \qquad (4)$$

The functions $N_i$ as well as their derivatives $\partial N_i/\partial x$, $\partial N_i/\partial y$ need to be known. The standard method for their calculation is to find them in a local system of coordinates where they are extremely simple and then to map them into the global system of coordinates. The derivatives with respect to x and y are calculated as:

$$\begin{bmatrix}\frac{\partial N_i}{\partial x}\\ \frac{\partial N_i}{\partial y}\end{bmatrix}=[J]^{-1}\begin{bmatrix}\frac{\partial N_i}{\partial \xi}\\ \frac{\partial N_i}{\partial \eta}\end{bmatrix} \qquad [J]=\begin{bmatrix}\frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi}\\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta}\end{bmatrix} \qquad (5)$$

where [J] is the Jacobian matrix.

These are now substituted in Eq. (2). In order to find a solution to the problem, the functional in (2) is minimized by setting it's first derivatives with respect to each unknown to zero. This produces an algebraic equation for each unknown. Rather than doing this for the whole solution domain, it is done for each element separately and the contributions from separate elements are summed up in a global system of equations.

To produce the elemental contributions, the terms in Eq. (2) need to be evaluated. This is done numerically by Gaussian quadrature.

$$\sum_{i=1}^{2}\sum_{j=1}^{2} W_i . W_j . f'(\xi_i,\eta_j) \qquad (6)$$

where $W_i, W_j$ are weights and $\xi_i,\eta_j$ are quadrature points. This integration is done locally and then, using the shape functions, mapped to the global system of coordinates. For a simple element like this two points in each spatial direction are sufficient. For more complex elements, more points may be required.

### Implementation on the MPP

Implementation begins by creating three arrays: SF1, SF2 and SF3 where SF1 holds the shape functions of the four nodes as four rows of four entries each. Each column holds the shape function for one quadrature point. Similarly, SF2 holds $\partial N_i/\partial \xi$ in four columns, each column corresponding to a quadrature point and each row to a nodal point of the element. SF3 has the same structure for $\partial N_j/\partial \eta$.

The first step in the computation of the elemental matrices is the calculation of the Jacobian (Eq. 5). It is stored in four arrays: RJAC1, RJAC2, RJAC3 and RJAC4. RJAC1 holds the first coefficient (RJAC(1,1) for 128 elements calculated in parallel. RJAC2 contains the second coefficient (RJAC(1,2)), RJAC3 contains RJAC(2,1) and RJAC4 contains RJAC(2,2). The determinant of the Jacobian is calculated and stored in array DETJ for all elements in parallel. The inverse of the Jacobian needed in Eq. (5) is now found for all 128 elements by a single divide of the RJAC arrays by the DETJ array. The results are placed back into the RJAC(i) arrays.

The derivatives of the shape functions with respect to x (Eq. (5)) are calculated by multiplying RJAC1 by the SF2 array, RJAC2 by the SF3 array and adding the results together. The calculation of the derivatives with respect to Y is found as the sum of RJAC3*SF3 and RJAC4*SF3.

The contributions to the elemental matrix in Eq. (6) are found by parallel array multiplication of each of the four DNDX(i) array by DNDX(j), including itself and adding these to the product of the DNDY(i)*DNDY(j) arrays. The material properties RXI ($\nu_x$), RYI ($\nu_y$) are taken from array MAT by shifting operations. From this, the 16 contributions to the elemental matrices are found and entered in a single array (RRR). Each element takes the first 16 rows on a single column.

The global equation assembly proceeds by moving each value in the elemental matrix into the global matrix.

To create the right hand side (RHS) of the system of equations, the same process is used. As is clear from Eq. (2), The current density J is multiplied by the shape functions in SF1 and integrated using Eq. (6). An elemental column vector is created and this is later put in a global RHS vector.

The method outlined can handle up to 128 elements in parallel. It is possible to extend this to any number but it requires handling multiple arrays. Since in a FEM analysis, most of the time is spent in the solution process this was not considered worth while. There are two problems associated with this method. One is the need for sequential insertion in the global matrix. The second, is in the number of arrays required. Since the number of arrays is relatively large (about 20 for a four node element), the only elements that can be handled in this fashion are elements with a small number of nodes (3 to 6 nodes). Other types of elements need special considerations. This may seem limiting but most FEM programs use these elements. For more complex elements, elemental matrices can be combined in arrays, with fewer parallel matrices at each step. Alternatively, the method of storage in the following section is used.

## DIRECT DEFINITION OF ELEMENTAL MATRICES

Another method for the definition of the elemental contributions is to assume that the distribution within the finite element is of the following form [1]

$$A(x,y)=ax+by+c \tag{7}$$

If this polynomial is written at the three nodes of the element in Fig. 1b, we get three equations in the unknowns a, b and c. By solving these equations we get:

$$A(x,y)=N_iA_i+N_jA_j+N_mA_m=\sum_{i=1}^{4} N_iA_i \tag{8}$$

where $N_i$, $N_j$, $N_m$ are the shape functions given by

$$N_i(x,y)=(a_ix+b_iy+ci)/2D \tag{9a}$$
$$N_j(x,y)=(a_jx+b_jy+cj)/2D \tag{9b}$$
$$N_m(x,y)=(a_mx+b_my+cm)/2D \tag{9c}$$

where:

$$a_i=y_j-y_m \qquad a_j=y_m-y_i \qquad a_m=y_i-y_j \tag{10a}$$
$$b_i=x_m-x_j \qquad b_j=x_i-x_m \qquad b_m=x_j-x_i \tag{10b}$$
$$c_i=x_jy_m-x_my_j \qquad c_j=x_my_i-x_iy_m \qquad c_m=x_iy_j-x_jy_i \tag{10c}$$

and $x_k$, $y_k$ are the x and y coordinates of node k, k=i,j,m. D is the area of element ijm.

The expressions in Eq.(9) are substituted into the functional F(A). Minimization of F(A) yields a characteristic equation for triangle ijm:

$$[K]_e\{A\}_e=\{F\}_e \qquad [K]_e=\begin{bmatrix} k_{ii} & k_{ij} & k_{im} \\ k_{ji} & k_{jj} & k_{jm} \\ k_{mi} & k_{mj} & k_{mm} \end{bmatrix} \qquad \{F\}_e=\begin{bmatrix} F_i \\ F_j \\ F_m \end{bmatrix} \tag{11}$$

where:

$$\begin{aligned} &k_{ii}=(a_ia_i+b_ib_i)/4D \quad k_{ij}=(a_ia_j+b_ib_j)/4D \\ &k_{im}=(a_ia_m+b_ib_m)/4D \quad k_{ji}=(a_ja_i+b_jb_i)/4D \\ &k_{jj}=(a_ja_j+b_jb_j)/4D \quad k_{jm}=(a_ja_m+b_jb_m)/4D \\ &k_{mi}=(a_ma_i+b_mb_i)/4D \quad k_{mj}=(a_ma_j+b_mb_j)/4D \\ &k_{mm}=(a_ma_m+b_mb_m)/4D \quad F_i=F_j=F_m=D*J/3 \end{aligned} \tag{12}$$

A global equation in the form $\{K\}\{A\}=\{F\}$ is assembled by accumulating the contributions of all elemental matrices (Eq. (11).

## Implementation on the MPP

The parallelization process can be divided into two phases. In the first phase, the elemental matrix is computed where the 9 coefficients are calculated in parallel. In the second phase, a number of elemental matrices are placed in the global matrix concurrently.

First, the elemental matrix $[K]_e$ in Eq.(11) is rearranged into the following form:

$$[K]_e=([K]_{e1}+[K]_{e2})/4D \tag{13}$$

where

$$[K]_{e1}=\begin{bmatrix} a_ia_i & a_ia_j & a_ia_m \\ a_ja_i & a_ja_j & a_ja_m \\ a_ma_i & a_ma_j & a_ma_m \end{bmatrix} \qquad [k]_{e2}=\begin{bmatrix} b_ib_i & b_ib_j & b_ib_m \\ b_jb_i & b_jb_j & b_jb_m \\ b_mb_i & b_mb_j & b_mb_m \end{bmatrix} \tag{14}$$

The two matrices in (14) can be further represented as

$$[k]_{e1}=[P]([P])^t+[Q]([Q])^t \tag{15}$$

where

$$[P]=\begin{bmatrix} a_i & a_i & a_i \\ a_j & a_j & a_j \\ a_m & a_m & a_m \end{bmatrix} \qquad [Q]=\begin{bmatrix} b_i & b_i & b_i \\ b_j & b_j & b_j \\ b_m & b_m & b_m \end{bmatrix} \tag{16}$$

Based on the expression in Eq.(10), [P] and [Q] are now represented as:

$$[P]=[P1]-[P2] \qquad [Q]=[Q1]-[Q2] \tag{17}$$

where [P1], [P2], [Q1] and [Q2] are the two parts of the expressions in Eq. 10a and 10b.

Special consideration is given to the calculation of D, the area of triangle ijm. In order to perform the divide operation in Eq. (13), an array, [D] is created

$$[D]=2([R]-[S]) \tag{18}$$

where

$$[R]=\begin{bmatrix} a_ib_j & a_ib_j & a_ib_j \\ a_ib_j & a_ib_j & a_ib_j \\ a_ib_j & a_ib_j & a_ib_j \end{bmatrix} \qquad [S]=\begin{bmatrix} a_jb_i & a_jb_i & a_jb_i \\ a_jb_i & a_jb_i & a_jb_i \\ a_jb_i & a_jb_i & a_jb_i \end{bmatrix} \tag{19}$$

These two arrays can be further represented in terms of the coordinates of the three nodal points:

$$[R]=([R2]-[R3])([S1]-[S3]) \tag{20a}$$
$$[S]=([R3]-[R1])([S3]-[S2]) \tag{20b}$$

where [S1] has $x_i$ propagated in all nine locations and [R1] has $y_i$ propagated throughout. Similarly, [S2] and [R2] contain $x_j$ and $y_j$ respectively and [S3] and [R3] contain $x_m$ and $y_m$ propagated throughout.

Finally, the elemental matrix of triangle ijm is obtained by performing one array divide operation:

$$[K]_e=([K]_{e1}+[K]_{e2})/D \tag{21}$$

At this point, the calculation of the elemental matrix has been parallelized. However, for the element used here, each array operation involves only 9 coefficients. A number of elemental matrices can be computed concurrently by creating a whole plane or nearly a whole plane of data before any array operation is carried out. As an example a rectangular mesh with 210 triangular elements is used (Fig. 2). The mesh can be partitioned into 8 sets of elements that have mutually exclusive nodes as shown. All nodal coordinates of elements in this set are placed in [P1], [P2], [Q1], [Q2] in Eq.(17), and [S1], [S2], [S3], [R1], [R2], [R3] in Eq.(20) to form nearly a whole plane of data. These elements can be assembled in parallel. Once the elements in a set have been assembled, a new set is treated until all elements have been assembled.

A Parallel Pascal code has been developed based on the above parallel algorithm and applied to perform global equation assembly on the MPP for the finite element mesh shown in Fig. 2. The total processing time is 114.47 ms including the time needed for local nodal numbering. A larger portion of the total processing time has been spent on forming whole planes of data. Since the x and y coordinates are stored in two arrays, considerable use of fast row and column propagation routines has been made. Thus, the assembly routines are not particularly efficient for elements with few nodal points.

There are some important elements, such as 3D solid and shell elements, etc., which are extremely complex and require considerable computer resources. For these elements, parallel assembly of the equations is a significant step towards improving solution times and, in some cases (boundary integral elements) may be more significant than the solution of the system of equations. For these elements, the method outlined becomes more efficient as the number of nodes per element approaches 128.

## CONCLUSIONS

Two methods for assembly of systems of equations arising from finite element analysis have been presented. One is particularly suited for elements with few nodal points while the other is for directly defined finite elements. The time involved in assembly is not considered to be significant compared with the time needed for solution.

## REFERENCES

[1] O. C. Zienkiewicz, The Finite Element Method in Engineering, third edition, McGraw-Hill Book Co., London, 1977.

[2] J. S. Wang and N. Ida, "Parallel algorithms for direct solution of large systems of equations", these proceedings.

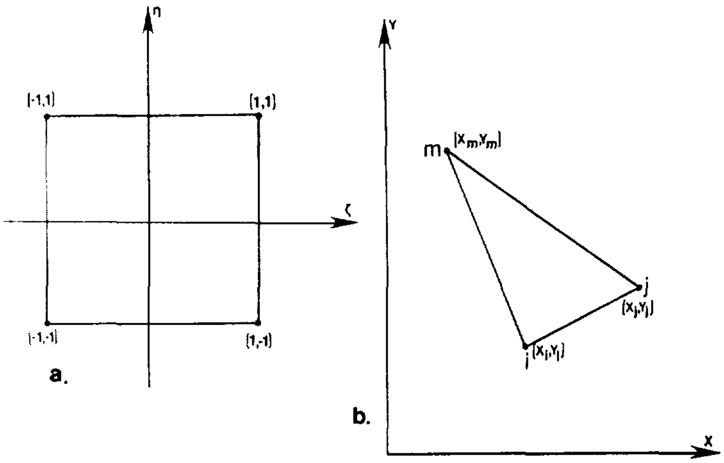


Figure 1. Two finite elements. a. Isoparametric element defined in a local system. b. Triangular element defined in a global system of coordinates.

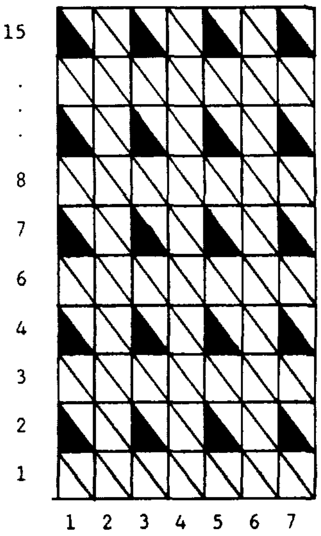


Figure 2. Elements in a mesh with mutually exclussive nodes.

Table 1. Processing time on the MPP for matrix assembly for different finite element meshes. Time is in seconds

| No. of Equat. | No. of Elements | Band-width | Time on the MPP |
|---|---|---|---|
| 128 | 93 | 63 | 0.183 |
| 256 | 189 | 127 | 1.296 |
| 512 | 441 | 127 | 3.488 |
| 1024 | 889 | 255 | 14.363 |

# THE FAST FOURIER TRANSFORM AS A TEST CASE FOR A SYSTOLIC DATA FLOW MACHINE

Doron Tal, John Comfort, Maria Martinez
School of Computer Science
Florida International University-
The State University of Florida at Miami
University Park, Miami, Florida 33199

## ABSTRACT

The Systolic Data Flow Machine (SDFM) is a new hybrid architecture concept that provides a new pattern for parallel processing. It is a two level architecture that combines the efficiency of systolic arrays with the elegance of a data flow architecture. The SDFM is based on the partitioning of data flow programs (graphs) into subgraphs of sufficiently small size so that they may be loaded into programmable systolic arrays, called processing elements. In this paper, we map a FFT algorithm onto a suggested processing element topology in order to demonstrate the utility of this approach, and to investigate the exploitation of the natural concurrency of this algorithm. This paper should be considered as an interim report from a major project to design and implement a SDFM computer system.

Keywords: FFT, Data Flow, Systolic Arrays, Parallel Processing, Mapability.

## INTRODUCTION

Recent trends in computer systems are the distribution of computation among physical processors, and the search for alternatives to the classical sequential (von Neumann) machine. Data flow machines [8,2,6,7,11], and systolic array processors [10,9,1]. are such alternatives.

The prime features which make the Data Flow model so promising are its capabilities to automatically distribute control down to the level of operations on scalar operands, and efficiently to exploit, on a large scale, an algorithm's inherent parallelism. Unfortunately, experience with Data Flow machines has revealed several serious problems. The absence of explicit storage, operand accumulation, the law of granularity, and the need to control and support large amount of interprocessor communications impose serious overhead problems.

Systolic arrays - large regular arrays composed of identical processors interconnected in a regular pattern - can efficiently support parallelism at an extremely fine level of granularity with no overhead. Systolic architecture is very attractive from the hardware viewpoint, for many processors interconnected by short communication paths could be effected by an economical VLSI design and implementation. It is also extremely attractive for the implementation of algorithms composed of a large number of similar computational cells. It does not seem to be directly applicable to support a general computing environment. Thus, it appears unlikely that computers based exclusively on either architecture will be able to compete with von Neumann based machines for general purpose computation.

These shortcomings have motivated the proposal of a new hybrid architectural concept - the Systolic Data Flow Machine (SDFM), which will provide a new pattern for parallel processing. It employs two levels, combining the efficiency of systolic arrays at the lower level with the elegance of a data flow architecture at the higher.

In this paper, we investigate the utility of this approach, and discuss the optimal exploitation of the natural concurrency of this application algorithm. To do this, we have mapped several FFT algorithm fragments onto a few systolic elements with a suggested topology. Then, we suggest mapability and performance criterions such as the number of allocated primitive processors (in a systolic array), and the number of active primitive processors and systolic arrays at a time.

## THE SDFM OPERATION

Conceptually, the SDFM is based on the partitioning of data flow programs (graphs) into subgraphs of sufficiently small size so that they may be loaded into programmable systolic arrays, called Processing Elements (PE's), each with local common memory. This memory may be used to contain structures accessible to all the processors in the PE. As the number of subgraphs required for the execution of a set of multiprogrammed processes will likely exceed the available number of PEs, PE faults - that is, input tokens appearing for which the designated subgraph is not presently assigned to a PE - will cause replacement of inactive resident PE subgraphs in the manner of a page fault. Figure 1 illustrates a set of physical PEs and a data flow program graph partitioned into a few blocks, some of which

CH2649-2/89/0000/0223$01.00 © 1988 IEEE

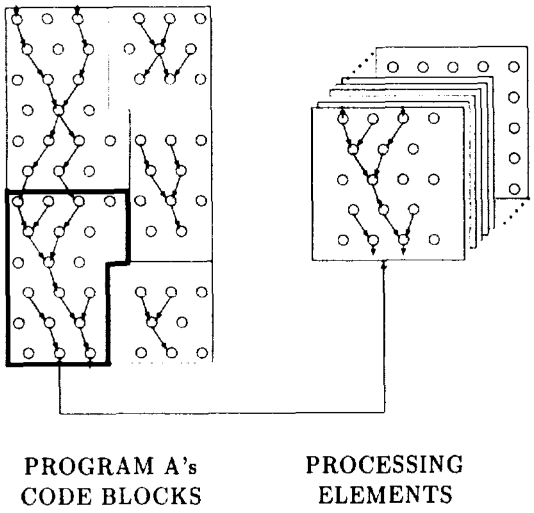


Figure 1: Data flow program graph blocks mapped to several processing elements.

are currently mapped to PEs (note the block marked by the thick frame). This approach is particularly attractive, since frequently invoked operating system procedures can reside permanently in PEs, eliminating the artificial temporal linearization of resident routines of an operating system used in a conventional machine.

### The Processing Element

The Processing Element (PE) implements the lowest granularity level of the SDFM. Each PE is a systolic array composed of a few Primitive Processors (PPs) connected in a regular geometry. This processors operate concurrently, passing scalar operands between them continuously. The topolgy being selected is a planar graph, where each node represents a PP, and has a connectivity with indegree 2 and outdegree 2 [1]. In order to support iteration the array is (conceptually) mapped onto a cylinder. Since only a subset of the hard-wired connections between PPs are (software) selected, many different configurations can be mapped to this physical pattern.

Further discussion of the SDFM architecture and its PE topolgy is given in [4,12].

## THE MAPPING OF A FFT ALGORITHM

### The FFT Algorithm

The Fast Fourier Transform is an algorithm that rapidly computes the discrete Fourier Transform. The algorithm is described by [5]:

$$X(n) = \sum_{k=0}^{n-1} X_0(k) e^{-j2\pi nk/N}$$

where $X_0(k)$ are the original (complex) data points in time or space domain, and $X(n)$ are the transformed points in the frequency domain.

For $N = 2^\gamma$ the FFT algorithm is a procedure for factoring an N x N matrix into $\gamma$ matrices. For example, if N=4, and $W = e^{-j2\pi/N}$.

$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \end{bmatrix} \underset{\cdot}{=} \begin{bmatrix} W^0 & W^0 & W^0 & W0 \\ W^0 & W^1 & W^2 & W3 \\ W^0 & W^2 & W^4 & W5 \\ W^0 & W^3 & W^6 & W9 \end{bmatrix} \begin{bmatrix} X_0(0) \\ X_0(1) \\ X_0(2) \\ X_0(3) \end{bmatrix}$$

The FFT algorithm factors the W matrix into the following submatrices:

$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \end{bmatrix} = \begin{bmatrix} 1 & W^0 & 0 & 0 \\ 1 & W^2 & 0 & 0 \\ 0 & 0 & 1 & W^1 \\ 0 & 0 & 1 & W^3 \end{bmatrix} \begin{bmatrix} 1 & 0 & W^0 & 0 \\ 0 & 1 & 0 & W^0 \\ 1 & 0 & W^2 & 0 \\ 0 & 1 & 0 & W^2 \end{bmatrix} \begin{bmatrix} X_0(0) \\ X_0(1) \\ X_0(2) \\ X_0(3) \end{bmatrix}$$

### Signal Flow Graph

A signal flow graph ([5]) can be constructed to represent the above matrix multiplication as shown in Figure 2.

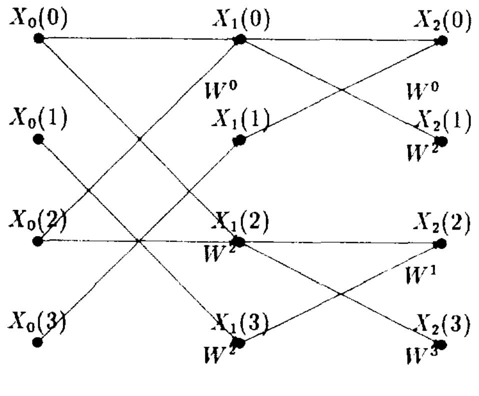


Figure 2: Signal flow graph for the matrix multiplication.

[1] Optimal choice of array geometry and the associated mapping problems are discussed in [3,14,13,15].

Each node is entered by two paths. A quantity from one array travels along a path, is multiplied by $W^p$, and proceeds into next array (no W shown implies $W^p = 1$). The inputs are added at each node. Each column of the graph corresponds to a factored matrix, of which there are $\gamma$.

### The Mapping of the FFT

As an example, we map the FFT algorithm with N=4 onto a single PE as shown in Figure 3.

The mapping is performed by insertion of additional "transmission" modes, and labeling (associating nodes with operators and/or routing functions.) At each point of insertion we use a routing node (e.g. a "crossover" node is inserted at each point where two edges cross.) Note, that the routing processors contribute to computation, since they are the analog of the data movement instructions in a conventional machine, or of the crowded communication buses in a data flow machine. The rest of the allocated PPs take on the dual functions of performing operations and transmitting data. The execution of such a data flow graph proceeds like a wavefront through the graph. We expect the PPs on each row to execute concurrently, as well as yielding operands to trigger PPs on the following row. Operands generated by PPs which reside on the last row are used as inputs to PPs comprising the first row (the cylinder structure facilitates iteration.) When all the iterations are completed, the results are the outputs of the PE. Let l, which will serve as an iteration counter, be initially set to $\gamma$ where $N = 2^\gamma$.

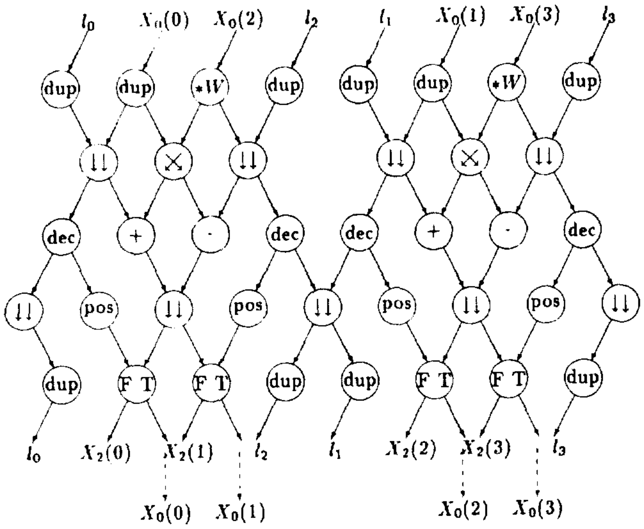


Figure 3: Mapping of a FFT algorithm onto a single PE.

### Generalization

While the PE has been shown as optimal for processing the FFT algorithm with a N=4, problems in the real world invariably involve handling FFT algorithms with much larger N.

```
For j=1 to $\gamma$
    For i=0 to N/2-1
        $X_j(i) := X(i) + X(i + (N/2 - 1)) * W^p$
        $X_j(i + (N/2 - 1)) := X(i) - X(i + (N/2 - 1)) * W^p$
    For i=0 to N/2-1
        $X(i * 2) := X_j(i)$
        $X(i * 2 + 1) := X_j(i + N/2 - 1))$
End
```

Since the destinations of all the operands inside an iteration are local, we can partition the computation graph to N/4 sub-graphs, each of which will be mapped to a single PE. Some of the outputs of a subgraph may be destined to subgraphs mapped to other PE's (external communication). This is accomplished by placing the PE's outputs on the ring bus, and transmitting them to the inputs of other PEs. For very large N (where the number of FFT subgraphs is larger then the number of physical PEs) we use virtualization. This elimination of the "residency requirement" facilitates the provision of many virtual PEs to accommodate the totality of the FFT data flow graphs.

## RESULTS

From studying the mapping of the FFT algorithm , conclusions can be obtain about system attributes such as: the ratio of local to global communication, granularity, instruction execution and communication time, parallelism and processor utilization. For this purpose, we suggest two mapability criteria: the processor allocation density (the ratio of allocated PPs to (allocated + free) PPs ratio) and the systolic efficiency (the ratio of local communication to (local + distant) communication ratio). All the PPs are being allocated if N mod 4 is 0 - otherwise, only one PE will have a lower density. Consequently, for large N the density will go to 1. Approximately, 53 percent of the PP's are utilized for communication purposes (routing, duplication, or crossing over). The rest of the PP's are used for simple instruction execution (multiplication, addition, and subtraction of scalar operands). A small fraction is also used for control instructions. The systolic efficiency is a bit lower then the results we got in other programs, and is 12 for N=4, and converges to 6 for large N.

The parallelism obtained by computing the FFT using the SDFM varies at the different architecture levels. The mapping shows that for $N = 2^\gamma$ there will be $2^\gamma$ inputs and $\gamma$ iterations of the systolic array. Each iteration is mapped onto a few PPs allowing for a simultaneous execution of scalar operators. Moreover, for large N the input can be partitioned in such a way that the iteration bodies are mapped, each to its own PE, allowing still further parallelism . Note that in this way, we achieve linear throughput and the degree of parallelism comparable with the number of available PEs. Thus, the parallelism achieved at the data flow level is optimal, as all the PEs are executing simultaneously, while at the PE level, 8 PPs (a single row) are active at a time.

The granule size -that is, the complexity of the tasks assigned to each processor – is one "butterfly" per processor. The granule size effects the complexity of the Individual PP's. If the granule size is increased by assigning multiple butterflies to each processor, the complexity of the PP's increases. If the granule size is decreased the PP's do only primitive floating point operations.

### FUTURE DEVELOPMENTS

We are investigating the performance of such a system by constructing a simulation model of a SDFM executing the mapped FFT algorithm.

To verify the correctness of the algorithm, and to validate partially the results of the simulation, the algorithm will be implemented on a small network. Each node in the network will be construed as a Transputer[2] connected to a multi-port link switch[3]. The action of each processor during each computational phase will require several steps: the data points must be input, the butterfly computation(s) must be performed, the destination Transputers must be selected, and the data points must be transmitted. Note that the first and last steps are performed concurrently, and that the final permutation of the data points into reverse binary order is done by the output selection step of this algorithm.

This research is a part of a three year project whose goals are the design of the systolic chip to use in the SDFM, the development of a language to document data flow programs, and the creation of a simulation/emulation environment to be used in verifying data flow programs, and in projecting the performance attributes of data flow systems not yet implemented. The language is being created as a superset of OCCAM, and the environment will be implemented using a network of Transputers. The project's ultimate goal is the development of a massively parallel general purpose computer system based upon the the SDFM concept.

[2] A Transputer(trademark of INMOS, Inc.) is a relatively fast reduced instruction set machine designed to function as a node in a network. Communication is accomplished through four 20 MHz bidirectional links.

[3] The INMOS C004 switch is a 32 wide link multiplexer. It is controlled through an additional link port. To connect any of the 32 link ports to any other requires 4 byte transmission times (about 1.2 microseconds).

## References

[1] M. Annaratone, E. Arnould, H.T. Kung, M. Lam, O. Menzilcioglu, and J.A. Webb. The wrap computer: architecture, implementation, and performance. *IEEE Transactions On Computer*, C-36(12):1523-1538, 1987.

[2] Arvind and D.E. Culler. Why dataflow architectures. In *The 4th Jerusalem Conference on Information Technology*, pages 27-32, 1984.

[3] R. Asbury, S.G. Frison, and T. Roth. Concurrent computers ideal for inherently parallel problems. *Computer Design*, 9:99-107, 1985.

[4] S. Bergman and D. Tal. Dedicated systolic arrays as nodes in a data flow machine. In W.R. Moore, editor, *Systolic Arrays*, pages 321-330, Adam Hilger, Bristol, 1987.

[5] O. E. Brigham. *The Fast Fourier Transform*. Prentice-Hall, Inc., Englewood Cliffs, New Jersy, 1974.

[6] A.L. Davis and R.M. Keller. Data flow program graphs. *IEEE Computer*, 15(2):26-41, 1982.

[7] J.B. Dennis. Data flow supercomputers. *IEEE Computer*, 13(11):48-56, 1980.

[8] A. Dubielewiez, A. Kalis, J. Kwiatkowski, and J. Ratajczak. An architecture of stand-alone multiprogramming data flow system. In *Euromicro Symposium*, pages 603-611, Brussels, September 1985.

[9] Special issue on systolic arrays. *IEEE Computer*, 20(7), 1987.

[10] H.T. Kung. Why systolic architectures? *IEEE Computer*, 15(1):37-46, 1982.

[11] W.G. Paseman. Applying data flow in the real world. *BYTE*, 10(5):201-214, 1985.

[12] D. Tal and S. Bergman. *The SDFM Architecture*. Computer Science Technical Report M268, Ben-Gurion University, Beer Sheva, July 1986.

[13] D.R. Wiilson, S.C. Winter, J.P. Gupta, and P. Espinasse. The execution of functional programs on the ctd multiprocessor. In *Euromicro Symposium*, pages 587-595, Brussels, September 1985.

[14] S.C. Winter. A cube type distributed interconnection of microcomputing elements. In *Euromicro Symposium*, pages 301-310, Madrid, September 1983.

[15] S. Yalamanchill and J.K. Aggarwal. Reconfiguration strategies for parallel architectures. *IEEE Computer*, 18(12):44-61, 1985.

# OPTIMAL TILINGS FOR ITERATIVE PDE SOLVERS *

Anthony E. Terrano
Department of Electrical and Computer Engineering
Rutgers University
Piscataway, NJ 08855

*This research supported in part by the National Science Foundation under grant MIP87-10829.

## ABSTRACT

The efficiency of a parallel algorithm for solving a partial differential equation on a multiprocessor computer is determined by the manner in which the problem is partitioned among the processors. Previous analyses have focussed on the relationship between the shape of the partition tile and the connectivity of the interconnection network, and have concluded that there is no simple rule governing the efficiency of a mapping. By extending the analysis to include geometrical considerations, in particular, the relative orientation of the update stencil and the partition tile, we have discovered a simple, universal procedure for deriving the optimal partition for a given iterative PDE algorithm. Based on this procedure, we derive two new partitionings for specific PDE algorithms which are more efficient than any previously known. The choice of partition is independent of the details of the multiprocessor architecture for a wide range of operating parameters.

Keywords: partial differential equations, communication demand, distributed memory multiprocessor, problem decomposition.

## INTRODUCTION

Multiprocessor computer architectures promise both great improvements in the amount of computational power which can be brought to bear on a given problem as well as significant reductions in the cost of a given amount of power. However, the efficient use of a parallel computer requires that each calculation be decomposed into independent pieces which can be solved concurrently. While a general solution to this problem has proven elusive, efficient algorithms and heuristics have been devised for a number of particular problems.

The efficiency of an implementation of a parallel algorithm can be determined by measuring the execution time as a function of the number of processors, $N$. The efficiency $E$, is then defined as the ratio of the observed speedup to the theoretical maximum:

$$E(N) = \frac{N \times (execution\ time\ for\ one\ processor)}{execution\ time\ for\ N\ processors} \qquad (1)$$

For distributed memory architectures, the limiting factor is the amount of data which must be exchanged between different processors for each operation. The best decomposition of a problem will minimize the ratio of communication to computation, $R_{cc}$, allowing the best possible approach to a linear speedup in the number of processors.

In the general case, all possible partitionings of the problem must be examined to find the one which minimizes $R_{cc}$. For specific classes of problems, a more detailed analysis can be performed. In this paper, we consider the iterative solution of the sparse linear equations which arise from discretizing a continuous PDE. These algorithms typically involve updating the current value of the solution at each grid point by means of a simple calculation involving the values at neighboring points. The collection of neighbors whose values are required for the update depends upon the precise algorithm, and is referred to as the stencil. With these algorithms many grid points may be updated simultaneously. The only constraint is that, for each point being updated, none of the other points in its stencil are also being updated at the same time.

## MAPPING

The best mapping of this problem onto a distributed memory architecture depends on the number of grid points $N_g$ versus the number of processors $N_p$. If we assume that typical problem sizes range upward from hundreds of grid points per side, with thousands of points being more nearly ideal, then a generous lower bound for $N_g$ will be $10^4$ for two-dimensional problems, and $10^6$ for three-dimensional ones. Most distributed-memory computers which have been built have $N_p$ ranging up to $10^4$; the largest values are still less than $10^5$. For the purposes of this paper, we make the reasonable assumption that $r = \frac{N_g}{N_p} > 1$.

Under this assumption, the most natural mapping of the problem into the architecture can be constructed as follows: first, embed a plane into the multiprocessor communication network, then tile the domain of the PDE problem with $N_p$ identical polygons, and finally assign one tile to each processor, with neighboring tiles assigned to neighboring processors under the embedding in defined in the first step. With this mapping, each processor will be solving a smaller, equivalent PDE boundary-value problem, with the boundary conditions at each stage determined by the solutions obtained thus far on the neighboring tiles. The interprocessor communication load is determined by the need to keep the boundary conditions current on each tile.

With this mapping, the efficiency of an implementation will be determined by the connectivity of the communication network, by the particular polygonal used for the tile, by the update stencil, and by the relative orientation of the tile and the stencil. Previous analyses of this problem have analyzed the effects of the only first two of these considerations [1] [2]. In particular, Reed, Adams and Patrick [2], have computed $R_{cc}$ for several different combinations of stencils and regular tilings of the plane. They found that, for some stencils a square tiling minimized the communication demands, while for others, the a hexagonl tile was better. They concluded that no simple relationship between a stencil and the preferred tiling exists, and that the best architecture for a given algorithm could only be determined by means of a detailed calculation of the communication demand for all possible stencils.

In this paper, we present an analysis which includes the effects of the relative orientation of the stencil and the tiling. By considering tilings which are oriented at an angle with respect to the coordinate axes, we reach qualitatively different conclusions from those in reference [2]. For any stencil, there exists an optimal tiling of the plane and this tiling can be easily derived from the shape of the stencil alone. By applying this construction to commonly occuring stencils, we have discovered two novel tilings of the plane: the diamond tiling and the oblique hexagon tiling. Furthermore, the construction is completely mechanical and thus can be performed automatically by a computer program.

The optimal decomposition of the plane for a given stencil is the one in which the tiles best approximate the shape and orientation of the stencils. By "best approximation", we mean the following: Given a stencil, identify the points which are the furthest from the center. Connect the points in clockwise order with line segments. The resulting shape will be a convex polygon, with a specific orientation relative to the stencil and to the coordinate axes. Such a polygon is called the convex hull of the stencil. If the polygon can be used to tile the plane, while preserving the required orientation, then it will be optimal tile for the stencil.

## TILING

In figure 1, we apply this construction to several common stencils. The convex hull for the five point, nine point cross and the thirteen point stencils is a diamond, for the seven point stencil, it is an oblique hexagon, and for the nine point star, it is a square. In figure 2, we present the resulting tilings of the plane. For completeness, we also show the hexagonal tiling analyzed in reference [2].

The diamond tiling consists of squares oriented at 45° with respect to the coordinate axes. The tile may be parametrized by the length, $k$, of its diagonal. The condition that all tiles be equivalent requires that $k$ by an even divsior of $n$. The number of points enclosed by a tile is $\frac{k^2}{2}$ and the perimeter is $2k - 2$. Finally, for comparison to the other tilings, it is useful to express $k$ in terms of $r$: $\frac{N_t}{N_p} = \frac{k^2}{2}$ or $k = \sqrt{2}r$.

| Stencil | Tile Shape | | | |
|---|---|---|---|---|
| | | oblique | | |
| | diamond | hexagon | square | hexagon |
| 5 point | $\approx 2.83r + 2$ | $\approx 3.46r + 2$ | $4r$ | $3r + 2$ |
| 7 point | $\approx 4.24r + 2$ | $\approx 3.46r + 3$ | $4r + 2$ | $4r + 2$ |
| 9 star | $\approx 5.66r + 2$ | $\approx 6.93r + 4$ | $4r + 2$ | $5r + 2$ |
| 9 cross | $\approx 5.66r + 8$ | $\approx 4.62r + 4$ | $8r$ | $6r + 4$ |
| 13 point | $\approx 5.66r + 8$ | $\approx 6.93r + 6$ | $8r + 4$ | $6r + 8$ |

Table 1: Communication demand for each pair of stencils and tiles.

The oblique hexagon tiling is constructed in two stages. First, for a given $l$, construct a hexagon with $l + 1$ grid points on each side and with the principal diameter making an angle of $-45°$ relative to the x-axis. In figure 2, the resulting hexagons are indicated with dotted lines. In order to form a partition of the problem, we must assign the grid points which lie on the dotted lines to a single tile. We accomplish this by moving the dotted lines down or to the left, or both, as appropriate. The solid lines in figure 2 indicate the resulting partition. The area of the $l$-hexagon, is $3l^2$, giving $l = \frac{r}{\sqrt{3}}$.

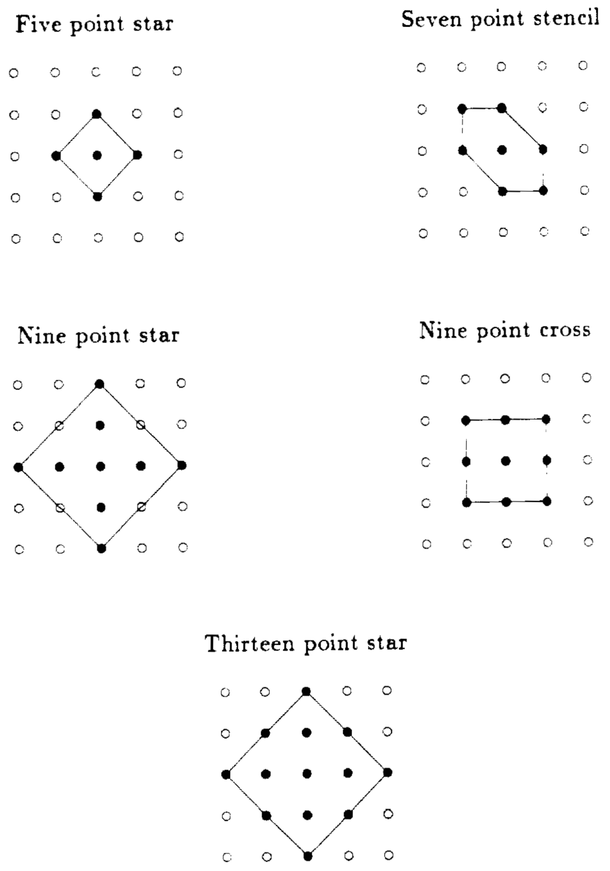
## CONCLUSION

For each of the stencils we have calculated the number of grid points whose values must be communicated to neighboring tiles, for both the diamond and oblique hexagon tilings. The results are summarized in table 1, along with the results obtained in reference [2] for the square and hexagonal tilings. As stressed in the introduction, the correct measure of communication demand is the amount of computation per word transferred. For this problem decomposition, the amount of computation is proportional to the number of grid points per tile, or $r^2$, with an additional dependency on the stencil size which is irrelevant for our purposes. Thus, the results presented in each column of the table are for identical amounts of computation.

## REFERENCES

[1] D. Vrasolovic, E. F. Gehringer, Z. Z. Segall, and D. P. Siewiorek. The influence of parallel deomposition strategies on the performance of multiprocessor systems. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 396–405, ACM, 1985.

[2] D. A. Reed, L. M. Adams, and M. L. Patrick. Stencils and problem partitionings: their influence on the performance of mulitple processor systems. *IEEE Transactions on Computers*, C-36:845–858, 1987.


Five point star

Seven point stencil

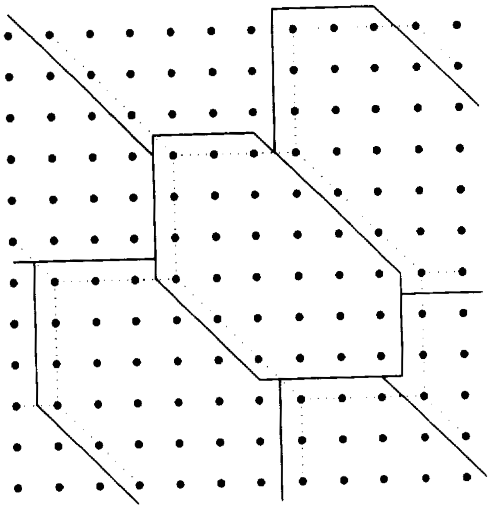
Nine point star

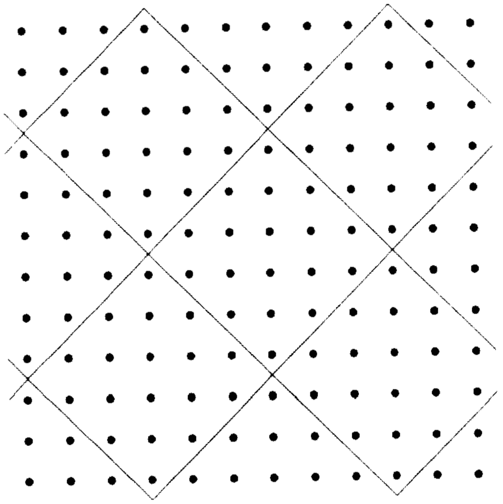
Nine point cross

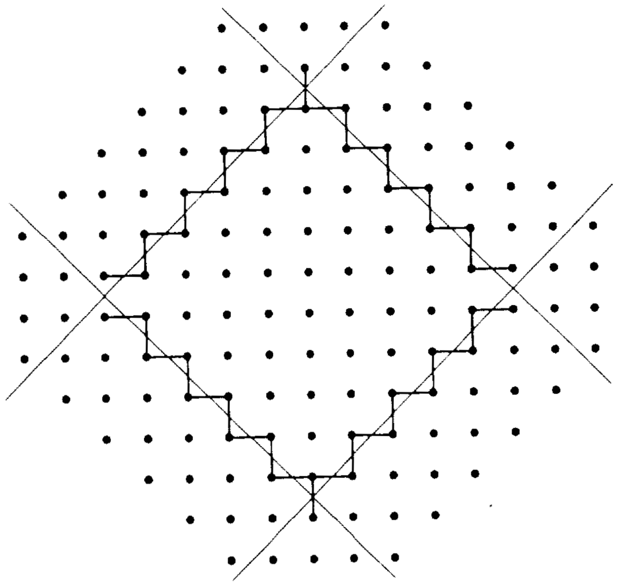
Thirteen point star

Common stencils for iterative PDE solvers.

Convex hulls of each stencil type.


Construction of the oblique hexagon tiling.


Decomposition into diamonds.


Transfers needed with five point stencil and diamond tiling.

# PARALLEL ALGORITHMS FOR DIRECT SOLUTION OF LARGE SYSTEMS OF EQUATIONS

Jian-She Wang      Nathan Ida

Electrical Engineering Department
The University of Akron
Akron, Ohio 44325

## ABSTRACT

A number of direct solution algorithms have been parallelized for use in conjunction with finite element analysis of large engineering problems. Parallel solution algorithms based on the Gauss-Jordan and Gauss elimination were implemented and compared. These parallel solvers are applied to large, dense or banded systems of equation arising from finite element analysis of 2-D and 3-D electromagnetic field problems. Both real and complex matrices are considered with emphasis on very large systems. The speedup obtained by parallelization on the MPP compared to sequential computers is almost three orders of magnitude. Although the MPP is used for implementation, most aspects of parallelization are general.

## INTRODUCTION

In engineering applications it is often necessary to solve large systems of equations that are either too large or require too much computer resources to be economically feasible on standard computers. For this type of problem a parallel machine is very attractive. The type of systems considered are those arising from the application of the finite element method (FEM) to engineering applications. The FEM is particularly computationally intensive, yet its various parts are either intrinsically parallel or can be parallelized. By using a parallel processor, considerably faster solution times can be achieved or, alternatively, larger problems can be solved.

The Gauss elimination and the Gauss-Jordan methods have been chosen for this work because of their extensive use in finite element applications. In most cases, dense, nonsymmetric, real systems are solved but similar methods for banded and complex systems are presented. Sparse systems are not considered here although, these can obviously be handled.

The MPP has been described elsewhere [1,2] in detail. For the purpose of this work, the MPP is configured as an 128*128 array with a 32 bit word length. For the solution of linear systems, the two most important aspects related to the MPP are the number of memory planes in the ARray Unit (ARU) and the size of the staging memory. The ARU contains 900 usable bit planes of memory. This limits the number of real arrays (128*128, 32 bit) in the ARU to 28. The staging memory is limited to 512 real arrays. Parallel Pascal callable I/O procedures can transfer only one 128*128 array in or out of the ARU at any one time. This makes it necessary for any array larger than 128*128 to be blocked into sub-arrays of 128*128. Thus, the smallest system considered is a 128*128 system of equations.

## A PARALLEL GAUSS-JORDAN ALGORITHM

For a system of equations of the form [A]{X}={B}, the parallel implementation of the Gauss-Jordan algorithm begins by loading [A] into one array and the right hand side (RHS) {B} into the first column of a second array. Assuming that the first column in [A] has been eliminated, these arrays look as:

$$
\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ & a'_{22} & a'_{23} & \cdots & a'_{2n} \\ & a'_{32} & a'_{33} & \cdots & a'_{3n} \\ & . & . & & . \\ & a'_{n2} & a'_{n3} & \cdots & a'_{nn} \end{bmatrix}_{1a}
\qquad
\begin{bmatrix} b_1 & 0 & 0 & \cdots & 0 \\ b'_2 & 0 & 0 & \cdots & 0 \\ b'_3 & 0 & 0 & \cdots & 0 \\ . & . & . & & . \\ b'_n & 0 & 0 & & 0 \end{bmatrix} \quad 1b
$$

where n=128. The prime indicates that the corresponding coefficients have been modified during elimination of the first column. To eliminate the second column, a pivot row array and a pivot element array are created using row and column broadcasting routines.

$$
\begin{bmatrix} 0 & a'_{22} & a'_{23} & \cdots & a'_{2n} \\ 0 & a'_{22} & a'_{23} & \cdots & a'_{2n} \\ 0 & a'_{22} & a'_{23} & \cdots & a'_{2n} \\ . & . & . & & . \\ 0 & a'_{22} & a'_{23} & \cdots & a'_{2n} \end{bmatrix}_{2a}
\quad
\begin{bmatrix} a'_{22} & a'_{22} & \cdots & a'_{22} \\ a'_{22} & a'_{22} & \cdots & a'_{22} \\ a'_{22} & a'_{22} & \cdots & a'_{22} \\ . & . & & . \\ a'_{22} & a'_{22} & \cdots & a'_{22} \end{bmatrix}_{2b}
$$

A pivot column array is created from (1a) as

$$
\begin{bmatrix} a_{12} & a_{12} & a_{12} & \cdots & a_{12} \\ a'_{22} & a'_{22} & a'_{22} & \cdots & a'_{22} \\ a'_{32} & a'_{32} & a'_{32} & \cdots & a'_{32} \\ . & . & . & & . \\ a'_{n2} & a'_{n2} & a'_{n2} & \cdots & a'_{n2} \end{bmatrix} \tag{3}
$$

Eq. (3) is divided by Eq. (2b) and multiplied by Eq. (2a) to create a modifier array

$$
\begin{bmatrix} 0 & a_{12}a'_{22}/a'_{22} & a_{12}a'_{23}/a'_{22} \cdots & a_{12}a'_{2n}/a'_{22} \\ 0 & a'_{22}a'_{22}/a'_{22} & a'_{22}a'_{23}/a'_{22} \cdots & a'_{22}a'_{2n}/a'_{22} \\ 0 & a'_{32}a'_{22}/a'_{22} & a'_{32}a'_{23}/a'_{22} \cdots & a'_{32}a'_{2n}/a'_{22} \\ & . & . \qquad\quad . & . \\ 0 & a'_{n2}a'_{22}/a'_{22} & a'_{n2}a'_{23}/a'_{22} \cdots & a'_{n2}a'_{2n}/a'_{22} \end{bmatrix} \tag{4}
$$

This array, with the exception of the pivot

CH2649-2/89/0000/0231$01.00 © 1988 IEEE

row, is subtracted from the original array in (1a). The result is a new coefficient array

$$\begin{bmatrix} a_{11} & 0 & a''_{13} & \cdots & a''_{1n} \\ 0 & a'_{22} & a'_{23} & \cdots & a'_{2n} \\ 0 & 0 & a''_{33} & \cdots & a''_{3n} \\ 0 & 0 & a''_{43} & \cdots & a''_{4n} \\ . & . & . & & . \\ 0 & 0 & a''_{n3} & \cdots & a''_{nn} \end{bmatrix} \tag{5}$$

The modification of the RHS during elimination is similar. Eq. (3) is divided by Eq. (1a) and multiplied by Eq. (1b) to generate an RHS modifier array This is subtracted from Eq. (1b) to obtain the new RHS array.

$$\begin{bmatrix} b_1 - a_{12}b'_2/a'_{22} & 0 & 0 & \cdots & 0 \\ b'_2-a'_{32}b'_2/a'_{22} & 0 & 0 & \cdots & 0 \\ . & . & . & & . \\ b'_n-a'_{n2}b'_2/a'_{22} & 0 & 0 & \cdots & 0 \end{bmatrix} \tag{6}$$

After n elimination steps, the original coefficient matrix is reduced to a diagonal system. To obtain the solution, an array of the diagonals is constructed

$$\begin{bmatrix} a_{11} & a_{11} & a_{11} & \cdots & a_{11} \\ a'_{22} & a'_{22} & a'_{22} & \cdots & a'_{22} \\ a''_{33} & a''_{33} & a''_{33} & \cdots & a''_{33} \\ . & . & . & & . \\ a''_{nn} & a''_{nn} & a''_{nn} & \cdots & a''_{nn} \end{bmatrix} \tag{7}$$

Eq. (6) is divided by Eq. (7), to obtain the unknowns x1 through x128:

## A PARALLEL GAUSS ELIMINATION ALGORITHM

The Gauss elimination algorithm follows similar steps. The steps in Eq. (1) through (4) are identical. In subtracting the modifier array in Eq. (4) from Eq. (1a), only the rows below the pivot row are modified. After (n-1) elimination steps, the original system (1a) is reduced to an equivalent upper triangular system: The right hand side is similarly modified.

$$\begin{bmatrix} a_{11} + & a_{12} + & a_{13} + \cdots + & a_{1n} \\ & a'_{22} + & a'_{23} + \cdots + & a'_{2n} \\ & & a''_{33} + \cdots + & a''_{3n} \\ & & & . \\ & & & a^{n}_{nn} \end{bmatrix}_{8a} \quad \begin{bmatrix} b_1 & 0 & 0 & \cdots & 0 \\ b'_2 & 0 & 0 & \cdots & 0 \\ b''_3 & 0 & 0 & \cdots & 0 \\ . & . & . & & . \\ b^{n}_n & 0 & 0 & & 0 \end{bmatrix}_{8b}$$

The solution of the system in Eq. (8) is performed using the following algorithm

$$x_i=b_i/a_{ii}, \quad b_k=b_k-a_{ki}x_i \tag{9}$$

where i=n,n-1,...,1 and k=i-1,i-2,...,1. In this algorithm, once an unknown is backsubstituted, the upper triangular system is reduced in order by one and then the RHS is modified.

A pivot column and a pivot element array are created as

$$\begin{bmatrix} a_{1i} & a_{1i} & a_{1i} & \cdots & a_{1i} \\ a_{2i} & a_{2i} & a_{2i} & \cdots & a_{2i} \\ a_{3i} & a_{3i} & a_{3i} & \cdots & a_{3i} \\ . & . & . & & \\ a_{ii} & a_{ii} & a_{ii} & \cdots & a_{ii} \\ 0 & 0 & 0 & \cdots & 0 \end{bmatrix}_{10a} \quad \begin{bmatrix} a_{ii} & a_{ii} & a_{ii} & \cdots & a_{ii} \\ a_{ii} & a_{ii} & a_{ii} & \cdots & a_{ii} \\ . & . & . & & . \\ . & . & . & & . \\ . & . & . & & . \\ a_{ii} & a_{ii} & a_{ii} & \cdots & a_{ii} \end{bmatrix}_{10b}$$

The RHS is divided by the pivot element array (masked operation) to solve for the ith unknown. From this, an RHS pivot array is generated. These arrays are:

$$\begin{bmatrix} b_1 & 0 & 0 & \cdots & 0 \\ b_2 & 0 & 0 & \cdots & 0 \\ . & . & . & \cdots & . \\ x_i & 0 & 0 & \cdots & 0 \\ . & . & . & & . \\ x_n & 0 & 0 & & 0 \end{bmatrix} \quad \begin{bmatrix} x_i & 0 & 0 & \cdots & 0 \\ x_i & 0 & 0 & \cdots & 0 \\ . & . & . & \cdots & . \\ x_i & 0 & 0 & \cdots & 0 \\ . & . & . & \cdots & . \\ x_i & 0 & 0 & \cdots & 0 \end{bmatrix} \tag{11}$$

Multiplication of Eq. (11b) by Eq. (10b) results in a modifier array:

$$\begin{bmatrix} a_{1i}x_i & 0 & 0 & \cdots & 0 \\ a_{2i}x_i & 0 & 0 & \cdots & 0 \\ a_{3i}x_i & 0 & 0 & \cdots & 0 \\ . & . & . & \cdots & . \\ a_{ii}x_i & 0 & 0 & \cdots & . \\ 0 & 0 & 0 & \cdots & 0 \end{bmatrix} \tag{12}$$

The modifier array in Eq. (12) is now subtracted from the RHS. After n=128 steps, the RHS array contains the n unknowns in its first column.

## BLOCK GAUSS-JORDAN AND GAUSS ELIMINATION

For the solution of any system with order larger than 128, the coefficient matrix is blocked in subarrays of 128*128. For each subarray the algorithm described in Eq. (1) through (7) is applied. A 512*512 system is chosen as an example since this is the largest array the ARU can handle. Any larger matrices will have to utilize the stager. In Fig. 1, the 512*512 coefficient matrix is blocked into 4*4 subarrays, while the RHS vector is stored in the first column of 4 corresponding subarrays or in the first four columns of one sub-array.

The Gauss elimination solution for a 512*512 system is similar to that of the Gauss-Jordan method described above other than the obvious changes described in Eq. (8) through (12).

Table 1 summarizes the number of operations required for solution on a sequential machine and on the MPP. Table 2 summarizes the solution times for a 128*128 and a 512*512 system of linear equations using the Gauss-Jordan and Gauss elimination methods on the MPP. The results are compared with those obtained for the same systems on a MicroVaxII computer. The highest speedup is achieved for a 512*512 system (largest problem that can reside in the ARU). The backsubstitution is the slowest of the two parts (essentially a sequential operation).

## SOLUTION OF BANDED SYSTEMS OF EQUATIONS

Fig. 2 shows the coefficient data structure of a 512*512 system of equations arising from finite element analysis, where only the shaded area has non-zero terms (semi-bandwidth less than or equal to 128). In Fig. 2a, 37.5% of the memory storage can be saved by considering only the non-zero blocks. In fig. 2b, 65.6% of the memory can be saved. With this storage scheme, a 1024*1024 matrix with a semi-bandwidth of 128 can reside in the ARU.

The solution times for a parallel, banded elimination algorithm are summarized in Table 3 and compared with those for full coefficient matrix of the same order (512*512).

## SOLUTION OF SYSTEMS WITH COMPLEX COEFFICIENTS

Application of the finite element method to the solution of eddy current problems in electromagnetic fields results in the following system of complex linear equations:

$$( A + jB )( X + jY ) = ( C + jD ) \tag{13}$$

On the MPP, complex data is stored in two sets of arrays. Complex calculations are resolved into two or more real parallel array operations. The basic operations required are implemented as:

$$\begin{aligned} &P+Q=(P1+Q1)+j(P1+Q1) \quad P-Q=(P1-Q1)+j(P1-Q1) \\ &P*Q=(P1*Q1-P2*Q2)+j(P1*Q2+P2*Q1) \\ &P/Q=((P1*Q1+P2*Q2)/(Q1*Q1+Q2*Q2)) \\ &\quad +j((P2*Q1-P1*Q2)/(Q1*Q1+Q2*Q2)) \end{aligned} \tag{14}$$

For the solution of a system of complex linear equations with order higher than 128, the complex coefficient matrix is blocked into subarrays of 128*128.

The solution times for a 128*128 and a 256*256 system of complex linear equations by the Gauss-Jordan and Gauss elimination methods are summarized and compared with those for solution of the same order system of real equations on the MPP. The results are shown in Table 4.

The solution time for a system of complex linear equations by Gauss' and Jordan's methods is about 4 to 5 times that needed to solve the same order system of real equations.

## SOLUTION OF LARGE SYSTEMS OF EQUATIONS

For problems of size larger than the capacity of the ARU, the stager must be used. The matrix is again subdivided into blocks of 128*128. Once all subarrays in the matrix are in correct stager addresses, part of the arrays are sent to the ARU for processing. The results are returned to the same stager addresses. This is repeated until the system is solved.

The division into subarrays is the same as in Fig. 2b for banded systems and as in Fig. 1 for dense systems except for the larger number of subarrays required. The RHS is placed in columns of a single array to save space.

Several banded systems with bandwidth≤128 of selected order ranging from 1024 to 16,384 have been solved on the MPP using the stager. The solution time (including data transfer between stager and ARU) is shown in Table 5. Table 6 gives the largest banded systems with different bandwidth that can be solved on the MPP under the limit of the stager size (32 Mb). Table 7 summarizes the solution times for two large, dense systems of equations (nonsymmetric).

## CONCLUSIONS

The implementation of solution algorithms on a massively parallel processor is quite efficient as long as the system fits in the ARU. Larger systems can also be solved with reduced efficiency. Even so, the solution is as fast or faster than on vector machines. An increase in size of the array and local memory could significantly improve performance.

## REFERENCES

[1] K.E. Batcher, "Design of a massively parallel processor ", IEEE Transactions on Computers, Vol. C-29, No.9, pp. 837-840, September 1980.

[2] K. E. Batcher, "Architecture of the MPP" IEEE Computer Society on Computer Architecture for Pattern Analysis and Image Database Management Proceedings, pp. 170-174, October 1983.

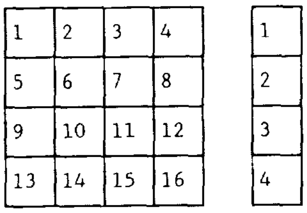


Figure 1 Blocking of a 512*512 system.

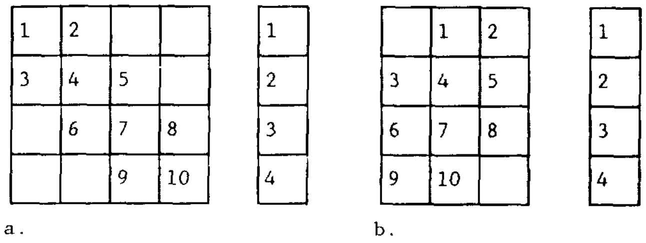


Figure 2. Two methods of blocking a 512*512 system with semi-bandwidth of 128 or smaller

Table 1. Number of operations needed for sequential and parallel solution with the Gauss-Jordan and Gauss elimination algorithms.

| | | Sequential solution | | | Parallel Solution | | |
|---|---|---|---|---|---|---|---|
| Method | Oper. | Diag./Triag. | Solution | Total | Diag./Triag. | Solution | Total |
| Gauss<br>Jordan | A/M<br>D | n(n-1)(n+3)/2<br>n(n-1) | 0<br>n | n(n-1)(n+3)/2<br>$n^2$ | amm(m+3)/2<br>amn | <br>m | amm(m+3) + a/2<br>m(am+1) |
| Gauss<br>Elim. | A/M<br>D | n(nn-1)/3<br>n(n-1)/2 | n(n-1)/2<br>n | n(n-1)(2n+5)/6<br>n(n+1)/2 | am(m+1)(2m+1)/6-1<br>am(m+1)/2-1 | am(m+1)/2-1<br>am | am(m+1)(m+2)/3-1<br>am(m+3)/2-1 |

A=add, M=multiply, D=divide, a=128, m=n/128, n=# of equations in the system.

Table 2. Comparison of solution times for the Gauss-Jordan and Gauss elimination methods on the Microvax II and the MPP. (Times in seconds).

| | Gauss-Jordan | | | Gauss Elimination | | |
|---|---|---|---|---|---|---|
| Order | μVax | MPP | Speedup | μVax | MPP | Speedup |
| 128: | | | | | | |
| Elim. | 11.57 | .07788 | 148 | 7.57 | .07795 | 97 |
| Sol. | 0.01 | .00643 | 1.6 | .12 | .0493 | 2.4 |
| Total | 11.58 | .08431 | 137 | 7.59 | .12927 | 59 |
| 512: | | | | | | |
| Elim. | 3476 | 1.728 | 2011 | 3165 | 1.231 | 2572 |
| Sol. | 0.23 | .02568 | 8.9 | 2.01 | .25615 | 5.6 |
| Total | 3476 | 1.754 | 1982 | 3169 | 1.588 | 1994 |

Table 3 Banded and full matrix solution times using Gauss elimination on the MPP. (in miliseconds).

| Step | Full Matrix | Banded Matrix | Speedup |
|---|---|---|---|
| Elimination | 1230.56 | 724.26 | 1.699 |
| Solution | 356.15 | 277.44 | 1.284 |
| Total | 1585.70 | 1001.71 | 1.584 |

Table 4. Comparison of solutions in real and complex variables on the MPP. (miliseconds).

| | Gauss Jordan | | | Gauss Elimination | | |
|---|---|---|---|---|---|---|
| Order | Real | Complex | Ratio | Real | Complex | Ratio |
| 128: | | | | | | |
| Elim. | 77.88 | 334.89 | 4.3 | 77.95 | 334.96 | 4.3 |
| Sol. | 6.43 | 12.43 | 1.9 | 49.30 | 244.58 | 4.96 |
| Total | 84.31 | 347.32 | 4.1 | 128.27 | 579.55 | 4.52 |
| 256: | | | | | | |
| Elim. | 343.68 | 1513.80 | 4.4 | 277.76 | 1196.30 | 4.3 |
| Sol. | 12.28 | 24.88 | 2.0 | 125.61 | 576.48 | 4.59 |
| Total | 355.96 | 1538.68 | 4.3 | 403.37 | 1772.77 | 4.4 |

Table 5. Solution times on the MPP for banded systems of different sizes. Semi-bandwidth is 128. (Time is in seconds).

| Size | Elimination | Solution | Total |
|---|---|---|---|
| 1024 | 1.556 | 0.651 | 2.206 |
| 2048 | 3.300 | 1.371 | 4.671 |
| 3072 | 5.015 | 2.077 | 7.093 |
| 4096 | 6.731 | 2.784 | 9.515 |
| 8192 | 13.594 | 5.611 | 19.205 |
| 12288 | 20.458 | 8.438 | 28.896 |
| 16384 | 27.321 | 11.264 | 38.264 |

Table 6. Largest systems solvable on the MPP

| Semi-bandwidth | Size of Systems |
|---|---|
| 128 | 21888 |
| 256 | 13184 |
| 384 | 9472 |
| 512 | 7552 |
| 1024 | 4352 |
| 2048 | 2944 |
| 2816 | 2816 |

Table 7. Solution of large, dense, nonsymmetric systems on the MPP. Time is in seconds.

| Size | Elimination | Solution | Total | CRAY X/MP |
|---|---|---|---|---|
| 1024x1024 | 7.809 | 1.327 | 9.136 | 36.523 |
| 2048x2048 | 51.353 | 4.638 | 55.990 | —— |

# SECTION II: APPLICATIONS

## Part 1: Oral Presentations

# STOCHASTIC SIMULATION OF CHARGED PARTICLE TRANSPORT ON THE MASSIVELY PARALLEL PROCESSOR

James A. Earl

Department of Physics and Astronomy
University of Maryland, College Park MD 20742

## ABSTRACT

Computations of cosmic-ray transport based upon finite-diference methods are afflicted by instabilities, inaccuracies, and artifacts. To avoid these problems, we have developed a Monte Carlo formulation which is closely related not only to the finite-difference formulation, but also to the underlying physics of transport phenomena. Implementations of this approach are currently running on the Massively Parallel Processor at Goddard Space Flight Center, whose enormous computing power overcomes the poor statistical accuracy that usually limits the use of stochastic methods. These simulations have progressed to a stage where they provide a useful and realistic picture of solar energetic particle propagation in interplanetary space.

Keywords: Cosmic-rays, Particle Transport, Interplanetary Medium.

## INTRODUCTION

The diffusion idealization, which has been almost universally invoked in discussions of cosmic-ray transport is easy to treat analytically. However, many observed phenomena give clear evidence for non-diffusive effects. One example is the so-called "scatter free" propagation of kilovolt solar electrons (Ref. 1), which is inconsistent with diffusion, but which can readily be interpreted in terms of a coherent mode of propagation. This mode is novel, but it is just a manifestatation in a dynamic situation of non-diffusive effects similar to those considered in the steady-state by classical transport theory (Ref. 2). Although these effects have been described analytically in References 3 and 4, the theory is very complicated. Consequently, there is a need for reliable numerical computations which bypass these complexities and yield concrete results suitable for comparison with observations. This paper describes computations of charged particle transport along a large-scale guiding magnetic field B whose spatial variations are characterized by the focusing length L, which is defined by

$$\frac{1}{L} = -\frac{1}{B}\frac{\partial B}{\partial z}, \tag{1}$$

where z is distance parallel to the guiding field. Particle trajectories are scattered by small-scale random magnetic fields whose effect is described by the mean free path λ. Note that the magnetic fields are visualized as static and that there is no interaction among particles in an extremely tenuous distribution of charged particles. This situation differs from those considered by plasma physics, but it is closely analogous to those treated by classical transport theory.

Over the past three years the computations have evolved through simulations of rectilinear transport along a constant guiding field, which were presented at the First Symposium on Massively Parallel Scientific Computation (Ref. 5), and of focused transport with constant focusing length and mean free path (Ref. 6), to the present simulations, which allow arbitrary variations of the focusing length and mean free path. In the interplanetary context, this formulation includes all important effects except those of convective motion of the background medium, which significantly affect the slow variations of cosmic-ray modulation, but which play a minor role during the rapid evolution of solar particle events. It includes the two essential aspects of charged particle transport. These are a strong inhibition of transport perpendicular to the guiding field and a strong anisotropy of the pitch-angle scattering by random fields.

## TRANSPORT EQUATIONS

Under the circumstances outlined above, particle transport is described by

$$\frac{\partial h}{\partial s} + \mu\frac{\partial h}{\partial z} = \frac{\partial}{\partial \mu}\psi e^{G}\frac{\partial}{\partial \mu}he^{-G}, \tag{2}$$

in which h is the number of particles per unit distance parallel to the guiding field, μ is the cosine of the pitch-angle. The parameter s = Vt, where V is particle velocity, plays the role of a temporal variable. The Fokker-Planck coefficient of pitch-angle scattering is given by

$$\psi = \frac{3(1-\mu^2)|\mu|^{q-1}}{2\lambda(2-q)(4-q)}, \tag{3}$$

where q is an index that measures the anisotropy of scattering (Ref. 7). The function G that appears in Equation 2 is defined in terms of ψ by

$$G(\mu) = \frac{1}{L}\int_0^{\mu}\frac{1-\nu^2}{\psi(\nu)}d\nu = \frac{(4-q)}{3}\frac{\lambda}{L}\mu|\mu|^{1-q}. \tag{4}$$

CH2649-2/89/0000/0237$01.00 © 1988 IEEE

In the discrete formulation, the continuous variables are replaced by a three-dimensional grid whose spacings are $\Delta z$, $\Delta\mu$, and $\Delta s$, and the derivatives appearing in equation (1) are replaced by their finite-difference analogs. The effect of these replacements is best described in terms of the particles flowing in and out of a cell, whose dimensions are $\Delta\mu$ and $\Delta z$, during a temporal increment $\Delta s$. These flows are illustrated in Figure 1, where the integer M refers to the pitch angle, and the integer Z refers to the distance z. Here, particles passing through the top and bottom of the box are those whose pitch angle changes because of scattering, and particles passing through the sides are those whose distance changes as a result of their motion parallel to the guiding field. The flows due to scattering can be described in terms of coefficients $p_M$ and $q_M$, which characterize the flows out of box M toward larger and smaller values of $\mu$, respectively, and whose numerical values can be derived from Equation 2. The difference equations that correspond to Figure 1 can be solved by standard numerical methods, but these methods are difficult to implement and subject to subtle errors, which are discussed in Reference 8.

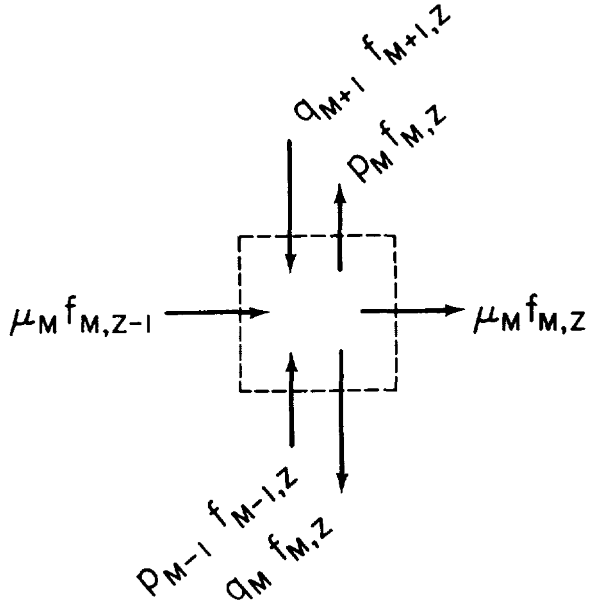


Figure 1. The number of particles in each cell of the $\mu$ - z plane changes, because scattering causes particles to flow in and out through the top and bottom boundaries, while motions in space cause them to flow through the vertical boundaries.

In the Monte Carlo formulation, the random history of a large number of particles is followed under the assumption that the coefficients p and q can be interpreted as probabilities in each temporal step that $\mu$ will change by $\Delta\mu$ toward more forward or more backward directions, respectively. Obviously, this defines 1-p-q as the probability that a pitch angle will not change. After their pitch angles are updated in each step, the particles move a distance $\mu\ \Delta s$.

In Figure 2, the transition probabilities are plotted as a function of $\mu$ for strongly anisotropic scattering, q = 1.8, similar to that occuring in interplanetary space and for strong focusing, $\lambda/L = 5$. These probabilities exhibit the same gross features as the Fokker - Planck coefficient, but there is a significant assymetry such that p is consistently larger than q. This leads to the systematic drift toward forward directions that is expected as a result of focusing. Note that the probability of going backward through $\mu = 0$ is only 0.13% in this example. This leads to a coherent mode of transport that is very different from diffusion, for any particle that reaches the forward hemisphere has a very small chance of going backward again.

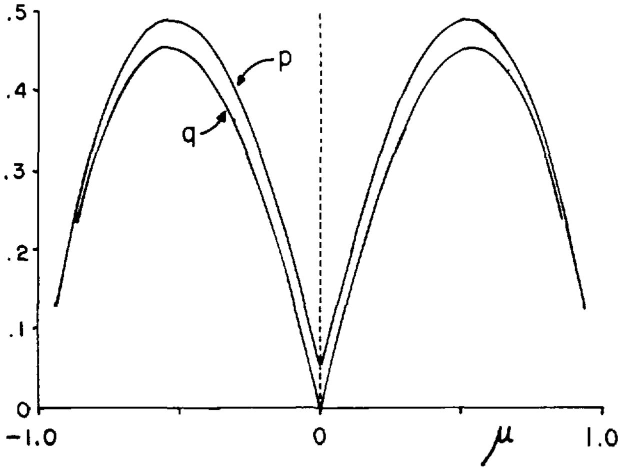


Figure 2. Forward and backward transition probabilites plotted as a function of pitch-angle cosine.

## THE ALGORITHM

To implement the Monte Carlo scheme outlined above, each particle was assigned an integer distance and an index M corresponding to $\mu$ that lies between 0 and 8. Because these parameters occupy only three bytes for each particle, there was plenty of storage for several parallel arrays of particles. Consequently, the results given below are based on 32 arrays which contained 32 * 16384 = 524288 particles. The fate of the particles was determined by a single parallel array of random integers ( ranging from -32767 to +32767) that was updated frequently. To implement changes in the pitch-angle cosine, forward and backward integers were assigned to each particle, according to its pitch angle and distance, in such a way that the probability of the current random number being larger than the integer is the corresponding forward or backward probability analogous to those poltted in Figure 2. Then the angular index was incremented for those particles whose current random integer was positive and greater than the forward integer, and decremented for those whose random integer was negative and less than the backward integer with its sign reversed. This approach satisfies the basic requirement that the probabilities of incrementing, decrementing and leaving unchanged the pitch-angle must add to unity. After the pitch-angles had been updated, each particle's distance was incremented. When the desired number of temporal steps had been carried out, particles were binned according to distance and pitch-angle.

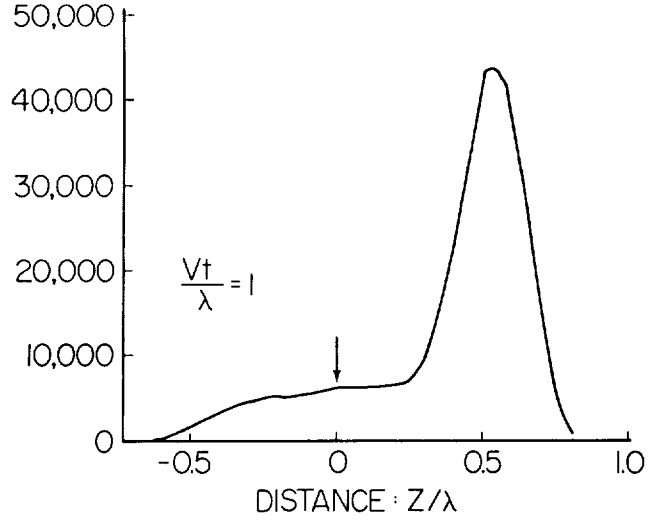


Figure 3. Density profile for rectilinear transport after a collimated injection at the arrow.

## SPATIAL AND TEMPORAL PROFILES

To set the stage for the simulations of interplanetary transport that are the main subject of this paper, it is useful to consider first the case of rectilinear transport, in which the guiding field is constant, and focusing does not occur. Figure 3 presents results obtained from the MPP as plots of the total number of particles in each distance bin. This sum over pitch-angles is a measure of the isotropic particle density. Because the total number of particles was large, statistical errors are small and, consequently, are not shown explicitly. However, slight irregularities in some parts of the curves give an indication of their magnitude.

This density profile describes a situation very shortly after the injection of a collimated beam of when the particles have had time to move a maximum distance of only one mean free path. The curve exhibits two features: an intense localized peak at the right and a broad wake spread on both sides of the arrow which indicates the point of injection. Qualitatively, the peak appears because particles become nearly uniformly distributed in the forward hemisphere, while very few particles penetrate to the backward hemisphere through the region of weak scattering at $\mu = 0$. This means that the particles in the forward hemisphere move with nearly the same average velocity parallel to the field, but statistical fluctuations in individual velocities give rise to a peak centered around the average displacement. Such features are designated as coherent pulses. They decay exponentially as particles slowly escape into the backward hemisphere and join the wake.

In the spiral pattern of the interplanetary magnetic field, the focusing assymetry that appears in Figure 2 tends to keep particles in the forward hemisphere. Consequently, the coherent pulse is long lived and intense relative to the wake. Moreover, the assymetry becomes very pronounced near the sun, for the ratio $\lambda/L$ varies approximately as $1/r$, where r is distance from the sun.

To illustrate these effects, Figure 4 shows snapshot profiles of number of particles vs. r for four equally spaced times after injection into a model interplanetary field at $r = 0$. These profiles describe a moving pulse whose width increases as it moves out, which is the qualitative behavior expected. However, the exact evolution of this width and the deceleration of the peak, which is evident from a careful examination of the figure, are details which crucially affect observations, but which are not adequately described by the analytic theory of focused transport.

In contrast with the rectilinear profile illustrated by Figure 2, the wake is virtually invisible in all of the four profiles. As was discussed above, this enhancement of the coherent mode is a consequence of focusing.

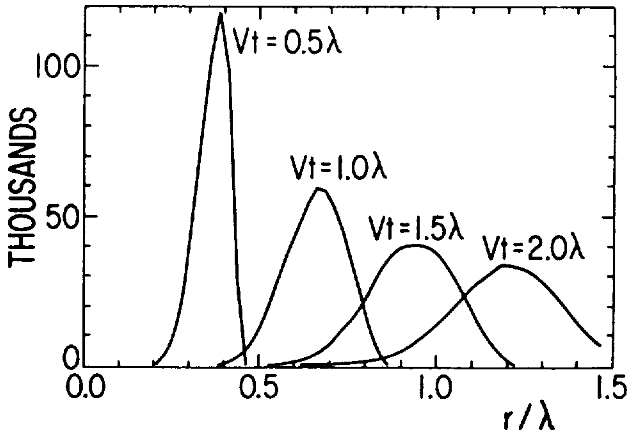


Figure 4. Profiles of isotropic particle density vs. distance from the sun for four different times after injection.

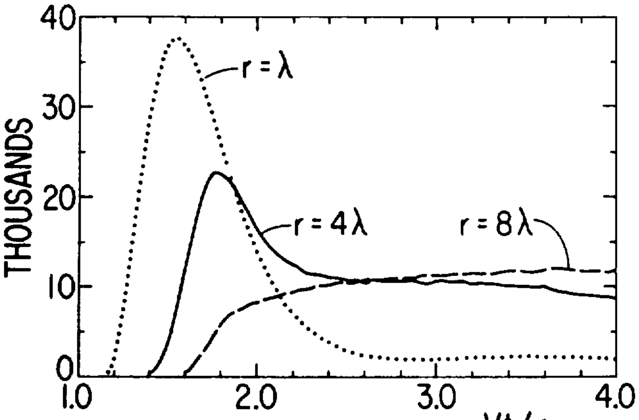


Figure 5. Profiles of intensity vs. time at a fixed distance.

Observations from spacecraft generally take the form of profiles of intensity vs. time at an essentially fixed position in space. In Figure 5, results from MPP simulations, which are presented in this form, show what happens when the coherent pulse sweeps over an observer at a fixed location. The dotted curve, which refers to a mean free path equal to the distance of the observer from the sun, defines a peak followed by a very small wake. As the mean free path becomes smaller, this pulse becomes less prominent relative to the wake ( solid curve ), until it is submerged in the onset ( dashed curve ). Examples of all three of these behaviors are seen among observed solar particle events, along with events whose extremely slow onset can be understood in terms of standard diffusion theory. Consequently, the MPP results make it clear that all types of prompt solar particle events, diffusive, abrupt onset and coherent, can be understood as morphological stages in a continuous sequence of profiles predicted for increasing mean free paths by the theory of charged particle transport. The same basic physics explains all types.

## ANGULAR DISTRIBUTIONS

The discussion above has focused on isotropic intensities in which particles are counted regardless of their directions. In fact both predicted and observed intensities are often highly anisotropic, and valuable information can be derived from a careful analysis of these anisotorpies. To illustrate this point, Figure 6 presents numbers of particles binned according to $\mu$ and summed over a coherent pulse. Evidently, the intensity in the forward hemisphere $\mu > 0$ is overwhelming larger than that in the backward hemisphere. The solid curve which gives the result of a MPP simulation is in very good agreement with the dashed curve, which gives the function exp ( G ) which is expected from theory.

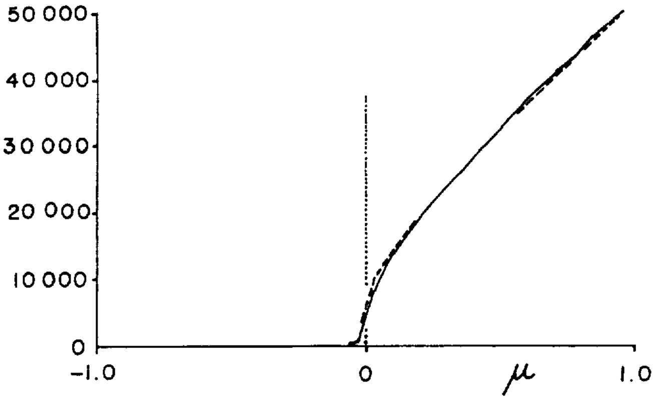


Figure 5. Angular distributions for particles summed over the coherent pulse.

## CONCLUSIONS

Results obtained on the MPP with the aid of Monte Carlo methods are equivalent in every detail to those based upon careful use of more traditional methods, but they are less subject to error and are closer to the physics. These characteristics offer tremendous advantages in the investigation of exotic transport regimes for which no theoretical description is available. In particular, the formulation of problems in which particles gain or lose energy leads to prohibitively large conventional computations, but their Monte Carlo versions are not significantly more complicated than the one described here. We intend to exploit these advantages in the investigation of two such problems: Adiabatic deceleration of cosmic-rays due to expansion of the solar-wind, and the loss of energy by electrons in radio sources due to synchrotron radiation. On a more immediate time scale, the current simulations, which embody an accurate and useful description of solar particle events, will be applied to the interpretation of observations.

## ACKNOWLEDGEMENTS

This research was supported by NASA under Grant NGR-21-002-066.

## REFERENCES

1. Lin, R. P., "Emission and Propagation of ~40 keV Solar Flare Electrons," Solar Physics, 15, 453, 1970.

2. Case, K. M., and Zweifel, P. F., "Linear Transport Theory", Addison-Wesley, Reading MA, 1967.

3. Earl, J. A., "Coherent Propagation of Charged Particle Bunches in Random Magnetic Fields", Astrophysical Journal, 188, 379, 1974.

4. Earl, J. A., "Analytical Description of Charged Particle Transport Along Arbitrarary Guiding Field Configurations", Astrophysical Journal, 251, 739, 1981.

5. Earl, J. A., "Simulation of Charged Particle Transport on the Massively Parallel Processsor", First Symposium on th Frontiers of Massively Parallel Scientific Computation, NASA Goddard Space Flight Center, Sept. 24 - 25, 1986

6. Earl, J. A., "Charged Particle Transport Calculations on the Massively Parallel Processor", Conference Papers. 20 th International Cosmic Ray Conference, Moscow, 3, 198, 1987

7. Jokipii, J. R., "Cosmic-Ray Propagation I, Charged Particles in a Random Magnetic Field", Astrophysical Journal, 146, 480, 1966.

8. Kota, J., Merenyi, E., Jokipii, J. R., Gombosi, T. I., and Owens, A. J., "A Numerical Study of the Pitch-Angle Scattering of Cosmic-Rays", Astrophysical Journal, 254, 398, 1982.

# SUITABILITY OF SIMULATION OF A POPULATION OF CHEMICAL POLYMERS ON THE MASSIVELY PARALLEL PROCESSOR

David Marshall Cohen

Department of Computer Science
The University of Iowa, Iowa City, IA 52242

## ABSTRACT

We describe our investigation of the suitability of the Massively Parallel Processor (MPP) for the execution of a program which simulates the degradation of the heparin polymer by the enzyme heparinase. The architecture of the MPP offers massive fine-grained parallelism, with the following limitations: (1) There is a single instruction stream, i.e., each of the 16,384 processing elements (PE's) executes the same instruction. (2) Each PE communicates directly with only four neighboring PE's, and the opposite edges of the grid of PE's are logically connected to each other. (3) Each PE has 1024 bits of local random access memory as well as (slower) access to a large (20 Mbyte) Staging Memory.

We report on the use of the MPP Pascal programming language to implement the simulation program, including the representation of polymer chains, the cleaving of chains into smaller chains, and the search for (stable) chains which are impervious to the actions of heparinase.

Keywords: Massively Parallel Processor, Polymer, Simulation, MPP Pascal, Heparin, Depolymerization

## INTRODUCTION

The Massively Parallel Processor (MPP) was built by Goodyear Aerospace Corporation for the NASA/Goddard Space Flight Center, which utilized it for processing data sent to earth from orbiting satellites (Ref. 1). In addition to image processing applications which included algorithms such as the Fast Fourier Transform and maximum likelihood classification, the MPP has been used for a wide variety of applications, including simulations of plasma electrodynamics, Ising spin exchanges, neural networks, and chemically reactive flows in two and three dimensions (Refs. 2-6).

We have conducted a test of the use of the MPP to simulate the enzymatically catalyzed degradation of a population of chemical (polymer) chains. The design of the algorithm to fit the computer architecture and the use of the MPP Pascal Programming language (Refs 7-8) to code the algorithm will be illustrated. We describe the algorithm and the MPP Pascal code for the algorithm in some detail.

## HEPARIN DEPOLYMERIZATION

Heparin is an acidic mucopolysaccharide mixture that is found in vertebrate blood. Despite its widespread clinical use as an anticoagulant, the exact structure of heparin remains unknown (Refs. 9,10). Heparin molecules vary in length (polydispersity) and in composition (microhetereogeneity). Rice and Linhardt (Ref. 11) have demonstrated that after degradation by the enzyme heparinase, five types of chemical molecules can be isolated, which together make up approximately 80% of the heparin molecules. These so-called "fundamental fragments" form the basic units of our simulation, and for purposes of discussion will be referred to as units F1, F2, F3, F4 and F5.

Chemically, the fundamental fragments are well-characterized oligosaccharides (Ref. 11), or small chains of sugar molecules. Heparinase is believed to catalyze the dissolution of the bond between any two fundamental fragments, with the exception of free F1-F1 dimer (two unit chain), which is called F4. A molecule of heparinase nonpreferentially ("randomly") attaches to a pair of adjacent fundamental fragments, enables the splitting of the bond at its site of attachment, and releases the product chains, with no net change to the enzyme itself. This process occurs with massive parallelism, owing to the presence of astronomical numbers of heparin and heparinase molecules.

Our objective in performing the simulation was to test the hypothesis that heparin polymer chains are a random arrangement of fundamental fragments. We simulated the degradation of heparin by heparinase, assuming a completely random arrangement of F1, F2, F3, F4, F5, and the same relative molar concentrations as had been measured experimentally (Ref 12). By comparing the simulated rate of appearance of free (unbound) fundamental fragments with observed rates of appearance, we were able to demonstrate that the observed data were not completely consistent with the consequences of the above-mentioned hypothesis.

One aspect of the model in particular makes it difficult to predict the rates of appearance of free fragments without the aid of a computer. As indicated earlier, fundamental fragments F1 through F5 are either impervious to heparinase or are cleaved at such a slow speed that they may be considered stable relative to the time scales of our observations. The complicating factor comes from the fact that F4 equals F1-F1 dimer or $(F1)_2$, provided the dimer is not bound to any other fragments. We do not observe F4 unless it is free. Hence a chain that is composed of four F1 fragments, i.e., $(F1)_4$, has two kinds of cleavable bonds: between the central two F1 units and between the terminal F1 units and their neighbors. If the first kind of bond is cleaved, the result is two F4 units; if the second kind of bond is cleaved, the

CH2649-2/89/0000/0241$01.00 © 1988 IEEE

result is one free F1 unit and a chain of three F1 units, $(F1)_3$, which is subject to further depolymerization and will yield one F1 unit and one F4 unit.

## THE MASSIVELY PARALLEL PROCESSOR

The Massively Parallel Processor belongs to the SIMD (single instruction stream, multiple data stream) category of parallel computers (Ref. 13). It achieves massive parallelism by doing arithmetic or logical operations on many short (1-bit) operands simultaneously.

The MPP is comprised of 5 principal parts (see Fig. 1). The Array Unit (ARU) consists of a 128x128 planar array of bit-slice microprocessors. Each of these processing elements (PE's) contains a variable length shift register (2-20 bits), several 1-bit registers, a full adder, logic circuitry, and 1024 bits of random access memory (Refs. 14-16). Communication between PE's occurs via the grid topology, with opposite edges logically joined together, thus forming a cylinder or a torus.

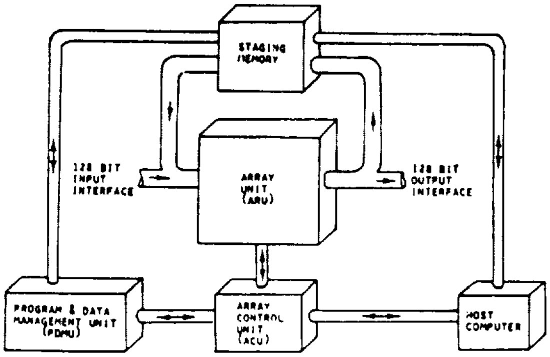


Figure 1. MPP System Block Diagram

The Staging Memory serves as a large (20 Mbyte) random access shared memory for each PE, with the 1024 bits of local memory functioning as a high-speed cache. Communication between the ARU and the Staging Memory is done in parallel, with 128 bits sent simultaneously between the two components. Using pipelining, the 16,384 PE's can all send (or receive) one bit to (or from) the Staging Memory in 12.8 microseconds.

The MPP runs as a peripheral processor, with the host machine currently a DEC VAX-11/780 minicomputer. The program in the host initiates the execution of a program in the MPP and can run concurrently with the MPP. Data may be transmitted between the MPP and the host via a high speed interface.

The Program and Data Management Unit (PDMU) is used for hardware testing and diagnostics. It is a DEC PDP-11/34 minicomputer, which played a larger role when the MPP first came on-line.

The Array Control Unit (ACU) consists of three parts, all of which can operate simultaneously. The Main Control Unit (MCU), a serial processor, performs all of the arithmetic not done by the array of PE's. The program running in the MCU invokes the use of the Array Unit by sending a subroutine call to the Processor Element Control Unit (PECU), the second component of the ACU. The PECU maintains a queue of requests and directs the computations to be performed by the PE's. Lastly, the Input/Output Control Unit (IOCU) handles the transmission of data into and out of the ARU.

Programming may be done in assembler, MPP Pascal, or MPP Parallel FORTH. Assembler language programs are written either in Main Control Language (MCL) for execution on the MCU or the IOCU, or in Processing Element Array Control Language (PEARL) for execution on the ARU (Ref. 17-19). An interactive symbolic debugger is available as well as a simulator of the MPP which run on the UNIX operating system or on the VAX/VMS operating system (Refs. 7,17).

MPP Pascal derives from the programming language Parallel Pascal, which was proposed by Reeves for use on SIMD computers (Ref. 20). We shall discuss MPP Pascal in detail below. John Dorband of the NASA/Goddard Space Flight Center designed and implemented a compiler for the programming language Parallel FORTH for use on the MPP (Ref. 21).

## MPP PASCAL

MPP Pascal is an extension of standard Pascal (Ref. 22) for use on SIMD computers. Aspects of the standard Pascal language that have not been retained include recursive procedures, sets, pointer variables and the character data type (Ref. 7). Three additions to Pascal have been made, namely (1) the "parallel array" data type; (2) the **where** statement; and (3) predefined procedures and functions for use with parallel arrays.

Corresponding bit positions in the local store of the PE's comprise a single "bit plane" of memory. Groups of continuous bit planes can be treated a single bit plane of integer values or reals. MPP Pascal allows for the declaration of parallel arrays of integers, reals, or subranges of integers (of any desired bit length). As one would surmise, all elements of a single parallel array have the same type, length of representation in bits, and location in the local store of the PE's.

MPP Pascal provides the **where** statement as an alternative to iteratively reading values from or writing values to the elements of a parallel array. In fact, owing to the architectural restrictions of the MPP, MPP Pascal does not allow the programmer to address locations in a parallel array by index values; the **where** statement is required.

The syntax of the **where** statement is the following:

```
where < boolean parallel array>
do < statement>
[   otherwise < statement>   ]
```

where < boolean parallel array> is any parallel array of 1-bit integers, with 0 signifying the boolean value of **false** and 1 signifying **true**. For each PE whose entry in the boolean parallel array is **true**, the MPP Pascal statement following the keyword **do** will be executed. If there is an **otherwise** clause in the **where** statement, then each PE whose corresponding entry in the boolean array is **false** will execute the statement following the keyword **otherwise**.

MPP Pascal extends the use of arithmetic operators (+ , -, *, /, **div**, **mod**), arithmetic functions (**abs**, **sqr**, **sin**, **cos**, **exp**, **ln**, **sqrt**, **arctan**), logical operators (**and**, **or**, **not**), and relational operators (= , < > , < , < = , > , > = ) to parallel array operands of appropriate parallel array types. The result of applying an arithmetic operator to two parallel arrays (say *A* and *B*) is a parallel array in which the value of element $[i, j]$ is

$$A[i, j] \quad < \text{op}> \quad B[i, j]$$

where < op> is the operator as it is used in standard Pascal. The result of applying an arithmetic function $f$ to a parallel array $A$ is a parallel array whose $[i, j]$ entry is $f(A[i,j])$. In the case of logical operators or relational operators applied to two parallel arrays (of appropriate type) called $A$ and $B$, the result is a parallel array of boolean (1-bit) values, in which the value of entry $[i, j]$ equals (again)

$$A[i, j] \quad < \text{op}> \quad B[i, j]$$

where < op> is the relational or logical operator as it is used in standard Pascal. In all cases, restrictions on operators and functions (no division by zero, **div** requires integer operands, ln(0.0) is undefined, etc.) carry over to individual elements of parallel array operands (Refs. 22-23).

MPP Pascal provides several predeclared functions for computing a scalar (non-array) value from the elements of a parallel array. Functions **min**, **max**, **sum**, and **prod** return the minimum, maximum, sum and product (respectively) of the elements of the (parallel array) parameter. In the case of a boolean parallel array, functions **all** and **any** return the logical conjunction and logical disjunction (respectively) of the elements of the (parallel array) parameter.

In addition to functions for calculating a single value from a parallel array, MPP Pascal provides three predefined functions for permuting the elements of a parallel array. The functions **shift** and **rotate** perform an end-off shift (filling with zeros) and an end-around shift, respectively. The direction of the rotation or shift is given by the directions of the compass (north, south, east and west) and is specified in the invocation of the function. A matrix transpose operation on a parallel array can be accomplished by invoking the function **transpose**.

The MPP Pascal run-time system automatically creates two parallel arrays of integers, *row_Index* and *col_Index*, whose utility cannot easily be overstated. The value of element $[i, j]$ is $i$ in array *row_Index* and $j$ in *col_Index*. To illustrate the use of these arrays, consider the following problem: we want to assign the value of 0 to all elements of rows 0, 1, and 2 of the (user-declared) parallel array *pixels* and the value of 20 to the rest of the elements. The following MPP Pascal statement suffices:

```
where row_Index < 3
do pixels := 0
otherwise pixels := 20
```

Note that in evaluation of the expression

```
row_Index < 3
```

the compiler generates assembly code that directs the MPP to compare *row_Index* with a parallel array all of whose entries have the value of 3.

## SIMULATION ALGORITHM

### Generation of Initial Population of Chains

**Overview**– The structure of the computations performed during the simulation of the model, together with the particular architecture of the MPP, allow for massive concurrency in the calculations (Refs. 12,24). Each PE represents the location of one oligosaccharide unit in a population of chains of oligosaccharides. (The PE's have been assigned an implicit linear ordering, corresponding to row-major ordering of the elements of the two-dimensional array of PE's.) The identity of each oligosaccharide, the locations of the ends of the polymer chains, and the types of chemical bonds (see below) are stored in the local memories.

The simulation begins with an initialization phase, in which the PE's compute the assignment of identities to the oligosaccharide units, the locations of the termini of the initial population of polymer chains, the types of the bonds between successive oligosaccharide units, and the seeds (initial values) for the random number generator.

**Random Numbers**– The random number generator uses the current value of its parameter (seed) to compute the next pseudorandom number. A linear congruential algorithm (Ref. 25) is implemented, generating uniformly distributed random numbers in the range [0.0, 1.0). Next we demonstrate the use of random numbers to select the identities of the oligosaccharide units.

**Selection of Units**– The choice of oligosaccharide units can be determined by assigning disjoint subintervals of [0.0, 1.0) to each type of oligosaccharide unit, with the proviso that the extent of a subinterval equals the desired fraction of units of that type in the entire population of units. It is readily seen that the collection of subintervals so defined covers the interval [0.0, 1.0). If the random number lies in the interval belonging to type "x" of oligosaccharide, then "x" is chosen for the identity of that particular unit. The (mole) fraction and accompanying subinterval for each of the fundamental fragments of the simulation are shown in Table 1.

Table 1. Mole fractions and intervals used for selecting the type of fundamental fragment to be assigned to a PE.

| Fragment | Fraction | Interval |
|---|---|---|
| F1 | 0.8184 | [0.0000, 0.8184) |
| F2 | 0.0398 | [0.8184, 0.8582) |
| F3 | 0.0589 | [0.8582, 0.9171) |
| F4 | 0.0607 | [0.9171, 0.9778) |
| F5 | 0.0222 | [0.9778, 1.0000) |

**Termini of Subchains**– After initially constructing a single (long) polymer chain, we proceed by dividing the chain into subchains, which will be the initial population of chains in the simulation. As in the case of the selection of the identities of the units, the simulation program uses the random number generator to determine the locations of the termini of subchains. A parameter *fractionTerminators* which has a value between 0.0 and 1.0 serves as the demarcation between the subinterval of [0.0,1.0) determining the presence of a terminus and the one which does not. The value of *fractionTerminators* used in the simulation is 5.91%. Roughly speaking, the value of *fractionTerminators* multiplied by the *populationSize* (a parameter equal to the number of oligosaccharide units in the population) approximates the number of chains that will be formed. In fact, one can calculate (Ref. 26) that if the algorithm for selecting the termini of the initial population of chains is repeated $N$ times (with different values for the initial seeds), then the average number of chains in the initial population will have a value of

$$\text{populationSize} \times \text{fractionTerminators}$$

and a variance of

$$\frac{1}{N} \times \text{populationSize} \times \text{fractionTerminators} \times (1 - \text{fractionTerminators})$$

### Simulation of Heparinase

After establishing an initial population of heparin chains, the simulation of depolymerization begins. The rate of the enzyme-catalyzed chemical reaction obeys the following differential equation (Ref 27):

$$\frac{dx}{dt} = \frac{-\alpha \times x(t)}{\beta + x(t)} \qquad (1)$$

where $x(t)$ is the number of cleavable bonds at time $t$ and time is measured from the start of the depolymerization reaction. The parameters $\alpha$ and $\beta$ are constants which are related to the maximum velocity and the Michaelis constant $K_m$, respectively, of Michaelis-Menten kinetics (Ref. 27). The number $\Delta x(t)$ of bonds to be cleaved at the current (discrete) time step is calculated from Equation 1, using a Taylor series method of second order (Ref. 28). The parameter *deltaTime* represents the duration of each time step and was kept at a fixed value for the entire simulation. Different values for *deltaTime* resulted in differences in the accuracy of the numerical integration and differences in the time required for running the simulation to completion.

There are three kinds of bonds: "cleavable", "noncleavable" and "nonexistent". A cleavable bond can be cleaved by heparinase, whereas a noncleavable bond can not. Noncleavable bonds are found only in the $(F1)_2$ dimer. Nonexistent bonds are considered to lie at the termini of the polymer chains, each designating the end of a chain. At each time step, a random number is calculated for each cleavable bond. If the random number belonging to a bond is below the value of the state variable *threshold* at the current time step, then the bond will be cleaved at the end of the time step. The formula for the *threshold* at time $t$ is

$$\text{threshold}(t) = \frac{\Delta x(t)}{x(t)} \qquad (2)$$

In this manner, the expected number (in the statistical sense) of bonds cleaved during each time step equals the number determined by Equation 1.

After the bonds have been cleaved, newly emergent chains of free $(F1)_2$ need to be detected, and the bond between the pair of F1 units changed to bond type "noncleavable". (Recall that a free F1-F1 chain is impervious to heparinase, and is, in fact, a single F4 unit.)

We define the value of *fractionCompletion* to be the ratio of the number of bonds cleaved or rendered noncleavable since the beginning of the depolymerization to the number of cleavable bonds in the initial population of chains. At the conclusion of each time step, the value of *fractionCompletion* is recalculated, using the formula

$$\textit{fractionCompletion} = 1 - \frac{x(t)}{x(0)} \qquad (3)$$

The simulation continues until a desired value of *fractionCompletion* has been achieved, at which time information may be written to disk files. Then the program either continues or terminates execution.

### MPP PASCAL CODE

In this section we present the code in MPP Pascal for the depolymerization of heparin molecules (chains). We use two globally declared parallel arrays called ***name*** and ***bond*** to hold the name of each oligosaccharide unit and the type of bond between that unit and its successor, respectively. Each PE stores information concerning one fundamental fragment in the simulated population. Recall that the PE's (and hence the fundamental fragments) have an implicit ordering by rows of the Array Unit. In order to clarify the presentation of the excerpts of the simulation program which follow, we have used **boldface** print for the names of procedures and functions.

The function **getNextStoppingFraction** returns the next value of *fractionCompletion* to be used as a stopping criterion for the numerical integration routine. The function **yetMoreStoppingFractions** returns a boolean value of **true** if there are more stopping fractions yet to be considered, and **false** otherwise. The depolymerization begins with the invocation of the procedure **depolymerization**.

```
procedure depolymerization;
begin
while yetMoreStoppingFractions do
  begin
  currentStop := getNextStoppingFraction;
  while fractionCompletion < currentStop do
    begin
     { Simulate one time step }
    oneTimeStep;

     { Find free F1-F1 dimers }
    modifyCleavableBonds;

     { Recalculate fractionCompletion }
    updateStateVariables
    end ;

     { Write data to disk }
  printInformation
  end
end ; { depolymerization}
```

Next we provide code for procedures invoked from procedure **depolymerization**, and then procedures invoked from within those procedures, etc. These procedures are all declared within procedure **depolymerization**, though for ease of exposition we have presented them separately.

```
procedure oneTimeStep;
  { Compute threshold and change bonds
   from "cleavable" to "nonexistent". }
var
  nextRand : parallelArrayType ;
begin
  { seed is global parallel array;
   randomNumber is a function that returns
   a parallel array of random numbers. }
nextRand := randomNumber (seed) ;
where (nextRand < threshold)
    and (bond = cleavable)
do bond := nonexistent
end ; { oneTimeStep}
```

```
procedure modifyCleavableBonds;
  {  Find all occurrences of free F1-F1 dimers.
    Look for a sequence of three units x, y, and z
    such that y and z are F1 units; x and z have
    nonexistent bonds; and y has a cleavable bond.
      Change the bond at y  to "noncleavable" }
var
  leftName, leftBond, rightBond:  parallelArrayType ;
begin
   { Shift array name to the left one position;
     shift array bond to the left;
     shift array bond to the right.
     Use these shifted arrays to find all sequences
     x, y, and z (as stated above). }
leftName := leftRowshift (name) ;
leftBond := leftRowshift (bond) ;
rightBond := rightRowshift (bond) ;

    { Modify bonds, changing them
      from cleavable to noncleavable. }
where (name =  F1) and (leftName =  F1)
  and (bond =  cleavable)
  and (leftBond =  nonexistent)
  and (rightBond =  nonexistent)
do bond :=  noncleavable ;
end ;  { modifyCleavableBonds}
```

```
procedure updateStateVariables;
var
  temp:  parallelArrayType ;
begin
where bond =  cleavable do temp :=  1
otherwise temp := 0 ;
numberCleavable := sum (temp) ;

    { Calculate new value for fractionCleavable, using
    global variable initialNumberCleavable, which equals
    numberCleavable at time t = 0. }
fractionCleavable
       :=  1 - numberCleavable/ initialNumberCleavable ;
end ;  { updateStateVariables}
```

Each of the next two functions takes a parallel array parameter *A* and returns the same array with the elements shifted one position downward (**leftRowShift**) or upward (**rightRowShift**) with respect to row major ordering of the elements of *A*. In the case of shifting downward, the [0,0] element of *A* is lost (destroyed), and the [127,127] element of the array returned by **leftRowShift** is zero. In the case of shifting upward, the [0,0] element of the array returned by **rightRowShift** is zero, and the [127,127] element of *A* is lost.

Each of the predeclared functions **shift** and **rotate** requires three parameters: a parallel array *A* and two integer values *ns* and *we*. If *ns* equals 1, the shift is upward (northward); if *ns* equals -1, the shift is downward (southward). If *we* equals 1, the shift is to the left (westward); if *we* equals -1, the shift is to the right (eastward). To illustrate, suppose that the ARU is a 3×3 array of PE's instead instead of a 128×128 array of PE's. If *A* were the following parallel array:

```
123
456
789
```

then the matrix returned by **leftRowShift** would be

```
234
567
890
```

and the matrix returned by **rightRowShift** would be

```
012
345
678
```

Here are the actual functions:

```
function leftRowShift (var A : parallelArrayType)
                                          : parallelArrayType ;
var
  ns, we, maxIndex : integer ;
  temp : parallelArrayType ;
begin
    { Shift each row to the left, wrapping around
     at the edges of the ARU. }
ns :=  0 ;  we :=  1 ;
temp :=  rotate (A, ns, we) ;

    { Shift rightmost column upward, filling with zero. }
ns :=  1;  we := 0 ;  maxIndex := 127 ;
where col_Index =  maxIndex
do leftRowShift := shift (temp, ns, we)
otherwise leftRowShift := temp ;
end ;  { leftRowShift}
```

```
function rightRowShift (var A : parallelArrayType)
                                          : parallelArrayType ;
var
  ns, we, minIndex : integer ;
  temp : parallelArrayType ;
begin
  { Shift each row to the right, wrapping around }
ns :=  0 ;  we :=  -1 ;
temp :=  rotate (A, ns, we) ;

   { Shift leftmost column down, filling with zero. }
ns :=  -1 ;  we :=  0 ;  minIndex :=  0 ;
where col_Index =  minIndex
do rightRowShift := shift (temp, ns, we)
otherwise rightRowShift :=  temp
end ;  { rightRowShift }
```

## DISCUSSION

The MPP may be compared with two other SIMD (parallel) computers, namely the Connection Machine (CM), manufactured by Thinking Machines Corporation, and the ICL-DAP computer, manufactured by Active Memory Technology, Inc. The Connection Machine series of computers (CM-1 and CM-2) resembles the MPP in the massive number (65,536) and word size (1-bit) of its processing elements. Unlike the MPP, the CM has a 16-dimensional hypercube interconnection topology, 4096 bits of random access memory at each PE, and a message-passing facility and router (Ref. 29-31). The ICL-DAP series of computers possess the grid (2-dimensional array) interconnection topology, like the MPP (Ref. 32). The grid itself is smaller (32x32 PE's), but each processing element possesses 32,768 bits of local random access memory, considerably more than belong to a processing element of the MPP.

Much attention has been shown in the last few years to algorithms for the Connection Machine (including an entire issue of the Communications of the ACM, see Ref. 33). To a lesser extent, some exposure has been given to algorithms running on the MPP (Refs. 34-35).

## CONCLUSION

We have described a simulation of the enzyme-catalyzed depolymerization of heparin and shown how it can be mapped to the architecture of the MPP. We explained the principal differences between MPP Pascal and standard Pascal. Our presentation of the actual program for the simulation illustrated the language constructs in MPP Pascal which allow for manipulation of the 16,384 processing elements in the Array Unit. The simulation included algorithms for circulating data among the PE's (using an implicit row-major ordering of the PE's) and for pattern matching.

## REFERENCES

1. Faiss, R.O. "The Goodyear SIMD Processors: Insights Gained from 20 Years of Usage," unpublished technical report, Goodyear Aerospace Corporation, Akron, Ohio.
2. Gledhill, I.M.A. and Storey, L.R.O., "Particle Simulation of Plasmas on the Massively Parallel Processor," (*Proceedings of Frontiers of Massively Parallel Scientific Computation*), NASA Conference Publication No. 2478, NASA Scientific and Technical Information Office, Greenbelt MD, 1987, pp. 37-46.
3. Sullivan, F. and Mountain, R., "A Fast MPP Algorithm for Ising Spin Exchange Simulations," (*Proceedings of Frontiers of Massively Parallel Scientific Computation*), ibid., pp. 53-56.
4. Barnden, J.A., "Simulation of an Array-Based Neural Net Model", (*Proceedings of Frontiers of Massively Parallel Scientific Computation*), ibid., pp. 59-68.
5. Hastings, H.M. and Waner, S., "Neural Nets on the MPP", (*Proceedings of Frontiers of Massively Parallel Scientific Computation*), ibid., pp. 69-74.
6. Carmichael, G.R., Cho, S.Y., Cohen, D.M., and Oguztuzun, M.H., "Chemical Network Problems Solved on NASA/Goddard's Massively Parallel Processor Computer", (*Proceedings of Frontiers of Massively Parallel Scientific Computation*), ibid., pp. 165-170.
7. Science Applications Research, "MPP Pascal Programmer's Guide", prepared for NASA/Goddard Space Flight Center, Greenbelt MD, Contract No. NAS-5-28200, Task No. 240-00, March 1988.
8. Fischer, J. (editor), "MPP User's Guide", prepared for NASA/Goddard Space Flight Center, Greenbelt Md 20771, by Science Applications Research, 4400 Forbes Boulevard, Lanham Md 20706, Contract No. NAS-5-28200, Task assignment No. 192, January 1986.
9. Linhardt, R.J., Rice, K.G., Merchant, Z.M., Kim, Y.S., and Lohse, D.L., "Structure and Activity of a Unique Heparin-Derived Hexasaccharide," *J. Biol. Chem.*, 1986, 261(31), pp.14448-14454.
10. Lindahl, U., Feingold, D.S., and Roden, L., "Biosynthesis of Heparin," *Trends Biochem. Sci.*, 1986, 11, pp. 221-225.
11. Rice, K.G. and Linhardt, R.J., "Study of Homogeneous, Defined Oligosaccharide Substrates of Heparin and Heparan Monosulfate Lyases," *Carbohydrate Research*, 1988, submitted.
12. Linhardt, R.J., Rice, K.G., and Cohen, D.M., "Non-Random Structural Features in the Heparin Polymer," *Biochem.*, 1988, submitted.
13. Hwang K. and Briggs F.A., *Computer Architecture* and *Parallel Processing*, McGraw-Hill, New York, 1984.
14. Gilmore P.A., "The Massively Parallel Processor," *Technical Report GER* – 17272, Goodyear Aerospace Corporation, May 1985.
15. Goodyear Aerospace Corporation, "General Description of the MPP," *Technical Report No. GER* – 17140, April 1983.
16. Batcher K.E., "Design of a Massively Parallel Processor", *IEEE Trans. Computers*, 1980, C-29, pp. 836-840.
17. NASA/Goddard Space Flight Center, "Computing on the MPP at the Goddard Image and Information Analysis Center (GIIAC)", version 2, February 1986.
18. Parker D., "MPP Main Control Language - MCL", Technical Report prepared by Science Applications Research (Lanham, MD 20706) for NASA/Goddard Space Flight Center, Contract No. NAS-5-29200, Task Assignment No. 177, September 1985.
19. Parker D., "MPP PE Array Language - PEARL", Technical Report prepared by Science Applications Research (Lanham, MD 20706) for NASA/Goddard Space Flight Center, Contract No. NAS-5-29200, Task Assignment No. 177, September 1985.
20. Reeves A.P., "Parallel Pascal: An Extended Pascal for Parallel Computers", *J. Parallel Distributed Computing*, 1984, 1, pp. 64-80.
21. Dorband J.E., "MPP Parallel FORTH User's Manual", Version 1.1.1, NASA/Goddard Space Flight Center, Greenbelt MD 20771.
22. Jensen K. and Wirth N., *Pascal User Manual* and *Report*, 3rd edn., revised by A.B. Mickel and J.F. Miner, Springer-Verlag, New York, 1985.
23. Welsh J. and Elder J., *Introduction* to *Pascal*, 2nd edn., Prentice-Hall International, Englewood Cliffs, N.J., 1982.
24. Cohen D.M. and Linhardt R.J., "Use of Computer Simulation on the Massively Parallel Processor to Study the Structural Features of Heparin", (*Proceedings of the Third International Conference on Supercomputing*, edited by L.P. Kartashev and S.I. Kartashev, International Supercomputing Institute, Inc.), 1980, volume 1, pp. 210-213.
25. Knuth, D.E., *The Art of Computer Programming*: *Volume* 2, *Seminumerical Algorithms*, 2nd edition, Addison-Wesley, Reading MA, 1981.
26. Cohen, D.M., "Validation of A Stochastic Algorithm for Choosing the Initial State of A Monte Carlo Simulation on a Parallel Computer," *SIAM J Sci. Stat. Computing*, 1988, submitted.
27. Lehninger, A.L., *Biochemistry*, 2nd edition, Worth Publishers, New York, 1975.
28. Johnson, L.W. and Riess, R.D., *Numerical Analysis*, 2nd edition, Addison Wesley, Reading MA, 1982.
29. Hillis, W.D., *The Connection Machine*, MIT Press, Cambridge MA, 1985.
30. Thinking Machines Corporation, "Introduction to Data Level Parallelism", *Technical Report* No. 86.14, April, 1986.
31. Gabriel R.P., "Massively Parallel Computers: The Connection Machine and Non-Von", *Science*, 1986, 231, pp. 975-978.

32. Active Memory Technology, Inc, "DAP Hardware", brochure, Active Memory Technology, Inc, Irvine, CA.

33. *CACM* , vol 29, No. 12, December, 1986.

34. Tilton, J.C., Hallada, W.A. and Ramapriyan, H.K., "An Overview of Algorithms for the Massively Parallel Processor," internal memo, Goddard Space Flight Center, Greenbelt, MD, April 1985.

35. Cohen, D.M., "Program Development on a Massively Parallel Computer," *Software– Practice* and *Experience*, 1988, submitted.

# ESTIMATING WATER FLOW THROUGH A HILLSLOPE USING THE MASSIVELY PARALLEL PROCESSOR

J. E. Devaney
Science Applications Research
Lanham, Maryland

P. J. Camillo and R. J. Gurney
NASA/Goddard Space Flight Center
Greenbelt, Maryland

## ABSTRACT

A new two-dimensional model of water flow in a hillslope has been implemented on the Massively Parallel Processor at GSFC. Flow in the soil both in the saturated and unsaturated zones, evaporation and overland flow are all modelled, and the rainfall rates are allowed to vary spatially. Previous models of this type had always been very limited computationally. This model takes less than a minute to model all the components of the hillslope water flow for a day. The model can now be used in sensitivity studies to specify which measurements should be taken and how accurate they should be to describe such flows for environmental studies.

## INTRODUCTION

One important part of the global hydrological system is a catchment, which separates rainfall into evaporation, overland flow, and infiltration. For a heavy rain, infiltration excess reaches the stream first as overland flow. Part of the infiltrated water may then flow rapidly below the surface to re-emerge downslope or enter the stream. This is usually referred to as saturated subsurface flow. The rest reaches the unsaturated zone. The flow there is vertical and horizontal, and the latter component may eventually contribute to the stream flow. Another component which can contribute to the stream flow is horizontal flow in a perched water table above the bedrock.

The primary output of catchment models is the hydrograph, in which the rainfall and fluxes to the stream from each of the above processes are plotted as a function of time. The rainfall rate and the sum of all the output fluxes are the usual data from a catchment, and a primary goal of catchment modelling is to understand the sensitivity of the output to the physical characteristics of the catchment, such as topography, cover type, soil characteristics, and antecedent moisture.

Ref. 13 define catchment models as being of three basic types, but with overlapping characteristics so they may be considered a continuum. The first is stochastic. These models are statistical, in which time series of measured hydrographs (output) are correlated to rainfall (input) using classical time series analysis techniques. This leads quite naturally to parametric models, their second class, in which the parameters of the stochastic models are related empirically to the physical properties of the catchment. The third class contains deterministic models based on the laws of conservation of energy, mass, and momentum, usually expressed as time and space dependent differential equations. As these almost always contain non-measurable parameters which must be calibrated, deterministic models are partly parametric.

There are many deterministic catchment models, but none of them includes all of the processes in the hydrological cycle. In part this is because we don't even know what they all are, due to the extreme complexity and variability of natural catchments. However, no existing model even includes all the processes previously described, because no serial computer can model them with a reasonable amount of computer time for a spatially variable catchment and for a long enough time period (Ref. 1,7,8,13.15).

The concept of partial (or contributing) areas is one basis of our understanding of how catchments distribute rainfall (Ref. 17). Due to the spatial variability of catchment characteristics (soils, cover, topography), different areas handle the rain in different ways. For example, if the rain rate exceeds the infiltration capacity for a particular area, then the excess rain becomes overland flow. Once the soil is saturated, the water can flow rapidly below the surface and parallel to it. This process is referred to as saturated subsurface flow. The water will re-emerge somewhere downslope, adding to overland flow. The areas change over time, so the saturated partial area which contributes to overland flow varies in time as well as in space.

We have tried to overcome the computing limitations by developing a model on the Massively Parallel Processor (MPP). The model consists of a set of partial differential equations, solved in parallel, and so adapts naturally to a parallel architecture. The MPP hillslope model includes the following components:

-- Surface retention

-- A complete surface energy balance (temperature and moisture) with separate evaporation rates from the soil, plants (with water extraction from the unsaturated zone), and surface retention

-- Overland flow

-- Saturated subsurface flow parallel to the surface

-- Horizontal and vertical flow in the unsaturated zone

-- Horizontal flow in an unconfined aquifer

Our model is a vertical slice of a hillslope, so it is basically a two-dimensional model. It may be considered three-dimensional only if the gradients are all downslope, not across the slope. It is based on a catchment model of Ref. 11, which is simply a series of uncoupled one-dimensional soil columns placed side by side. We have improved their design by allowing for horizontal flow in the unsaturated zone between the columns, and including the soil and surface temperatures.

We decided at the beginning of this research effort to create one-, two-, and then three-dimensional models in succession. The one-dimensional model (Ref. 6) was compared to a similar one which runs on a serial machine (Ref. 5,10) to make sure the equations are solved correctly on the MPP, and as a timing benchmark. After the two-dimensional model is completely tested, we plan to develop a three-dimensional version.

Our use of a parallel processor significantly reduces the execution time. Typically a 24 hour period may be modeled in about one CPU minute. Ref. 11 state that their model does not use excessive computer time on a serial machine, but they only present results from 6 hour simulations.

## THE TWO-DIMENSIONAL MODEL

The specifications for each of the components of the model given in the first section are described here as flux and continuity partial differential equations. The method of solution is also briefly described.

### Unsaturated Zone

Moisture flow is modeled as described in Ref. 5, except we now have a horizontal component in the soil moisture flux. The surface temperature is modeled by the force-restore method.

Boundary value fluxes must be specified for moisture at the top and bottom of the hillside (vertical direction) and at the hillslope divide and surfaces (horizontal direction). The top boundary flux is the infiltration or evaporation rate, computed from the surface energy balance. The horizontal flux into the hillslope at the divide is zero. The horizontal flux at the hillslope surface depends on whether that cell is saturated. If it is and the sum of the vertical fluxes plus the horizontal flux into the cell from the interior of the hillslope would cause soil moisture to exceed saturation, then the flux onto the surface is set to whatever value is needed to keep moisture just as saturation. Otherwise, it is zero. This is the mechanism which allows subsurface return flow.

### Saturated Zone

The water table height in each column is HB. The horizontal flux is QB, and the vertical flux is QZ. The fluxes and vertical boundary conditions are calculated by the one dimensional Boussinesq equation (Ref. 14). The flux into the water table from the unsaturated zone is modelled as the vertical hydraulic conductivity of the layer, and the bottom boundary condition is an input parameter representing an impervious layer or upward or downward seepage.

The flux at the catchment divide is set to zero. At the seepage face the height HB is a fixed input parameter. Therefore the time derivation of HB is zero for the last column, and the discretized form of this derivative may be solved for the horizontal flux at the seepage face. This is the saturated zone flux which contributes to the hydrograph.

### Overland Flow

If the surface water height is larger than a critical value, the overland flow flux is determined by Manning's equation (Ref. 7).

The infiltration rate is basically the Green-Ampt model (Ref. 9), with the usual modification which replaces the depth of the wetting front with the cumulative infiltration:

$$I(t) = a + b/\int_0^t I(t')\,dt' \qquad (1)$$

### Surface Energy Balance

The energy balance equation provides the surface fluxes:

$$G = R + LE + LH \qquad (2)$$

All fluxes are positive downward. G is the heat absorbed by the soil, R is the net radiation flux, LE is the evapotranspiration energy flux, and H is the sensible heat. After finding the solution, the surface moisture flux $q_\theta$ is set equal to the soil evaporation rate, and G is used in the force-restore model. The surface temperature needed to evaluate the fluxes is known from the force-restore equation. The latent and sensible heat fluxes are the usual resistance formulations. We imagine the soil and vegetation as one surface with the temperature $T_s$. We also allow for some surface water storage. This affects the evaporation rates, because the surface resistance is zero for the fraction of the evaporation which comes from the stored water.

### Method of Solution

The soil moisture and temperature continuity equations are solved by calculating the spatial derivatives of the moisture fluxes and then computing the time integral using numerical models.

The soil is divided into cells by creating a grid of N layers and M columns of varying widths $\Delta z_i$ and $\Delta x_i$ respectively, which are input parameters. At a specified time the fluxes at the interior boundaries are calculated. The surface energy balance equations are evaluated and all boundary conditions applied. The continuity equations are of the form:

$$\frac{d\vec{y}}{dt} = \vec{f}(t,\vec{y}) \qquad (3)$$

The vector $\vec{y}$ represents the state of the system in the unsaturated zone and $\vec{f}(t,y)$ represents the model equations. This is solved with an Adams-Bashforth predictor-corrector method (Ref. 3,16). This solution is described in detail in Ref. 5. Since double precision is not available on the MPP, the form of the predictor-correction equations with the calculations done with the derivatives instead of the backward differences was used. New values of the state vector, $\vec{y}(p)(t+\Delta t)$ are predicted in terms of the previous derivatives. The derivatives are recalculated from the model equations, and then the corrected value of the state vector, $\vec{y}(c)(t+\Delta t)$, is obtained.

The difference between $\vec{y}(p)$ and $\vec{y}(c)$ is a reliable estimate of the discretization error, and the software determines if each element of this difference lies within a user-specified window. If all differences are smaller than this window, the integration step size ($\Delta t$) is doubled, leading to increased computational efficiency and reduced roundoff errors. If any difference is too large, the step size is halved. Doubling of the time step was accomplished by saving the previously calculated derivatives and using them. Thus, maximum accuracy could be retained. Where the time step could be doubled because the errors are small enough but there were insufficient back derivatives, doubling was postponed until there were sufficient back data. When the error window checks required that the time step be halved, three of the required derivatives for the predictor-corrector were available, and two were missing. The Runge-Kutta method was used to calculate these needed derivatives. The continuity equations for surface and saturated flow are solved using a Runge-Kutta method throughout.

## UTILIZING THE MPP ARCHITECTURE FOR SPEED

Since identical calculations were needed at each soil cell, the mapping of the two dimensional model of the hillslope was accomplished by assigning an individual processing element to each soil cell (see Fig. 1). Thus, the local memory of each processor contains the values which belong to that cell, i.e. moisture, position, thickness, depth, conductivities, etc. Surface temperature, deep soil temperature, cumulative infiltration, overland flow, and saturated flow were all stored as vectors in the same array as the moisture values since they were part of the state vector.

The first step in the solution required calculation of the fluxes at the interior boundaries of the soil cells. These calculations involved only array arithmetic and nearest neighbor (in one direction for horizontal fluxes and in the other direction for vertical fluxes) calculations. Since the interconnect scheme of the MPP is a nearest neighbor network, all of the array arithmetic and nearest neighbor calculations could be done in parallel. The next step in the solution required the surface energy balance equations be evaluated and the boundary conditions applied. These all involved vector calculations. Numerous input vectors were required to do these calculations over the course of a model run. Some were time dependent vectors such as the air temperature across the surface of the hillslope throughout the day and some were static throughout the model run, such as surface slope, surface roughness, and surface vegetation properties. These vectors were packed into array columns. To get the vector data to a convenient place to do calculations, the row and column broadcast capability of the MPP was used. This allows fast broadcast of one element from each row (column) to the other processor memories in the same row (column) (see Fig. 2).

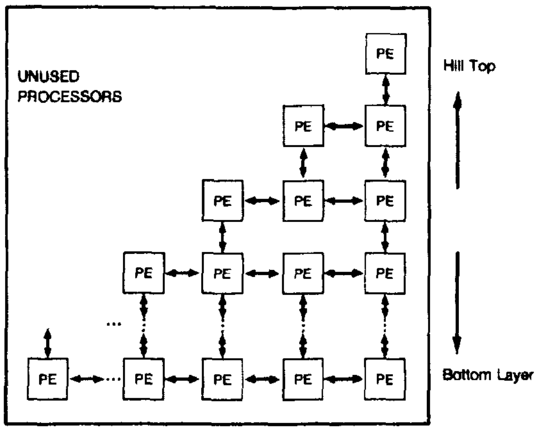


Figure 1. One processing element is assigned to one soil cell

It is not necessary that the broadcast row (column) be composed only of elements in a horizontal (vertical) direction but merely that one element per column (row) be selected. The MPP's capability to select arbitrary areas of an array for calculation via boolean masks allowed the completed vector calculation results to be placed for example into the processor memories of only the surface of the hillslope. This combination of data movement via broadcast and boolean selection enabled the vector calculations to be done simply. In addition, since many of the vector calculations were similar, it was possible to do more than one set at a time.

Once the derivatives were calculated, the predictor-corrector equations were used and the differences between them found. The tests on the

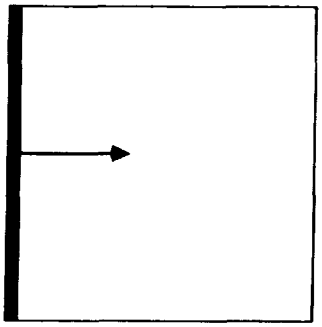


Figure 2. The row and column broadcasting feature of the MPP allows quick movement of data for vector calculations

halving (doubling) converted to a hardware instruction on the MPP and could thus be done in parallel. This global testing ability of the MPP was also used to decide if whole blocks of code needed to be executed or could be skipped. This occurred for example with the infiltration calculations under surface saturation. If no part of the surface was saturated, then these calculations could be skipped entirely. This also contributed to the overall speed of execution.

In summary, the program's speed was achieved through array arithmetic (masked and unmasked), parallel data movement through nearest neighbor communication and row and column broadcasting, and global testing of conditions using 'any' or 'all' for the purpose of choice in the next set of calculations. All of these fitted naturally with the MPP architecture and the computational requirements of the model. A comparison of the times (see Table 1) for the model as it has evolved from a 14 layer, one-dimensional limited flow model to the current two dimensional model shows that a single day of data run through the model requires only about a minute of CPU time.

Table 1. Timing measurements comparing MPP and a serial processor for 24 hours of data processed.

| | |
|---|---|
| One Dimensional Model<br>(14 soil layers, no rain, vertical flows only) | |
| IBM (Full processing capability): | 4 sec |
| MPP (14/16384 processors): | 10 sec |
| Two Dimensional Model<br>(102 soil layers, 102 soil columns, horizontal and vertical unsaturated flows, saturated flow, overland flow, one hour of rain) | |
| MPP: | 57 sec |

MODEL OUTPUT

We have not yet completed unit testing of all the processes in the model. Here we present the results of one test, which includes the surface energy balance of and infiltration into an initially very dry sandy loam soil.

The hillslope is divided into 102 columns of width .5 meters each. The first column has 100 soil layers of thickness .1 m and the bottom two layers .5 meters. The last column has only the bottom two layers. The slope is a line drawn from the top of the first column to the top of the last, so the area modeled is a right triangle with height 11 meters and base 60 meters. These soil cells plus the additional cells for temperature, infiltration, overland flow and saturated flow use approximately one-third of the Array Unit Processor capacity.

The initial volumetric moisture in the unsaturated zone is set to .05 $m^3 m^{-3}$ everywhere. To model a sandy loam we have set the parameters in the hydraulic conductivity and matric potential models to $\theta_s = .375$, $K_s = 2.8 \times 10^{-5}$ $m s^{-1}$, $\psi_s = -.43$ m, and $b = 5$. These values were derived from fits to the characteristic curves measured during an experiment near Phoenix in 1972 (Ref. 12). They were reused for each of the 6 days modeled here. Ref. 4 show how these data were fitted to the surface energy balance model. The rainfall rate was 1.6 cm $h^{-1}$ for the first 3 hours.

Perhaps the most important result is that the simulations took approximately 1 minute of CPU time per 24 hour period, or 6 minutes for the entire 6 day run. In numerical simulations on earth science problems, computer runs of an hour or more are not uncommon. In such a time, it is feasible to simulate 2 months or more of model time on the MPP. This will allow for simulations of many storms and inter-storm periods.

Figure 3a and 3b show the force restore solutions to the surface and deep soil temperatures as functions of time and column number. Time zero is the start of the simulation, which here is midnight. Column 1 is at the hillslope divide and column 102 is at the seepage face. It is difficult from these plots to project the daily maximum value onto the time axis, but for each day this occurs at 2 p.m. The temperatures range from 22 to 40 ($^{o}$C), increasing as the soil surface dries. The temperatures in the last three columns show some problems, which we are examining.

Figure 4a shows soil moisture in the top soil layer as a function of time and position. The rapid rise as the initially dry soil absorbs all the rain and the subsequent decline over the next 5 days are physically realistic.

Figure 4b shows the soil moisture profile in column 50 (halfway down the hillslope) as a function of time. This shows that the moisture never penetrates deeper than the top 5 layers, or .5 meters. It also shows that after 2 days the surface exhibits small oscillations about a value of .05 (same as in Fig. 4a), increases to a value

ORIGINAL PAGE
BLACK AND WHITE PHOTOGRAPH

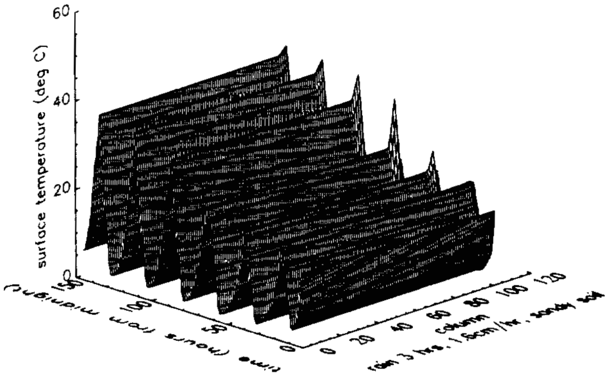


Figure 3a. Force-restore solution for the surface temperature as a function of time and position on the hillslope

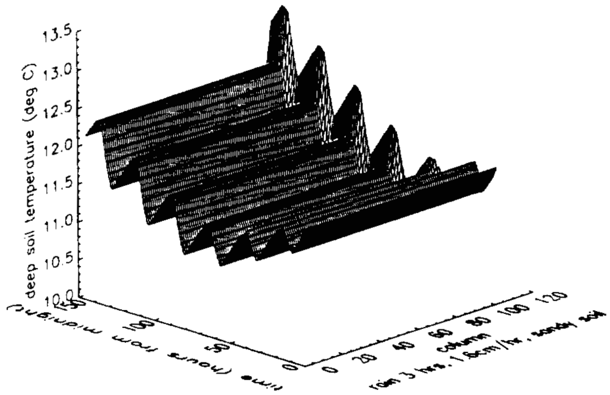


Figure 3b. Force-restore solution for the deep soil temperature as a function of time and position on the hillslope

of about .12 at about .3 meters then decreases to an unchanging value of .05 below .5 meters. Thus, the dynamic zone seems to be the top .5 meters.

Figure 4c shows the variation of the top cell soil moisture as a function of time. The effects of infiltration and evaporation, as well as of capillary action, can be seen.

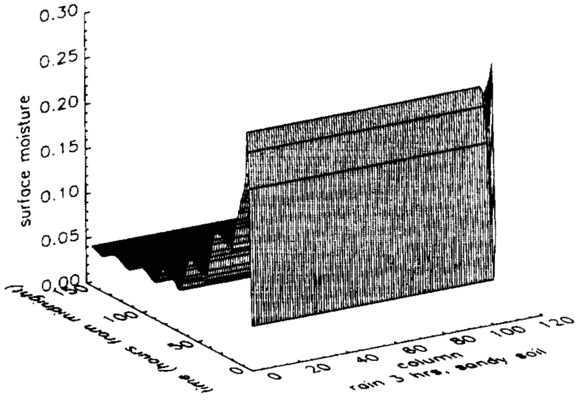


Figure 4a. Surface soil moisture as a function of time and position on the hillslope

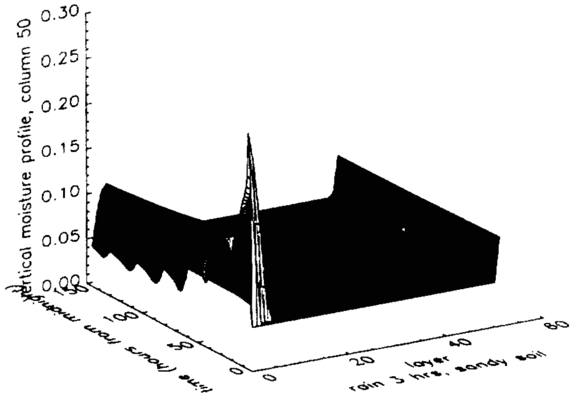


Figure 4b. Soil moisture profile for column 50 as a function of time

Figure 5a shows the infiltration rate as a function of time and position. The maximum rate shown here $(4.4 \times 10^{-4}$ cm $s^{-1})$ equals the rain rate, 1.6 cm $h^{-1}$. Figure 5b shows the cumulative evaporation everywhere as 4.8 cm, exactly equal to the cumulative rainfall. For this simulation, then, all the rain immediately infiltrated into the soil surface. Figure 5b also shows that the cumulative infiltration calculation is correct. There is no surface retention.

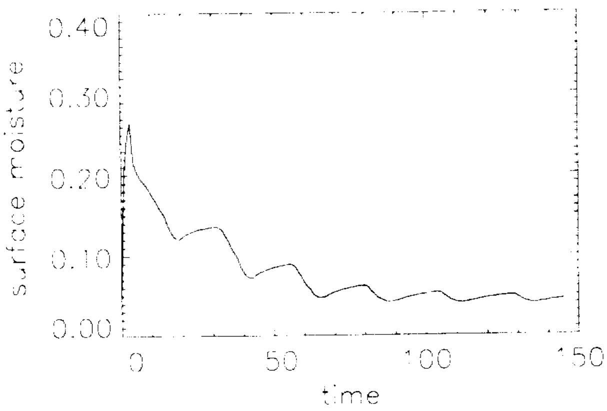


Figure 4c. Surface moisture for column 50 as a function of time

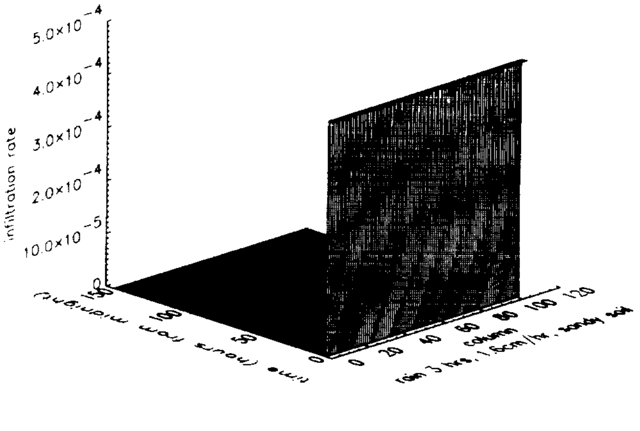


Figure 5a. Infiltration rate as a function of position and time

The surface energy balance fluxes are plotted in Figs. 6a-6d. The net radiation (Fig. 6a) is the data used to drive the energy balance model. These are the same very day, as we simply reused the 24 hour data set each day. The latent heat flux (Fig. 6b) decreases each day as the soil dries out. The sensible heat flux (Fig. 6c) exhibits peculiar behavior, being predominantly positive (towards the soil in the sign convention of Eq. 2) for the first 4 days and negative thereafter. Finally, Figure 6d shows the soil heat flux. It is positive during the day as it should be for a soil surface which is getting warmer every day, but it is also 50% of net radiation, a rather large value. The problem is not in the values for thermal conductivity and heat capacity, as may be seen in Figure 7. These vary with soil moisture as they should.

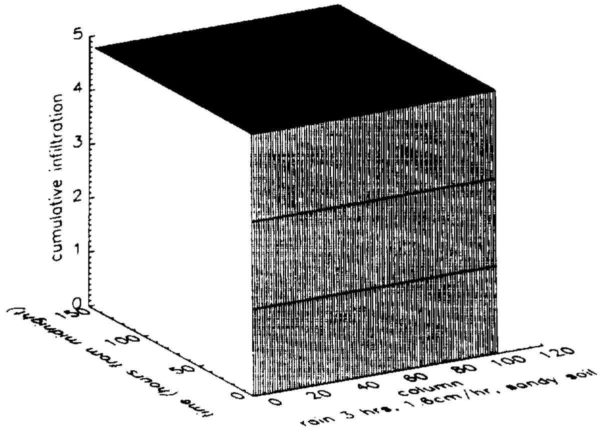


Figure 5b. Cumulative infiltration as a function of position and time

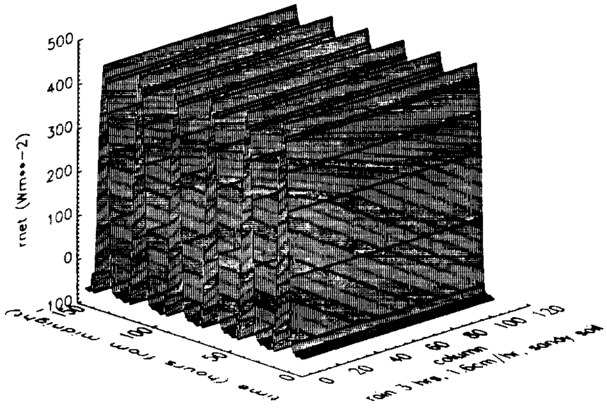


Figure 6a. Surface net radiation as a function of time and position

These peculiarities in the surface fluxes are most likely due to the use of the same net radiation every day, which cannot be representative of all the surface conditions modeled here. This is being checked out by using modeled instead of measured radiation.

ORIGINAL PAGE
BLACK AND WHITE PHOTOGRAPH

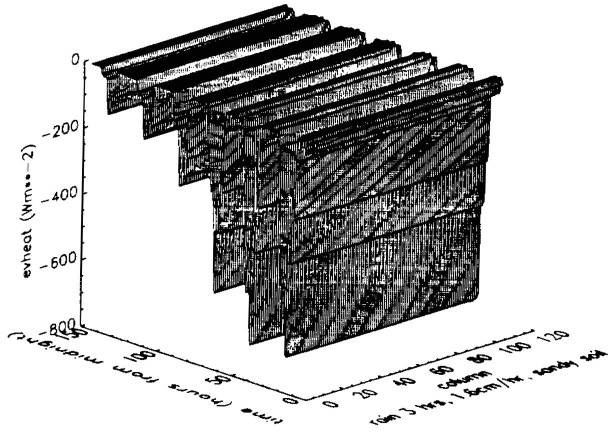


Figure 6b. Latent heat flux as a function of time and position

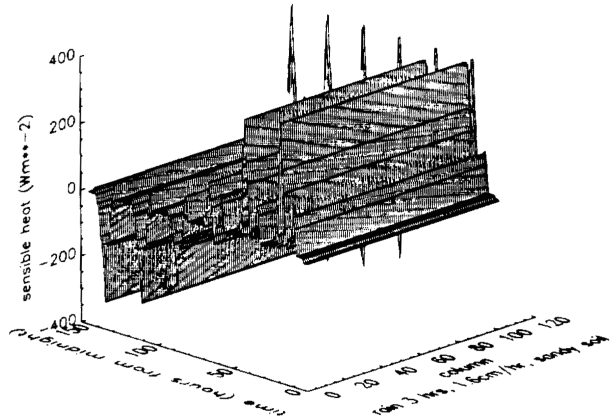


Figure 6c. Sensible heat flux as a function of time and position

SUMMARY

We have presented a new model of the hydrological response of a hillslope to rain. It runs on a SIMD parallel architecture computer, the Massively Parallel Processor, at Goddard Space Flight Center. Its major advantage over other models of its type is its much reduced execution times (due to the parallel architecture of the MPP) from what one gets on a serial machine. This allows the model to include more of the hydrological processes than any other model has been able to, including saturated subsurface flow and a sophisticated surface energy balance.

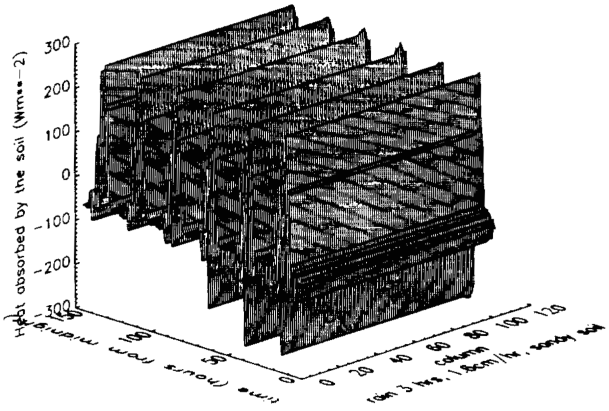


Figure 6d. Soil heat flux as a function of time and position

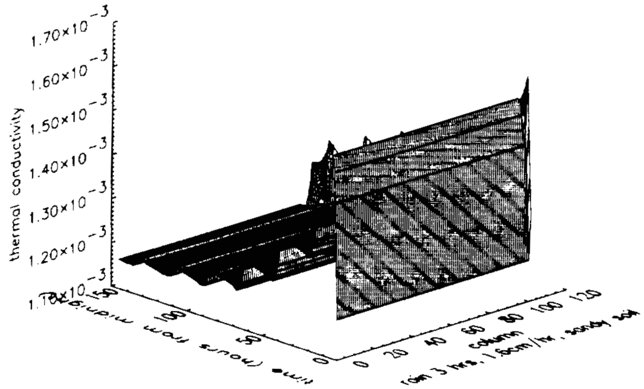


Figure 7a. Thermal conductivity of the top soil layer as a function of time and position

ACKNOWLEDGEMENT

One of the authors, J. E. Devaney, was partially supported by NASA Contract NAS5-28200.

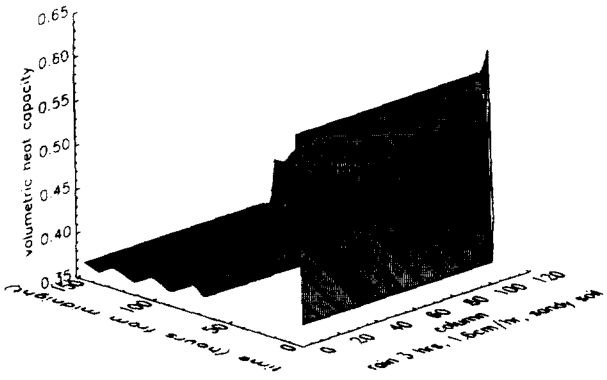


Figure 7b. Heat capacity of the top soil layer as a function of time and position

REFERENCES

1. Bathurst, J. C., "Physically-based Distributed Modeling of an Upland Catchment Using the Systeme Hydrologique Europeen," Institute of Hydrology, Wallingford, Oxon, UK.

2. Becker, F., P. J. Camillo and B. Choudhury, "Review of ET Estimation by Means of Satellite Data," Proceedings of the ISLSCP Conference on Satellite Data Algorithms, Pasadena, CA, January 1987.

3. Devaney, J. E., P. J. Camillo and R. J. Gurney, "A SIMD Implementation of a One-Dimensional Energy-Moisture Balance Model," submitted to Proceedings of the ISLSCP Conference on Satellite Data Algorithms, Pasadena, CA, January 1987.

4. Booth, A. D., Numerical Models, Academic Press, New York, 1957.

5. Camillo, P. J. and R. J. Gurney, "A Resistance Parameter for Bare Soil Evaporation Models," Soil Science, 141, 1986, 95-105.

6. Camillo, P. J., R. J. Gurney and T. J. Schmugge, "A Soil and Atmospheric Boundary Layer Model for Evapotranspiration and Soil Moisture Studies," Water Resources Research, 19, 1983, 371.

7. Devaney, J. E., P. J. Camillo and R. J. Gurney, "A SIMD Implementation of a Distributed Watershed Model," Proceedings of the Second International Conference on Supercomputing, Santa Clara, California, May 3-8, 1987.

8. Eagleson, P. S., Dynamic Hydrology, McGraw-Hill, New York, NY, 1970.

9. Freeze, R. A., "Three-Dimensional, Transient, Saturated-Unsaturated Flow in a Groundwater Basin, Water Resources Research, 7, 1971, 929-941.

10. Green, W. A. and H. A. Ampt, Studies on Soil Physics, 1. The Flow of Air and Water Through Soils, Journal of Agricultural Soils, 4, 1911, 1-24.

11. Gurney, R. J. and P. J. Camillo, "Modelling Daily Evaporation Using Remotely Sensed Data," Journal of Hydrology, 69, 1984, 305-324.

12. Hillel, D. and G. M. Hornberger, "Physical Model of the Hydrology of Sloping Heterogeneous Fields," Soil Science Soc. Am. J., 43, 1979, 434-439.

13. Jackson, R. D., "Diurnal Changes in Soil-Water Content During Drying," in Field Soil Water Regime, R. R. Bruce et al. (eds.), Soil Sci. Soc. Am. Proc, Special Publ. 5, 1973.

14. Kirkby, M. J. (ed.), Hillslope Hydrology, Wiley-Interscience, Chichester, UK, 1972.

15. Pikul, M. F., R. L. Street, I. Remson, "A Numerical Model Based on Coupled One-Dimensional Richard and Boussinesq Equations," Water Resources Research, 10, 1974, 295-302.

16. Smith, R. E. and D. A. Woolhiser, "Overland Flow on an Infiltrating Surface, Water Resources Research, 7, 1971, 899-913.

17. Teddington, A., Modern Computing Methods, Philosophical Library, New York, 1958.

18. van de Griend, A. A. and E. T. Engman, "Partial Area Hydrology and Remote Sensing," Journal of Hydrology, 81, 211-251.

# Implementation of a 3D Thermal Analysis Code on the CM-2 Connection Machine Computer

R.E. Cline, Jr.
*rec@snll-arpagw.llnl.gov*
*Division 8233, Sandia National Laboratories, Livermore, CA 94550*
*(415)-294-1395*

B.M. Boghosian and B. Nemnich
*bmb@think.com and bruce@think.com*
*Thinking Machines Corporation, 245 First Street, Cambridge, MA 02142-1214*
*(617)-876-1111*

## Abstract

This paper describes progress toward the full implementation of a version of TACO3D, a 3D thermal analysis code, for use on the CM-2 Connection Machine. We have implemented a combination of a parallel conjugate gradient solver, which runs on the CM-2, and a serial calling program, which resides on the VAX front-end in order to assess the feasibility of utilizing this type of massively parallel computer in the engineering production environment. Preliminary results have shown this implementation, running on a 16K processor CM-2, is over 5 times faster than a single XMP processor in the solution of a $\sim$ 30,000 node 3D application. We will also discuss future plans to complete the optimum implementation of the entire code.

**Keywords:** finite element, diffusion, parallel computing

## 1 Introduction: Parallel Processing in the Production Environment

Application of parallel processing in the production environment requires the development of algorithms that efficiently solve problems of interest to design engineers. Aside from fast floating point processing modern production codes require the availability of input/output (I/O) devices for the storage and retrieval of temporary information as well as online databases of material properties. Modern production codes also rely heavily upon the ability to present the numerical results in an easily understandable graphical form. With the availability of floating point performance of over 100 Mflops, mass storage and graphical frame buffers, massively parallel processors are now being considered as viable additions to the engineering production environment.

Many of the currently available examples of engineering applications of parallel processing solve simplified problems which have been tailored to specific architectures. The true measure of the utility of parallel processing in the production environment should be based upon the ability to solve larger problems and solve them faster than currently possible on Cray class supercomputers. In order to assess the feasibility of incorporating a massively parallel processor into the production environment we have undertaken the translation of TACO3D, a finite element heat transfer code currently used for production applications at Sandia National Laboratories, for use on a number of different computational platforms. The structure of the code allows this process to performed in a stepwise manner, as will be discussed later. This paper describes the progress of an implementation for use on the Thinking Machines Corporation CM-2 and results from several application problems. The current phase of implementation uses a conjugate gradient solver, running on the CM-2, interfaced to the main program, which resides on the front-end machine.

## 2 The Connection Machine Computer

The CM-2 Connection Machine system is a massively data parallel computer consisting of up to $2^{16} = 65536$

bit serial processors arranged in a hypercube topology. Each processor has 64K bits of local memory, so a full machine has an overall memory capacity of 512 megabytes. The processors and their associated memories are arranged in hardware with 16 to a chip. Every pair of chips, *i.e.* 32 processors, is supported by 32-bit floating-point unit (FPU) resulting in 2048 FPU's per full machine. A nominal performance figure, averaged over a variety of applications, is in the vicinity of 2.5 GFlops, with a peak performance rating of more than an order of magnitude greater than that. The CM-2 operates in SIMD, single instruction multiple data, mode and is further characterized by a sophisticated communications network linking all of its processors.

In addition to the memory and computational capacity of the CM-2, the machine has an I/O structure that allows for parallel input/output. This is accomplished by I/O *controllers*, of which there are eight on a full CM-2. An 80 bit wide I/O bus connects each of these I/O controllers with the device that it controls. The I/O devices currently available fill the need for mass storage and graphics capability essential in the production environment. The DataVault, is a parallel disk drive unit, capable of transfering data to or from the CM-2 at approximately 40 Mbytes/second. This system could be used to enable the solution of problems that are larger than the current memory size of the machine. The second device is a high-resolution graphics display that can receive and display data stored in the CM-2 at approximately 1 Gbit/second. In addition to the analysis of results, this interface would enable the development of interactive mesh generators for large problems.

The CM-2 Connection Machine system uses a conventional front-end computer. Currently, it may be either a VAX with the ULTRIX operating system, or a Symbolics 3600-series Lisp Machine. The front-end machine supports the operating environment and all programming is done via the front-end. The control structure of program is maintained by the front-end, which issues commands to the CM-2 processors whenever necessary. Because the front-end could not possibly broadcast instructions directly to the Connection Machine processors as fast as the latter could accept them, high-level instructions are instead sent to a *sequencer* which in turn translates a corresponding packet of *microcode* from a *control store* into many instructions of *nanocode* and broadcasts these to the processors for execution.

In order to make the granularity of the machine somewhat flexible, the Connection Machine system allows the use of *virtual processors* (VP's). This is software that allows the programmer to treat one physical processor (ΦP) as though it were really some larger number of processors in a fully transparent manner. It thus enables makes the machine to simulate a system with more processors. The abstraction of virtual processors is supported by segmenting the local memory of each processor, and by having the sequencer multiplex over these memory segments as necessary. The degradation in the speed at which a program executes varies linearly in the number of virtual processors used; this is because the different virtual processors within a single physical processor operate sequentially.

Connection Machine processors are arranged in a Boolean hypercube configuration. Thus, for a Connection Machine with 65536 processors, a processor address is a sixteen-bit *cube address* that specifies the location of the processor on a sixteen-dimensional hypercube. Using a Gray code mapping it is possible to embed a one-dimensional array in a binary hypercube of $n$ dimensions in such a way that nearest neighbors are preserved. By applying the Gray code to $m$ successive subsets of bits within the sixteen bit cube address, it is possible to embed an $m$-dimensional Cartesian array into the hypercube so that nearest neighbors are preserved, as long as $m \leq n$. It is therefore possible to address processors on a Cartesian grid of $m$ dimensions, where $m \leq 16$, through the use of this Gray coded *grid address*.

Interprocessor communication can be either *regular* or *irregular*. An example of regular communications might be every processor getting data from some specified memory location of the processor whose grid address differs by $-1$ in some direction, e.g. along the $x$-axis. The Gray code mapping allows this to be done along unique cube wires with no resulting conflicts. An example of irregular communications, on the other hand, might be every processor getting data from processor 0. In this case, some data might have to traverse many cube wires to reach its destination, e.g. to get from processor 0 to processor 65535 it is necessary to traverse 16 cube wires, and some form of control is necessary to resolve resulting conflicts in the communications requests.

Regular communication is done by having each processor simply send its message over the cube wires directly; in this case, there is no possibility of contention. Irregular communication also uses the cube wires, but only in a manner that is controlled by special purpose adaptive communications hardware called the *router*. Any processor can use the router to send information to any other processor with a command that is appropriately called SEND. It is also possible to specify logical and/or arithmetic operations to be performed in case of a "collision", i.e. two data being sent to the same location within the same processor simultaneously.

When a processor wants to SEND data, it hands that data to the router. The router keeps track only of the *relative* address of the destination processor with respect to the current cube address. The relative address is continually updated as the messages move through the network toward the destination processor; when it reaches zero, the message is at its destination cube position and is delivered to the processor there. The router is responsible for buffering conflicting communication requests when more than one message wants to move along the same edge of the hypercube.

Due to the lack of contention, regular communications, using the cube wires directly, are going to be faster than irregular communications, using the router. As we shall see in the next two sections, finite element analyses in some geometries can be done with regular communications only, while in other geometries irregular communications are required.

## 3 The TACO Code

TACO3D is a three-dimensional finite element code for the solution of heat transfer problems in arbitrary geometry. It can perform both linear or nonlinear analyses on either steady-state or transient problems. Material properties, transport coefficients, and boundary conditions, may be specified as functions of space and time for inhomogeneous heat transport, or of temperature for nonlinear heat transport.

TACO3D also has a variety of specialized features that make it a valuable engineering production tool. These include the treatment of enclosure radiation, bulk nodes, and master/slave internal surface conditions which are useful for handling the material interfaces. The code has user-friendly input/output options such as free-field data input format, and user-specified functional representation of any independent variable. Graphical output plots in time or space are easily obtained by a variety of post-processing tools.

Here we adopt the notation from Reference [1], which should be consulted for additional details.

The basic equation solved by the TACO3D code is the diffusion equation:

$$\rho c_p \dot{\theta} = (k_{ij}\theta_{,j})_{,i} + Q \equiv \hat{L}\theta. \tag{1}$$

The boundary conditions can be any combination of Dirichlet conditions

$$\theta = \theta_s, \tag{2}$$

or Neumann conditions

$$k_{ij}\theta_{,j}n_i + q = 0, \tag{3}$$

and an initial condition must be specified for $\theta$ as well,

$$\theta = \theta_0 \tag{4}$$

Here, $\theta$ is the temperature, the commas denote differentiation with respect to the spatial coordinates, the overdot denotes differentiation with respect to time, $\rho$ is the mass density, $c_p$ is the specific heat, $k_{ij}$ is the thermal conductivity tensor, $Q$ is the internal volumetric heat generation rate, $n_i$ are the components of the unit normal to the boundary surface, $\theta_s$ is the temperature boundary condition which is a specified function of space and time, $q$ is the surface heat flux which is a specified function of space time and temperature, and $\theta_0$ is the temperature initial condition which is a specified function of position.

The flux boundary conditions may be written in the general form:

$$q = \sum f(\theta,t)\left(\theta^a - \theta_q^a\right)^b, \tag{5}$$

where $f$, $a$, $b$, and $\theta_q$ depend on the specific type of boundary condition used, e.g. conduction, convection, radiation, etc. This may be recast into the form:

$$q = f_0(\theta,t) + h(\theta,t)(\theta - \theta_q), \tag{6}$$

where the $f_0$ term is necessary to account for the case when $a = b = 0$. In what follows, we shall assume that the Neumann boundary conditions are expressed in this form.

Because our dependent variable is a scalar quantity, its finite element representation is particularly simple:

$$\theta(x_i,t) = \sum_j N_j(x_i)\theta_j(t), \tag{7}$$

where the $\theta_j(t)$ are nodal temperatures, and the $N_j(x_i)$ are basis splines. Employing the Galerkin principle, we can insert this representation into the dynamical equation, Eq. (1), and the boundary conditions, Eq. (3), to get the matrix equation,

$$C \cdot \dot{\theta} + K \cdot \theta = F, \tag{8}$$

where

$$C_{ij} \equiv \int_V N_i \rho c_p N_j dV \tag{9}$$

$$K_{ij} \equiv \int_V N_{i,\alpha} k_{\alpha\beta} N_{j,\beta} dV + \int_{S_N} N_i h N_j dS \tag{10}$$

$$F_i \equiv \int_V N_i Q dV + \int_{S_N} N_i (h\theta_q - f_0) dS, \tag{11}$$

where $S_N$ denotes the subset of the boundary surfaces on which Neumann conditions are specified. This must be suplemented by the Dirichlet conditions on

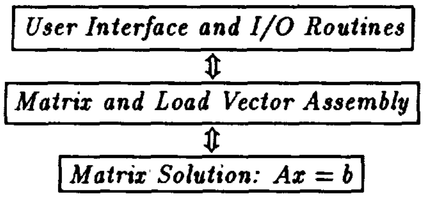

User Interface and I/O Routines
Matrix and Load Vector Assembly
Matrix Solution: $Ax = b$

Figure 1: Diagram of the coarse grain structure of TACO3D

the components $\theta_j(t)$ corresponding to nodes that lie on boundary surfaces on which Dirichlet conditions are specified.

The integration in time of Eq. (8) is accomplished by a generalized midpoint method, described in Reference [1]. The essential point is that time is discretized, so that the value of the temperature is represented by $\theta_k$ at time $t_k$. Then, $\theta(t)$ and $\dot{\theta}(t)$ in Eq. (8) are represented by various linear combinations of $\theta_k$ and $\theta_{k+1}$ at time $t_k$. Exactly which linear combinations are used depends on what sort of time differencing (e.g. forward-explicit, Crank-Nicholson, backward-implicit, etc.) is desired. Differencing Eq. (8) in this manner yields an equation of the form:

$$K_k^* \cdot \theta_{k+1} = F_k^*, \qquad (12)$$

that must be solved at time step $t_k$ for the new temperature, $\theta_{k+1}$. The matrix $K^*$ and load vector $F^*$ are linear combinations of $C$, $K$, and $F$ at timesteps $t_k$ and $t_{k-1}$. For example, in a steady-state analysis, the $\dot{\theta}$ term in Eq. (8) vanishes, so that $K^* = K$ and $F^* = F$, and Eq. (12) needs to be solved only once.

When the dynamical equation, Eq. (1), or boundary conditions, Eq. (3), are nonlinear, the matrix equation, Eq. (8), will also be nonlinear. That is, $K^*$ will depend on $\theta$, and/or $F$ will depend nonlinearly on $\theta$. In this sort of situation, Eq. (12) is solved iteratively by the TACO code. That is, $K^*$ and $F^*$ are evaluated based on $\theta$ at the last iteration, and are used to solve for the next iterate. As a first guess, the value of $\theta$ at the last time step is used.

Thus, regardless of whether the analysis is steady-state or time-dependent, and linear or nonlinear, the main computation done by the code is the solution of the matrix equation, Eq. (12). This must be done at least once at every time step.

# 4 Implementation of the Conjugate Gradient Algorithm for TACO3D

The coarse structure of TACO3D, Figure 1, consists of a user interface, input-output routines, the global conductance matrix and load vector assembly and the solution of the matrix equation, Eq. (12). The natural breaking point for the first step in the code conversion is at the level of the matrix solution. It was decided to implement the conjugate gradient routine on the CM-2, written in C-Star and Paris, and interface it to the main FORTRAN program running on the front-end.

The matricies generated by Taco are in general found to be symmetric positive-definite. These conditions allow some simplifications of the generalized conjugate gradient algorithm. The Cray version of TACO3D used in this study includes the conjugate gradient routine from SPDPACK [3], a package of routines developed by J. Grcar and J. Meza of Sandia. The detailed algorithm used in both the CM-2 and Cray XMP versions of the solver is:

```
To solve $Ax = b$:
        $M = \mathrm{diag}(A)$
        $k = 0$
        $x_0 = 0$
        $r_0 = b$
        $p_0 = z_0 = M^{-1}r_0$
    loop:
        $k = k + 1$
        $\alpha = z_{k-1} \cdot r_{k-1} / p_{k-1} \cdot A \cdot p_{k-1}$
        $x_k = x_{k-1} + \alpha p_{k-1}$
        $r_k = r_{k-1} - \alpha A \cdot p$
        $z_k = M^{-1} r_k$
        $\beta = z_k \cdot r_k / z_{k-1} \cdot r_{k-1}$
        $p_k = z_k + \beta p_{k-1}$
    goto loop
```

The bulk of the computational effort is in computing $A \cdot p$, but only one of these matrix-vector dot-products is required per iteration.

Taco solves the finite-element problem in terms of nodal equations rather than elemental equations. That is, an $n$-node problem involves solving a system of $n$ linear equations at each time step. A very natural approach is to assign each node to one virtual processing element. The result is that each processing element contains data pertaining to one of the nodal equations, *i.e.*, one row of the matrix and its right-hand-side element.

|  |  | Grid Solver |  | Routing Solver |  | Cray Solver |
|---|---|---|---|---|---|---|
| Problem | Nodes | *grid* | *ms/iter* | *VPR* | *ms/iter* | *ms/iter* |
| 11 x 11 x 11 | 1331 | 32 x 16 x 16 | 10.5 | 1 | 38.8 | 30 |
| 21 x 21 x 21 | 9261 | 32 x 32 x 32 | 16.6 | 1 | 41.5 | 49 |
| 31 x 31 x 31 | 29791 | 32 x 32 x 32 | 16.6 | 2 | 88.5 | 159 |

Table 1: Comparison of the Cray XMP-24 and the CM-2 solution times for the regular cubic solid with varying mesh sizes. The solution time for the CM-2 is found to scale linearly with the virtual processor ratio (VPR).

|  |  | Routing Solver |  | Cray Solver |
|---|---|---|---|---|
| Problem | Nodes | *VPR* | *ms/iter* | *ms/iter* |
| Dome16 | 2544 | 1 | 42.0 | 13.4 |
| Dome32 | 5088 | 1 | 42.3 | 27.6 |
| Dome64 | 10176 | 1 | 42.7 | 53.0 |
| Dome128 | 20352 | 2 | 81.0 | 106.4 |
| Dome256 | 40704 | 4 | 180.7 | - |

Table 2: Comparison of the Cray XMP-24 and the CM-2 solution times for the hemispherical dome problem with varying mesh sizes. The Dome256 problem was too large to run on the XMP-24.

### 4.1 Routing Version

The most general way to represent the matrix $A$ is to store row $n$ in processing element $n$ as a vector of pairs $(N_i, W_i)$, the node numbers and weights, respectfully, corresponding to the $i$ nonzero entries in that row of the matrix. This is a parallel analog to the IJA storage scheme used in PCGPAK [4,5].

All the vectors, such as the right-hand-side $b$, are simple to represent: processor $n$ gets element $n$ of the vector. Vector addition then becomes a simple addition of two values within each processor; the whole vector sum is computed at once in parallel, and it involves no interprocessor communication. A vector dot-product is a simple multiplication of two values within each processor, followed by a sum-reduction across all processors. The sum-reduction is an $O(\log n)$ operation on the CM system.

Computation of the matrix-vector dot-product $A \cdot p$ is a little more complicated. Each processor computes an element of the product, given by $\sum_i W_i p_{N_i}$. The values $W_i$ and $N_i$ are local to the processor, but the value $p_{N_i}$ is not. However, with the use of a single Paris instruction each processor can get the needed value from another processor, to which it has a pointer, using the parallel *GET* instruction. This must be done $i$ times each iteration, and is by far the speed-limiting operation in this approach.

The router in the CM-2 system is a message *sending* system; *GET*s are implemented as multiple *SEND*s. A *SEND* instruction tells each selected processor to send a datum to a processor to which it has a pointer. The messages are all sent in parallel through a message "routing network". This ability to efficiently pass messages through pointers to other processors is what separates the Connection Machine computers from other massively-parallel architectures.

We can take advantage of the symmetry of the matrix to make it go somewhat faster using *SEND*s. Since $A_{jk} = A_{kj}$, we can arrange, for some $i$, to have $N_i = j$ in processor $k$, and $N_i = k$ in processor $j$ and $W_i = A_{jk}$ in both. That is, for step $i$ of the matrix-dot-vector operation, pairs of processors will be exchanging their vector elements using a *SEND*. The weight $W_i$ does not need to be sent, since it is the same at both ends of the exchange.

This is the general-purpose "routing" implementation; it deals only with the solution of the matrix problem, and takes no advantage of the geometry of the problem which lead to the matrix.

### 4.2 Grid Version

If the geometry of the problem consists of a parallelepiped mesh a special purpose solver can be used to obtain large performance gains. One can configure the CM as an $n$-dimensional grid of virtual processors, with each dimension taking on an integral power-of-2 length. As discussed above, it is much faster to shift data from neighbor to neighbor along a grid direction than to use the general-purpose *SEND* instructions.

A large class of problems solved by Taco involve 3D parallelepiped element grids. For each node, Taco generates contributions from the 6 directly-connected neighbor nodes and the 20 "diagonal nodes". Therefore, each internal node typically has 26 nonzero matrix elements in each row plus the diagonal.

We can take advantage of the geometry by configuring the CM as a 3-dimensional grid large enough to hold the grid of nodes in the problem. Then, the layout of the nodes in the CM reflects the layout of the nodes in the real problem. All communication can be done locally, since all matrix row contributions are nearby on the CM grid.

If the problem is mapped onto the CM grid in this manner, the $N_i$ aren't needed; we can simply use the convention that $W_i$ refers to the coupling of the node in direction $i$, hence this implementation uses less memory per processor.

However, there are some drawbacks. First, the grid must be regular. Second, the CM system software currently requires each dimension to be a power-of-2 in length; if we have, say, a 33 x 33 x 33 grid of nodes, we need a 64 x 64 x 64 grid of processors, and we are only using a little more than 1/8 of the processors. One can assume that on average, a little less than half, *i.e.* $(3/4)^3$, of the configured processors will be used if the grid generators don't know about the power-of-2 limitation.

Nevertheless, the grid version is so much faster that it is the right choice whenever it is topologically possible to use it. The speed increase is sufficiently large that it may be beneficial to tune the mesh generation package toward the production of a parallelepiped grid.

## 5 Results and Timings

We ran several sample problems on both a Cray XMP 24 with scatter-gather hardware and a 16K-processor CM-2 at the Naval Research Laboratory.

The first class of problems are rectangular solids divided up into equal rectangular elements. A constant-temperature boundary condition is imposed on one side. This problem is an obvious fit to the grid solver. The results for problem using several various mesh sizes are listed in Table 1.

Note that the Cray times scale linearly with the number of nodes in the problem. The CM grid solver scales better than linearly because nearest-neighbor communications scale better than linearly. Also note that the times are the same for the 21 x 21 x 21 and 31 x 31 x 31 problems with the CM grid solver, because it needed to use the same number of virtual processors to solve the problem.

The second set of problems involves a more complicated geometry. It is a hemispherical volume of three materials: the curved "dome" shell, a disk-shaped plate, and the interior volume. This mesh is generated by spinning a 2D mesh about the symmetry axis of the dome shell, resulting in a grid that is periodic in the $\theta$ direction. A 3D solution is required due to the application of nonsymmetric heat load applied to the surface of the dome. The results listed in Table 2 show that the CM-2 versions perform the solution step up to 5 times faster than the Cray version.

## 6 Conclusions and Future Work

We have presented here the initial step toward the complete conversion of TACO3D to run in the Connection Machine environment. Results for a parallel implementation of the conjugate gradient routine have been shown to be faster than the Cray XMP-24 results. The greatest speed increases have been demonstrated for a grid version of the solver, which requires a parallelepiped mesh. The more general solver, which utilizes the router communication scheme, is still able to perform at up to 2 times the speed of the Cray solver on a 16K CM-2. One of the major gains in performance comes from the fact that the CM-2 algorithm scales linearly with the number of virtual processors required to solve the problem, where the Cray implementation scales linearly with the number of nodes in the finite element mesh. Extrapolating the CM-2 timings to a full 64K processor, Figure 2, shows that this machine is capable of up to 8 and 30 times the performance of the Cray XMP for the router and grid versions of the solver respectively.

The next step in the conversion of the TACO3D code would be the generation of the global conductance matrix and load vector using the CM-2. The parallelization of these tasks appears to be simplified by the fact that they can be broken into independent operations on individual elements of the mesh. The only communication necessary in these steps would be the transfer of the assembled matrix and load vector into processor geometry required for the solver.

The capability of solving large finite element problems may also require the renovation of several related processes, including mesh generation techniques and graphical analysis of the resulting information. These are both areas in which the computational power of the CM-2 may be useful. It may also be possible to solve problems which are several orders of magnitude larger through the use of the DataVault mass storage system.

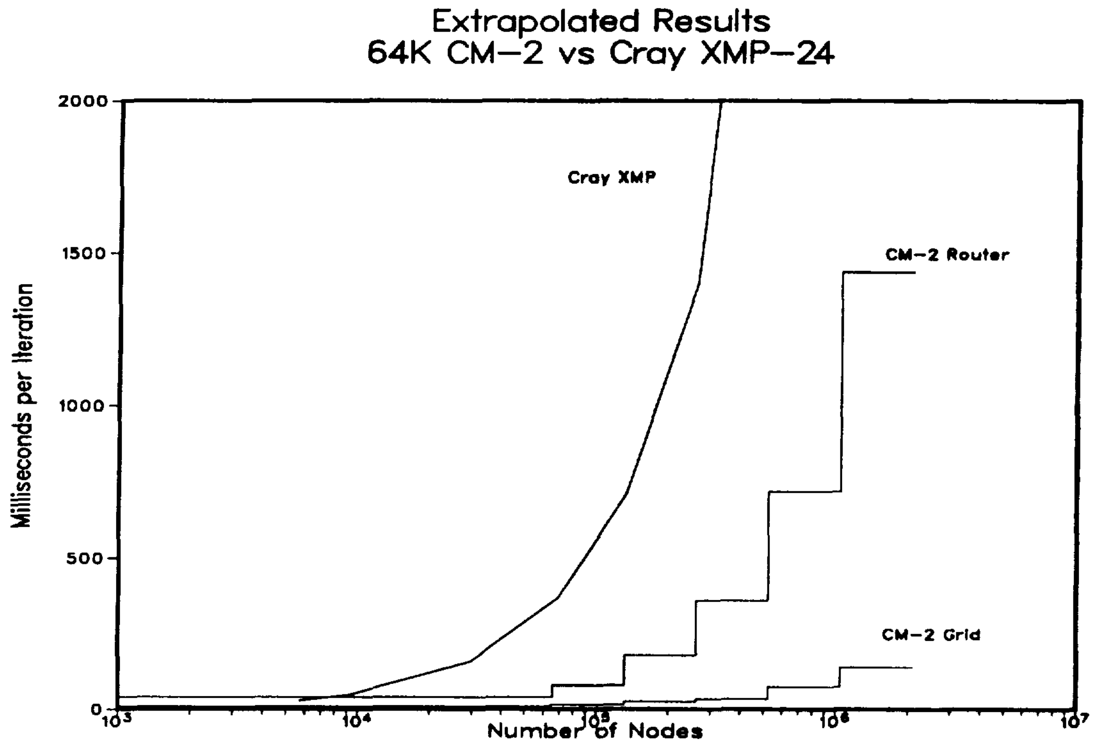


Figure 2: Comparison of the scaling of solution times between a Cray XMP-24 and the extrapolated results for a 64K CM-2

## 7 Acknowledgements

This work was supported by the United States Department of Energy under contract DE-AC04-76DP00789. REC wishes to thank J. Meza, R. Whiteside and W. Mason for many helpful discussions. We also wish to thank the NRL Connection Machine Facility for the use of their CM-2.

## References

[1] W.E. Mason, *TACO3D - A Three-Dimensional Finite Element Heat Transfer Code,* Sandia Laboratories Report SAND83-8212, April, 1983.

[2] G.H. Golub and C.F. Van Loan, *Matrix Computations,* Johns Hopkins University Press, Baltimore, 1983, Chapter 10.

[3] J. Grcar and J.C. Meza, *SPDPACK - A Package for Solving Symmetric Positive Definite Systems of Linear Equations,* Sandia National Laboratories Report, to appear.

[4] S.C. Eisenstat, M.C. Gursky, M.H. Schultz and A.H. Sherman, *Yale Sparse Matrix Package. I. The Symmetric Codes,* Technical Report 112, Yale University Department of Computer Science, 1977.

[5] S.C. Eisenstat, M.C. Gursky, M.H. Schultz and A.H. Sherman, *Yale Sparse Matrix Package. II. The Nonsymmetric Codes,* Technical Report 114, Yale University Department of Computer Science, 1977.

PRECEDING PAGE BLANK NOT FILMED

# FRACTAL GRAPHICS AND IMAGE COMPRESSION ON A SIMD PROCESSOR

S F Reddaway, A Wilson
Active Memory Technology
Reading
England

A Horn
Programming Research Group
Oxford University
England

## ABSTRACT

Fractal based Iterated Function Systems are producing very highly compressed graphics, images and movies, as well as being a promising approach to pattern recognition. The processing requirements are severe, and this paper examines the algorithms best suited to an SIMD processor array (in particular the AMT DAP510) both for encoding and image generation (decoding). One image generation algorithm, RCTA, has been implemented, and another quite different algorithm, ITA, is being implemented that will enable real time movies to be played. The paper both introduces and contributes to the IFS literature.

Keywords: Iterated Function Systems, Fractals, SIMD, Array Processor, Image Compression, Graphics, Movies.

## INTRODUCTION

Images are not random arrays of pixels, but have structure; if this structure can be discovered, there is great scope for data compression. The aim of compression may be later regeneration, or it may be to help recognise or match patterns. The work may start from an image or movie, or conversely the aim may be to generate graphics or animation. The applications potential is very wide.

Parts of images often have some similarity to other parts, often with a change of scale. The newly emerging science of Iterated Function Systems (IFS) exploits the self-similarity of images as a means of compactly defining images. Compression ratios as high as 10,000:1 have been claimed for some images, and even higher ratios for movies. The generation of images from the compact form has been reported to take up to 30 minutes on a Masscomp 5600 and compression has been reported (Ref. 1) to take up to 100 hours of man-machine interaction!

This paper has overlapping purposes. Of most direct interest to the conference is how an SIMD computer with an array of 1024 processors (the AMT DAP510) achieves high performance on some IFS algorithms. It is also a partial review of, and contribution to, the IFS literature from more of an engineering than mathematical angle. This is to help readers understand the algorithms and get into this rapidly expanding field. Ref. 1 is an easy introduction to IFS algorithms, and Refs. 2-5 are more detailed and up to date. Finally, some alternative machine architectures are discussed.

## IFS IMAGE REGENERATION THEORY

### Boolean Images

In simple IFS images, the compact form is represented by n contractive affine transformations with 6 coefficients each that map the whole image onto part of the image:

$$x' = ax + by + e$$

$$y' = cx + dy + f$$

The 6 coefficients are a to f, and there are n sets of these. Provided all the transforms are contractive, a unique binary image is defined solely by the recursive property that application of any transform to points in the image produces other points in the image. The image is known as the attractor of the

264 INTENTIONALLY BLANK

IFS, because iterative application of the transforms to arbitrary starting points converges to the attractor. This iteration can either be done by repeatedly applying every transform to the current set of points (pixels) and merging points falling into the same pixel, or else choosing a sequence of transforms according to probabilities, using them to progressively transform the coordinates of a point, and recording the pixels visited. The probabilities used in the second method do not affect the image (attractor) eventually generated, provided they are non-zero; however, the image is covered most uniformly and quickly if the probabilities are proportional to the area into which a unit of area is transformed. (The latter is given by ad - bc). Both methods converge at a rate dependent on the least contractive transform. If a fern is being generated, the least contractive transform has a linear contraction of about 15% and about 40 iterations are needed. (log (n)/log (1/S), where n is the required linear resolution, say 512, and S is the linear contraction, 0.85). Thus the first regeneration algorithm must generate about 40 intermediate images of steadily increasing accuracy, whilst the second algorithm should discard the first 40 points before recording pixels. The first algorithm lends itself to working in image space, with the transforms compressing, distorting and moving (binary) pixels arranged in raster arrays in the computer memory; it is thus termed the Image Transformation Algorithm (ITA). The ITA algorithm can be applied to points (pixels) held as coordinates, but the merging step (which is essential to prevent a combinatorial explosion of the number of points) is then more awkward. A key aspect of the ITA algorithm is that image resolution can start low and grow steadily. The second algorithm has been termed the Random Coordinate Transformation Algorithm (RCTA) and performs arithmetic on point coordinates.

**Grey and Colour Images**

A simple way of providing some colour is to associate a colour with each transform and colour a pixel according to the (last) transform that wrote the pixel. If the picture has been divided into segments that each have separate ets of IFS transforms, then the segments can be separately coloured.

However, a better approach to grey and colour is to use "measure theory", or in simple terms to include intensity. With RCTA, a count is kept of the number of times a pixel is visited, and the probabilities may be adjusted to achieve intensity effects. Full colour can be achieved by generating an intensity image and two colour difference images which may be at a lower resolution. With ITA a weight is used instead of a probability, and the intermediate (and final) images are typically held as 8-bit pixels; the pixel values are carried through the transformation, being added or interpolated when pixels are merged, and multiplied by the weight before being added to the next image. Either the weights are normalised in such a way that average intensity levels remain constant, or else the image will need to be renormalised at intervals. Colour can be dealt with as before, with intensity and 2 colour difference images.

Negative weights or probabilities can be permitted. The implication is that the transformed image is subtracted instead of added. In RTCA the pixel counter is decremented.

**Effect of Probabilities on Image Quality**

Ref.6 proves that RCTA converges to the same result as ITA; however, convergence is slow. For example, if it is desired that a particular pixel should have a 95% probability of being within 20% of the true intensity, it needs to be visited about 100 times. Similar accuracy with the ITA algorithm requires working to 4 or 5-bit precision, although one or two more bits may be desirable for dealing with low intensity parts of the picture.

**Extensions to the simple IFS representations**

One extension to the simple scheme is to have several partially connected images.

Each image will have its own set of contractive transforms which may map its own image onto part of itself, or else map one of the other images onto part of itself. One of the images is the desired output. This extension is described in Refs. 2 and 7, and does not add any basic problems to the image generation algorithms. The RCTA algorithm can be adapted by keeping independent points for each image, but recording the pixel hits only for the output image. Either the points for each image are updated in turn, or, more economically, points are updated only when new points are needed to serve the output image. Pixel storage is required for the output image only. The ITA algorithm may require storage for all images, but can take images in non-IFS form more easily than the RCTA algorithm.

Another extension, which can be viewed as a variant of the above, is to compose the final image from several separate images. This can either be by spatial segmentation referred to earlier, or it can be by adding overlapping images. The latter could provide a basis for compression. An ordered overlay of images is also possible, controlled by zero pixels in one of the images. The compressed IFS form is extended from the simple case by:

a) dividing the list of transforms into separate images,
b) specifying the output image, and
c) adding to each transform an eighth number specifying the source image.

The techniques of this section are called "condensation" in Ref.2.

**Animation and Movies**

Movies could be made with a succession of independent images. However, it is better to specify how transform coefficients change with time, as well as sometimes introducing new transforms (or images) and killing old ones. Thus every coefficient can be accompanied or replaced by rate of change coefficients, which do not need to be updated as frequently. Indeed, the update rate could be related to how quickly the rate of change is changing (ie acceleration) for that coefficient or transform. Thus the new data could be tagged to say which coefficient it applies to,.

One technique to produce interesting graphics is to progressively interpolate between two unrelated images.

**IMAGE GENERATION ON DAP**

An adaptation of the RCTA algorithm has been implemented on the AMT DAP 510, which is a SIMD computer with 1024 bit-organised Processing Elements (PEs). The ITA algorithm is currently being implemented on the DAP, and offers the prospect of eventually being an order of magnitude faster than the RCTA algorithm could ever be. For high quality colour images, the prospective improvement is more than two orders of magnitude, because of the effect of probabilities. The following sections are an update of Ref.8. The improvement in the RCTA algorithm of using the local random bit to choose between two transforms rather than selectively ignoring one transform is due to A Horn.

**Adaptation of the RCTA algorithm for the DAP**

Each of the 1024 PEs generates in parallel a semi-independent point sequence as follows:

a. Two Monte Carlo selections of the next transform are made globally.

b. Each PE uses a local random bit to decide which of the two transforms to use.

c. New points are converted from coordinate space to a 1024 x 1024 image space.

Fortran Plus code for the inner loop is:

```
      DO 100 J = 1, n
c     K1 and K2 are scalars that choose
c     the next transforms.

      K1 = choosevs1 (J)
      K2 = choosevs2 (J)
c     get a random plane.  Cost 200
c     cycles, including generation

      rp = rps (,,J)

c     perform transform in 1024 PEs.
c     6600 cycles
```

```
      cc = merge (c(K1), c(K2), rp)
      t = x * cc
      aa = merge (a(K1), a(K2), rp)
      bb = merge (b(K1), b(K2), rp)
      ee = merge (e(K1), e(K2), rp)
      x = x * aa + y * bb + ee
      dd = merge (d(K1), d(K2), rp)
      ff = merge (f(K1), f(K2), rp)
      y = t + y * dd + ff

c     convert to integer.  500 cycles
      ix = x
      iy = y

c     write dots to image space.  6600
c     cycles (special putdots)

      call putdots (ix, iy)

100   continue
```

**Note the implemented code includes 2500 cycles for scaling and positioning the coordinates prior to "putdots". The above code assumes the transform coefficients have been changed to make this unnecessary.**

**Illustration of semi-independence**
**If the globally selected pairs of transform sequences for an image with 7 transforms, T1, T2, .... T7, are:**

```
      T5  T3  T5  T7  T4  T2  T5  ...
      T1  T5  T2  T2  T5  T3  T5  ...
```

**and the corresponding random bits in 2 PEs are:**

```
      1   0   0   1   0   1   1   ...
      1   1   0   1   1   0   0   ...
```

**then the transform sequences in the 2 PEs are:**

```
      T5  T5  T2  T7  T5  T2  T5  ...
      T5  T3  T2  T7  T4  T3  T5  ...
```

## Generation of the Transform Choice Integers

Fortran - Plus Code:

```
c     pr is the cumulative probability,
c     rand1 and rand2 are each 1024
c     random numbers and choose1 and
c     choose2 are each 1024 integers

       choose1 = L1
       choose2 = L1
       do 200 J = L1 + 1, Ln

      choose1 ((rand1 + pr(J)).LT.1.) = J
      choose2 ((rand2 + pr(J)).LT.1.) = J

200    continue

c      convert choose integers from
c      matrices to vector sets.
       call convmv 2 (choose1)
       call convmv 2 (choose2)
```

**Performance of RCTA on the DAP 510**

| | |
|---|---|
| cycles for 1024 new points | 14K |
| new points/second | 740K |
| typical time/image | 1 second |

(A Masscomp 5600 workstation apparently goes at about 10K new points second).

**RCTA Movies on the DAP**
The technique of adjusting coefficients is illustrated by the movie of a waving fern. Note that the fern is changing shape, not merely rotating.

A fern has 4 transforms. Loosely speaking, 2 generate the first branches on either side of the stem, one generates the start of the stem and one causes the structure repetition including a gentle curve. The 2 coefficients (out of the 24) that control this curvature are progressively changed. Adding 2 successive images introduces motion blur.

The binary version of this simple RCTA movie has the following performance on the DAP510:

| | |
|---|---|
| New points/sec | 550K |
| Points/basic image | 238K |
| Points/displayed image | 476K |
| Display resolution | 1024 x 1024 |
| Frames/sec | 2.3 |
| Approx data required | 50 Bytes |

**The ITA Algorithm on the DAP**
IFS transforms are usually expressed for performing arithmetic on point

coordinates. The ITA (Image Transformation Algorithm) algorithm moves raster image data around memory. The primitive operations are:

a) linear shrink (can be negative) along a raster axis
b) shear (or skew) along an axis

c) translation along an axis

In terms of 2 x 2 transform matrices, a) can be written:

$\begin{pmatrix} x & 0 \\ 0 & 1 \end{pmatrix}$ or $\begin{pmatrix} 1 & 0 \\ 0 & y \end{pmatrix}$ for shrinks along x or y axes,

and b) could be written:

$\begin{pmatrix} 1 & u \\ 0 & 1 \end{pmatrix}$ or $\begin{pmatrix} 1 & 0 \\ v & 1 \end{pmatrix}$ for shears along x or y.

It is usually fastest to do the shrinks first, so as to do most work with the least data. Thus one way of rewriting a general 2 x 2 transform is:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ v & 1 \end{pmatrix} \times \begin{pmatrix} 1 & u \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} x & 0 \\ 0 & y \end{pmatrix}$$

Solving this gives:

$$x = a$$
$$y = (ad - bc)/a$$
$$v = c/a$$
$$u = ab/(ad - bc)$$

This rewrite will often give satisfactory parameters for the primitives; however, one of the "shrinks" may be an expansion, or else an angle of shear may be inconveniently acute. Other rewrites with the primitives are possible, and, fortunately, well-behaved parameters can always be found.

**Image Mapping**
Because image iterations are built from parts (which in turn are derived from the whole image), the implementation is using a 2D sheet mapping (Ref.9).

**Shear Primitive**
Shears do not change the number of pixels, and, because ITA is a robust convergent algorithm, interpolation is of little value; thus a row of pixels is shifted along its length by a whole number of pixel positions. Different rows in a sheet are shifted by different amounts. This is achieved by activity control of shifts of 1, 2, 4 and 8.

**Shrink Primitives**
A shrink is split into a power of 2 shrink and a shrink of between 0.5 and 1. The former might be re-used. Shrinks reduce the number of pixel rows. The new pixels can be formed in various ways. Selecting some rows and discarding others is one way; a better way is to interpolate so as "pixel mass" is conserved. Pixel mass is conserved in the power of 2 shrinks, and in the other shrink a new row of pixels is either an unchanged row, or else the average of two rows. The shifting for the power of 2 shrinks is done with Parallel Data Transforms (Ref.10), and for the other shrink by shifts of 1, 2, 4, 8 .. applied to a whole sheet-row, with activity-controlled overwriting.

**Translation**
The intention is to eliminate translation as a separate primitive by achieving translation alignment with DAP sheet boundaries as part of other primitives, and whole-sheet translations by addressing the correct sheets.

**Performance on ITA**
Many factors will affect performance. If resolution is increased appropriately through the iterations, then performance is dominated by the last one or two iterations only. An image with many highly contractive transforms will mean a lot of control work and work on partially filled sheets, but if all transforms are highly contractive few iterations are needed. For Boolean images control work will be more important, but for many bits/pixel, multiplication by the transform weight is not negligible. The overlap of transforms affects performance.

For this type of work the programming level and effort is very relevant; a target of 500 cycles per bit-plane of final image is reasonable, but initial high level implementation may be more than an order slower. For a 1024 x 1024 colour image with 8-bit pixels, plus two 8-bit colour difference images at half resolution, the above target performance is 600 msec. A Boolean image of the same size would be 50 msec.

**Movie Performance**

Because one frame is similar to the previous one, image iteration can start with the previous image. This might typically speed up by a factor of 1.5. But if data is supplied as rate of change of transform coefficients, then only rapidly changing transforms need be updated frequently; the contribution of each transform can be saved, and on update the old can be subtracted and the new added. On the DAP510, only the relevant part of the framestore is updated. If the movie contains a lot of rapid movement then the DAP510 will not be able to prevent the movement looking jerky at 1024 x 1024 resolution; then the resolution can be dropped to 512 x 512.

## COMPRESSION ALGORITHMS

Whilst IFS fractal graphics and animation have a promising future independent of the compression of natural images, the greatest interest is in compression, either for later regeneration or else as an aid to pattern recognition and matching. Less has been published about compression, maybe because it has mostly been done with manual interaction. We have not implemented compression on the DAP, but have considered some algorithms. These are: some variations on the collage theorem approach, 2D projections of 3D primitives, image segmentation, the use of moments, and the Fractal Transform.

**Collage Theorem**

This theorem is most easily pictured for Boolean images that represent shapes, but it is more general. If a shape is approximately covered by multiple copies of itself that are contracted, sheared, rotated and translated, then the corresponding IFS maps (transforms) collectively define an attractor (image) that is also an approximation of the original image (shape). The smaller distorted copies can overlap, but it is desirable that they are significantly shrunk, as the attractor is up to 1/(I-S) times bigger than the error in "covering" or "tiling" the shape. Here S is the shrink factor (<1) in the least contractive direction in the least contractive map (transform), and an error measure such as Hausdorff distance is used. Errors in the "tiling" of a smooth shape may result in fractal shaped attractors, but the technique can be applied to arbitrary images. Both the theory and the technique can be extended to grey and colour images, where a different distance metric is used, and overlapping distorted copies are added. The aim is still to produce approximate images.

**Application of the Collage Approach**

Ref. 2 describes interactive use of the Collage theorem; the human is good at pattern recognition. A more automatic approach might be to try all sensibly different transforms and measure how good a fit to part of the image is achieved for each one, with a view to selecting a set of the best fits (smallest distance measures) while avoiding heavy overlap between pairs in the set. This initial approximation should avoid the local, minima difficulties of Ref 11.

"All sensibly different" transforms implies a large number. If the criterion is differing by at least one pixel position, then for a linear resolution of n, there are approximately n**6 different transforms, and the average number of pixel pairs involved in comparing the distorted image against part of the original is nearly n**2. The work thus appears to vary as n**8; advanced algorithmic techniques involving re-use of intermediate results, for example with convolution methods, should reduce this dependence to nearer (log n)* n**6. This work is still prohibitive for full resolution images; for example, a 1000 x 1000 image would involve 10**24 pixel operations with direct methods, or 10**19 operations with advanced methods.

An encouraging approach is to perform the initial collage search with a very coarse version of the image; a 10 x 10 image would require around 4*10**6 operations, which is quite feasible in real time. Even for Boolean originals, coarse images are grey; the above analysis has not explicitly allowed for choosing the transform weights.

Having selected a limited set of good transforms (plus weights) that do not greatly overlap, the set can be iterated in detail to improve the fit. This can either be done as a collage applying each transform to the original image, or by generating the attractor (including intensity) of the set. Either way, smooth variation of the IFS parameters gives smooth variation of the image. At this iteration stage the image resolution can be increased. When further improvement seems to be limited with that set of transforms, a full resolution attractor is generated and subtracted from the original image. This remainder (which may include negative pixels) can then be put through the full cycle again to generate an attractor that forms a further "layer" of the image. The process can continue until a satisfactory approximation is achieved. It would seem desirable to keep the number of transforms in each set quite small in order to ease the optimising work in the iterations.

The above "layering" is a form of segmentation of pixel intensity, with the final pixel value formed by adding the layers. Another approach is to spatially segment the image (or image remainder) prior to choosing transform sets. The IFS sets may then be chosen so the attractors do not overlap (or the overlapping intensities are small), or else an ordering can be specified with later segments overwriting earlier ones. Segmenting may be automatic or manually assisted; for the former, the algorithm will involve rules that may be difficult to devise effectively, especially for non-specific images.

Repeated patching-in of variations of an object can be a powerful tool, and can be achieved with multi-screen techniques. The automatic recognition of such possibilities may be difficult; looking for good matches of parts of the original images mapped onto other parts could be one way that might be feasible for 2D distortions. Real world pictures are 2D images of a 3D world, and it would seem that this is used to advantage in the "sunflower" picture publicised by Barnsley's group (Ref.2). The individual sunflower primitive is described as an overlay of 4 IFS attractors; this would be one screen, which is then overlayed (by "condensation") onto the final image once for each flower in the picture. It would seem, however that the sunflower "primitive" must be a 3D IFS with different projections used for the different flowers. This is a good model of the real world, but it is difficult to see that IFS structure being automatically generated from a single photograph; with human knowledge of the real world it becomes feasible.

**Moment Theory**

If 2D space is viewed as the space of complex numbers, then the nth moment of an image can be defined as the sum of the products of the pixel values and the nth power of the pixel coordinate. The IFS literature contains some discussion of moments, and in particular the idea that the first n moments of an image can be calculated and that an IFS can then be sought with an attractor (plus intensity) having approximately the same moments. If high order moments are relatively unimportant, a reasonable approximation to the image might be obtained with, say, 10 or 20 moments and the approach can be used recursively as with the collage approach. The advantage is that the image information has been reduced to 10 or 20 numbers and it is these that are used to fit the IFS approximation. The powers of the complex coordinate can be calculated once and for all, and stored; with coarse resolutions of order 100 x 100, the storage space should not be a problem.

**Fractal Transform**

A recent press release from Iterated Systems (1988) claims to have discovered a "Fractal Transform" that could compress and regenerate images. This is either with exact reproduction and typical compression ratios of 10 to 100:1, or else approximately, with much higher compression ratios. The speed for either compression or regeneration of under 10 seconds for a 256 x 256 image of 8-bit pixels on a SUN workstation indicates that an extensive optimisation process cannot be involved in the compression; possibly moment theory is involved. For the time being ISI are keeping the details confidential.

## SUITABILITY OF THE DAP FOR COMPRESSION

### Collage approach

Central to the Collage approach is the measuring of the difference between pairs of images. This may either be between a small number of pairs of quite big images in the optimisation stage, or else be between a large number of small image pairs (one or more to a PE) in the initial "try everything" stage. A family of algorithms for performing similar measurements has been studied on the DAP (Ref.12); the algorithms can be extended to the re-use of intermediate results for the try everything stage. The work is done entirely in raster pixel space, rather than in coordinate space. The distortions (transforms of the original image) are a very minor part of the try everything stage, because partially transformed data is used many times over. For example, a line of pixels resulting from a rotation and a linear shrink along the line, will be matched against lines resulting from many different translations, skews and linear shrinks in the orthogonal direction.

The core work in the try everything stage is differencing pixels and summing the differences, together with a little data movement that is mainly broadcasting data. In mapping arrays onto the DAP, "crinkle" mapping ("domain partitioning") of arrays is important. This ensures that most of the summation is within PEs, that there is little data movement and that the multi-grid aspects of changing resolution are efficient (see for example Ref.13).

In the optimisation stage the work is mainly generating intermediate resolution images with the ITA algorithm, starting with a good approximation; also important is measuring the difference between images.

An indication of DAP 510 performance on measuring the difference between images can be derived from the above reference as less than 5 msec for a pair of 1024 x 1024 images of 8-bit pixels. The try everything stage for 10 x 10 resolution would take about 40 msec. Clearly, there is a trade-off of speed against quality. For TV it is difficult to predict the quality achievable in real time. The use of rate of change techniques (see 2.5) can greatly improve the compression ratio and the compression speed.

### Moment Theory Approach

The calculation of moments could be done quickly on the DAP. With the complex coordinate powers pre-computed, each moment requires 2 multiplies and 2 adds per pixel; with 8-bit pixels and a 256 x 256 image, each moment could take as little as 2 msec.

## ASSESSMENT OF ARCHITECTURES AGAINST ALGORITHMS

For image generation the RCTA and ITA algorithms have very contrasting requirements. Aside from random number generation, which the DAP can do very fast (Ref.14), the RCTA algorithm requires arithmetic on coordinates, and incrementing one out of a raster of counters. A normal scalar machine can perform these operations fairly efficiently. An MIMD array could perform quite powerfully, provided it (a) had a fast random number generator, (b) had enough space for a complete raster in each processor, and (c) was effective in summing the rasters across the processors. An SIMD array without local indexing is likely to be fairly slow in converting its arrays of data from coordinate space to raster space. This is the case on the DAP, although its fast central addressing of the array memory means that it takes only about half the total time. A bit-organised arrav like the DAP would be able to tailor arithmetic precision with considerable advantage, because the precision requirement is modest. The implemented code uses 24-bit floating point for convenience. A factor of about 3 on the arithmetic (about 2 overall) could be achieved by further tailoring, but this is not worthwhile because the ITA algorithm is a much faster prospect. Decision-making with random bits suits a bit-organised machine. With the modified RCTA algorithm there is no shortage of parallelism for the arithmetic, especially for high quality pictures. The DAP 510 measured achievement of 600,000 points/second on RCTA is good, but might be bettered on some architectures.

The ITA algorithm has the potential for being at least an order of magnitude faster than RCTA on the DAP, and more than two orders for high quality images. However on most machines RCTA will still be faster than ITA except for very precise images. At its heart, ITA requires data to be moved around memory in a raster-like way, but with fairly flexible control patterns. It also requires some pixel-precision array arithmetic. This is almost ideal for the DAP; even higher performance could be achieved if more flexible array routing hardware were provided. Neither conventional scalar machines nor MIMD arrays provide the fast and flexible array shifting capability required for high performance. The DAP 510 performance on a 1024 x 1024 colour image with about 5% pixel accuracy might eventually be about 400 msec or about 2.5 million 6-bit pixels/second; the RCTA algorithm for similar precision would need to generate about 400 million points for a standard deviation of 5%, so the RCTA equivalent performance would be about 1000 million points/sec. That is a difficult figure for any machine to match with the RCTA algorithm. (The DAP610 is four times more powerful than the DAP510).

The Collage approach to compression involves highly parallel low precision work, change of resolution and use of the ITA algorithm. Thus the DAP architecture looks to be almost ideal. Calculating moments for the moment theory approach is also close to ideal. An advantage of DAP over specialised hardware is that IFS work is likely to be part of a wider application. For example, compression might be used as a first step to pattern recognition. Doing the whole job in the DAP is a great advantage.

**APPLICATIONS**

If the results live up to the claims made for it, then the prospects for these techniques, and future developments of them, are extremely bright. The techniques are wide, covering graphics, animation, TV, movies and images of all kinds, with impact on transmission, storage and pattern recognition. The potential application fields are very wide, covering medical, defence, (for example, image understanding), TV etc. The leading company in the field is Iterated Systems, whose first product is known as VRIFS, which was first implemented on a SUN; the company now has a DAP 510, and is porting code onto the DAP.

**REFERENCES**

1. Barnsley M F and Sloan A D. "A Better Way to Compress Images", Byte Magazine, January 1988. pp 215 - 223.

2. Barnsley M F, Jacquin A, Renter L, Sloan A D. "Harnessing chaos for Image Synthesis" Computer Graphics, vol 22, number 4, August 1988. pp 131 - 140. (Siggraph 1988).

3. Barnsley M F. "Fractal Modelling of Real World Images" in "The Science of Fractal Images", edited by H Peitgen and D Saupe. (Springer - Verlag, 1988).

4. Barnsley M F. "Iterated Function Systems" in Lecture Notes"Chaos and Fractals: the Mathematics behind the Computer Graphics" American Mathematical Society, August 1988.

5. Barnsley M F. "Fractals Everywhere". (Academic Press, 1988).

6. Elton J. "An Ergodic Theorem for

Iterated Maps" Ergodic Theory and Dynamical Systems 7 (1987) pp 481 - 488.

7. Barnsley M F, Berger M A and Soner H M. "Mixing Markov Chains and their Images". Preprint.

8. Reddaway S F and Wilson A. "Regeneration of Images from IFS Codes on a Processor Array", Poster Paper, SIAM Annual Meeting, Minneapolis, July 1988.

9. Reddaway S F. "Mapping Images onto Processor Array Hardware" in "Parallel Architectures and Computer Vision", edited by I Page (Oxford, 1988).

10. Flanders P M and Parkinson D. "Data Mapping and Routing for Highly Parallel Processor Arrays" Future Computing Systems 2 1987 pp 183 - 224.

11. Levy-Vehel J and Gagalowicz A. "Shape Approximation by a Fractal Image" Eurographics 1987 (North Holland, 1987). pp 159 - 180.

12. Reddaway S F. "Some Fast Measurements on Images" paper presented at SIAM Annual Meeting, Minneapolis, July 1988.

13. Reddaway S F. "Achieving High Performance Applications on the DAP" paper at CONPAR 1988, to be published by Cambridge University Press, 1989.

14. Smith K A, Reddaway S F and Scott D M. "Very High Performance Pseudo-random number generation on DAPs" Computer Physics Communications 37, 1985, pp 239 - 244.

# APPLICATION OF MASSIVELY PARALLEL MACHINES TO MOLECULAR DYNAMICS SIMULATION OF FREE CLUSTERS

L. L. Boyer and P. J. Edwardson*
Complex Systems Theory Branch
Naval Research Laboratory, Washington, D.C. 20375-5000
*NRC-NRL Postdoctoral Associate

## ABSTRACT

Molecular dynamics is used to examine melting of a free cluster of up to 1000 sodium and fluoride ions. An algorithm designed originally for the Distributed Array Processor (DAP) is implemented on the Connection Machine (CM), and their performance for this problem is compared with that of the Cray 1. The CM would be twice as fast as the DAP using twice the number of processors as the DAP's 4096 if not for saturation of the VAX front end. Saturation of the front end increases the run time by 50% in typical cases.

Keywords: Molecular Dynamics, Massively Parallel Processing, Connection Machine, Distributed Array Processor, Single-Instruction-Multiple-Data Computers, Melting, Sodium Fluoride, Ionic Clusters.

## INTRODUCTION

Molecular dynamics (MD) simulation is becoming an increasingly important tool for research in materials science, owing to new methods for deriving realistic interatomic potentials and advances in computer performance. Even though future major advances in computer performance are likely to involve parallel computation in some form, relatively few MD calculations have actually been carried out on massively parallel machines. Here we discuss some techniques available for MD simulations which exploit the power of massively parallel computation, focusing especially on a relatively simple approach for simulating the dynamics of free clusters of particles using massively parallel single-instruction multiple-data (SIMD) computers.

Molecular dynamics simulations are usually carried out for systems with periodic boundary conditions. The use of periodic boundary conditions has some advantages over cluster simulations: 1) it presumably requires fewer particles to simulate a "bulk" property and 2) the pressure on the system is easily controlled, since all of space is filled. In addition, periodic boundary conditions can be exploited in developing algorithms for massively parallel systems. Specifically, for systems with little or no diffusion, one can map the particles to processors in a manner which allows easy transfer of data between neighboring particles. This clearly leads to great efficiency if the particles do not interact with long range forces. Pawley and coworkers (Ref. 1-3) have employed this approach in a number of MD calculations using the ICL Distributed Array Processor (DAP) at the University of Edinburgh (Ref. 4). An algorithm which exploits periodic boundary conditions for computing long range forces, the so called $P^3M$ method (which scales as NlnN, where N is the number of particles), also has been applied recently in MD calculations using the DAP (Ref. 5).

The first MD simulation of free clusters using a massively parallel computer were carried out on the DAP (Ref. 6). An algorithm called the row-column difference (RCD) method was introduced which proved to be remarkably efficient for the DAP. The DAP is an SIMD machine consisting of 4096 single-bit processing elements connected on a square grid. In the RCD method the coordinates of a group of 64 particles are put in the rows of one of the natural 64x64 matrices and the coordinates of another group of 64 particles are put into the columns of another natural matrix. Subtracting the two matrices gives the relative coordinates between all possible pairs formed from the two sets of particles. Forming a double loop over all distinct sets of 64 particles one can compute and accumulate the forces due to all pairs by doing them 4096, or very near to 4096, at a time. Some special care must be taken when computing the interactions involving only one set of 64 particles (see Ref. 6 for detail). Even though this "brute force" approach is an $N^2$ algorithm, i.e., the time scales as the square of the number of particles, it efficiently exploits the parallel architecture. Thus, it can be more efficient than NlnN algorithms for N up to a thousand or more. As mentioned above, simulation of certain phenomena require both free surfaces and long range forces: For such problems the $N^2$ algorithm is essential.

The simulation of free clusters, i.e., systems with free boundaries rather than periodic boundaries, has some advantages over bulk calculations. Obviously, free-cluster calculations provide the most realistic zero pressure simulations. Moreover, one can study both bulk and surface properties by the same technique provided the system is sufficiently large. Calculations of thermal expansion of NaF (Ref. 6) suggest that "sufficiently large" is approximately 500 hundred or more particles.

CH2649-2/89/0000/0275$01.00 © 1988 IEEE

While surface effects are clearly of great interest in their own right, the proper simulation of certain phenomena requires both large systems and surfaces; two examples being melting and the response of a system of charged particles to an electric field (Ref. 7-8).

## APPLICATION OF THE RCD METHOD

The RCD method is most easily illustrated by considering a specific example: assume we have a computer with 16 processing elements (PE's) on a 4x4 NEWS grid. Let our system contain at least 8 atoms located at positions $(x_i, y_i, z_i)$. The position coordinates are simply a set of numbers stored in some convenient fashion among the PE's whose values have no relation to the structure of the PE's. The procedure begins by transferring sets of 4 of these numbers to rows and/or columns of the natural sized 4x4 matrices. In particular let

$$XR - \begin{matrix} x_1 & x_2 & x_3 & x_4 \\ x_1 & x_2 & x_3 & x_4 \\ x_1 & x_2 & x_3 & x_4 \\ x_1 & x_2 & x_3 & x_4 \end{matrix}$$

be the matrix whose rows are the x coordinates of the positions of atoms 1-4, and

$$XC - \begin{matrix} x_5 & x_5 & x_5 & x_5 \\ x_6 & x_6 & x_6 & x_6 \\ x_7 & x_7 & x_7 & x_7 \\ x_8 & x_8 & x_8 & x_8 \end{matrix}$$

be the matrix whose columns are the x components of atoms 5-8. Then the relative separations of all pairs from these two sets of atoms are given in the matrix

R - SQRT((XR-XC)**2 + (YR-YC)**2 + (ZR-ZC)**2)

where YR, YC, ZR and ZC are the analogous matrices for the y and z components. If the atoms interact with the Coulomb force then the x components of this force are given by elements of FXC-(XR-XC)/R**3 . The forces on atoms 1-4 are obtained by summing the rows of FXC and the forces on atoms 5-8 are the negative of the values derived from summing the columns of FXC. Techniques for handling interactions between atoms in a single set of four and in partly filled sets are discussed in Ref. 6.

In an MD simulation the time required to update the positions and velocities is usually a negligible fraction of the time required to compute the forces. This is certainly the case for our problem. Observe that three separate operations are required to compute the forces using the RCD method; 1) spreading data from a vector to form a matrix with identical rows or columns, 2) performing the arithmetic operations required to compute the forces, and 3) collecting the results by summing over rows and/or columns. Obviously, the spreading and collecting operations must be performed efficiently for the RCD method to be a viable approach, since this time is independent of the functional form of the potentials. The calculations carried out on the DAP employed a functional form consisting of the Coulomb interaction plus 4 exponential terms (Ref. 9), which resulted in approximately 60% of the total time being spent on the arithmetic operations.

The particular Connection Machine used to carry out the calculations reported here consisted of 512 chips, with 16 PE's per chip. Communication between chips occurs along wires with connections prescribed by a 9 dimensional hypercube. Communication between PE's within a chip are much faster. Floating point operations on the CM were performed by 256 floating point units accompanied by "Sprint" routers to handle communications between the PE's of designated chip pairs and the floating point units. NEWS programming for the Connection Machine is accomplished by a software package, which gives the user some freedom to select the number of "virtual" processors in a virtual NEWS grid. Increasing the VP ratio (number of virtual processors per physical processor) permits the pipelining of data through the floating point units, which can give a substantial increase in performance (Table 1).

Table 1. Timing results (sec per time step) for MD simulations on the CM (8192 PE's), DAP (4096 PE's) and Cray XMP-24 computers.

| | | | 512 ions | | 1000 ions | |
|---|---|---|---|---|---|---|
| Computer Type | Block Size | VP Ratio | CM Time | Elapsed Time | CM Time | Elapsed Time |
| CM | 128 | 2 | 0.34 | 1.26 | 1.18 | 4.42 |
| CM | 256 | 8 | 0.23 | 0.48 | 0.76 | 1.44 |
| CM | 512 | 32 | 0.23 | 0.32 | | |
| DAP | 64 | | | 0.5 | | 2.0 |
| Cray | | | | 0.3 | | 1.14 |

On the other hand, the RCD method becomes less efficient when the number of processors exceeds about $(N/2)^2$. These two effects combine to give only small improvement in going from a VP ratio of 8 to 32 for the N - 512 system. For the 1000-ion cluster we were limited to a 256 grid by memory constraints. The most efficient calculation for the 512 ion cluster was achieved for the 512x512 NEWS grid; specifically, 0.32 sec per time step. (The elapsed time is larger than the CM time because the host, which in our case is 1 processor of a VAX 8800, is 100% saturated by our process. Using larger NEWS grids dramatically lessens the burden of the host, but even for a VP ratio of 32, the host is still the bottle neck.) This compares with 0.5 sec for the DAP (negligible time on the DAP host) and 0.3 sec on the Cray XMP-24. About 75% of the CM time was spent spreading and collecting data, whereas on the DAP these operations required only about 40%. On going from N-512 to N-1000 ions, the DAP time increases by a factor of 4, rather than $(1000/512)^2$ - 3.81, because the same number of operations are required for an N-1000 or an N-1024 cluster. The Cray scales by the expected factor of 3.81. On the CM, the calculations for the N=1000 ion system are somewhat more efficient than would be expected from the timing of the 512 ion system.

## RESULTS FOR NaF

Molecular dynamics simulation of 216- and 512-ion clusters of NaF have been performed to study various properties: thermal expansion, melting, diffusion and responses to external electric fields, specifically, electrostriction and infrared absorption (Ref. 6-8). One of the more intriguing results of these calculations pertains to the melting transition and the presence of a rather dramatic kink in the plot of "temperature" vs energy. Temperature is placed in quotation marks because very long simulations are needed near the transition to obtain a good average kinetic energy, from which the equilibrium temperature is determined. Interestingly, a kink occurs for both melting (increasing energy) and freezing (decreasing energy), the details of which permit one to make the following statement with confidence. The energy range for which solid and liquid phases coexist in these clusters is much less than the latent heat.

This poses a fundamental question: Does the presence of the kink and associated absence of two-phase coexistence result from not having a truly macroscopic sample? Or, does two phase coexistence result when a system is not in perfect isolation long enough to achieve equilibrium? To help answer this question we have performed calculations which simulate melting for a 1000-ion cluster of NaF. Results for temperature (T) vs half the total energy (E/2) are shown in Fig. 1. A slope of unity on such a plot indicates exact equipartition of energy, which results for perfectly harmonic systems.

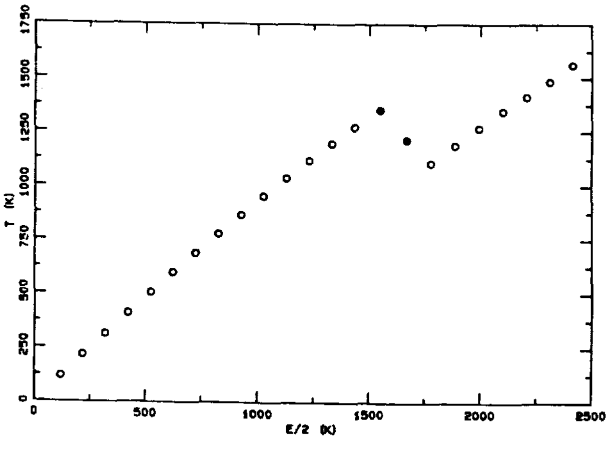


Fig 1. Plot of temperature vs half the total energy, both expressed in Kelvin, for a 1000-ion cluster of NaF. The energy has been shifted to give zero at T=0 . Filled circles indicate that longer simulation could well change the "temperature", while open circles results showed no drift in the kinetic energy. The experimental melting temperature is 1265 K.

The simulation was started with the ions in a perfect cube shaped microcrystal at low temperature. Constant energy simulations were performed for ~10 psec to determine the temperature at each energy. Near the melting transition longer simulations, up to ~100 psec, were carried out in an effort to avoid superheating. The energy of the system was changed by scaling the velocities, and the points in Fig. 1 were obtained by increasing the energy of the previous point. The solid points are used to indicate that longer simulations could well produce a further change in temperature. The results obtained thus far do not show substantial differences from the simulations of the 512-ion system. For both systems the vertical dimension of the kink is ~250 K and the horizontal dimension agrees well with the experimental latent heat. Simulation of the corresponding freezing curve is underway and will be reported elsewhere.

## REFERENCES

1. M.T. Dove, G.S. Pawley, G. Dolling, and B.M. Powell, Mol. Phys. 57, 865 (1986); and references therein.

2. K. Refson and G.S. Pawley, Mol. Phys. 61, 669 (1987); ibid., 693 (1987).

3. G.S. Pawley, Solid State Commun. 53, 817 (1985).

4. R. W. Hockney and C. R. Jesshope, "Parallel Computers", (adam Hilger Ltd., Bristol, UK, 1981).

5. A. Brass, Ph.D. Thesis, University of Edinburgh, 1986.

6. L. L. Boyer and G.S. Pawley, J. Comp. Phys., 78, 405 (1988).

7. L. L. Boyer, "Application of Vector and Parallel Machines to Molecular Dynamics of Large Clusters", Proc. of Third Inter. Conf. on Supercomputing (L. P. Kartashev and S. I. Kartashev, editors, International Supercomputing Institute, Inc. 1988) vol. 1, p. 187.

8. L. L. Boyer, P. J. Edwardson and R. E. Cohen, "Ab Initio Calculation of Electrostriction and Field Induced Instability of NaF", Bull. Am. Phys. Soc. 33, 627 (1988 March Meeting).

9. L. L. Boyer, Phys. Rev. B 23, 3673 (1981).

# AN OVERVIEW OF CURRENT CONNECTION MACHINE APPLICATIONS AT MRJ

R. Michael Hord

MRJ, Inc.

## ABSTRACT

The Perkin-Elmer Advanced Development Center at MRJ installed the first commercial Connection Machine™ (CM) supercomputer in August 1986. A second 16K CM was installed in March 1987 and was upgraded to a CM–2 in January 1988. A Data Vault and Frame Buffer have also been added.

Previously, a variety of CM applications at MRJ have been reported. These include maze solving, line of sight, text processing, HI-CAMP target tracking, automatic target detection, FFT, image processing primitives, image model matching, stereo matching, knapsack, neural networks, nonlinear network optimization, thermal diffusion, wave equation, pyramid linking for image segmentation, and rapid 3D rendering.

This paper surveys diverse current CM applications at MRJ. Both contract and internal R&D activities are addressed. These include: 3D electromagnetic scattering models using both finite difference and method of moments; signal processing; system scheduling; synthetic aperture radar pattern recognition; robotic arm control; optimal resource allocation; traveling salesman; 3D shape recognition; large matrix solution and eigenvalue determination; Monte Carlo techniques; computational fluid dynamics; communications network reconstruction; multispectral image analysis; strategic defense simulation; battlefield analysis; optical system ray tracing; logistics models; and symbolic computation. A FORTRAN to CM language semiautomated translator facilitated these efforts.

**Keywords:** Applications, Scattering, Images, Signals, Operations Research, Engineering, Fluids

The Perkin-Elmer Advanced Development Center at MRJ installed the first commercial Connection Machine™ (CM) supercomputer in August 1986. A second 16K CM was installed in March 1987 and was upgraded to a CM–2 in January 1988. A Data Vault and Frame Buffer have also been added. The CM produced by Thinking Machines Corporation in Cambridge, Massachusetts, is a fine-grain massively parallel supercomputer. The CM–1 has a Symbolics 3675 host and is characterized by a 4MHz clock and 4K bits per processor. The CM–2 is a second generation of this technology with a 16-fold increase in memory per processor, floating point hardware, faster clock, and a much increased input/output rate. The CM languages in use at MRJ are PARIS, *LISP, and C*.

Previously, a variety of CM applications at MRJ have been reported. These include maze solving, line of sight, text processing, HI-CAMP target tracking, automatic target detection, FFT, image procesing primitives, image model matching, stereo matching, knapsack, neural networks, nonlinear network optimization, thermal diffusion, wave equation, pyramid linking for image segmentation, and rapid 3D rendering.

This paper surveys diverse current CM applications at MRJ. Both contract and internal R&D (IR&D) activities are addressed.

*Electromagnetic scattering* models have been implemented on the CM for computing the near and far fields of diverse 3D bodies. Initial efforts addressed both finite difference time domain models and method of moments models, and considered a range of materials, shapes, and sizes of both simple and complex objects. The finite difference time domain code addressed nonhomogeneous time domain composite dielectric or perfectly conducting scatterers. The scattering object size on a full 64K sized CM–2 can be as large as 24 lambda x 24 lambda x 10 lambda, where lambda is the wavelength of the incoming wave.

The method of moments code addresses homogeneous dialectric scatterers. Matrices are complex and dense and are solved to size 4000 x 4000.

A system study addressing the insertion of a CM into a large operational system for expedited *system scheduling* was successfully concluded. A benchmark activity executed a scheduling algorithm on a variety of computers including an IBM 3090/200, Cray X–MP1, and Cray Y–MP8 for comparison with the CM–2 with 8K, 16K, and 32K processors.

Vector and Parallel Computing for *Large Scale Network Optimization,* research performed by Professor Stavros Zenios of the Wharton School of the University of Pennsylvania and Dr. Robert Lasken of the Advanced Development Center at MRJ won in March 1988 a special award in the Gordon Bell competition organized by the IEEE computer society. The research was performed in part on the CM and supported by MRJ. The topic was drawn from the field of Operations Research and deals with the optimal flow of a commodity through a network characterized by nonlinear costs. Runtime comparisons were made with Cray and other computers showing an advantage for the CM in speed and problem size.

A variety of operations research/control theory problems were addressed by the CM under contract. Multiply articulated *robot arms* of complex configuration were manipulated in the same physical workspace without collision and without constraint (cable wrap, velocity, acceleration, jerk, etc.) violation in minimum time.

A large set of diverse *resources* was *allocated* to competing demands in an optimal way with provision for both rapid replan

CH2649-2/89/0000/0279$01.00 © 1988 IEEE

and minimum disruption upon changes in demand. *Orbit optimization* planning was also performed. Several other operations research problems have been studied in a CM context. These include parallel algorithms for the traveling salesman problem for which near optimal solutions for very large tours are obtained very quickly.

The *traveling salesman* problem seeks the optimal (shortest) path among N nodes uniformly distributed in a plane based on a Euclidean distance metric. MRJ devised a two-step approach that obtains a good first guess solution by fractal seeding and then refines that solution using a barter method. Eight thousand nodes were solved to within 1% of optimal in 20 minutes on the CM-1.

Under DARPA funding, the CM is being employed to develop pattern recognition algorithms for *synthetic aperture radar* images of the ocean surface.

Perkin-Elmer makes a *ranging/imaging sensor* with active laser ground scan illumination. This IR&D effort uses the CM to convert range data to elevation information, thereby resolving range ambiguity, and then applies 3D pattern recognition methods together with reflectance images to detect objects in a ground scene. For this project the ranging data were processed to find net covered camouflaged trucks among trees. The laser ranging data were produced by the Perkin-Elmer 3-D Line Scanner. This laser line scanner looked down at Earth from an air breathing platform flying at a 229 meter altitude. The scanner sweeps a collimated beam of laser radiation through a 120 degree arc. The laser radiation has a wavelength of 0.85 microns. From the returning laser radiation both the range and surface reflectivity are recorded.

Processing included two broad areas. The first area included removing artifacts in the data. These artifacts are ambiguities caused by the collection process, which are not naturally found in the scene. The second area was pattern recognition. This effort succeeded in highlighting trucks hidden in the image.

Trucks represent a good focus for pattern recognition. Automatically identifying trucks in an image is of military interest for intelligence purposes or for targeting purposes. Methods that work for trucks should extend to other military targets such as tanks and mobile missiles because they are all of similar but unique size and shape.

Automatic techniques are necessary because the laser line scanner produces large amounts of data; in the unprocessed data there is low contrast between the trucks and the background, and there is not enough manpower to analyze the data in a timely manner. By highlighting the trucks, automatic target recognition solves these problems.

*Stereo image matching* was used to generate a 3D scene model for interactive manipulation. Using two airborne visible USGS images of Tysons Corner, VA, acquired from unknown altitudes and at different times of day, an apartment complex was displayed at arbitrary azimuth and elevation angles and range with an arbitrary illumination direction in about a second per view.

The major functions of the 3D scene model are:

1) registration (2D)
2) determination of the axis of shift
3) feature extraction
4) image segmentation
5) matching
6) elevation computation
7) model display.

One unusual characteristic of this method is that the matching works with a 5D feature vector associated with each pixel. One set of five features that works well is:

1) neighborhood average
2) edge magnitude
3) edge direction
4) constant false alarm rate
5) max–min texture.

Structures analysis using NASTRAN is a major MRJ business area. This IR&D project's first task was to design a method of using the CM for the computationally intensive sections of NASTRAN runs. The second task sought methods for the analysis of dynamic structures. The primary thrust of this effort to date has been the development of *linear algebra* tools: matrix solvers, matrix inversion, and eigenvector determination. Versions of these tools are available for dense unstructured matrices and special versions are also available for certain matrix types. For example, for block tridiagonal matrices, sizes of up to 64K x 64K are accommodated.

Calculation of the behavior of *optical systems* is known to be computationally intense. An example of such a calculation is the examination of the behavior of the diffraction pattern formed by an objective lens or mirror under varying conditions of obscuration by, for example, secondary optics support structures. There are a number of ways to perform this computation, including analytically (symbolically) and by use of Fourier transforms. In the present case, the use of *Monte Carlo* techniques for determining the diffraction pattern by direct calculation is explored, making use of the substantial computational capabilities of the CM to produce results rapidly.

A program, written in the parallel computing language *LISP, calculates the interference pattern produced by a user-selected number of randomly chosen rays passing through the chosen aperture relative to the central reference ray. These interference patterns are ultimately summed, and their square gives the observed luminous intensity of the diffraction pattern. A number of typical apertures can be specified, including those for Newtonian and Cassegrain reflector designs, as well as some more unusual choices. The program user may also select the aperture size and the wavelength of light in use and observe the effect that these choices have on the resulting pattern in the focal plane.

The CM and its host computer provide a particularly effective model for this problem, as the system structure parallels that of the physical model: the host computer plays the role of the aperture, while the CM is analogous to the focal plane. Light rays passing through the aperture and forming patterns over the entire focal plane correspond to the movement of data from the front-end computer to the various CM processors in parallel. This correspondence of structure makes the implementation of the central computation routines quite straightforward, while the substantial graphics capabilities of the Symbolics host computer allow the development of a convenient and high-powered user interface. The details of the implementation also serve to illustrate such valuable parallel processing techniques as problem replication and segmentation, the former being used for the parts of the problem much smaller than the configured number of processors, with the latter being employed where the problem was bigger than the machine.

*Turbulent flow* around vehicles moving through the atmosphere has previously been modeled in FORTRAN on sequential computers in the WAKE program. This code is being migrated under this project to the CM for faster execution.

A second computation fluid dynamics activity addressed ***gaseous diffusion polution modeling***. The CM solves the diffusion equation in three dimensions to propagate smoke plume size and shape subject to effects of the surrounding terrain, local wind velocity, and other model parameters. A time averaged model in 512 x 512 x 21 cells generates a graphic display.

Using terrain models, doctrine, equipment constraints, and other criteria, a CM-based system derives the topology of probable *military communications networks* from signal externals. A demonstration of this capability addressed the Fulda Gap area of Germany in which spoke/hub relationships are extracted from among hundreds of emitters. The approach is to use a knowledge-based system for situation assessment—the knowledge is separate from the control and the chaining of applicable knowledge is driven by goals and data.

*Image Processing* research is developing a range of CM-based software for image processing, pattern recognition, and display. Specific items involved principal components data compression, shape recognition in two and three dimensions, frame buffer and Megatek display tools, registration of terrain elevation data to maps, flythrough DTED demonstrations with cast shadows, and so forth.

A diverse project used the CM to perform a variety of functions in support of a remote sensing system. Computationally intense background suppression was performed on the data. The data for this activity are derived from a mobile facility with a Perkin-Elmer sensor termed a *Fourier Transform Spectrometer*. One example application is the sensing of spectral data associated with the smoke from a Consolidated Edison client facility to determine whether the fuel used was coal or natural gas.

Launch Missiles is a program that demonstrates the CM's ability to run *strategic defense simulation* games. In this demonstration, one-thousand missiles are launched from sites in the Soviet Union. A few minutes into the flight, each missile MIRVs into eight warheads (making a total of 8,000 warheads). The warheads continue along their flight path until they reach their targets. Radar dishes are stationed along the flight path to monitor progress. A total simulation takes about 3 minutes real time while simulating approximately 30 minutes of flight time.

The simulation has two displays. The first shows a polar map of Earth and plots the course of each missile or warhead. The second display simulates six radar screens and plots each missile or warhead as it would be seen from this radar position. All displays are updated in real time. During the simulation it is possible to place a new radar dish at any point on the map. The radar will immediately begin displaying from its new position.

Each missile can have its launch position, launch time, MIRV time, flight time, and destination independently controlled. The problem of Gauss is solved separately for each missile.

The CM is responsible for three separate activities during the simulation. First, for each step in the simulation the CM calculates the new position and velocity of each missile or warhead. After calculating the new position, the CM calculates how this warhead would be seen in each of the radar displays (if it can be seen). And finally, the CM calculates which bits to turn on or off in each of the two displays. In each of these activities the CM does the work for all 8,000 warheads simultaneously using 8,000 processors.

*Camouflage, Cover, and Deception* (CC&D) techniques by military units include using trees for concealment, nets and tarpaulins over emplacements, background matching paints, decoys, and a variety of other hiding methods. Image analysts seek to exploit reconnaissance pictures for opposition deployment and order of battle information. To facilitate rapid image exploitation in the presence of CC&D activity, a phase 1 prototype soft-copy image analyst workstation was developed on the CM.

This workstation consists of a Symbolics 3675 host for the CM, a high-resolution color image display unit, a map projection unit, an ancillary computer running an expert system, and a print station that issues a formatted exploitation report. The Symbolics-CM-display system performs image manipulation under operator command. The expert system, which in phase 2 will be integrated onto the Symbolics, functions as an analyst associate to improve the productivity of analysts with low-to-moderate skill levels under time pressure. The expert system addresses tactical situations using a specified ER (exploitation request), doctrine, terrain, weather, and collateral reports to advise the operator regarding the most effective image manipulation algorithms to apply for enhancing the digital imagery.

A substantial program has aimed at finding better ways to exploit the CM's parallel advanced architecture. *FORTRAN to CM-language semiautomatic translation,* improved fast Fourier transforms, linear algebra, and other primitive math tools have been addressed. Interprocessor communication efficiency has been emphasized.

The CM code translator software converts FORTRAN or LISP programs into LISP-PARIS for parallel processing on the CM. The translator will also convert to LISP, C, and C-PARIS.

The translator is used mainly for converting large blocks of FORTRAN code into PARIS, a parallel-operation language, so that individual instructions can be run in parallel on the CM. (There are two CM's available to the user. A CM–1, front ended by two Symbolics 36xx, contains 16K processors. There is also a CM–2 16K processor machine front ended by both the VAX 8300 and the Symbolics 36xx.) Input is a file or a string containing working code written in the original programming language. Output, which has been translated to the selected language, is sent to either a file or the screen.

The translator resides on both the Symbolics 36xx and VAX 8300. A user needs some basic knowledge of the machine, editor, and operating system upon which the program is run as well as knowledge of the output language to integrate the output into operational code.

Text processing research examines the utility of the CM for rapid document search and retrieval. An innovative and powerful approach termed *Euclidean Concept Space* (ECS) was identified and developed. ECS defines mapping the documents of a text data base into points of an N dimensional space (N>20) based on content. Then the tools of analytic geometry are employed to explore relationships among the documents and their meaning.

Many applications not described here are being pursued as well. We are gratified to note that almost every application area we've examined of relevance to our client base responds well to massively parallel methods.

# A HIGH PERFORMANCE PARALLEL APPROACH TO MEDICAL IMAGING

**G. Frieder**

School of Computer and
Information Science
Syracuse University
Syracuse, NY 13224

**O. Frieder**

Bell Communications Research
445 South Street
Morristown, NJ 07960

**M. R. Stytz**

Air Force Institute of Technology
assigned to
University of Michigan
Ann Arbor, MI 48109

## ABSTRACT

Research into medical imaging using general purpose parallel processing architectures is described and a review of the performance of previous medical imaging machines is provided. Results demonstrating that general purpose parallel architectures can achieve performance comparable to other, specialized, medical imaging machine architectures is presented.

A new back-to-front hidden-surface removal algorithm is described. Results demonstrating the computational savings obtained by using the modified back-to-front hidden-surface removal algorithm are presented. Performance figures for forming a full-scale medical image on a mesh interconnected multiprocessor are presented.

Keywords: parallel processing, medical imaging, mesh, data parallel, image processing pipeline, hidden-surface removal.

## INTRODUCTION

Medical imaging is inherently computationally intensive. There are many computational aspects of medical imaging: the most prominent being the massive amount of data to be processed[1], preferably in real-time, the need for long range retention of data, and the need for data manipulation and display of the resulting three-dimensional (3D) images of complex anatomical structures, again preferably in real-time. These displays differ from the more familiar CAM/CAD and other graphics applications in that they do not have underlying geometrical structures which can be exploited to reduce the computational burden. Only a limited amount of abstraction from the raw data is permitted because disease diagnosis is based upon departures from the norm, and high level models would reduce, or eliminate, these differences. These stringent imaging and performance requirements combined with the associated heavy computational demands indicate that a massively parallel approach is worthy of consideration.

This research has two objectives. First, unify, capitalize and expand previous work in medical image processing. Second, demonstrate that a medical imaging machine operating in a general purpose massively parallel architecture can achieve the performance of specialized medical imaging machines operating on specialized architectures. Hence, in terms of cost and reliability, they are a viable approach to the interactive display of medical data. The research has been through two phases. The first phase, described in [Sty88], mapped the parallel primitives of the Voxel Processor machine described in [Gol87] onto mesh architectures. In the second phase, described here, we develop a new algorithm which reduces the computational overhead without sacrificing image quality and apply it to simulated mesh computers. The results of this investigation provide a foundation for cost effective, high quality three-dimensional images for disease diagnosis and treatment.

This paper is organized as follows. The next section provides a brief review of medical imaging terminology and a discussion of previous medical imaging machines. Section 3 presents a modified back-to-front algorithm which reduces the computational time required to generate an image. Section 4 describes the proposed medical image processing pipeline. Section 5 presents performance results obtained when the pipeline uses the modified back-to-front algorithm in the demonstration architecture. The final section contains our conclusions and outlines our future work.

## MEDICAL IMAGING: TERMINOLOGY AND PREVIOUS APPROACHES.

Medical imaging employs many of the image quality enhancement operations used in other graphics applications, and consequently has adopted many graphics terms as well. We define these terms below.

**Anti-aliasing** (see [Cro77] and [Cro81] for examples) is a graphics operation used to smooth the jagged edges which would otherwise appear in the final two-dimensional (2D) image when the scene is rotated.

The **cuberille model**, described in [Her79], describes the contents of a volume with identically sized, tightly packed, cube shaped small volumes called cuberilles. Each cuberille represents one or more properties of the associated volume within the irradiated space. A three dimensional set of cuberilles, therefore, represents the entire irradiated volume. Cuberilles have four characteristic properties: They are all of the same size, they are discrete, they lie in one of three mutually perpendicular directions, and they are very small relative to the object being imaged. The cuberille array is naturally represented within a computer by a three-dimensional (3D) array, in which each array element's value is the density value of the corresponding cuberille.

**Hidden-surface removal** ( see [Fol83], [Fri85], [Fuc79], and [Mea82]) is a graphics operation used to remove from the final 2D image those portions of the scene which the observer can not see. Hidden-surface removal can be accomplished using, for example, a z-buffer or back-to-front readout algorithm. A z-buffer algorithm makes use of two buffers. The refresh buffer, which is always required when forming a

[1] The average computerized tomography procedure generates one million voxels per single slice of a patient scan, magnetic resonance imaging and ultrasound procedures produce similar amounts of data. In 3D medical images, which are the subject of this paper, there can be 100 million data points generated per patient scan.

CH2649-2/89/0000/0282$01.00

digital image, is used for storing pixel intensity values. The other buffer is a z-buffer which is used to store the z-value of the each cuberille that is currently mapped to a screen pixel. The z-buffer is initialized to the largest representable z-value and the refresh buffer is initialized to the background value. Each cuberille is then scan converted. The conversion yields a depth z(x,y) at screen position (x,y). If the newly computed z(x,y) is less than the z(x,y) stored in the z-buffer, then the current cuberille is closer to the observer than the cuberille previously stored at (x,y), so the z value and intensity of the current cuberille replaces the z value and intensity stored in each buffer at position x,y.

Another class of hidden-surface removal techniques is back-to-front readout of the cuberilles in the scene. Front is defined as the point(s) in the scene closest to the observer. Back is defined as the point(s) furthest from the observer. The algorithm is simpler to implement than the z-buffer algorithm and requires less space since there is no z-buffer to maintain. The operation of the algorithm requires that the entire data set be accessed in back-to-front order relative to the observer. This is accomplished by correctly oriented the scene relative to the three coordinate system axis. Once the correct viewer-object orientation is achieved, the cuberilles in the scene are read-out in back-to-front order and mapped onto the image display.

**Pixel** is an abbreviation for picture element. A single pixel is the smallest element of a picture that can be displayed, it is essentially a single point on the display screen.

**Shading** (see, for example, [Che85], [Fuc83], [Hef85], or [Pho75]) is a graphics process whereby the appearance of a visible surface is altered to account for the number and types of light sources illuminating the surface, the surface texture, surface color, surface reflectance, and the position and orientation of surrounding surfaces.

Five medical imaging machines were examined with a view toward defining their capabilities. Figure 1 contains the performance figures for the five machines. In brief, their performance can be characterized as follows. The Image Overlay Machine, [Far85], is a high resolution, high-speed machine implemented on a general-purpose mainframe and workstation. The graphics processing algorithms are encoded in software, and the common 3D image formation operations are avoided, with a 3D effect being provided by color and image rotation. The Pixel-Planes machine, [Fuc85], is a medium resolution, medium speed machine which is implemented as a special purpose processor. The Pixel-Planes graphics processing algorithms are hard-coded into the hardware of the machine, with an apparent 3D view formed using shading, shadows, and hidden-surface removal techniques. The Voxel Processor, ([Gol87]), is a medium-resolution, high-speed machine which is implemented on a special-purpose architecture. The graphics processing algorithms are encoded in hardware, with the apparent 3D image formed using shading and hidden-surface removal techniques. The True 3D machine, ([Rob85] and [Rob86]), is a low-resolution, high-speed machine which is implemented on a general-purpose architecture using a special purpose graphics display device. The graphics processing algorithms are placed in software, with the 3D image formed using a varifocal mirror and a series of frame buffers to provide a true 3D image (this is the only machine of the five which does form a real 3D image). The Medical Image Processing Group (MIPG) machine, [Her86], is a low to medium resolution, slow machine which is implemented on a general-purpose architecture. The graphics processing operations are placed in software, with shading, motion, and hidden-surface removal algorithms employed to create a 3D effect in the displayed scene. A thorough survey of medical imaging modalities and medical imaging machine architectures can be found in [Fri88].

MEDICAL IMAGING MACHINES

| | Farrell's Image Overlay Machine | Fuch's Pixel Planes Machine | Voxel Processor Machine | Robb's True 3D Machine | MIPG Machine |
|---|---|---|---|---|---|
| ARCHITECTURE | Mainframe and workstation | Smart frame buffer | Parallel/ pipeline | Mainframe and workstation | CT scanner microprocessor |
| PROCESSING STRATEGY | Use masking and coloring to simplify processing | Use linear equations to describe scene | Parallel operations on 64 disjoint subcubes of scene | Mainframe performs major computations, workstation takes on minor computational load | Indentify organ early in processing |
| OBJECT MODEL USED | Cuberille/voxel | Convex polygons | Cuberille/voxel | Voxel | Cuberille/voxel |
| SHADING | Back-to-front increase in lighting | Based on light source and surface normal | Gradient shading | None | Distance only, contextual, gradient, or Phong shading |
| ANTIALIASING | 3 x 3 local average | 4 x 4 local average | Supersampling to double resolution | None | Supersampling to double resolution |
| HIDDEN-SURFACE REMOVAL | Overwrite unnecessary objects in scene | Z-buffer | Back-to-Front | Z-Buffer | Modified Z-buffer |
| RESOLUTION (in pixels) | 1024 x 1024 | 512 x 512 | 512 x 512 | 128 x 128 | 256 x 256 |
| PERFORMANCE | Exceeds real-time speed requirements | 30k polygons/ sec, 10-20 sec for a medical image | 25 frames/sec formed and output | 25 preprocessed frames/sec. output | Slow |
| PARALLELISM | Minimal | SIMD | MIMD | None | None |

**Figure 1**: Medical Imaging Machines

Even in this brief survey it is evident that there is no one architecture or approach which dominates the medical imaging field. In broad terms, these machines can be characterized as follows. The serial machines sacrifice image quality or speed in order to accomplish their image processing tasks. On the other hand, the parallel processing machines are special purpose architectures and therefore relatively costly for their performance, especially in a medical imaging environment. Additionally, the special purpose parallel processing machines sacrifice the processing flexibility provided by software encoding of algorithms for the speed advantage that comes from hardware encoding in order to achieve their real-time performance. Note that each machine has a suite of capabilities which distinguishes it from the other four. These capabilities, such as software encoding of algorithms, use of general purpose architectures, apparent 3D, high speed, and high resolution, are not mutually exclusive. These capabilities should be able to be combined within one medical imaging machine to provide a richer, more flexible imaging environment for the clinician. Our research has concentrated on melding the advantages possessed by each of these machines into one unit which is fast and has high resolution while remaining cost effective.

We began by implementing the parallel operational primitives of the Voxel Processor within a mesh architecture. This choice was motivated by three factors. First, the Voxel Processor operational primitives are geared toward parallel processing, albeit on a special purpose architecture. Second, the parallel primitives employed in the Voxel Processor are scalable. Third, the stages operate independently except for message passing to move the data through the machine. This aspect provides flexibility when tackling the performance bottlenecks in each stage because modifications to a stage are transparent to all other stages in the pipeline. Thus, we were able to quickly implement the machine with confidence in its ability to scale to full-size medical images and in its ability to accommodate improvements to the operation of each stage. We employ the cuberille data model so as to reduce the packing/unpacking overhead in the processing of the images (note that this does not preclude any compression of data for retention purposes). Anti-aliasing is accomplished using super sampling to high resolution with averaging to a low resolutionfinal image as described in [Cro81]. Pixel intensity values are assigned using an 8 bit gray scale. Our evaluation is done on various mesh architectures that are simulated using C under HP-UX 5.3 on a Hewlett-Packard 9000 Series 300 with a CPU running at 16Mhz. Message passing is used to simulate the node interconnections of the architectures. While the environment is simulated, the algorithms are actually executed under the assumption that each node is actually a HP machine, thus actual displays are produced. The elapsed time required for each stage to perform its processing is obtained using the timing calls provided in HP-UX. Total elapsed time for formation of a single image is taken to be the sum of each stage's processing time plus the communication time between stages.

The image processing machine consists of a multistage pipeline which operates on a three-dimensional scene formed by interpolation of computerized tomography, magnetic resonance imaging, or ultrasound image data. Figure 2 portrays the image processing pipeline used to obtain the baseline and modified back-to-front implementation results.

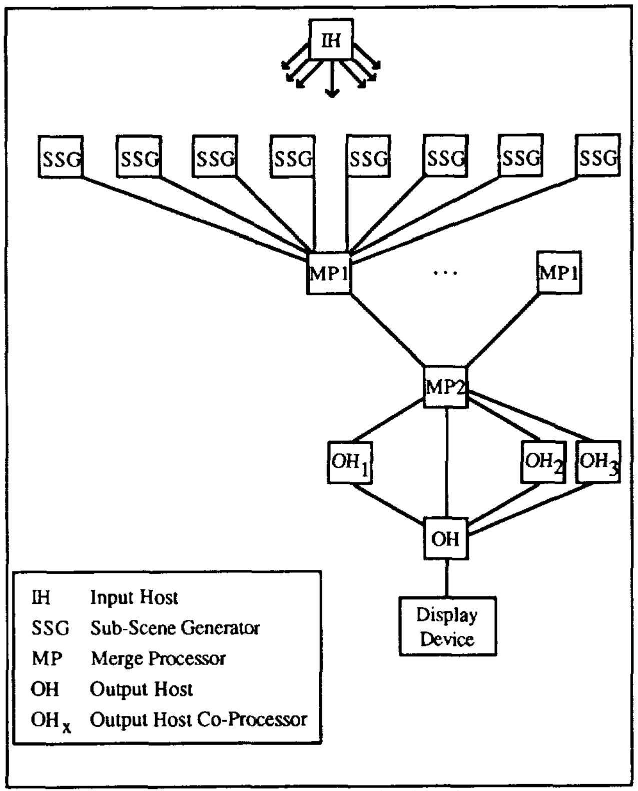


Figure 2: Image Processing Pipeline

Each stage of the pipeline operates as follows. The Input Host gathers user scene editing inputs and broadcasts them to all the processors in the machine as well as performing the initial data distribution of the floating point cuberille values to the Sub-Scene Generator (SSG) processors. The data which forms the 3D scene is equally distributed among the Sub-Scene Generators with each SSG receiving a continuous volume for it to operate upon. Figure 3 demonstrates how the image volume is divided into eight octants, and the numbering scheme used to identify each octant. Each octant is recursively divided by repeatedly applying the Figure 3 scheme until the data volume has been divided into as many suboctants as there are SSGs, at which time each volume is sent to the corresponding SSG. Each Sub-Scene Generator performs anti-aliasing, scene rotation and object space to image space mapping on its subset of the image volume based on the user inputs broadcast by the Input Host. When a SSG finishes processing, it sends its output 2D scene, consisting of a floating point density value and an integer z value for each coordinate, to its Merge Processor in the next stage. Each Merge Processor (MP) gathers the output of the eight SSG's or MP's which send it input and performs a back-to-front merge of the eight input scenes to form its larger output scene. The final pipeline stage, consisting of one MP,merges the eight scenes from the previous stage of MP's into the final, full-scale scene. The last stage MP output is divided among the Output Host and three coprocessors (Output Host$_1$, Output Host$_2$, and Output Host$_3$) for shading to provide a three dimensional effect before final display by the Output Host. Shading is performed in software, even though image shading hardware exists, to demonstrate that the pipeline operates correctly within the realm of the simulation environment. The results obtained for the final stage are not presented as they are based on a serial mode of operation for the display device, and this mode masks the performance speedup obtained from parallel processing. The questions of shading and image display will be addressed separately as they have a great impact on the performance of the pipeline.

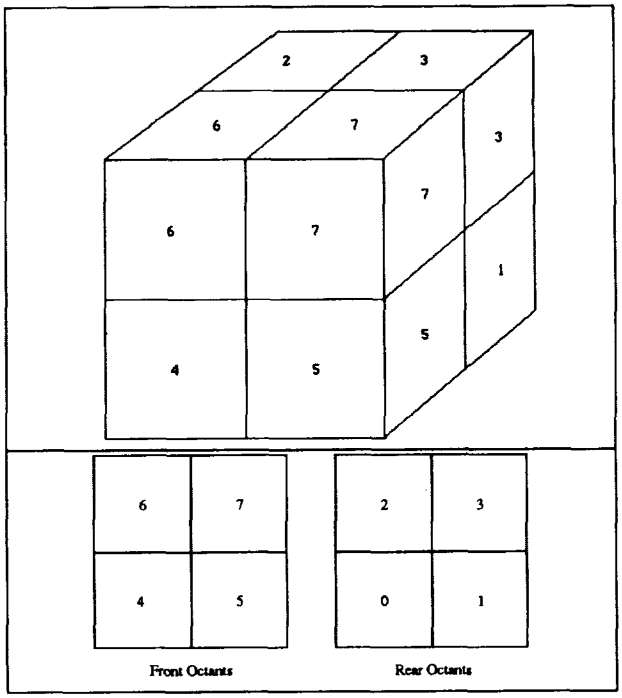


Figure 3: Octant Numbering Scheme

Timing results were obtained using a message packet size of 64k bytes and a 16 x 16 x 16 scene at each SSG. As our goal was identifying the bottlenecks in the pipeline, production times for each stage were gathered as an integral part of each simulation run. Refer to Figure 4, where the elapsed time spent in each stage of the baseline pipeline is presented. Note that the SSG stage, where back-to-front readout of the suboctants is performed, is the main bottleneck in the image processing pipeline. This bottleneck motivated our development of a new back-to-front hidden-surface removal algorithm.

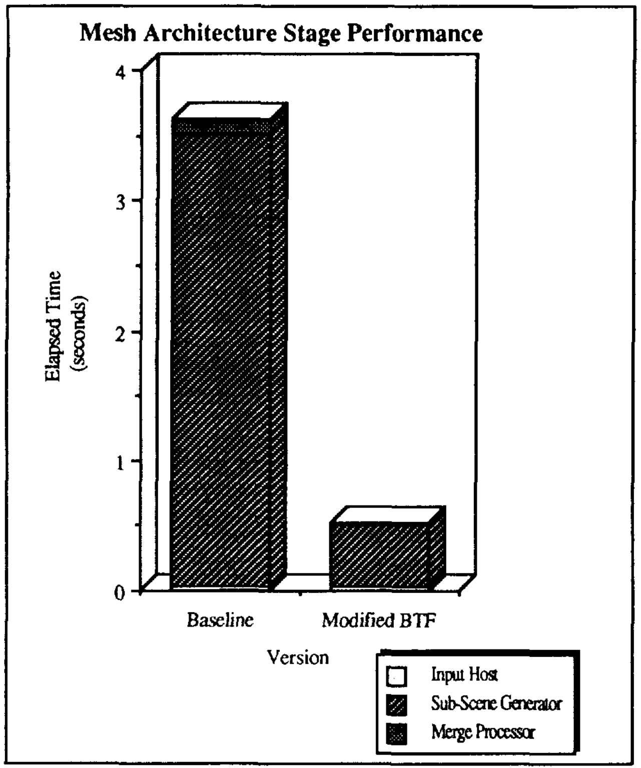


Figure 4: Mesh Stage Performance Results

## THE MODIFIED BACK-TO-FRONT ALGORITHM

In the second phase of the research, we developed an improved back-to-front algorithm which reduces the amount of computation required in the Sub-Scene Generator (SSG) stage. The algorithm produces a two-dimensional slice of the scene which is useful for scene editing purposes (see [Rob86]). Once the user has achieved the desired orientation of the scene, we then switch to the full three-dimensional (3D) scene processing scheme. *This approach permits rapid editing of the imaged* volume as well as full 3D viewing of the volume of interest by the user. Our algorithm is based on two observations. First, if graphic output is placed in the output buffer in back-to-front order, then the final image contains only those portions of the scene observable from the viewer's position. Second, any portion of the scene which is closer to the back of the scene and further from the scene's cutting plane than is the geometric center of the scene can not be observed from the user's position. We capitalized on these two observations and developed a new hidden-surface removal algorithm which reduces the computation burden at each SSG by roughly a factor of 4 as compared to the standard back-to-front algorithm. A brief description of the algorithm follows, it is not complete as it does not allow for rotation of the scene but is presented to give a feel for the algorithm's operation.

Assume an imaged volume of size N x N x N, where N is a power of two, and a cutting plane through the volume which is perpendicular to the y-z plane. The origin lies at the front, lower, left of the scene. First, divide the imaged volume into eight equal sized octants of size N/2 x N/2 x N/2. Division in this manner allows us to perform the remainder of the scene subdivision process with shifts rather than divides. The octants thus formed are labeled as shown in Figure 3. We then place the observer at the center front of scene, coordinates N/2, N/2, 0. A back of scene reference point is placed at coordinates N/2, N/2, $\frac{N\sqrt{2}}{2}$

Next, compute each of the eight octant's center coordinate as follows:

| | |
|---|---|
| For octants 0, 2, 4, 6: | x coordinate = N/4 |
| For octants 1, 3, 5, 7: | x coordinate = 3 * N/4 |
| For octants 0, 1, 4, 5: | y coordinate = N/4 |
| For octants 2, 3, 5, 7: | y coordinate = 3 * N/4 |
| For octants 0, 1, 2, 3: | z coordinate = 3 * N/4 |
| For octants 4, 5, 6, 7: | z coordinate = N/4. |

Let *i* represent octant i of the scene, *ij* represent the octant j which lies within octant i, and so on. Let C represent the center of the scene, $C_i$ represent the center of octant i, and $C_{ij}$ *represent the center of octant j, which lies within octant i,* and so on. $D_{rp}(x)$ returns the distance from a point x to the back-of-scene reference point. $D_p(x)$ returns the distance from the point x to the scene's cutting plane. Perform the following steps in back-to-front order on each of the eight octants . First, determine if the octant contains data which is visible from the observer's position. The octant is obscured if $D_{rp}(C_i) < D_{rp}(C)$ and $D_p(C_i) > D_p(C)$. If these conditions are met, ignore the remainder of octant *i*. If either test is not met, process the remainder of the octant. Octant *i* now becomes a parent octant of dimension M x M x M, where M=N/2. Now, treat parent octant *i* as though it were the entire scene and determine in back-to-front order which of its eight octants are visible as follows. First, if *i* encloses 2 x 2 x 2 primitive data elements write *i* to the output buffer in back-to-front order. Otherwise, divide *i* into eight new octants *i0... i7* using the method employed above to divide the entire scene. Determine which of the new octants *ij* are visible to the observer. The visibility determination is made by following the steps originally performed to determine which scene octants are visible. First, the new suboctant centers, $C_{ij}$, are computed as shown above, and then the distance from the new octant centers to the cutting plane and the back-of-scene reference point are determined. *If* $D_{rp}(C_{ij}) < D_{rp}(C_i)$ and $D_p(C_{ij}) > D_p(C_i)$ then do not process the remainder of *ij*. If either test fails, consider *ij* to be a parent octant and perform the octant division and distance determination steps for each of the new octants *ijk*. Continue processing smaller and smaller portions of each scene octant *i* until each element has been either cast away or written to the output buffer. When processing concludes, the two-dimensional slice of the volume defined by the cutting plane will be in the output buffer.

The modified back-to-front algorithm rapidly determines the front of the scene as defined by the cutting plane by recursively examining the scene to determine the set of cuberilles the cutting plane intersects. This determination is made by applying two tests to each octant. The first test determines if there is another octant between the current octant and the observer, this test is accomplished by computing the distance from the current octant to the *back-of-scene reference* point and comparing this to the distance of the parent octant to the back-of-scene reference point. The second test determines if the cutting plane passes through the current octant. This test is accomplished by comparing the distance from the octant center to the cutting plane with the distance from the octant center to the furthest point in the octant. These tests allow the SSG to cast away substantial portions of its subscene early in the hidden-surface removal process, thereby reducing the computation performed at the SSG. This is in contrast to the other hidden-surface removal algorithms described in Section 2 which must examine each element in the SSG's subscene to construct the required view of the image.

Refer back to Figure 4, where the baseline image processing pipeline performance using the modified back-to-front algorithm is presented. The results were gathered on a simulated 16 node mesh connected multiprocessor using 64k byte message packets with a 16 x 16 x 16 scene at each SSG. The reduction in time spent in the SSG stage when the modified back-to-front algorithm is used instead of the standard back-to-front algorithm is demonstrated by the results presented in the figure. The SSG stage was then timed independently to gather performance data for various SSG scene sizes, ranging from 8 x 8 x 8 to 64 x 64 x 64 , at a constant packet size of 64k for both the standard and modified back-to-front algorithms. These results are presented in Figure 5. Note that the amount of time spent at the SSG stage decreases linearly with decrease in scene size at the SSG, or, equivalently, with the increase in the number of nodes used in the SSG stage. These promising results led us to develop the front end to an image processing pipeline, based on the baseline image processing pipeline described above, which is capable of matching the performance exhibited in special purpose medical imaging machines.

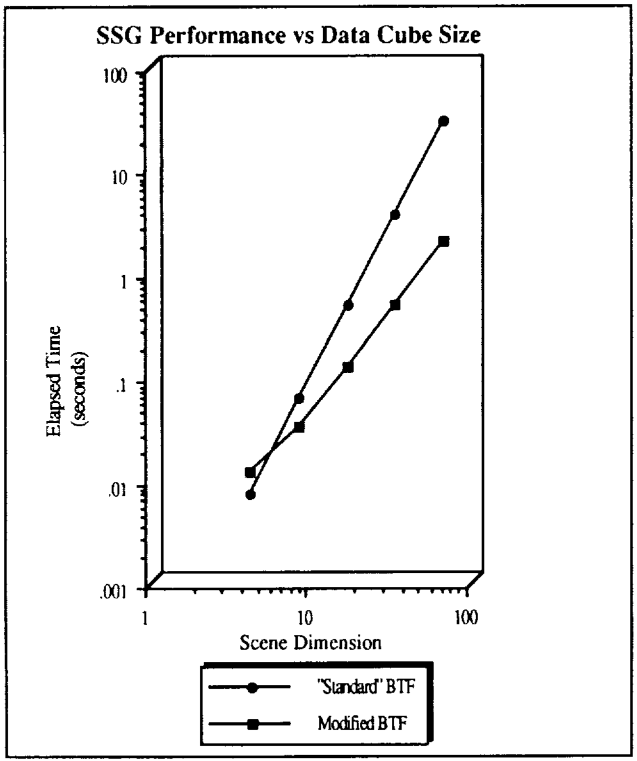


**Figure 5:** Sub-Scene Generator Performance

## FULL SCALE MEDICAL IMAGING PIPELINE

To demonstrate the performance of the full-scale medical imaging pipeline when operating within a general purpose parallel processing machine, the simulation results were extended to 512 x 512 x 128 cuberille medical images (33.5 million data elements), which is a reasonably sized volume of CT scanner data. This extrapolation is valid because our data allows us to employ a data parallel approach, as defined in [Tuc88], for programming the machine. Recall that the PEs in each stage operate in concert, with no interference from other stages. As a result, the timing results for a given stage remain valid as the overall image size is increased because the amount of work performed by a PE in a given stage remains constant.

The graphics pipeline we use to obtain these performance figures is modified from the baseline pipeline described earlier. This new configuration is motivated by the observation that image shading is now the performance bottleneck. Because the time required to shade an image decreases linearly with decreasing image size, the performance of the shading stage can be improved by dividing the shading operation among many processors which each shade a very small portion of the overall scene. To move the shading operation forward in the pipeline, the processors in the shading stage must exchange the information needed to shade their portions of the scene. The first two stages remain as depicted in Figure 2. For ease of exposition, the scene dimension at each SSG is the dimension without supersampling, and the time to perform anti-aliasing operations is not included within the SSG figures we use below. The third stage performs Sub-Scene Generator output shading using the gradient shading algorithm described in [Sty88]. The simulation environment only permits software shading, at a cost of one shading node per SSG. An actual implementation could replace the software shading nodes with a hardware shading capability and thereby reduce the multiprocessor node count and the time required to process an image. The fourth stage performs image translation and merging. A naive implementation of the remainder of the pipeline would have subsequent Merge Processor (MP) stages forming successively larger 2D scenes, with the last stage of MPs sending their output to the Output Host for display. Adopting this configuration for the pipeline results in elapsed times in later stages of the pipeline which nullify the benefits realized by multiprocessing. To better highlight the performance available, we omit the scene merging times from this discussion and concentrate instead on the operation of the first four stages (the front end) of the pipeline.

## RESULTS

The timing results for the first four stages of the full-scale image processing pipeline described in the previous section are based on a message packet size of 64k bytes. The mesh processors are assumed to run at the speed of the simulation CPU, which is 16Mhz. The scene size is 512 x 512 x 128 cuberilles and the observer is placed at the front, center of the scene. Each of the four stages of the pipeline were timed independently in the simulation computer, and the individual stage results combined to determine the performance achieved by the first four stages of the full-scale pipeline. The results are conservative in their assessment of each stages' true performance because of hardware and software limitations inherent in the simulator used for the research. Hardware and software improvements can be made in the actual machine which would yield substantial performance improvements. For example, multiprocessor computers with nodes running significantly faster than the 16Mhz of our simulator are available. Since the machine's performance is presently constrained by the amount of computation performed at each node, rather than communication between stages, an increase in the CPU speed at each node translates to decreased elapsed time at each stage resulting in improved machine performance. In addition, the code for each stage was not optimized in that function calls and recursion are freely used, especially in the SSG stage. Unwrapping the recursion and eliminating the function calls will yield further significant performance improvements in each stage. Replacing the software shading in the third stage with a hardware shading capability at the end of the pipeline would yield a significant performance improvement, thereby removing the shading bottleneck from the pipeline.

There are two components to the performance of the pipeline we have proposed: The image production rate and the elapsed time through the pipeline. The image production rate, $T_r$, is the rate at which images emerge from the pipeline. In general, $T_r$ is equal to the processing speed of the slowest stage. In the full-scale medical imaging pipeline, $T_r$ is equal to the amount of time spent in the shading stage when the modified back-to-front algorithm is used, and is equal to the elapsed time in the SSG stage when the standard back-to-front algorithm is used. The elapsed time through the pipeline, $T_e$, is defined as the amount of time which must elapse before the image corresponding to a user input is displayed, ie., the time required for an input to be converted into an image. Both components are used to describe the full-scale pipeline's performance. The results below describe the pipeline's performance when producing a continuous stream of images using the modified back-to-front algorithm.

Placing a 64 x 64 x 64 scene at each SSG gives a $T_r$ for the four stages of 1.97 seconds and a $T_e$ of 4.15 seconds employing 128 SSG nodes on a mesh connected multicomputer using software shading. This is not acceptable performance in a clinical environment, therefore a higher degree of parallelism is required to achieve our goal of real-time performance. By decreasing the subscene at each SSG from 64 x 64 x 64 to 32 x 32 x 32 cuberilles, we incease the number of SSG nodes required to 1k. However, $T_r$ decreases to .49 seconds and $T_e$ decreases to 1.03 seconds. By removing the recursion and using state of the art technology processors at each node this machine approaches real-time speeds using software shading.

Multicomputers capable of supporting 128 SSG nodes are well within the scope of today's technology, but the elapsed time is well beyond the real-time range. The 32 x 32 x 32 SSG scene size offers the most promise at this time. The numbers of SSG nodes required is small enough, 1k, that computers with this number of nodes are available. The performance, .49 seconds to form an image, combines high image quality with low image formation times.

## CONCLUSIONS

Medical imaging on mesh connected architectures is practical, especially when the modified back-to-front algorithm is used to accomplish the initial scene editing operations. The front end of the image processing pipeline described above offers a unique combination of high image quality, high speed, and scalability. Its performance along these three dimensions using state of the art technology compares favorably with the special purpose parallel architecture medical imaging machines described in Figure 1. The machine is suitable for medical imaging applications within a general purpose parallel processing architecture.

We have several projects underway which will bring about the realization of a full-scale medical image processing pipeline. Of primary importance is developing methods for subscene merging and scene display which do not cancel the benefits which accrue from parallel processing. One bottleneck remaining in the pipeline is the time required to move the image from the end of the pipeline to the display. This bottleneck arises from the fact that in the simulator pixels are written to the display buffer serially by a single processor. This bottleneck is not caused by the algorithms employed or the pipeline architecture, but is solely due to the serial display hardware. To eliminate this performance constraint, an investigation of display methodologies which scale along with the multiprocessor size is underway. A second bottleneck is the time required to merge the subscenes into the full scene. This processing takes place in the later portions of the pipeline, and like the image display process itself, largely cancels the benefits which come from parallel processing. We are investigating alternative pipeline architectures and algorithms which eliminate merge processing. Third, we are investigating methods for reducing the elapsed time required for the 3D image formation process. We expect that this can be done by changing the image/Sub-Scene Generator data assignment algorithm and by employing an image processing pipeline that overlaps the operations of the stages. The data assignment is made such that each SSG has data from each scene octant, and the octants are processed sequentially in back-to-front order by the pipeline. Preliminary results for this new data assignment scheme are promising and will be presented in a future paper when they are more conclusive. Finally, we are attempting to quantify the performance improvement achieved when hardware shading replaces software shading in the pipeline. We plan on implementing the pipeline on an actual, scalable multicomputer to further validate our conclusions with actual performance measurements.

## REFERENCES

[Che85] Chen, Lih-Shyang; Herman, Gabor T.; Reynolds, R.Anthony; and Udupa, Jayaram K. "Surface Shading in the Cuberille Environment," *IEEE Computer Graphics and Applications*, vol. 5, no. 12, pp. 33-43, December 1985.

[Cro77] Crow, Franklin C. "The Aliasing Problem in Computer-Generated Shaded Images," *Communications of the ACM*, vol. 20, no. 11, pp. 799-805, November 1977.

[Cro81] Crow, Franklin C. "A Comparison of Antialiasing Techniques," *IEEE Computer Graphics and Applications*, vol. 1, no. 1, pp. 40-48, January 1981.

[Far85] Farrell, Edward J.; Yang, Wen C.; and Zappulla, Rosario. "Animated 3D CT Imaging," *IEEE Computer Graphics and Applications*, vol. 5, no. 12, pp. 26-30, December 1985.

[Fol83] Foley, James D., and Van Dam, Andries. *Fundamentals of Interactive Computer Graphics*. Reading, Massachusets: Addison-Wesley Publishing Company, 1983.

[Fri85] Frieder, Gideon; Herman, Gabor T.; Meyer, Craig; and Udupa, Jayaram. "Large Software Problems for Small Computers: An Example from Medical Imaging," *IEEE Software*, vol. 2, no. 5, pp. 37-47, September 1985.

[Fri88] Frieder, G.; Frieder, O; and Stytz, M.R. "Medical Imaging: Computers Assisting Medicine," *submitted for publication*.

[Fuc79] Fuchs, Henry; Kedem, Zvi M.; and Naylor, Bruce. "Predetermining Visibility Priority in 3-D Scenes(PreliminaryReport)," *Computer Graphics*, vol. 13, no. 2, pp. 175-181, August 1979.

[Fuc83] Fuchs, Henry; Abram, Gregory D.; and Grant, Eric D. "Near Real-Time Shaded Display of Rigid Objects," *Computer Graphics*, vol. 17, no. 3, pp. 65-69, July 1983.

[Fuc85] Fuchs, Henry; Goldfeather, Jack; Hultquist, Jeff P.; Spach, Susan; Austin, John D.; Brooks, Frederick P. Jr.; Eyles, John G.; and Poulton, John. "Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes," *SIGGRAPH '85*, vol. 19, no. 3, pp. 111-120, July 1985.

[Gol87] Goldwasser, Samuel M., and Reynolds, R. Anthony. "Real-Time Display and Manipulation of 3-D Medical Objects: The Voxel Processor Architecture," *Computer Vision, Graphics and Image Processing*, vol. 39, pp. 1-27, 1987.

[Hef85] Heffernan, Patrick B. and Robb, Richard A.

"Display and Analysis of 4-D Medical Images," *Proceedings of the International Symposium: CAR '85, Computer Assisted Radiology*, Berlin, West Germany, pp. 583-592, 1985.

[Her79] Herman, Gabor T., and Liu, Hsun Kao. "Three-Dimensional Display of Human Organs from Computed Tomograms," *Computer Graphics and Image Processing*, vol. 9, pp. 1-21, 1979.

[Her86] Herman, Gabor T. "Computerized Reconstruction and 3-D Imaging in Medicine," *Technical Report MIPG108*, Dept. of Radiology, University of Pennsylvania, February 1986.

[Mea82] Meagher, Donald. "Geometric Modeling Using Octree Encoding," *Computer Graphics and Image Processing*, vol. 19, pp. 129-147, 1982.

[Pho75] Phong, Bui Tuong. "Illumination for Computer Generated Graphics," *Communications of the ACM*, vol. 18, no. 6, pp. 311-317, June 1975.

[Rob85] Robb, Richard A. *Three-Dimensional Biomedical Imaging*. 2 vols. Boca Raton, Florida: CRC Press, 1985.

[Rob86] Robb, R.A.; Heffernan, P.B.; Camp, J.J.; and Hanson, D.P. "A Workstation for Interactive Display and Quantitative Analysis of 3D and 4D Biomedical Images," *Proceedings of the Tenth Annual Symposium on Computer Applications in Medical Care*, Washington, D.C., pp. 240-256, October, 1986.

[Sam88] Samet, Hanan and Webber, Robert E. "Hierarchical Data Structures and Algorithms for Computer Graphics," *IEEE Computer Graphics and Applications*, vol.8, no. 3, pp. 48-68, May 1988.

[Sty88] Stytz, Martin R.; Frieder, G.; and Frieder, O. "On the Exploitation of a Commercially Available Parallel Processing Architecture for Medical Imaging," *Proceedings of the Symposium on the Engineering of Computer-Based Medical Systems*, Minneapolis, Minnesota, pp. 49-59, June, 1988.

[Tuc88] Tucker, Lewis W. and Robertson, George G. "Architecture and Applications of the Connection Machine," *IEEE Computer*, pp. 26-38, August 1988.

# PARALLEL ALGORITHM FOR THE SOLUTION OF NONLINEAR POISSON EQUATION OF SEMICONDUCTOR DEVICE THEORY AND ITS IMPLEMENTATION ON THE MPP

J. P. Darling
The Johns Hopkins University Applied Physics Laboratory
Laurel, MD 20707

I. D. Mayergoyz
Electrical Engineering Department and Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742

## ABSTRACT

The solution of the nonlinear Poisson equation of semiconductor device theory is important for the design of sub-micron devices used in VLSI circuits. A new algorithm for the numerical solution of this equation has been described in Ref. 1. This algorithm is inherently parallel and is thus well suited to implementation on a computer with large parallelism. This paper describes the initial implementation and testing of this algorithm on NASA's Massively Parallel Processor (MPP).

Keywords: Modeling of semiconductor devices, Nonlinear Poisson equation, Parallel algorithm, Global convergence, MPP implementation, Parallel Pascal.

## INTRODUCTION

Numerical modeling of semiconductor devices is becoming increasingly important in the design of sub-micron devices used in VLSI circuits, as well as in the characterization of materials and processes used in the manufacture of these circuits. A complete model of the device requires the solution of three coupled equations: the Poisson equation for electrical potential combined with the continuity equations for electron and hole currents. Because electron and hole densities are nonlinear functions of potential, the first equation is known as the nonlinear Poisson equation.

Solution of the nonlinear Poisson equation is directly applicable to determining the thermal equilibium value of the potential within the device. Furthermore, problems involving reversed biased p-n junctions can be reduced to the solution of the nonlinear Poisson equation using standard low-level injection approximations.

The conventional approach employed for the solution of the nonlinear Poisson equation is based on the Newton-Raphson method applied to simultaneous discretized equations. This approach has two major difficulties. First, the Newton-Raphson method possesses only local convergence, and thus global convergence using this technique is not assured for an arbitrary initial guess. Second, the amount of computation and storage required to solve the simultaneous equations is large, especially for high-density meshes, requiring the use of high-performance computers with a large main memory capacity.

A new, inherently parallel algorithm for solving the nonlinear Poisson equation has been described in Ref. 1. This algorithm has none of the disadvantages of the existing method. First, it has guaranteed global convergence for an arbitrary initial guess. Second, explicit algebraic formulas are used to update the value of electric potential at each mesh point, minimizing the amount of storage required per mesh point. Third, the calculations used to update the value of electrical potential at each mesh point can be performed concurrently for all mesh points, making the algorithm especially attractive for implementation on a parallel computer.

The MPP was selected as a good candidate for implementing this algorithm because of the large parallelism it possesses—16,384 Processing Elements (PE). The Single Instruction Multiple Data (SIMD) architecture of the MPP matched well with the algorithm characteristics of updating the potential at each mesh point, using the same steps, in parallel. Also, the two-dimensional interconnection structure of the MPP's Processing Elements is ideally suited to the algorithm's nearest neighbor communications requirements.

Subsequent sections of this paper are organized as follows. The next section develops the discretized equations for implementing the two-dimensional version of the algorithm. Using these equations, the major computational steps of the algorithm are described. The following section provides a brief description of the parallel architecture of the MPP, highlighting the features exploited by our numerical implementation. The fourth section describes the specifics of the algorithm implementation using the Parallel Pascal language of the MPP. Refinements made to the algorithm implementation to improve run-time efficiency are also discussed. The last section summarizes the algorithm implementation, and outlines future work.

## NONLINEAR EQUATION

Poisson's equation (1) describes the variation of electric potential within the MOSFET device shown in Figure 1.

$$\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} = \frac{q}{\epsilon_s}(n - p - D) \ . \qquad (1)$$

Here, $n$ and $p$ represent the mobile electron and hole concentrations, respectively, $D$ is the concentration of ionized impurities, $q$ is the electron charge, and $\epsilon_s$ is the permittivity of silicon.

At thermal equilibrium in a nondegenerate semiconductor the concentrations of mobile electrons, $n$, and holes, $p$, are approximated by Boltzman's statistics. By using these statistics in equation (1), we end up with the nonlinear Poisson equation (2),

$$\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} - \frac{qn_i}{\epsilon_s}\left\{\exp\left(\frac{q\phi}{kT}\right) - \exp\left(\frac{-q\phi}{kT}\right)\right\} = \frac{-qD}{\epsilon_s} \ , \qquad (2)$$

where $n_i$ represents the intrinsic mobile carrier concentration, $k$ is Boltzman's constant, and $T$ is absolute temperature. This equation applies for any position within the boundaries of the device in Figure 1.

There are three classes of boundary conditions that apply to the MOSFET geometry of Figure 1. The boundary conditions for the

CH2649-2/89/0000/0289$01.00 © 1988 IEEE

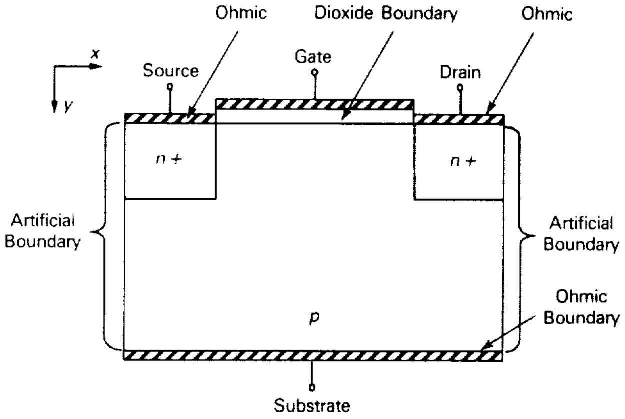


Figure 1. MOSFET Device Geometry With Boundary Conditions.

potential at the ohmic contacts (source, drain, and substrate) can be determined by assuming charge neutrality. The artificial boundaries in Figure 1 are intended to isolate the MOSFET device. This isolation is well approximated with a zero Neumann boundary condition. The dioxide-semiconductor interface, under the gate electrode of Figure 1, is well approximated by an impedance-type boundary condition as described in Ref. 1.

Equations (3), (4), and (5) describe the boundary conditions for the ohmic boundaries, the artificial (Neumann) boundaries, and the impedance-type boundary, respectively.

$$\phi = \frac{kT}{q}\, ln \left( \frac{\frac{D}{2} + \sqrt{\frac{D^2}{4} + n_i^2}}{n_i} \right) . \tag{3}$$

$$\frac{\partial \phi}{\partial \nu} = 0 . \tag{4}$$

$$\phi + \frac{\delta \epsilon_s}{\epsilon_{ox}} \frac{\partial \phi}{\partial \nu} = V_g + \frac{\delta}{\epsilon_{ox}} Q . \tag{5}$$

Here, $\delta$ represents the thickness of the dioxide, $V_g$ is the voltage applied to the gate, $Q$ represents the charge trapped at the dioxide interface, $\epsilon_s$ is the permittivity of silicon dioxide, and $\nu$ is an outward normal.

For a numerical solution the problem is discretized by choosing two sets of mesh lines, one set parallel to each of the $x$ and $y$ axes of Figure 1. The mesh lines are specified by:

$$x_i \; ; \quad i = 0, 1, 2, \ldots i_{\max} - 1, i_{\max}$$

$$y_j \; ; \quad j = 0, 1, 2, \ldots j_{\max} - 1, j_{\max} .$$

with both $x_o$ and $y_o$ specifying the top left corner of the MOSFET device of Figure 1. Furthermore, the mesh line spacing must form a monotonically increasing sequence, such that $x_i < x_{i+1}$ (and $y_j < y_{j+1}$) for all $i(j)$. The intersection of these mesh lines forms a set of mesh points at which the physical quantities of interest will be calculated. Each mesh point can be uniquely identified by the pair $(i,j)$, which will be used in subsequent equations to denote the values of physical quantities at specific mesh points (these indices were chosen to conform with the MPP's convention for identifying PEs).

The discretized version of the two-dimensional nonlinear Poisson equation, which describes the electric potential at mesh point $(i,j)$, is given in Equation (6),

$$\frac{2}{g_i + g_{i-1}} \left( \frac{(\phi_{i+1,j} - \phi_{i,j})}{g_i} - \frac{(\phi_{i-1,j} - \phi_{i,j})}{g_{i-1}} \right) + \frac{2}{h_j + h_{j-1}} \left( \frac{(\phi_{i,j+1} - \phi_{i,j})}{h_j} - \frac{(\phi_{i,j-1} - \phi_{i,j})}{h_{j-1}} \right) - \frac{qn_i}{\epsilon_s} \left\{ \exp\left( \frac{q\phi_{i,j}}{kT} \right) - \exp\left( \frac{-q\phi_{i,j}}{kT} \right) \right\} = \frac{-qD_{i,j}}{\epsilon_s} , \tag{6}$$

where $g_i$ and $h_j$ are the distances between adjacent mesh lines in the $x$ and $y$ directions, respectively, and are defined by $h_j = y_{j+1} - y_y$ and $g_i = x_{i+1} - x_i$. Combining terms of $\phi_{i,j}$ with identical subscripts, and introducing the notation: $\Phi_{i,j} = \frac{q\phi_{i,j}}{kT}$, $\beta = \frac{qn_i}{\epsilon_s}$, and $f_{i,j} = \frac{-qD_{i,j}}{\epsilon_s}$, we arrive at equation (7).

$$\alpha^{N}_{i,j}\, \Phi_{i,j+1} + \alpha^{S}_{i,j}\, \Phi_{i,j-1} + \alpha^{E}_{i,j}\, \Phi_{i-1,j} + \alpha^{W}_{i,j}\, \Phi_{i+1,j} - \alpha^{C}_{i,j}\, \Phi_{i,j} - \beta\, (\exp(\Phi_{i,j}) - \exp(-\Phi_{i,j})) = f_{i,j} . \tag{7}$$

The mesh constants $\alpha^S_{i,j}$, $\alpha^N_{i,j}$, $\alpha^E_{i,j}$, $\alpha^W_{i,j}$, and $\alpha^C_{i,j}$ are given in terms of mesh lines by Equations (10), (11), (12), (13), and (14). (Note that these constants are defined only for nonboundary mesh points).

$$\alpha^S_{i,j} = \frac{2kT}{q(h_j - h_{j-1})h_j} = \frac{2kT}{q(y_{j+1} - y_{j-1})(y_j - y_{j-1})} . \tag{8}$$

$$\alpha^N_{i,j} = \frac{2kT}{q(h_j - h_{j-1})h_{j-1}} = \frac{2kT}{q(y_{j+1} - y_{j-1})(y_{j+1} - y_j)} . \tag{9}$$

$$\alpha^E_{i,j} = \frac{2kT}{q(g_i - g_{i-1})g_i} = \frac{2kT}{q(x_{i+1} - x_{i-1})(x_{i+1} - x_i)} . \tag{10}$$

$$\alpha^W_{i,j} = \frac{2kT}{q(g_i - g_{i-1})g_{i-1}} = \frac{2kT}{q(x_{i+1} - x_{i-1})(x_i - x_{i-1})} . \tag{11}$$

$$\alpha^C_{i,j} = \alpha^E_{i,j} + \alpha^W_{i,j} + \alpha^N_{i,j} + \alpha^S_{i,j} . \tag{12}$$

The discretized forms of the boundary conditions, and the values of the mesh indices for which they apply, are given by Equations (13) through (18).These equations include the effects of external voltages applied to the MOSFET device, which include the gate voltage, $V_g$, the source voltage $V_s$, the drain voltage $V_d$, and the substrate voltage $V_{\text{sub}}$. The end points of the gate electrode are demarcated by the indices $i_{gs}$, and $i_{ge}$. The maximum indices of mesh lines in the x and y directions are given by $i_{\max}$ and $j_{\max}$.

**Ohmic Boundaries**

$$\Phi_{i,j} = \frac{qV_s}{kT} + ln \left\{ \frac{\left( \frac{D_{i,j}}{2} + \sqrt{\frac{D_{i,j}^2}{4} + n_i^2} \right)}{n_i} \right\} \quad \begin{matrix} \text{for } j = 0, \\ 0 \leq i \leq i_{gs} \ . \end{matrix} \tag{13}$$

$$\Phi_{i,j} = \frac{qV_d}{kT} + ln \left\{ \frac{\left( \frac{D_{i,j}}{2} + \sqrt{\frac{D_{i,j}^2}{4} + n_i^2} \right)}{n_i} \right\} \quad \begin{matrix} \text{for } j = 0, \\ i_{ge} \leq i \leq i_{\max} \ . \end{matrix} \tag{14}$$

$$\Phi_{i,j} = \frac{qV_{\text{sub}}}{kT} + ln \left\{ \frac{\left( \frac{D_{i,j}}{2} + \sqrt{\frac{D_{i,j}^2}{4} + n_i^2} \right)}{n_i} \right\} \quad \begin{matrix} \text{for } j = j_{\max}, \\ 0 \leq i \leq i_{\max} \ . \end{matrix} \tag{15}$$

**Side (Neumann) Boundaries**

$$\Phi_{i+1,j} = \Phi_{i,j} \qquad \text{for } i = 0, \qquad 1 \leq j \leq j_{\max} - 1 \ . \tag{16}$$

$$\Phi_{i-1,j} = \Phi_{i,j} \qquad \text{for } i = i_{\max}, \qquad 1 \leq j \leq j_{\max} - 1 \ . \tag{17}$$

**Dioxide Boundary**

$$\Phi_{i,j} = \frac{((y_{j+1} - y_j)\ \delta Q + \epsilon_{ox}\ V_g)q}{((y_{j+1} - y_j)\ \epsilon_{ox} + \delta\ \epsilon_s)kT}$$

$$+ \frac{\delta\ \epsilon_s\ \Phi_{i,j+1}}{\epsilon_{ox}(y_{j+1} - y_j) + \delta\ \epsilon_s} \qquad \text{for } j = 0, \qquad i_{gs} < i < i_{ge} \ . \tag{18}$$

According to the algorithm in Ref. 1, simultaneous Equations (7) are solved using the following iterations:

$$F_{i,j}^{(m+1)} = \alpha_{i,j}^N\ \Phi_{i,j+1}^{(m)} + \alpha_{i,j}^S\ \Phi_{i,j-1}^{(m)}$$

$$+ \alpha_{i,j}^E\ \Phi_{i-1,j}^{(m)} + \alpha_{i,j}^W\ \Phi_{i+1,j}^{(m)} - f_{i,j} \ , \tag{19}$$

and

$$\alpha_{i,j}^C\ \Phi_{i,j}^{(m+1)} + \beta\ (\exp(\Phi_{i,j}^{(m+1)}) - \exp(-\Phi_{i,j}^{(m+1)})) - F_{i,j}^{(m+1)} = 0. \tag{20}$$

The algorithm starts by choosing an (arbitrary) initial guess for potential at all nonboundary mesh points ($1 \leq i \leq i_{\max} - 1$), and then proceeds as follows: (1) Use Equation (19) to calculate $F_{i,j}^{(m+1)}$ from $\Phi_{i,j}^{(m)}$ at all nonboundary mesh points in parallel. This computation step is referred to as the outer loop, with index $m$. (2) Insert $F_{i,j}^{(m+1)}$ into Equation (20) which then must be solved for $\Phi_{i,j}^{(m+1)}$ at all mesh points in parallel. Because equation (20) is transcendental and cannot be solved for $\Phi_{i,j}^{(m+1)}$ directly, the one-dimensional iterative Newton method is employed. Formula (21) is the one-dimensional Newton-method formulation of Equation (20), which is applied to find $\Phi_{i,j}^{(m+1)}$ until the selected convergence criterion is satisfied.

$$\Phi_{i,j}^{(m+1)(n+1)} = \Phi_{i,j}^{(m+1)(n)} -$$

$$\frac{\alpha_{i,j}^C\ \Phi_{i,j}^{(m+1)(n)} + \beta\ (\exp(\Phi_{i,j}^{(m+1)(n)}) - \exp(-\Phi_{i,j}^{(m+1)(n)})) - F_{i,j}^{(m+1)}}{\alpha_{i,j}^C + \beta\ (\exp(\Phi_{i,j}^{(m+1)(n)}) + \exp(-\Phi_{i,j}^{(m+1)(n)}))} \ . \tag{21}$$

The repetitive application of Formula (21) is referred to as the inner loop, with index variable $n$. (3) The value of $\Phi_{i,j}^{(m+1)}$ found by Formula (21) is compared to $\Phi_{i,j}^{(m)}$. If their difference, for all nonboundary mesh points, is less than some convergence threshold, the algorithm terminates with $\Phi_{i,j}^{(m+1)}$ as the solution. (4) Otherwise, computations proceed again to step 1.

## MPP ARCHITECTURE

Before describing the specifics of the implementation of the above algorithm, it is appropriate to describe briefly the MPP architecture. Complete details on the MPP are available to the interested reader in Ref. 2. We present only the architectural details of the MPP that are significant to the understanding of the implementation of the above algorithm.

The MPP operates as an attached processor to a host computer, which is currently a VAX 11/780. The host computer performs high-level control, provides disk storage, and supports software development tools for the MPP.

There are three basic units of the MPP: (1) the Array Unit (ARU); (2) the Array Control Unit (ACU); and (3) the Staging Memory that reformats and buffers data passing between the ARU and the VAX host.

The ARU consists of 16,384 PEs, which are interconnected to form a two-dimensional mesh with 128 PEs on a side. PEs that are not on the boundary of the matrix have a direct connection to their four nearest neighbors, designated as its East, West, North, and South PEs. The boundary processors have only three nearest neighbors; the remaining connection for these PEs is determined by a software-configurable topology control register. For our implementation, the boundary PEs were left unconnected.

In addition to the four communication paths, each PE has a single-bit ALU, 35 single-bit registers, and 1024 bits of local RAM. One of the single bit registers is the mask register, which is used to control the activity of the PE. The purpose and operation of this mask register will be described later.

The PE instruction set includes the standard arithmetic and logical operations, as well as inter-PE data transfer primitives. Moreover, the ARU hardware supports array reduction operations. Reduction operations provide a parallel method for finding a selected quantity; for example, the maximum value across all PEs in the ARU. Both the inter-PE data transfer primitives and array reduction operations were architectural features exploited by our implementation.

The single-bit ALUs in the MPP PEs mean that the PE instructions operate on single-bit operands. Multiple-bit fixed-point arithmetic is emulated on the MPP by performing multiple single-bit instructions. The MPP can be programmed to perform arbitrary precision fixed point arithmetic as well as floating point arithmetic.

The MPP is classified as Single Instruction Multiple Data (SIMD) architecture. This means the following. At each instruction cycle, a Single Instruction (SI) is executed concurrently by all PEs, acting on Multiple Data (MD) points. In the MPP, there exists the capability to individually "mask" off PEs, disabling them from executing current instruction. The control of this mask register is available to the programmer. The stream of parallel operations for execution in the ARU is generated by the MPP's Main Control Unit, which will be described in a subsequent section.

The MPP Stager facilitates movement and reformatting of data between the VAX host and the MPP array unit. It connects to the MPP through the s-plane, and to the VAX through a 6-MByte/sec, 16-bit parallel bus. For each PE in the ARU, the Stager contains an additional 16K bits of RAM memory. These data can be transferred between the ARU and Stager at a high bandwidth (180 MBytes/sec.). Data to be transferred from the VAX to the ARU (i.e. an input data set) is first transferred to the Stager at a 12-MByte/sec. rate. The Stager reformats the data, and then transfers the data to the ARU, through the s-plane, at the high rate. Because of the low bandwidth between the Stager and the VAX, it is desirable to minimize the amount of data traversing this path.

The Main Control Unit (MCU) controls both the Array Unit and the Stager. It is a 16-bit minicomputer that runs the MPP assembly language code. It contains separate controllers for the Array Unit (PECU) as well as the Stager Unit (IOCU). The PECU and IOCU are slave processors, taking commands from the MCU and then operating in an independent fashion, in parallel with the MCU as much as allowed by the constraints imposed by the executing program.

The MPP can be programmed at the microcode level, the assembly language level, or using a high-level language. The usual tradeoffs for selecting the level of programming apply to the MPP as if it were any other computer. The MPP currently supports the PASCAL and FORTH languages.

The MPP implementation of PASCAL is called parallel Pascal (PP) (Ref. 3). It is an extension of the standard PASCAL language to include constructs that are directly supported by the MPP hardware. These extensions are: (1) parallel data-type declaration, for parallel arrays that reside in the MPP array memory; and (2) parallel operators and functions for operating on parallel arrays. The Where-do-otherwise statement for selected parallel assignments uses the mask feature of the PEs.

PP makes the power of the MPP available in a high-level language format. This gives the MPP the appearance of a general purpose computer, especially from the PP code level. The only nonstandard aspect about programming the MPP using PP is that the programmer must specify the movement of data from the VAX host to the Stager memory and then into the Array Unit. This is due in part to the limited memory available in the Array Unit and the Stager. The programmer is in the best position to decide when data can be moved and where local PE storage is available.

## IMPLEMENTATION SPECIFICS

Coding of the algorithm to run on the MPP was considerably simplified by the use of PP, which provided function calls to implement the ARU's hardware primitives (such as the array reduction and inter-PE data transfer operations). In addition, PP provided microcode support for all basic floating point operations, as well as higher order functions (i.e. exponentials).

This implementation provided for the use of grid sizes up to 128 × 128 mesh lines. Meshes that were smaller than this were mapped into the Northwest corner of the MPP ARU; see Figure 2. The unused processors in this figure were masked from performing any undesirable operations (i.e., divide by zero).

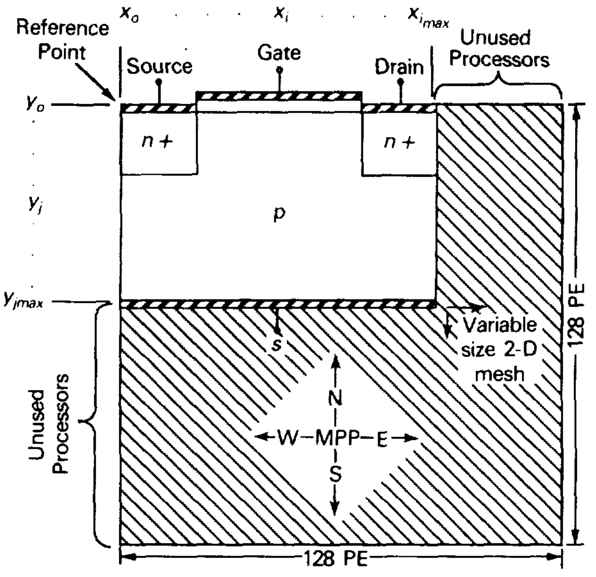


Figure 2. Variable Size MOSFET Geometry Mapped to the MPP Processor Array.

In PP, Equation (19) was efficiently implemented by four shift operations, four floating point multiples, and four floating point additions. For example, a shift operation transfers all values of $\Phi_{i,j}$ north by one mesh line (from $\text{PE}_{i,j}$ to $\text{PE}_{i,j-1}$) where it then is multiplied by $\alpha^S_{i,j}$. Subsequent shift and multiply operations, for the remaining three directions, would generate the other terms of Equation (19). The boundary PEs (i.e. $i = 0, i = i_{\max}, j = 0, j = j_{\max}$) do not implement Equation (19).

Implementation of Equation (21), while slightly more complex mathematically, is still quite straightforward. While this equation doesn't require any inter-PE data transfers, considerably more computation is required to calculate the exponentials and to perform the floating point division. Moreover, different PEs might require a different number of iterations of Equation (21) before satisfying the convergence criteria. Initially, we chose to disable any PE that had satisfied the convergence criteria from performing any more iterations of the Equation (21), the inner loop.

The boundary conditions were implemented separately for each boundary, according to Equations (13) through (18). Most of these are simple expressions, except for the dioxide boundary Equation (18), and require only a shift or an assignment operation.

This initial implementation was validated by comparing the results with those from a previous implementation in FORTRAN running on a VAX. Figure 3 shows an example of results generated by our implementation on the MPP. It illustrates the variation of electric potential within the MOSFET device as a function of both the $x$ and $y$ coordinates. This implementation used a mesh constructed from 128 × 128 mesh lines. Simplistic (constant) doping profiles were used in this example, with doping in the $n$ region $D = 1 \times 10^{18}$ cm$^{-3}$, in the $p$ region $D = -1 \times 10^{18}$ cm$^{-3}$, $\delta = 0.05\ \mu m$, $Q = 0$, and $V_g = V_s = V_d = V_{\text{sub}} = 0$ volts.

Some efforts have been made to improve run-time efficiency. The most inefficient aspect of the algorithm implementation was realization of the boundary conditions. In the later case, only about 200 PEs were doing any useful operations at a given time; the remaining PEs were sitting idle, for a PE utilization of approximately 1%.

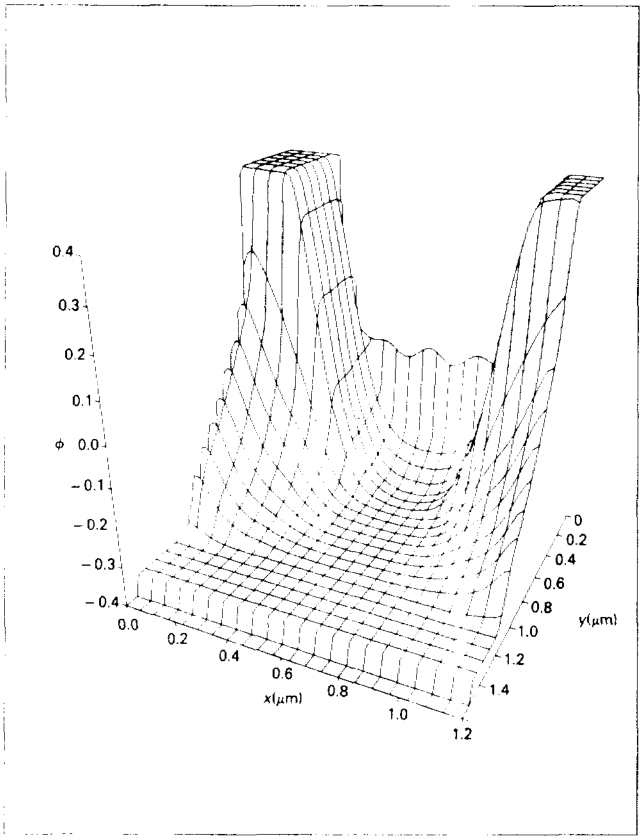


Figure 3. Plot of Potential Versus Position in a MOSFET Device.

To improve efficiency, we reorganized the computations to include the boundary condition computations into the general nonboundary mesh point equations. The motivation for this was the fact that implementing the boundary condition equations required only shifts and multiplies, which were already being performed in all PEs during the solution of Equation (19). Because the boundary PEs are not involved in the realization of Equation (19), and thus remain idle, they could possibly be used during this period to improve execution efficiency.

Substituting in the boundary conditions into Equation (19) at the appropriate pre-boundary mesh points, new values for the mesh constants were derived. The modified $\alpha$'s for each boundary condition, and the mesh indices for which they apply, are presented as Equations (22) through (27).

### Ohmic Boundaries

$$\alpha^C_{i,j} = \alpha^E_{i,j} = \alpha^W_{i,j} = \alpha^N_{i,j} = \alpha^S_{i,j} = 0 \tag{22}$$

$$\text{for } j = 0,\ 0 \le i \le i_{ge};$$

$$j = 0,\ i_{ge} \le i \le i_{\max};$$

$$j = j_{\max},\ 0 \le i \le i_{\max};$$

and $\Phi_{i,j}$ given by Equations (16), (17), and (18) remains constant $\forall m$ (for all outer loop iterations).

### Neumann Boundaries

*Boundary PEs*

$$\alpha^C_{i,j} = \alpha^E_{i,j} = \alpha^W_{i,j} = \alpha^N_{i,j} = \alpha^S_{i,j} = 0$$

$$\text{for } i = 0,\ 1 \le j \le j_{\max} - 1;$$

$$i = i_{\max},\ 1 \le j \le j_{\max} - 1,$$

*Pre-boundary PEs*

$$\alpha^W_{i,j} = 0; \alpha^C_{i,j} = \alpha^E_{i,j} + \alpha^N_{i,j} + \alpha^S_{i,j} \text{ for } i = 1, 1 \le i \le j_{\max} - 1\ . \tag{24}$$

$$\alpha^E_{i,j} = 0; \alpha^C_{i,j} = \alpha^W_{i,j} + \alpha^N_{i,j} + \alpha^S_{i,j} \text{ for } i = i_{\max} - 1, 1 \le j \le j_{\max} - 1\ . \tag{25}$$

### Dioxide Boundary

*Boundary PEs*

$$\alpha^C_{i,j} = \alpha^E_{i,j} = \alpha^W_{i,j} = \alpha^N_{i,j} = \alpha^S_{i,j} = 0$$

$$\text{for } j = 0,\ i_{gs} < i \le i_{ge}\ ;$$

$$\Phi_{i,j} = \frac{(y_1 - y_0)\ (\delta\, Q + \epsilon_{ox}\ V_g)q}{((y_1 - y_0)\ \epsilon_{ox} + \delta\ \epsilon_s)\ kT},\ \forall m \tag{26}$$

*Pre-Boundary PEs*

$$\alpha^C_{i,j} = \alpha^E_{i,j} + \alpha^W_{i,j} + \alpha^S_{i,j} + \alpha^N_{i,j} \left\{ \frac{(y_1 - y_0)\ \epsilon_{ox}}{\epsilon_s \delta + (y_1 - y_0)\ \epsilon_{ox}} \right\}$$

$$\text{for } j = 1,\ i_{gs} < i < i_{ge} \tag{27}$$

The second source of inefficiency for this algorithm has been in the inner loop. Here, some PEs require more iterations of Newton's method before satisfying the convergence criteria. It is important to note that solving Newton's method, Equation (21), requires considerably more computation steps than the outer loop, Equation (19).

From our observations of the algorithm's execution characteristics, after the first few iterations of the outer loop, at most, three iterations of Newton's method (the inner loop) were required before all PEs satisfied the convergence criteria. The PEs that required two or three iterations were those whose mesh positions were in a region that was undergoing large changes of $F_{i,j}$, the inner loop forcing function. Since the value of potential in these PEs is very likely to change again during the next iteration, determining their accuracy to high precision is inefficient.

Rather than let each PE determine for itself when it had satisfied the convergence criteria, Newton's method could be applied a fixed number of times by all PEs. This method trades off the time spent computing in the inner loop against the total number of outer loop iterations required for termination of the algorithm. Table 1 summarizes the trade-off results we have collected. It presents the number of outer loop iterations (and the total number of inner loops executed) required for algorithm termination as a function of the number of inner loop applications, for different mesh sizes.

Table 1. Inner Loop versus Outer Loop Trade-off Results.

| Maximum inner loop iterations | Mesh Size | | |
|---|---|---|---|
| | 32 × 32 | 64 × 64 | 128 × 128 |
| 1 | •OL = 82<br>†TIL = 82 | OL = 170<br>TIL = 170 | OL = 285<br>TIL = 285 |
| 2 | OL = 84<br>TIL = 168 | OL = 182<br>TIL = 364 | OL = 309<br>TIL = 618 |
| ≥3 | OL = 85<br>TIL = 183 | OL = 182<br>TIL = 381 | OL = 309<br>TIL = 631 |

•OL is the number of outer loop iterations.
†TIL is the total number of inner loops performed.

These experimental results indicate that computing the solution of Equation (20) to a high accuracy, by allowing a large number of inner loop iterations, does not improve and can actually retard (see Table 1) the global convergence of the algorithm. This characteristic appears to be a function of the initial guess used.

Intuitively, this make sense by the fact that an accurate solution to Equation (20), for an initial guess that is far from the actual solution, can cause the new estimate to remain farther away than a less accurate computation. This can result in an increase in the total number of outer loop iterations to achieve global convergence.

## SUMMARY

This paper has described the implementation of a new, inherently parallel algorithm for solving the nonlinear Poisson equation of semiconductor device theory on the MPP. This parallel implementation has also provided insight into techniques for improving run-time efficiency of this algorithm, including integrating the boundary conditions into the standard mesh point equations, and the interrelationship between the number of inner loop iterations and the number of outer loop iterations required for global convergence.

Plans for additional work with this algorithm include extending the algorithm to the modeling of a three-dimensional MOSFET device. Next, electron and hole continuity equations can be incorporated into the discretized implementation. Furthermore, realistic doping profiles can be added using a program similar to SUPREM (Ref. 4.).

## REFERENCES

1. I. D. Mayergoyz, "Solutionof the Nonlinear Poisson Equation of Semiconductor Device Theory," *Journal of Applied Physics*, 1986, 59, p. 195.
2. *The Massively Parallel Processor*, J. L. Potter, Ed., MIT Press series in scientific computation, MIT Press, Cambridge, Mass., 1985.
3. A. P. Reeves, "Parallel Pascal, An extended Pascal for Parallel Computers," *Journal of Parallel and Distributed Computing*, 1984, Vol. 1, pp. 64–80.
4. D. A. Antoniadis and R. W. Dutton, "Models for Computer Simulation of Complete IC Fabrication Process," *IEEE Journal of Solid-State Circuits*, April 1979, Vol. SC-14, No. 2, pp. 412–430.

# DARPA SENSOR NATIONAL TESTBED: HARDWARE AND SOFTWARE ARCHITECTURE

D.R. Guarino, R.P. Kruger, S. Sayre, T. Sos, C.J. Turner, C.L. Winter
Science Applications International Corporation

## ABSTRACT

A heterogeneous network of parallel computers developed for complex distributed processing applications is described. Network computers include a Connection Machine, a Butterfly multiprocessor, a WARP systolic array, a Symbolics and several SUN Workstations; an Ethernet and high-bandwidth APTEC bus supports data transfers. Distributed applications are built from individual processes executing on computers in the network. A powerful asynchronous communication facility is built upon the multiple computer operating systems to provide uniform message passing, global memory variables and remote process execution services to processes. An Executive Controller and the LISP+ functional language provide a method of integrating distributed processes into an application with transparent control of network resources and communications. Additional applications can be rapidly built from existing processing to support experiments in distributed and parallel applications.

Keywords: Message Passing, Data Flow, Software Backplane, Distributed Computing, Parallel Computing, Parallel Processing

## INTRODUCTION

Efficient imagery exploitation is a complex activity that requires the application of diverse algorithms. Technical challenges include the detection and recognition of image signatures; understanding of non-imagery based cues such as terrain; automated hypotheses generation and inference; information fusion; and control of distributed applications. These algorithms require diverse machine architectures to be efficiently implemented (Ref. 1-2), and flexible control strategies to manage distributed resources. We describe the Defense Advanced Research Projects Agency (DARPA) Sensor National Testbed (SNTB) as an integrated solution to support rapid and flexible development of heterogeneous, distributed processing applications.

A major objective of the SNTB was to develop a general purpose programming environment for distributed and parallel applications. This environment was designed to increase programmer productivity by shielding application developers from machine dependencies, and to support a laboratory atmosphere for experimentation in distributed and parallel applications.

The result is a loosely-coupled heterogeneous network of computers integrated with a flexible, distributed communications facility for asynchronous message passing, and a central control facility for applications development. Applications are integrated from individual processes executing on any computer in the network; thus, algorithms can be implemented on the most applicable machine architecture. Communications and control between processes is provided across computers and across operating systems.

## HARDWARE DÉSCRIPTION

The hardware architecture of the SNTB is a heterogeneous network of computers integrated into a single environment. Major hardware subsystems are either uniprocessors or homogeneous multiprocessors of varying granularity as shown in Figure 1. An APTEC high bandwidth data bus provides a means to centrally store and share imagery and other large data quantities among computers. Ethernet low-bandwidth message-handling protocols are used to support message passing and sharing of small data transfers.

The SNTB is designed to facilitate efficient implementation of the individual algorithms (numeric or symbolic) comprising an application by providing five different machine architectures to which an algorithm can be mapped. Multiple SUN Workstations and a VAX provide traditional computing with one central processor running UNIX. The SYMBOLICS Lisp machine has hardware optimized for executing programs in that high level language. Multiple instruction, multiple data (MIMD) processing is performed on the Butterfly multicomputer, while single instruction, multiple data (SIMD) processing is performed on the massively parallel Connection Machine. The hardware suite is completed with a WARP linear systolic array. The Butterfly, Connection Machine and WARP are the primary computing resources of the SNTB.

The Butterfly, produced by Bolt, Beranek and Newman is a network of up to 256 identical processor nodes. Each node is a Motorola 68020 microprocessor with a floating point co-processor and 1 to 4 MBytes of local memory. A proprietary "butterfly" switch network allows local memory to be tightly coupled and shared between processor nodes. Thus, the Butterfly achieves its processing performance by using multiple low-cost processors cooperatively. The SNTB currently has a 28 node Butterfly with 76 MBytes of total memory.

The MIMD architecture and shared memory of the Butterfly can be used to effectively exploit control level parallelism. Each identical processor can execute its own set of instructions, meaning different processors can run different programs on different data sets. And the shared memory architecture provides a program execution environment in which tasks can be distributed among processor nodes without regard for the physical location of the task data, although some performance is lost if the data does not reside in local memory.

The Butterfly uses the Chrysalis Operating System which supports both heterogeneous and uniform programming. In the heterogeneous mode, each processor node may run a different program. Under the uniform system, each node, or set of nodes, runs the same program. Software development is performed on a SUN Workstation host and downloaded into the Butterfly for execution. Programming languages include C, Fortran, and Scheme. Current testbed applications include parallel methods for Monte Carlo Simulation and R-Tree based spatial search.

The Connection Machine is a SIMD architecture with high processor granularity built by Thinking Machines Corporation. Each processor is interconnected with both a local grid (N, E, S, W) hardware interconnect and a highly efficient global packet switched interprocessor hypercube network. The Connection Machine is the first architecture to fully exploit data level parallelism.

The initial Connection Machine model CM-1 contains up to 65,536 bit serial microprocessors, and 32 MB of total physical memory. The SNTB has a 16,384 node CM-1 for a total of 1,000 million instructions/second (MIPS) capability. The successor to the CM-1, the CM-2, will support parallel 32 and 64 bit floating point arithmetic and increase the total physical memory by a factor of 16 over the CM-1. It is capable of 3500 MFLOPS and was introduced in the fourth quarter of 1987.

The Connection Machine uses a DEC VAX 8000 series computer, linked over a special high speed interface, as a host for software development and control. The C* and *LISP languages for programming the Connection Machine are extensions of the standard languages. Library functions logically create virtual nodes to match the number of processors to the problem. For example, image processing on a 512 x 512 pixel image is easily performed on a 16K node machine by mapping one image pixel to each of 256K virtual processors. Existing SNTB applications include SAR imagery detection and spatial clustering operations.

The underlying architecture of the WARP is a linear systolic array consisting of 10 powerful computing cells. It was developed as Carnegie Mellon University and built by General Electric Company. Each cell is capable of 10 MFLOPS, giving the 10 cell array a combined maximum throughput of 100 MFLOPS. Two independent I/O buses can pipe data through the array at 20 MBytes per second or store data in any cell's 128 KBytes of memory.

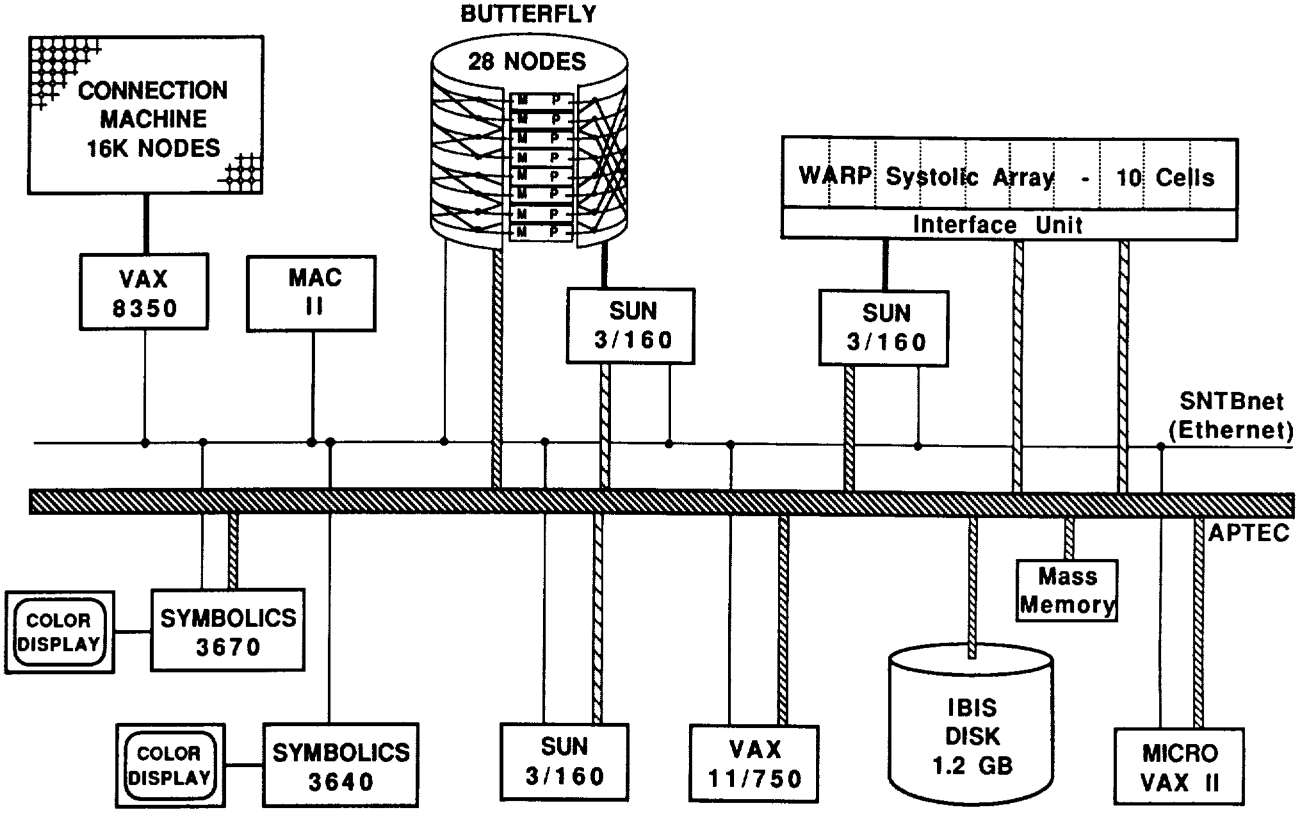


FIGURE 1. SENSOR NATIONAL TESTBED ARCHITECTURE. Lightly shaded interfaces to the Aptec bus are not currently implemented.

There are three main hardware components to the WARP: an external SUN host processor, the WARP systolic array, and an interface unit with two cluster processors. The SUN is used for program development and WARP control. In operation, data is loaded into the memory of the cluster processors. An interface unit then converts between fixed and floating point formats while the data is transmitted from one cluster processor through the array and to the other cluster processor. The cluster processors are also programmable. The APTEC data bus feeds data into or out of the cluster memories directly without passing through the SUN host.

The WARP can be programmed in either W2, a high-level C like language that gives the programmer full control over each processing cell and the data flow through the array, or in APPLY which only supports homogeneous programming. In this mode, each cell executes the same program on different data. W2 allows the WARP array to be programmed for heterogeneous processing where each cell may execute a different program, usually on the same data as it is passed through the array. Current testbed applications of the WARP include a true systolic algorithm for SAR imagery detection.

Two distinct communications networks are available for passing data and messages between processors on the testbed. Images and other large data items can be transferred over the APTEC bus and stored on the IBIS disk drive. The APTEC DPS-2400 is a synchronous, high-speed computer bus with a transfer rate of 24 MByte/second over two 12 MByte/second, read/write, uni-directional buses. The APTEC network includes 9 MBytes of mass memory and a high-speed 1.2 GByte IBIS disk. The disk and mass memory are useful for storage of intermediate results. The APTEC host computer is a MicroVAX II and acts as the controller for the APTEC network file system. The Ethernet is a slow speed bus for the transfer of messages and other control data between computers on the SNTB.

## SOFTWARE DESCRIPTION

Several layers of software have been added on top of the operating systems of the individual computers. This additional software provides the various layers of abstraction shown in Figure 2 and consists of three primary components: Communication and System Services, an Executive Controller Agent, and Network Vision Software. Together, this software integrates the various hardware components of the SNTB into a single programming environment called the Virtual Computer.

### Communication and System Services

For an application to use multiple machines in the SNTB, processes executing on each machine must not only share data, but they must know when to execute. This implies the need for both message passing and control mechanisms. Although this capability is provided through the Ethernet bus with TCP/IP software, it has several disadvantages. First, the protocols are at too low of a level and do not directly support interprocess data sharing or control. Second, the protocol implementations and syntax vary between machines. And third, each application must derive and implement its own requirements. These factors lead to longer development time and reduced software portability.

These limitations initiated development of the SNTB Communications and System Service (CSS). The CSS is a distributed service used to provide flexible asynchronous inter-process communication among processes executing on the SNTB. Additionally, it provides system and network wide functional tools and abstractions upon which the LISP+ and Data Flow Model of the Executive Controller are implemented. The basic services are provided by communicating Local System Servers (LSS), one on each machine in the testbed, while uniform interfaces to each LSS are provided by libraries of functions that are linked with the process' software.

**Message Passing** The basic inter-process communication paradigm provided by the CSS is asynchronous message passing between logically named, distributed client processes called agents. One of the main goals of the CSS is to hide details of the actual locations of agents from other communicating agents. This location independence is realized by providing logical named references. Logical names are managed by the collective LSSs and are known as io-ids or just ids. Ids are unique within the CSS and are dynamically created or deleted by agents.

Each id is associated with an object in the CSS. Objects are nodes in directed acyclic graphs (DAGs) used for message routing and storage. Each node in the graph has a separate and unique id, while each leaf also has a unique machine location in the SNTB, allowing CSS messages to be queued at and removed from the leaves of the graph. In a simple communication between two agents, the graph would have two nodes which are also leaves, and the message would flow along the edge of the graph from one node to the other.

The CSS provides a powerful facility for mapping ids. This provides the abstraction of "connecting" nodes in the DAG where the intermediate, non-leaf nodes of the graph are mapped ids. Each node may be mapped to one or more other nodes which may be mapped to other intermediate nodes before being mapped to a leaf node. The only limitation is that a circular path cannot be formed within the graph. The result is a very flexible and controllable message routing facility.

Agent message passing is accomplished through functional constructs that take on various forms of send and receive. An agent can send to any id in the CSS and optionally wait for a reply. Likewise, an agent can receive from any id previously associated with the agent. (This association constraint is necessary to support some receive functional abstractions provided by the interface libraries.) Both synchronous and asynchronous receive operations are supported. A synchronous receive is provided by waiting for a message to appear on a particular id; this allows processes to synchronize with each other. An asynchronous, or conditional receive, allows a process to poll for messages and continue execution if none are currently available.

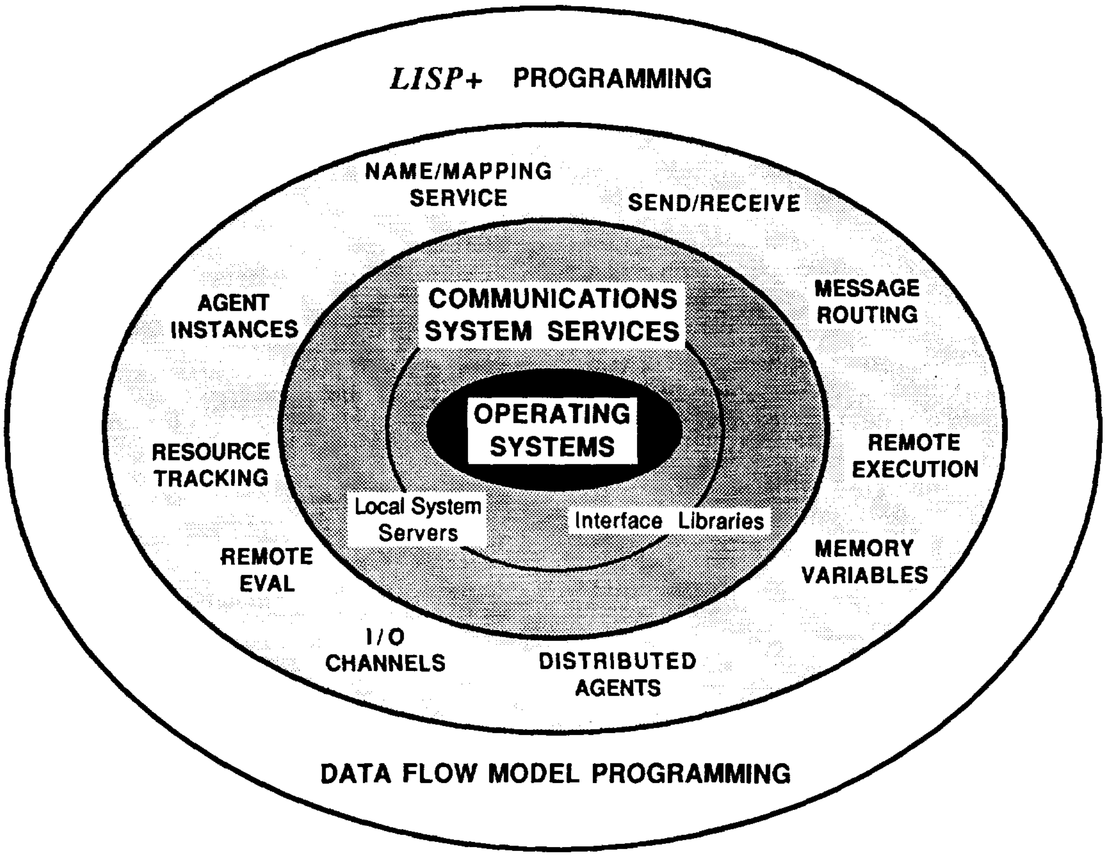


FIGURE 2. SNTB FUNCTIONAL ABSTRACTION LAYERS.

Individual messages have priorities associated with them to control the way in which they are queued. The priority is specified when the message is sent, or defaulted to a preset value. Messages with higher priority are de-queued to satisfy a receive request before those of lower priority. This facilitates some degree of control over message traffic that would not otherwise be available.

**Other Services** Besides application agent message passing, the CSS provides other important services built upon the general message passing paradigm discussed above. These include global memory variable,d remote process execution, resource tracking and message logging.

The CSS provides a mechanism for sharing global memory between agents in the form of memory variables. This abstraction is currently supported with a special kind of io-id. The queue for memory ids is implemented such that a send overwrites the current message, and a receive returns a copy of the last stored message. With this implementation, variable assignment and access is performed using send and receive constructs identical to those for message passing. Except for the agent that created the memory variable id, other agents are not aware of its special status.

The capability to remotely spawn agents from a single location is provided through the CSS with a special message construct. Each LSS has the responsibility of spawning processes through its local operating system. When an LSS receives this message-based command, it tries to execute the process contained in the message's data field and returns the status in a reply message. A similar special message construct is used to terminate an existing process.

Resource management is an important issue in the control of most distributed systems. The resources of the SNTB can be thought of in terms of the io-ids and processes that have been created through the CSS. In order to provide access to the state of the CSS, each LSS responds to a special status message construct. The reply contains a snapshot of the io-id and process state on the LSS's local machine. The message contains the number of sends and receives for each id, the number of stored bytes in the queue for each id, and id and process history.

A message logging capability is provided within the CSS to help with debugging SNTB applications. Both local and global message logging are supported. Local message logging causes a copy of all message traffic through an LSS, and a log of all LSS processing errors, to be written out to a file on the local machine. Global message logging causes a copy of all messages through an LSS to be sent to a particular id in the network. In this way, one location can keep tract of all messages in the CSS.

**Interface Libraries** With each LSS implementation, a set of primitives was developed in the programming language of the LSS. These primitives could be used to interface with the LSS on each machine. However, they do not provide the uniform, language independent agent interfaces supported on the SNTB. For this reason, Communication Interface Libraries (CILs) were developed as a part of the CSS.

Built upon the primitives available to each LSS, these libraries provide exactly the same functionality for using the CSS to every agent in the SNTB. There are currently two libraries, one in Common LISP (LCIL) and one in C (CCIL), although a CIL can be easily developed for any language available on a machine as long as the language interface to the operating system and LSS primitives is also available.

The CILs primary goal is to provide a uniform and portable interface between each agent and the LSS. The CILs however, also provide the additional capability for hiding the details of message transfer from the agent; multiple destinations in message send operations; and local error and consistency checking.

In summary, the CSS provides a uniform interface to its client processes, regardless of the machine the client is executing on. This uniformity promotes modular client development and portability of software where, in many cases, software is source code compatible across a few of the different machines in the SNTB. Such capabilities allow greater flexibility of experimentation in distributed and parallel processing. The paradigm of the CSS is used for both intra-machine and inter-machine communication between agents since it does not limit an application to any one data structure or communication format. The form and content of messages are left to the application, as is the way the messages themselves are used.

## Executive Controller

The Executive Controller (EC) provides the user with a single interface to the entire testbed, thus treating the SNTB as a virtual computer. The EC allows the user to easily configure a distributed application. It is designed to facilitate the process of controlling and experimenting with distributed applications on the testbed. As such, it can maintain information about SNTB computers, application agents and the communications within an application.

The EC is an application agent designed for a special purpose in the SNTB. It derives all its capabilities from self contained knowledge and the CSS, and is given no system level privileges that are not also available to other agents through the CSS. Its purpose is to provide several functions commonly required in distributed applications so that new applications can be quickly and easily developed on the SNTB. This frees the programmer to concentrate on details of the application rather than on details of the SNTB.

The EC provides two different models of computation that can be used: a graphics oriented dataflow model, and a functional configuration language, LISP+. For both models, the programmer must supply information about each agent in the application. This information includes agent input and output behavior, startup parameters, host computer, and data dependencies. The EC uses this information when starting up agents and configuring the communication channels between them.

**Data Flow Model** The Executive Controller provides a set of graphical, user oriented tools for experimenting with a distributed application from a data flow view point. The user can easily configure an application, activate and deactivate agents and specify data routings with the aid of mouse sensitive icons while the entire application agent suite is displayed on the screen. Mouse sensitive objects on the display allow the user to selectively view agent I/O information, equivalencies and host processors.

The user configures an application by connecting data paths between agents on the screen. Assuming the I/O behavior of the agents is compatible, the EC will define all necessary io-ids and communication channels through the CSS. This process is transparent to the user. Control. functions such as conditionals are handled by special user control agents designed for that purpose.

For example, a single detection agent may operate on multiple resolutions of imagery, while a subsequent clustering agent may be specific to high resolution. Under the data flow model, the user would have to provide a special agent that directed the data according to the imagery resolution. This agent would have a single input and two outputs. Its logic would be very simple: IF (data from high resolution imagery) THEN (send data to output A) ELSE (send data to output B).

Once the application is configured it may be executed. During execution, the EC will collect and display various status information such as individual agent execution times and processing advancement. Additionally, inter-agent messages and intermediate results are saved for later examination. This information is available to the user through the mouse sensitive icons.

The data flow model is useful for experimenting with distributed systems because of its graphical interface, but its limitations soon become exceeded. For example, it requires the network configuration to be static once execution has begun. Many distributed and parallel applications must dynamically reconfigure their communication channels in response to external stimuli (data) or intermediate results. Although this could be handled by using special control agents as discussed above, it would rapidly become cumbersome and awkward.

**LISP+** The Executive Controller provides another means to configure and control a distributed application on the SNTB. This method allows the SNTB to be programmed as a virtual computer through the functional language LISP+. LISP+ is a powerful command interpreter that allows individual agents to be treated as functions in a programming language. Through LISP+, the EC shields the programmer from operational knowledge such as agent host identity, number of agent instances, message passing, and global variables by providing the required scheduling and routing operations.

LISP+ is designed as a language for configuring applications on the SNTB. It is implemented by adding functions such as defvar+, let+ and setf+ to Common LISP. These additional functions are simply special versions of the standard implementation that use the global memory variables of the CSS instead of local variables. Additionally, when LISP+ executes a function that is really an agent, the EC first performs the desired communications routing. As in the data flow model, these operations are transparent to the programmer.

In LISP+, conditional control is handled by the constructs of the Common LISP language. Special control agents, as in the data flow model, are not needed. This further allows the user to concentrate on the application problem rather than on control.

The EC also maintains an agenda where program fragments can be executed. Thus, an application can have a special agent (for example, and inference agent) that generates program fragments and places them on the agenda for execution. This gives an application the capability to dynamically respond to its data environment or to multiple competing demands placed upon it.

### Network Vision

The testbed contains several Symbolics and SUN Workstations that can be used for display of intermediate processing results and other information. One example might display imagery and associated terrain data overlays, as well as the results of image understanding operations all on a single display. Since this processing will likely be performed on different machines in the SNTB, Network Vision (NV) software was developed to direct all results to a single display of the virtual computer.

Network Vision is composed of a server and a set of libraries that operates in much the same way as a remote windowing system such as X-windows, but at a much "higher" level. The purpose of the NV software is to provide the application developer with basic image display routines used to display information from agents running on any computer in the SNTB. Current capabilities include operations such as window creation, deletion, and management, as well as image, terrain, detection and cluster display, and some simple graphics capabilities.

The NV server is an autonomous agent and performs all of the window management functions for agents in the application. Communication between an application agent and the server takes place via messages through the CSS. Agents do not directly communicate with the NV server, but instead use the NV library calls.

The NV library functions are used by the application agents. Currently a library exists for the Common LISP and C programming languages. The agent callable functions are written in a flexible keyword style to permit the use of default argument values. Each request is checked for simple errors, reformatted for the NV server and passed through the CSS. The NV system may be used in a synchronous manner, where the calling process suspends until the requested operation has been acknowledged; or display commands may be executed asynchronously, in which case no reply is sent.

All actions performed by the NV server currently take place relative to a window on the Symbolics color screen or SUN Workstation. Multiple display windows are managed. Initial placement of windows is determined by other windows which are currently active on the screen. The user can subsequently reposition the windows with the normal move/reshape mouse-based operations. The display windows are identified to the user by an agent name or other description.

## SUMMARY

We have described the heterogeneous computer network and integrated software of the DARPA Sensor National Testbed. Network computers include several von Neumann architectures, a SIMD and a MIMD machine, and a linear systolic array. Upon this foundation, SAIC has built the distributed asynchronous communications, control and display software required to integrate these diverse parallel architectures into a flexible testbed.

Application programming on the SNTB is a three stage process. First the application agents are developed on the individual machines in one of the standard programming languages available. Here the programmer is free to use any debugging tools provided on the local operating system to help in agent development. Second, agents are interfaced with the CSS. In the software, this is simply a matter of using functions for receiving and sending information. Third, the agents are integrated into the distributed application. The Executive Controller and Network Vision are tools available during this stage.

Once a number of agents are developed through the first two stages, they can be rapidly combined to form different applications. In this way, the SNTB aids in the rapid prototyping of new applications from existing application agents. Thus, the SNTB supports experimenting with and rapid prototyping of distributed and parallel applications.

Although the testbed was initially developed for complex image exploitation tasks, it provides a general purpose programming environment for distributed and parallel applications. This environment permits the user to construct multiple application agent level execution scenarios without requiring him to have detailed knowledge of the system testbed layers, thereby improving programmer productivity.

## ACKNOWLEDGMENTS

This work was performed under the Advanced Digital Radar Imagery Exploitation System (ADRIES) program, DARPA Contract No. DACA76-86-C-0011. The authors would like to acknowledge the support of DARPA and their contracting agent, the U.S. Army Engineering Topographic Laboratories.

## REFERENCES

1. Cantoni, V., and Levialdi, S., "Matching the Task to an Image Processing Architecture," *Computer Vision, Graphics, and Image Processing*, Vol. 22, 301-309, 1983.

2. Reeves, A.P., "Parallel Computer Architectures for Image Processing," *Computer Vision, Graphics, and Image Processing*, Vol. 25, 68-88, 1984.

3. Kruger, R.P., "DARPA SAR National Testbed Mission, Architecture and selected applications," AFCEA Eleventh Annual Seminar, March 1988.

4. Sayre, S.M., "ADRIES Communication System Functional Design Description," SAIC September 1988.

# EFFICIENT MANAGEMENT OF SENSORY DATA FOR AN AUTONOMOUS SUBMERSIBLE UTILIZING A PARALLEL PROCESSING ARCHITECTURE

Alhad Chande, Sondra Shapiro, and Art Acampora

Martin Marietta Aero & Naval Systems
Baltimore, Maryland

## ABSTRACT

Today's Remotely Operated Vehicles (ROVs) provide sensory data, so that the control of the underwater vehicle can be managed by an operator on the surface ship. To remove the operator from the control loop, creating a fully autonomous vehicle (untethered), requires new and improved sensor configurations and sensor post-processing techniques. In an autonomous vehicle, multielemental array sonar sensors are exploited to sense the surrounding environment, so that a 3-D obstacle surface can be reconstructed in real-time. The sensor hardware placed strategically around the vehicle provides forward and peripheral fields of view of the surrounding environment, in terms of massive amounts of data. Because of the limited space available on the submersible, obsolete sensor information must be discarded. This paper describes a systematic method of managing sensor information, utilizing a bit serial systolic array, so that irrelevant data storage and archival are rendered unnecessary.

Keywords: Systolic Arrays, Underwater Sensing, Obstacle Avoidance, Object Identification, Sonar, Lidar, Multisensor Fusion, Automated Sensing.

## SUBMERSIBLES

Manned oceanic submersibles of the early 1960's have matured into free-swimming and bottom-crawling Remotely Operated Vehicles (ROVs) with significant advances in operational efficiency in the hazardous undersea environments. The ability to execute deep sea exploration missions will have a positive impact on the industrial and military strength of a nation. The cost of manned exploration has been prohibitive, similar to that of space exploration, and large investments in this area have not been economically justifiable, with the exception of limited advances made for oil and natural gas explorations, at depths of 6000 feet. The parallel advances in mobile robotics, undersea sensors and processing, and, artificial intelligence and digital computing, have provided a technical base for the research and development of advanced teleoperated underwater vehicles and Autonomous Underwater Vehicles (AUVs).

The primary focus of contemporary autonomous vehicular research has been dichotomized into sensing and control. In terms of Draper's informator-effector model (Ref. 3), Figure 1 shows a three-tier control architecture for an AUV. The informator in this architecture consists of a pre-informator with apriori information for the autonomous system, and the in-situ informator deals with current data and real-time path planning. The effector is the physical real-time servo-controller for the actuators and propulsion system. The requirements for control of manned spacecraft or deep space probes are easier to predict, primarily because space transit is resistance free, and also, because on-board computers perform the critical function of monitoring forces affecting the ship and its attitude control. Contrarily, a six degree-of-freedom AUV requires a controller that can adapt to the action-reaction nature of the fluid medium engulfing it, and continually compensate for deviations from the desired trajectory due to hydrodynamic forces. This generates the need for a more frequent sensing of the surrounding obstacle environment, to allow for sensor post-processing delays and latencies of an autonomous controller and its servocontrollers. This paper deals with autonomous obstacle avoidance and object identification for AUVs, which constitutes the autonomous "informator," that is indespensible for efficient route planning and its plan execution.

## NECESSITY FOR MULTISENSOR FUSION

Multisensor fusion is the process by which specific unified data relating to an entity, the fusion entity, is obtained by the integration of information from multiple sensors. The fusion entity for the AUV is a reconstructed model of the 3-D world geometry of the obstacles in its vicinity. The typical model update rate is estimated to be 500 milliseconds or less for an AUV transit velocity of 2 to 8 knots. The number of sensor elements on the submersible could be up to 4000 elements, and these would supply meaningful range, azimuth, and elevation three-tuples after post-processing throughput delays. The fusion process commences to reconstruct a 3-D world geometry model of the obstacle surfaces that intrude the approximately 8000 cubic feet that encompass the mobile submersible. This estimate is based on a sensor ranging maxima of 2000 feet. This occurs after the sensor post-processors have range, azimuth, and elevation data from each sensor element suitably ingested and collated. This type of sensor fusion is termed as direct, autonomous, or local fusion, and is radically different from classic global fusion. Typically, global fusion deals with a battlefield scenario, where the fusion entity is in the form of intelligence reports for human perusal and analysis, and the sensors are widely distributed over different mobile or stationary platforms.

The real-time fusion algorithms on the Martin Marietta ASAP$^{TM}$ (Advanced Systolic Array Processor) (Ref. 4) spatio-temporally fuse 3-D range data from multiple and disparate types of sensors (Sonar/Lidar) into a cogently compressed world geometry of surfaces of avoidance; these algorithms and ASAP$^{TM}$ together, comprise the Fusion Processor. Further, the systolic array updates this surface of avoidance geometry incrementally, based on either sensor updates or compensations due to ship movement. The "systolic" action of the ASAP$^{TM}$

CH2649-2/89/0000/0303$01.00 © 1988 IEEE

provides a persistence to the reconstructed world geometry as it is viewed from a mobile submersible. The key issue in the obstacle and terrain avoidance functionality is the ability of the Fusion Processor to represent large amounts of range data, and provide the AUV controller with an updated report, within 100 to 500 milliseconds, detailing obstacle surfaces that must be avoided. Additionally, motion compensation for six degrees-of-freedom, especially to adjust for the vehicle's crab angle (actual path resulting from cross-currents), must be made at least every 100 milliseconds based on navigation fixes. This is also necessary during sensor-blackout down times resulting from the send-receive time periods of sensor operation.

The Martin Marietta fusion algorithms for the bit-serial ASAP$^{TM}$ are designed to successfully handle the spatio-temporal fusion for different densities of 3-D sparse image data in the required short durations.

## SYSTEM ARCHITECTURE

The entire electromagnetic (EM) spectrum is minimally suited for underwater sensing, because EM waves have high attenuations at high source powers, therefore, sensing underwater is predominantly acoustic. Maximum ranging distances of such sensors is small, with the exception of blue-green light of 480-580 nanometers. Light of these wavelengths can penetrate around a hundred meters, while high frequency (200 KHz) active acoustic ranging devices can range up to one thousand meters. Even though underwater laser devices suffer from transmissivity problems in turbid waters, they have the potential of acquiring images that are crisper that television,.for ten times the range of camera imaging. However, these Lidar (LIght Detection And Ranging) sensors are non-operational at ranges less that ten meters, so, cameras can be deployed for up to ten meters. The multi-beam sonar arrays can range up to one thousand meters with decreasing resolutions. Consequently, the front-end to an autonomous intelligent controller requires the use of range overlapped Lidar and Sonar devices of varying range maximas and resolutions. The sensor post-processors and fusion processor algorithmically extract and reconstruct a 3-D obstacle world model from data streams of successive swaths of multiple sensors strategically placed around the AUV. See Figure 2, for a forward looking overlapped sensor suite and a hardware block schematic of an autonomous sensing system.

The system architecture, shown in Figure 3, indicates how the systolic array receives ingested and collated data from the sensor post-processors and compresses it into a world model. This iconoclastic method departs from the conventional approach of incorporating large peripheral memories, such as disks, since the subsequent archival and data management from several such storages for several sensors results in a non-convergent processing scenario. Such a radically new approach to multisensor fusion, requires a single disk drive with limited memory, for the purposes of tagging analyzed symbolic versions of high resolution imaging processes, that are initiated by the low resolution world model, which resides in the systolic array. Such high resolution imaging processes are carried out in the high resolution sensor post-processor, comprising highly reconfigurable Datacube (Ref.2) hardware.

## WORLD MODEL

Restricted space on the AUV makes it infeasible to carry mass storage systems onboard with large amounts of data, that must further be archived and processed on demand. Moreover, peripheral storage data archival would scarcely lead to real-time performance, due to latency and I/O delays. Consequently, sensor data storage is mostly high speed random access, and is directly accessible by the processing elements (PEs) of the systolic array. This requirement leads to the need for a world model that stores only relevant information to the task at hand, and a world model that compresses its data storage to a minimum size. To achieve this, the world model stores, at bit level, the occupancy state of the environment surrounding the AUV as unknown, occupied, or empty. The task at hand is to

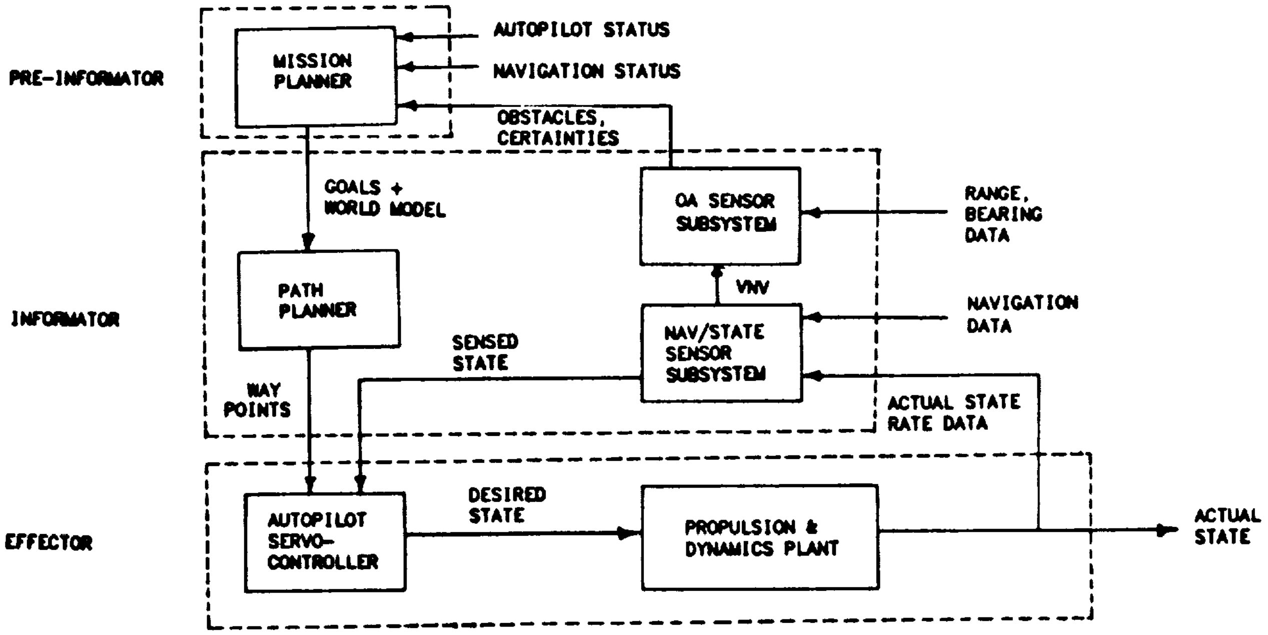


Figure 1. Informator - Effector Model

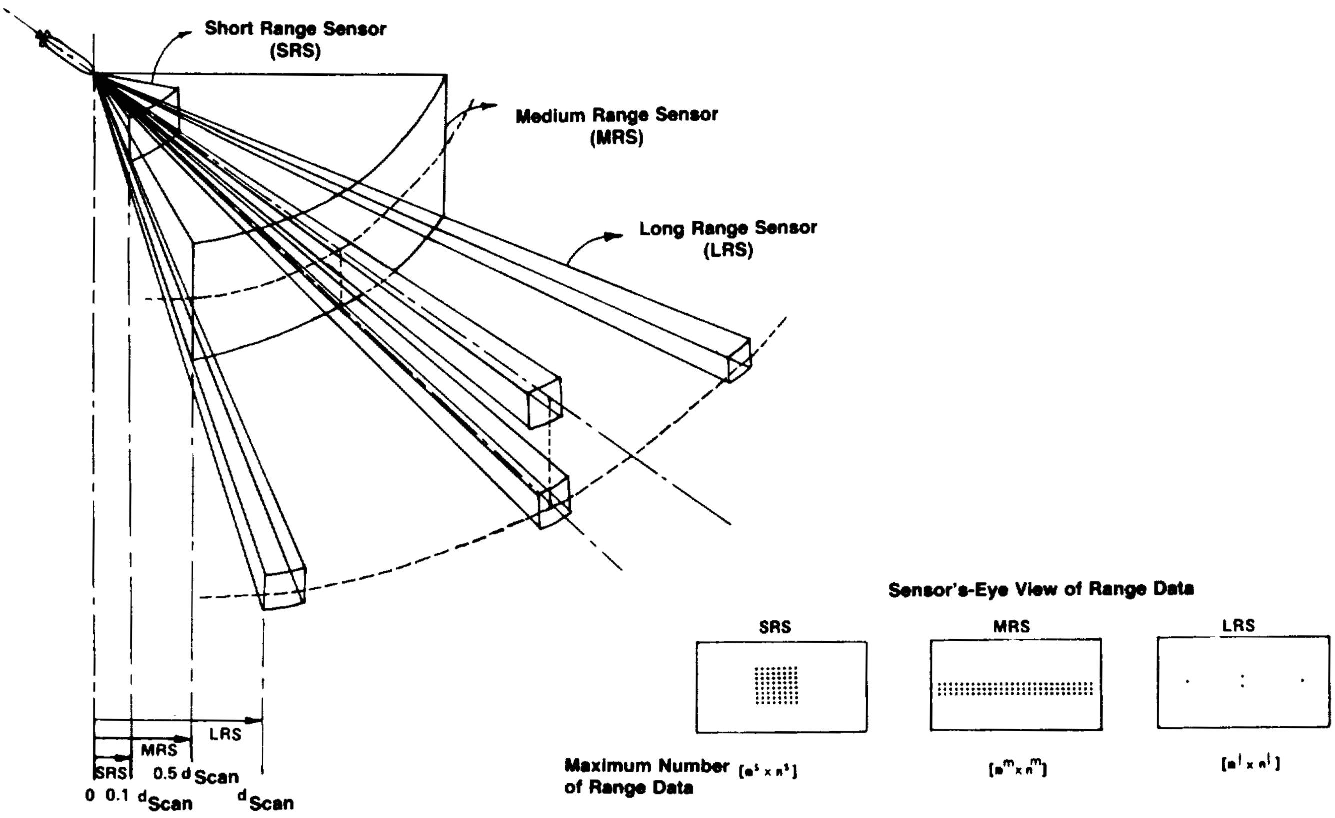


Figure 2. Prototypical Multisensor Configuration

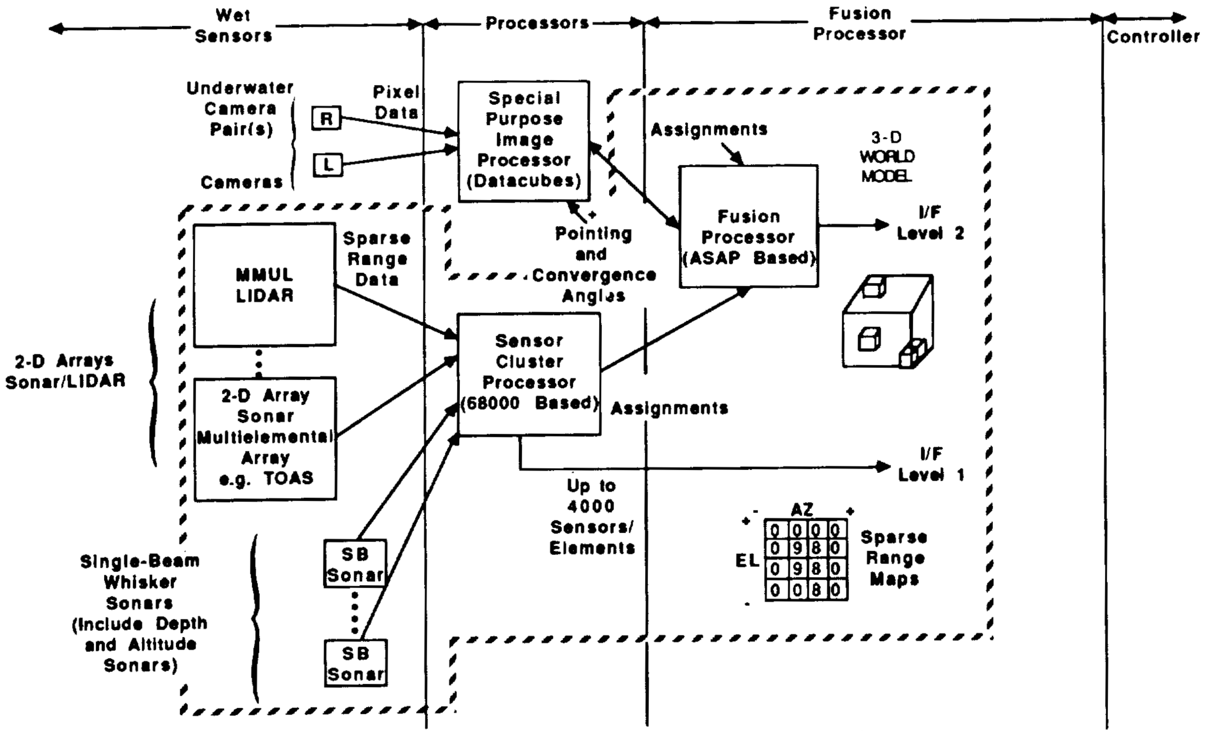

Wet Sensors
Processors
Fusion Processor
Controller
Underwater Camera Pair(s)
Cameras
R
L
Pixel Data
Special Purpose Image Processor (Datacubes)
Pointing and Convergence Angles
Assignments
Fusion Processor (ASAP Based)
3-D WORLD MODEL
I/F Level 2
MMUL LIDAR
Sparse Range Data
2-D Arrays Sonar/LIDAR
2-D Array Sonar Multielemental Array e.g. TOAS
Sensor Cluster Processor (68000 Based)
Assignments
I/F Level 1
Up to 4000 Sensors/ Elements
Single-Beam Whisker Sonars (Include Depth and Altitude Sonars)
SB Sonar
SB Sonar
AZ
EL
0 0 0 0
0 9 8 0
0 9 8 0
0 0 8 0
Sparse Range Maps

Figure 3. Autonomous Sensing System Architecture

utilize sensor data and to reconstruct the three dimensional spatial occupancy of the real world around the submersible. The volume of space represented by the world model is defined in terms of the submersible's position, so that the monitored volume always surrounds the submersible. Therefore, the total storage required can be fixed, and sensor data that no longer occupies the current volume represented is deleted from the world model. Deleted data can always be resensed, if the region it represents is revisited.

Processing required to generate and maintain this world model is triggered by sensor data, in the form of range, azimuth, and elevation three-tuples, tagged with vehicle position, to represent occupied and empty volumes around the AUV. These processes must be accomplished in real-time, since a moving vehicle cannot afford to continue storing outdated sensor information. Since the processing required is massive, and is characterized by repetitive operations applied to different data, a single instruction multiple data (SIMD) parallel computer architecture is apt for this application. System constraints, such as physical size limitations, data size of the compressed world model, and associated computational load per unit volume sensed, indicate that the Martin Marietta Advanced Systolic Array Processor (ASAP$^{TM}$) is a particularly suitable parallel processor for this application.

## SYSTOLIC ARRAY FOR A FUSION PROCESSOR

The Martin Marietta Advanced Systolic Array Processor (ASAP$^{TM}$) is a SIMD array processing unit, composed of 16 single bit processing cells. The cells are arranged in a 4 x 4 array with the processors physically connected to form a mesh. Each processor in the mesh can communicate directly with its north, south, east, and west neighbors, and multiple ASAP$^{TM}$ chips can be configured to form larger arrays.

The ASAP$^{TM}$ is a micro-coded machine with programmable micro-code control for the array of PEs and the input/output modules. The on-chip local memory in each PE provides the high speed random access data storage required by the application. Each processor element can directly access additional RAM space, as needed. Also, the micro-code word size is ample to provide the opportunity for low level parallelism within a processing element itself.

## ALGORITHM OVERVIEW

The data store in the Fusion Processor is distributed and stored among the PEs. The sensor data ingest process receives sensor data, then, with vehicle position and orientation information, it is converted into a set of empty and occupied volumes corresponding to the vehicle's frame of reference. The result of this process is a sensor data store that represents three dimensional spatial occupancy, as perceived through the sensors from the vehicle's frame of reference.

When the vehicle's current position and orientation indicate the need to motion compensate, the following algorithm runs on each processor of the SIMD machine. This results in an updated sensor data store, in which all sensor data is relative to the vehicle's frame of reference.

```
  for each occupancy state value per processor
      old_position <-- occupancy state position in world
                          model
      new_position <-- 3-D coordinate transformation
                          on old_position
      if (new_position is within world model)
*         then occupancy state @new_position <--
                    occupancy state @old_position
          delete occupancy state @old_position
      endif
  endfor
```

Since the occupancy states are distributed among the PEs, step * may require interprocessor communication. The processing required is directly proportional to the distance between the processors, which is in terms of the processor mesh interconnect. Processor array sizes ranging from (16 x 16) to (32 x 32) accommodate the application adequately, and result in tolerable interprocessor communication overheads. Larger array sizes correspondingly have increasing communication overheads rendering the application non-real-time; these overheads can be alleviated by a hypercube interprocessor interconnect, such as that found on the Connection Machine (Ref. 6).

## INTELLIGENT SENSING

In a mobile robot, such as an AUV, the information directly acquired by the sensing system rarely has an adequate representation of cognitive relationships defining the contextual structure within the environment, due to the impracticality of storage requirements. To reconstruct an adequate representation of the world surrounding the vehicle, both high and low resolution sensor data need to be judiciously analyzed, so that the related cognitive overhead is alleviated by efficient cueing and localizing of potential obstacles. This cueing efficiency is a function of the toggling capability between the high and low resolution sensors, afforded by the Fusion Processor (see section 3 & 4). Due to the incorporation of multiple sensors to achieve such cueing efficiencies, there is a resultant overwhelming quantity of sensor data that must be processed before it can be made useful for maneuvering commands.

Two types of intelligent sensing applications that are currently being investigated are Obstacle Avoidance and Object Identification, with an attempt to eliminate the need for involved scene analyses of all obstacles within a field of view (see Figure 4). Obstacle Avoidance is the process of scanning the surrounding environment for any path obstruction without shape descrimination, and determining the best route around the obstructions to arrive at the final endpoint. Such Obstacle Avoidance functionalities can be achieved primarily by low resolution sonar sensors, and if necessary tertiarily by high resolution Lidar sensors. Object Identification is the process by which localized obstacles are analyzed for eventual identification, with the help of data from high resolution sensors, such as Lidars, and other apriori data constructs stored in the sensor post-processor memory.

A decision mechanism for prioritizing between Obstacle Avoidance or Object Identification processes is imperative for autonomous sensing; for which Artificial Intelligence is a suitable candidate. Even though the incorporation of cognitive knowledge is essential to these types of decision making processes, there are practical deficiencies in relying solely on Artificial Intelligence concepts for a complete solution to the obstacle avoidance and object identification problem.

Eventually, a "context cueing" performed solely by Artificial Intelligence techniques normally suffers from a severe lack of real-time data that further improves context definition, or suffers from a need to post-process large quantities of data, so that it can be utilized for context analysis; a process that invariably abounds for an autonomous sensing system aboard a mobile robot. For example, in Computer Vision and Image Processing, most of the research has been confined to the use of one type of sensor, such as a camera, for all data input excluding apriori data, and the responsibility for the recognition process depends entirely on the reasoning mechanism used, and not on the efficient management of the data being stored; this has been a serious shortcoming for object identification. This paper explores schemes for more accountable and efficient data management, rather that open-ended Artificial Intelligent methods that could accommodate numerous instances, but also need recoding for every previously unaccountable instance that subsequently needs to be included. Conventional database management also does not apply, since this is an autonomous system with no human in the loop. Consequently, radical data management methods, such as described in Section 3, need to be investigated.

To further explore the issues concerning efficient data management of high resolution data, it is necessary to look at typical data rates. The sensor information is sent from the sensor to the post-processing computer memory at a rate of thirty frames per second. Typically, high resolution sensor data being transmitted comprises a stream of image frames of 512 x 512 pixels each. Looking at Figure 5, storing all the high resolution images at every point in the autonomous vehicle's trajectory is not only unnecessary, but also impractical. The amount of data that would be stored would overflow any onboard computer memory. To eliminate this overflow of data, selected regions in the surrounding environment are chosen as candidates for high resolution sensing, by the triggering derived from the low resolution sensing system. The sensor pointing angles are used to designate the pan and tilt mechanisms of the high resolution sensors, and data acquisition is triggered. From the real-time analysis of the packets of such consecutive frames of data, a condensed symbolic contextual output as a probable "object" is stored in another section of memory, with a tagged association to the low resolution data "obstacle" source in the Fusion Processor. Each time this data analysis occurs, the transmitted high resolution data stored in memory is then condensed to occupy a smaller quantity of memory in a different area of the disk. Therefore, the disk space that was used for data ingest and analysis can be overwritten with new data. In the interim, most of the disk space can be continually reused with new sensor data. This reusability of limited disk space is imperative, in a mobile autonomous system. The usually involved Computer Vision task is thus rendered as an "intelligent" scratch-pad memory consuming task, and image data archival overheads are considerably reduced.

Succinctly stated, for an autonomous system, such as an underwater vehicle, sonar array data is considered low resolution information that is exploited for the representation of a model of the surrounding environment. Based on hand-offs from low resolution information, the appropriate assignment of pointing angles is derived, and the high resolution information can now be acquired through camera or Lidar sensors. Appropriate pointing angles best achieve the robustness of a multisensor system, and achieve adequate data gathering efficiencies for an autonomous sensing system of a mobile robotic vehicle. An Artificial Intelligence-based decision mechanism allocates sensors and enables triggering between the high and low resolution sensors. Additionally, key events that occur randomly during vehicle maneuvers contribute to sensor prioritization, due to the decision mechanisms. Such methods reduce the need for the construction of a global database.

## CONCLUSION

Systolic arrays provide a significant advantage to multisensor

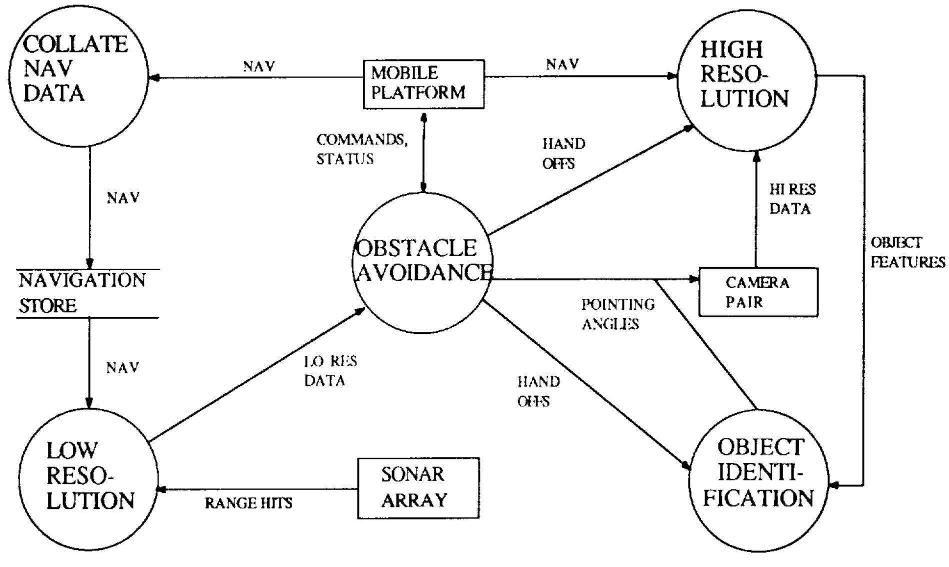


Figure 4. Obstacle Avoidance and Object Identification for Autonomous Vehicles

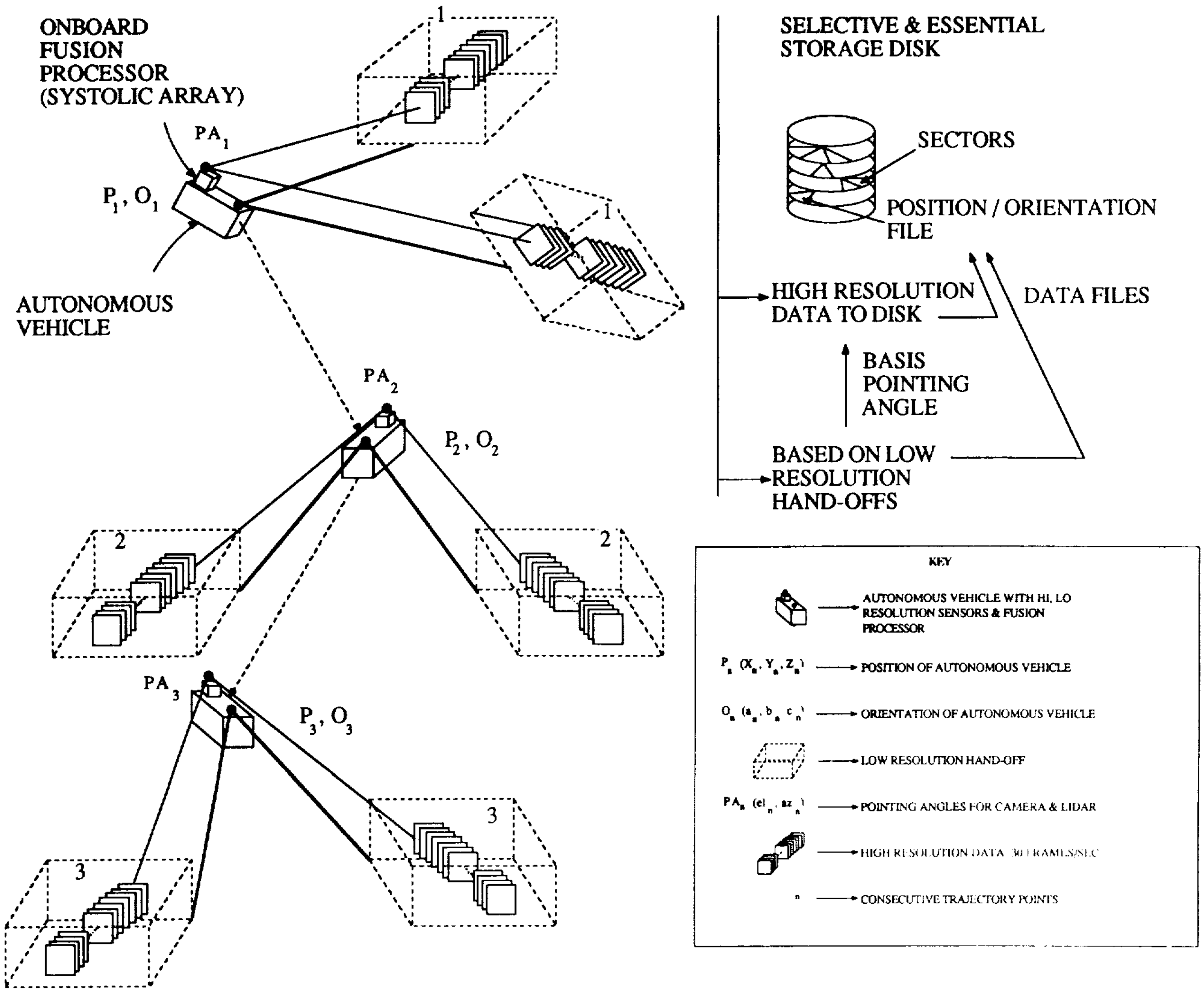

ONBOARD FUSION PROCESSOR (SYSTOLIC ARRAY)
1
PA1
P1, O1
AUTONOMOUS VEHICLE
1
PA2
P2, O2
2
2
PA3
P3, O3
3
3
SELECTIVE & ESSENTIAL STORAGE DISK
SECTORS
POSITION / ORIENTATION FILE
HIGH RESOLUTION DATA TO DISK
DATA FILES
BASIS POINTING ANGLE
BASED ON LOW RESOLUTION HAND-OFFS
KEY
AUTONOMOUS VEHICLE WITH HI, LO RESOLUTION SENSORS & FUSION PROCESSOR
Pn (Xn, Yn, Zn) → POSITION OF AUTONOMOUS VEHICLE
On (an, bn, cn) → ORIENTATION OF AUTONOMOUS VEHICLE
LOW RESOLUTION HAND-OFF
PAn (eln, azn) → POINTING ANGLES FOR CAMERA & LIDAR
HIGH RESOLUTION DATA 30 FRAMES/SEC
n → CONSECUTIVE TRAJECTORY POINTS

Figure 5. Intelligent Sensing

fusion research. Not only do the massive number of processors of an array accomplish repetitive and similar processing for numerous sensing elements simultaneously, but their mesh connectedness also promotes implicit data associativity without explicit software allocations. Eventually, autonomous vehicles will need to be equipped with smart skins, comprising processing-sensor arrays, that feed into a massively parallel data fusion association center. The incorporation of multiple systolic arrays as sensor post-processing firmware will considerably advance the development of sensory interactive systems for autonomous mobile robotics in the near future.

## REFERENCES

1. Chande, A.M., and Noon, K.M., "Obstacle Avoidance and Navigational Sensing for an Autonomous Underwater Vehicle," (*Proceedings of the 1986 SPIE Symposium on Optical and Optoelectronic Engineering*), October, 1986.

2. Datacube, Incorporated, *MAXVIDEO Image Processing Hardware*, 4 Dearborn Road, Peabody, Massachusetts.

3. Draper, C.S., "Control, Navigation, and Guidance," *IEEE Control Systems Magazine*, Vol. 1, No. 4, December, 1981.

4. Haug, A.J. and Graybill, R.B., "The Martin Marietta Bit-Serial Systolic Processor," (*The Second Symposium on the Frontiers of Massively Parallel Computation*), October, 1988.

5. Herr, W. and Collins, K., "MUST - A Large Versatile AUV Technology Testbed System," (*Proceedings of the Underseas Defense Conference*), USD, 1987.

6. Hillis, W.D., "The Connection Machine," *The MIT Press*, Cambridge, MA, 1985.

7. Liddle, D., "TROJAN: Remotely Operated Vehicle," *IEEE Journal of Oceanic Engineering*, Vol. OE-11, No. 3, July, 1986.

8. Noon, K.M. and Chande, A.M., "Sense Analysis Simulator for Sensor Fusion on an Autonomous Underwater Vehicle," (*Eastern Simulation Conference*), April, 1987.

9. Robinson, R.C., "National Defense Application of Autonomous Underwater Vehicles," *IEEE Journal of Oceanic Engineering*, Vol. OE-11, No. 4, October, 1986.

# SECTION II: APPLICATIONS

## Part 2: Poster Presentations

# SIMULATIONS OF CONPOSIT, A SUPRA-CONNECTIONIST ARCHITECTURE FOR COMMONSENSE REASONING

John A. Barnden

Computing Research Laboratory, New Mexico State University
Box 3CRL, Las Cruces, NM 88003.

## ABSTRACT

A computational architecture called "Conposit" is outlined. Conposit manipulates very-short-term complex symbolic data structures of types that are useful in high-level cognitive tasks such as commonsense reasoning, planning, and natural language understanding. Conposit can be straightforwardly implemented as a large neural/connectionist network, and therefore provides a way of bridging the gap between high-level cognitive information processing and neural networks. Conposit's data structures are, essentially, temporary configurations of symbol occurrences in a 2D array of registers. Each register is implementable as a neural subnetwork whose activation pattern realizes the symbol occurrence. The data structures are manipulated by condition-action rules that are realizable as further neural subnetworks attached to the array. In simulations, Conposit has performed symbolic processing of types previously found difficult for connectionist/neural networks. This paper concentrates on a version of Conposit, simulated on the Massively Parallel Processor, embodying core aspects of Johnson-Laird's mental model theory of human syllogistic reasoning. This version illustrates Conposit's power and flexibility, which arises from unusual data-structure encoding techniques called "Relative-Position Encoding" and "Pattern-Similarity Association"

Keywords: Cognitive Modeling, Commonsense Reasoning, Connectionism, Neural Network, Knowledge Representation, Syllogism, Mental Model.

## INTRODUCTION

The challenge presented to connectionism by high-level cognitive processing — which includes commonsense reasoning, planning, and some aspects of natural language understanding — is gaining increasing recognition. The main technical difficulties are listed in Refs. 1-4, 7, 8, and elsewhere in the connectionist literature, and include the well-known variable-binding problem and the problem of accounting for complex, temporary, novel data structures.

Ref. 6 reports experiments with a version of Conposit that incorporates production rules for commonsense reasoning, one of which can be paraphrased as

*IF: a person X loves a person Y who*
*loves a person Z (different from X)*
*THEN: X is jealous of Z.*

This exercises Conposit's handling of variable bindings.

The version of Conposit described below engages in a particular type of commonsense reasoning, namely syllogistic reasoning, by embodying some core aspects of the Johnson-Laird's "mental model" theory (Refs. 9-12). The main goal of the work was to verify that the techniques developed for other types of processing in Conposit (Refs. 4-7) were flexible enough to be extended in a natural way to the distinctly different type of processing required by the mental model theory — and in fact no new features have had to be added.

Conposit is currently concerned only with short-term processing: there is no adaptive learning capability at present, and long-term memory consists entirely of the fixed set of condition-action rules (but see the suggestions in Refs. 4,7 for a long-term memory of data structures). It is closer to the "localist" than to the "distributed" end of the spectrum of connectionist systems.

## BRIEF SKETCH OF CONPOSIT

Conposit is currently defined as a computational architecture whose components can be straightforwardly implemented in connectionist terms. Details are reported in Ref. 7 (or Ref. 4 for an earlier formulation).

In Conposit, a *"Relative-Position Encoding"* technique is used as the foundation for complex *short-term* data structures. These reside in a 32×32 array of *registers*. This array is called the *configuration matrix* (CM). The values in registers are usually rapidly changing. Each register can be implemented as a small connectionist subnet that holds a dynamically changing activity pattern implementing the register's value, and that is connected to neighboring registers and other components.

A register's value consists of a "symbol" and a vector of binary "highlighting flag" values. A symbol may have a specific representational function, such as denoting a particular person or a particular type of relationship among people. Any symbol can be placed in any register, and all registers

CH2649-2/89/0000/0311$01.00 © 1988 IEEE

have the same set of highlighting flags. *Temporary structure is encoded mainly in the adjacency relationships among values in CM registers.* For instance, if a register contains a symbol denoting the class of all possible situations in which one person loves another, and has a certain highlighting flag in the ON state, then any adjacent register that has another specific highlighting flag ON is deemed to represent, temporarily, a specific loving situation.

See, for example, the representation of the proposition that John loves Mary in the upper portion of Figure 1, which shows an 8 × 8 region of the CM.

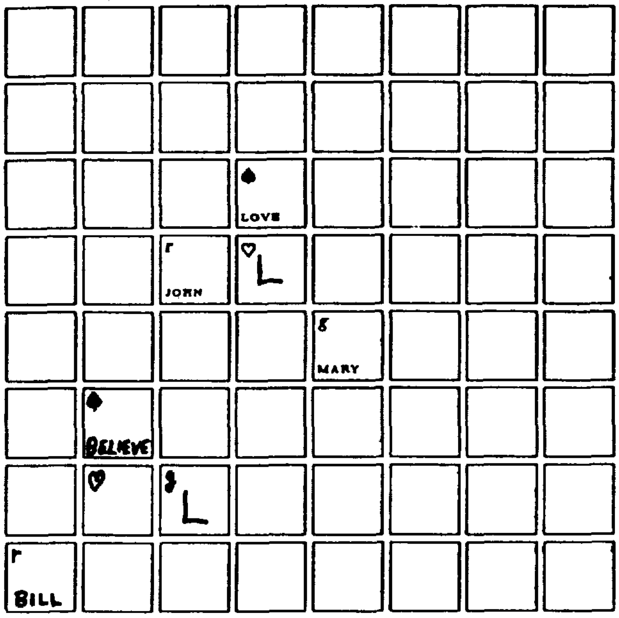


Figure 1. "Bill believes that John loves Mary."

Each square stands for a register, and capitalized words and letters stand for symbols. The word JOHN stands for a symbol denoting a particular person John known to the system. The LOVE symbol denotes the class of all conceivable loving situations. The L symbol may be ignored for now. The registers with no symbol shown contain a null symbol that does not denote anything. The denotations of symbols are considered to be borrowed by the registers they occur in at any moment: a register containing a non-null symbol denotes what that symbol denotes. Hence, in the figure there are registers that — temporarily — denote John, Mary and the love-situations class. The other signs within squares show ON states of highlighting flags, which in this example are all referred to by the names of colors. An 'r' indicates that the register is red-highlighted (i.e. the red flag is currently on); similarly 'g' for green, heart sign for white, and spade sign for black. One important function for highlighting is to help specify the representational relationships temporarily holding between adjacent registers. For instance, a white-highlighted register is deemed to denote a member of the class denoted by any neighboring black-highlighted register. Therefore the upper white register in the figure denotes some love situation. Further, if a register denotes a love situation, then any adjacent red register (here, the one containing JOHN) denotes the "lover" and any adjacent green one (here, the one containing MARY) denotes the "lovee". Note that the *absolute* positions of the symbols and highlighting states are irrelevant, as are the *directions* of the adjacency relationships.

Complex data structures can be split up into pieces by a *shared-symbol association* technique. Shared-symbol association relies on the stipulation that two registers containing the same symbol are considered to represent the same entity. The real power comes from the sharing of variable-like "unassigned symbols". By appearing within a data structure, an unassigned symbol can be viewed as having a temporary denotation dictated by the role of the symbol in the structure. The letter 'L' in Figure 1 indicates an unassigned symbol, which temporarily comes to name the hypothetical loving situation by being in the white-highlighted register in the loving-subconfiguration. The Figure shows how the proposition that *Bill believes that John loves Mary* can be encoded by two separate register-value subconfigurations that are linked by the sharing of the L symbol.

In this shared-symbol association technique, two or more registers contain the *same* symbol, and to that extent contain similar activity patterns at the connectionist level of description. The notion of similarity here is simple and all-or-none (i.e. not graded), but other versions of the technique could be based on more sophisticated, and perhaps graded, notions of similarity of connectionist activity patterns. Shared-symbol association is thus a simple instance of the class of "Pattern-Similarity Association" techniques, which are discussed briefly in Ref. 6.

The processing of the short-term data structures in the CM is performed by internal "circuitry" (i.e. system components that are mapped straightforwardly into a connectionist implementation) mediating mainly neighbor-neighbor interaction within the CM, and external "circuitry" outside the CM but attached to it. The external circuitry embodies "hardwired" *condition-action processing rules.* Rules can detect particular configurations of symbols and highlighting states in the CM by means of highly parallel detection circuitry that involves further two-dimensional register arrays isomorphic to the CM (Refs. 4,7), and can in response send complex sequences of signals to the CM. A rule can embody conditionals testing the CM state, loops, and a simple form of non-recursive routine calling. A rule operates on the CM in a highly SIMD, register-local, parallel fashion: each action on the CM is performed by sending to each register an identical "command signal" in parallel, whereupon different registers change state differently, according to their own current states and those of their immediate neighbors.

A command signal can have one of a number of effects, such as making each register that has specified highlighting flags ON or OFF change the states of some flags, and/or accept a new symbol value, and/or broadcast its symbol value to the other registers (via a central relay station attached to the CM and called the Parallel Distributor). It is also possible for a signal only to have an effect on a single, randomly chosen register with specified highlighting, rather than on each such register. A command signal may also require that, for a register to respond, either some or all of its neighboring registers be in a specified highlighting state. Refs. 4, 7 detail how the signals can be used to process data structures, and, in particular, to find free space for, and then create, new data structures in the CM.

A tentative mapping of the model to connectionist networks that appear to be biologically reasonable is sketched in Refs. 4, 7. In particular, it is suggested that the CM could be realized as a localized group of thin cortical columns. It is this suggestion that motivates the choice of dimension two and size 32×32 for the CM (see Ref. 4). A non-biological version of the approach could be based on a CM of other dimensions and sizes.

## JOHNSON-LAIRD AND SYLLOGISMS

Consider the syllogism

**Some chemists are beekeepers.**
**All beekeepers are householders.**
**Therefore, some chemists are householders.**

To simplify a little, Johnson-Laird maintains that we make such a syllogistic inference by constructing a mental model of the form illustrated in Figure 2.

```
C     =     B     =     H

C     =     B     =     H

(C)         (B)   =     (H)

(C)                     (H)

                        (H)
```

Figure 2. A Johnson-Laird syllogistic mental model.

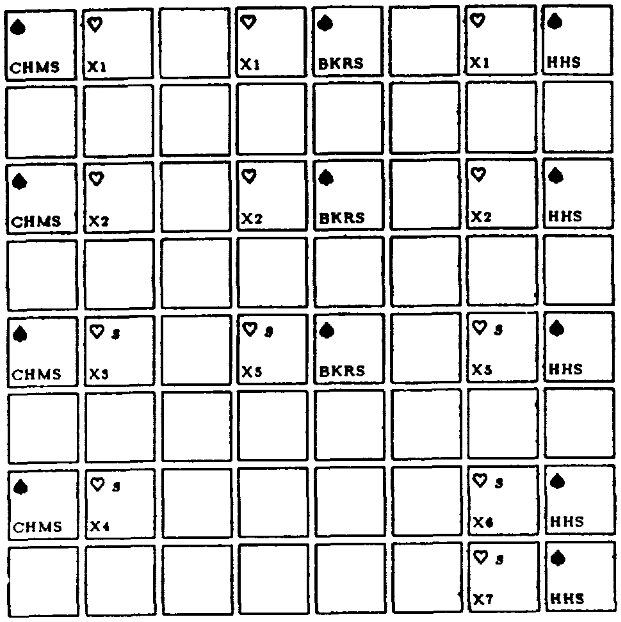
This mental model is an abstract data structure made up of "tokens" (shown by the capital letters) and identity links between tokens (shown by the equality signs). There is an arbitrarily selected number of tokens C standing for chemists. An arbitrarily selected proper non-empty subset are related by identity links to beekeeper tokens B, and all beekeeper tokens are so linked to householder tokens H. The parentheses in the figure indicate that the enclosed tokens are optional. The conclusion that some chemists are householders arises from noticing that some chemist tokens are linked by chains of equality tokens to householder tokens. There is much arbitrariness in the construction of a mental model. For instance, the number of tokens in a particular model is arbitrarily chosen, as is the number marked as optional. There is also leeway in how the links are placed. The mental model serves as a sort of internalized, highly abstract "example" situation conforming to the premises of the English syllogism. Naturally, the "conclusion" read off from a mental model might merely be an artifact of the particular example it embodies, and therefore be invalid. In response to this, Johnson-Laird postulates that the system attempts to construct several different mental models conforming to the premises in an attempt to falsify any particular putative conclusion before outputing it. The attempted-falsification process will fail in the present case, but should succeed if in the above syllogism contained "some beekeepers" rather than "all beekepers".

Johnson-Laird's theory is able to explain certain syllogistic preferences, difficulties and errors exhibited by human subjects. He does not specify any implementation of mental models in neural net terms.

## JOHNSON-LAIRD SYLLOGISTIC REASONING IN CONPOSIT

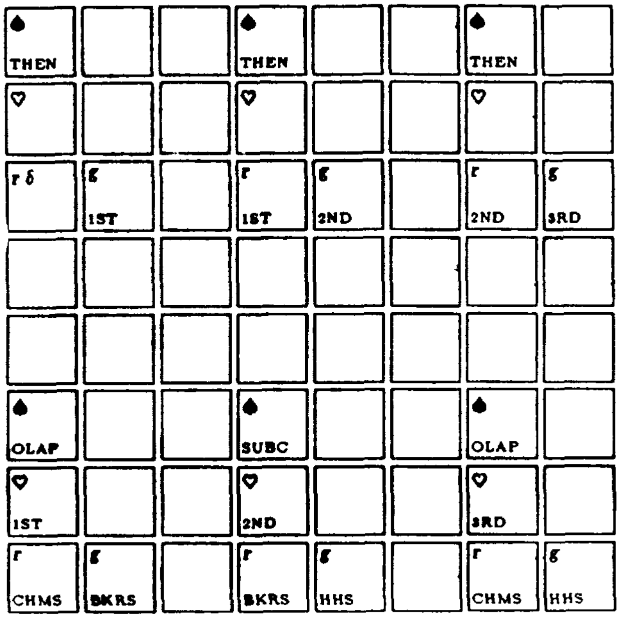
Conposit straightforwardly represents mental models, and constructs them from propositional CM subconfigurations that encode syllogism premises. I have not yet addressed the following aspects of Johnson-Laird's approach: (i) the understanding or generation of natural language; (ii) a thorough attempted-falsification process — the current Conposit is given the conclusion, and merely checks its validity with a single model randomly generated from the premises; or (iii) negative premises and conclusions ("no X are Y" and "some X are not Y"), which require special representational and processing features. The correction of the last two deficiencies is not difficult, however, and will be described elsewhere.

Figure 3 shows the CM version of a syllogistic model derived from the premises in the Section 3 syllogism. The CHMS, BKRS and HHS symbols denote the classes of all conceivable chemists, beekeepers and householders respectively. The X1 to X7 are distinct unassigned symbols. Each Johnson-Laird "person token" is implemented as a pair of adjacent CM registers, one of which (the black one) temporarily represents a class of person, and the other of which (the white one, containing an Xi symbol) represents a particular though indefinite member of the class. (Recall the use of white/black adjacent highlighting in the love-situation representation in Figure 1). Each Xi symbol is thereby considered to denote a person for the time being. The Figure shows the person tokens positioned in a regimented way, but in the actual simulation they are randomly positioned in the CM, and the white-highlighted register in each pair is a random neighbor of the black register. The function of Johnson-Laird's identity links is taken over by symbol-sharing, which is therefore being used for its *standard* function of making different CM registers represent the same thing. In the figure an 's' indicates special highlighting signifying that the token is optional.



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| ♠<br>CHMS | ♡<br>X1 | | ♡<br>X1 | ♠<br>BKRS | | ♡<br>X1 | ♠<br>HHS |
| | | | | | | | |
| ♠<br>CHMS | ♡<br>X2 | | ♡<br>X2 | ♠<br>BKRS | | ♡<br>X2 | ♠<br>HHS |
| | | | | | | | |
| ♠<br>CHMS | ♡ s<br>X3 | | ♡ s<br>X5 | ♠<br>BKRS | | ♡ s<br>X5 | ♠<br>HHS |
| | | | | | | | |
| ♠<br>CHMS | ♡ s<br>X4 | | | | | ♡ s<br>X6 | ♠<br>HHS |
| | | | | | | ♡ s<br>X7 | ♠<br>HHS |

Figure 3. CM version of a syllogistic  mental model.



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| ♠<br>THEN | | | ♠<br>THEN | | | ♠<br>THEN | |
| ♡ | | | ♡ | | | ♡ | |
| r δ | g<br>1ST | | r<br>1ST | g<br>2ND | | r<br>2ND | g<br>3RD |
| | | | | | | | |
| | | | | | | | |
| ♠<br>OLAP | | | ♠<br>SUBC | | | ♠<br>OLAP | |
| ♡<br>1ST | | | ♡<br>2ND | | | ♡<br>3RD | |
| r<br>CHMS | g<br>BKRS | | r<br>BKRS | g<br>HHS | | r<br>CHMS | g<br>HHS |

Figure 4. The statement of the syllogism displayed in the text.

The mental model in Figure 3 is constructed from representations, analogous to the one for John loving Mary in Figure 1, of the two premises of the syllogism. These premise representations are shown at the bottom left and bottom middle of Figure 4. Conposit is also given a propositional representation for the syllogism's conclusion (bottom right of Figure 4) and checks that the mental model is consistent with this given conclusion. The OLAP and SUBC symbols denote the classes of all conceivable class-overlap situations and subclass situations respectively. The 1ST, 2ND and 3RD symbols are arbitrary, distinct, unassigned symbols. None of these five symbols is dedicated to syllogistic reasoning. The registers containing 1ST denote the situation of chemists overlapping with beekeepers (i.e. of some chemists being beekeepers). The registers containing 2ND and 3RD are analogously interpreted.

The construction of the mental model has two main phases. A hardwired rule called *Rule_Some* detects the subconfiguration for the first premise (Figure 4, botttom left), and constructs, in a another part of the CM, the chemist and beekeeper tokens in Figure 3. (It creates randomly many chemist tokens, six on average, then constructs beekeeper tokens using the same unassigned symbols as in a random subset of the chemist tokens, and, finally, randomly constructs three extra beekeeper tokens on average.) Another, similar, rule called *Rule_All* detects the subconfiguration for the second premise and constructs some householder tokens with the same unassigned symbols as in the beekeeper tokens, and then constructs some extra householder tokens. Finally, *Rule_Some* comes into play again by detecting the subconfiguration for the conclusion (bottom right of illustration) and checking that there is at least one chemist token and householder token sharing a symbol. In cases where the conclusion is invalid (such as in the amended Section 3 example), Conposit sometimes does and sometimes does  not construct a mental model consistent with the conclusion, because of the  randomness. It would be trivial to get Conposit to repeat the whole process in an attempt to randomly alight on a falsifying model.

*Rule_Some* and *Rule_All* work with *any* classes in syllogisms, not just the chemist, beekeeper and householder classes. There is no replication of rule circuitry for the different classes. Achievement of this effect in a more standard type of connectionist system would cause considerable difficulty.

Ref. 4 describes versions of *Rule_Some* and *Rule_All* in complete detail. That paper also describes the rule *Note_Next* that fires three times, once in response to each of the propositional CM subconfigurations at the top of Figure 4. These state the order in which the premise subconfigurations are to be considered. (The THEN symbol denotes the class of all conceivable succession situations.) *Note_Next* moves highlighting of two special sorts around in the CM with the result that *Rule_Some* and *Rule_All* are triggered in the right order.

*Rule_Some* checks the conclusion in our example as follows. It marks all the white registers in chemist and householder tokens with special highlighting flags "member_of_class1" and "member_of_class2" respectively. Part of this marking process is to spread such highlighting to all registers with the same symbol. All that is left to do is to detect the presence of some register marked with both "member_of_class1" and "member_of_class2". We have here a traditional marker passing process, *but working over highly temporary data structures.*

### Simulation Results

Elapsed simulated time depends on values for signal-travel distances, signal-travel speeds, and combinatorial-logic delays (e.g. within CM registers) that are based on broad assumptions about how Conposit could be realized as a biologically reasonable neural net (Ref. 4), rather than just as an abstract connectionist net. The main parameter values are as follows:

| | |
|---|---|
| distance between rule circuitry and the CM: | 50mm |
| long distance transmission speed: | 10mm/ms |
| basic time for register's response to a signal: | 10ms |
| overhead of random register selection: | 5ms |

The values of the last two parameters listed appeal to fast non-spike inter-neural communication in local circuits (see Ref. 4 for a discussion). Notice the long distance of 5 centimeters between CM and rule circuitry. The 10mm/ms value appears to be about the maximum speed for transmisison of neural impulses over long distances in cortex.

The following average timings were observed over one set of twelve experiments conducted (one syllogism per experiment).

| | |
|---|---|
| processing of a whole syllogism | 2526ms |
| detection phase of a rule | 98ms |
| a *Note_Next* execution | 76ms |
| a *Rule_Some/All* execution on first premise | 905ms |
| a *Rule_Some/All* execution on second premise | 602ms |
| a *Rule_Some/All* execution on conclusion | 180ms |

## CONCLUSION

The average syllogism-processing time of about 2.5 seconds seems small enough to be psychologically realistic. It is hard to discern timings for human syllogistic reasoning in Johnson-Laird's experimental reports, partly because of the need for a natural language understanding phase. The experiments all appear to have allowed a time much longer than two and a half seconds. E.g. in the experiments of Ref. 11 subjects were given either ten seconds or as long as they liked. According to figures of Bara (personal communication), the faster human subjects work a simple syllogism in a time comparable to the two and a half seconds needed by Conposit.)

It is probably not biologically plausible for rules like Rule_Some and Rule_All to be *hardwired* as in the current Conposit version, partly because of the difficulty of seeing how the rule circuitry could be developed on the basis of experience. However, the basic processing techniques developed will be central also in more realistic systems in which a high-level production rule such as Rule_Some would itself be a *data structure* in one of a possibly large *set* of CMs (Refs. 4,7). In such systems, which are under investigation, individual rule execution could be *faster* because of faster subconfiguration creation, and because there would be the possibility of massive parallelism among rules in different CMs.

The power and flexibility of Conposit arises from its Relative-Position Encoding and Pattern-Similarity Association techniques for encoding data structures. These techniques are unusual for connectionism, although the are loosely related to methods found elsewhere (see Ref. 6).

## ACKNOWLEDGMENTS

This research is being supported in part by USAF under grant AFOSR-88-0215 and by NASA through the MPP Working Group.

## REFERENCES

1. Barnden, J.A. On association techniques in neural representation schemes. *Procs. 5th Conf. of the Cognitive Science Society*, Rochester, NY, 1983.
2. Barnden, J.A. On short-term information processing in connectionist theories. *Cognition and Brain Theory, 7* (1), 1984.
3. Barnden, J.A. Diagrammatic short-term information processing by neural mechanisms. *Cognition and Brain Theory, 7* (3&4), 1985.
4. Barnden, J.A. Complex cognitive information-processing: a computational architecture with a connectionist implementation. Tech. Rep. 211, Computer Science Dept., Indiana University. 1986.
5. Barnden, J.A. Simulation of an array-based neural net model. In *Proceedings of the First Symposium on the Frontiers of Massively Parallel Scientific Computation.* NASA Conference Publication 2478. 1987.
6. Barnden, J.A. The right of free association: relative-position encoding for connectionist data structures. *Procs. 10th Annual Conf. of the Cognitive Science Soc.* Hillsdale, N.J.: Lawrence Erlbaum, 1988.
7. Barnden, J.A. The power of some unusual connectionist data-structuring techniques. In J.A. Barnden & J.B. Pollack (Eds), *Advances in Connectionist and Neural Computation Theory, Vol. 1*, Norwood, N.J.: Ablex, to appear.
8. Dyer, M.G. Symbolic NeuroEngineering and natural language processing: a multilevel research approach. In J.A. Barnden & J.B. Pollack (Eds), *Advances in Connectionist and Neural Computation Theory, Vol. 1*, Norwood, N.J.: Ablex, to appear.
9. Johnson-Laird, P.N. *Mental models.* Harvard University Press: Cambridge, Mass., 1983.
10. Johnson-Laird, P.N. Reasoning by rule or model? *Procs. 10th Annual Conf. of the Cognitive Science Soc.* Hillsdale, N.J.: Lawrence Erlbaum, 1988.
11. Johnson-Laird, P.N. & Bara, B.G. Syllogistic inference. *Cognition, 16* (1), pp.1-61, 1984.
12. Johnson-Laird, P.N., Oakhill, J. & Bull, D. Children's syllogistic reasoning. *The Quaterly J. of Experimental Psych., 38A*, 35-58, 1986.

# Massively Parallel Computing Applied to the One-Dimensional Bin Packing Problem

Judith O. Berkey

George Mason University
Fairfax, Virginia

## ABSTRACT

This paper presents a parallel bin packing algorithm for packing $N \geq n^2$ pieces in a massively parallel computing environment of $n^2$ processors operating in SIMD mode. A new heuristic approach is developed that improves packing efficiency by a careful structuring of the input. Results are compared with parallel versions of the traditional Next Fit and First Fit packing algorithms.

Keywords: Bin packing, SIMD Algorithms, Parallel Processing, Analysis of Algorithms

## INTRODUCTION

Bin packing is an optimization problem that plays an important role in many combinatorial problems existing in the areas of computer science and operations research (Ref. 3). In this problem, a set of n objects having weights between 0 and 1 is placed in a set of unit bins so as to minimize the number of bins used. The optimal solution for this problem is known to be NP-Hard but a number of heuristics have been developed that find solutions with provable and acceptable bounds (Ref. 2).

This paper presents a parallel algorithm for packing $N \geq n^2$ pieces in a massively parallel computing environment of $n^2$ processors operating in SIMD mode. The new heuristic approach that is developed utilizes a grid topology that structures the input to increase packing efficiency without sorting the data set. The results of implementing this algorithm on a Connection Machine 2 System are compared with the results obtained from parallel versions of two traditional bin packing algorithms.

A parallel algorithm must yield an acceptable solution while maximizing the utilization of the processors and minimizing the total interprocessor communication time (Ref. 4). It is therefore important to choose an appropriate representation for both the pieces and the bins. The algorithm should distribute the pieces to the processors in such a manner that both the movement of pieces between processors and the packing time for the bins is minimized. The grid topology that we use increases the packing efficiency while decreasing the amount of internode communication that is necessary.

## THE GRID PACK ALGORITHM

The algorithm begins by dividing the pieces into two classes. Pieces that are between 0.5 and 1.0 in size are labeled "bin starter" pieces. The remaining pieces, that is, those with size between 0.0 and 0.5, are "bin filler" pieces. The goal of the algorithm is to pack pieces in the bin starter class of size $p_i$ into a bin with capacity as near as possible to $(1-p_i)$. Note that the pieces do not have to be presorted. The division can be done as part of the data input step.

The bin starter pieces are packed first. Observe that since pieces of size greater than 0.5 cannot be combined, each one must be packed in a separate bin. Thus, at this point, the packing is optimal. This packing is done in parallel and the resulting bins are maintained in each processor's local memory. Then the algorithm packs the remaining pieces by matching pairs of bin starter and bin filler pieces whose combined size fills a bin as completely as possible. We want to minimize the amount of empty space remaining in the bin after a packing cycle has been performed. It has been shown that processing the pieces in order of decreasing size will increase the packing efficiency (Ref. 2). Our structuring of the input allows us to do this without actually sorting the input data.

Packing continues in this fashion until all bin filler pieces have been processed. A piece that cannot be packed in any bin in the set of bins resulting from the bin starter packing is labeled as "well-traveled." The

CH2649-2/89/0000/0317$01.00 © 1988 IEEE

well-traveled pieces will be combined using a next fit packing in the final phase of the algorithm. The packing algorithm is presented in Figure 1.

```
Algorithm Parallel-Pack

begin
   in parallel
      send pieces to processors
      pack bin starter pieces
   Pack-Loop
      begin
         find all bins that can pack bin-filler piece
            then
               pack piece in the smallest cap. bin
            else
               mark piece as well-traveled
         until all pieces have traveled
      end
   if piece is well-traveled
      then
         Next-Fit pack piece
end
```

**Figure 1. Grid Pack Algorithm**

## IMPLEMENTATION

The Grid Pack Algorithm was implemented on a Connection Machine 2 in *Lisp. The input data, consisting of a list of pieces, was divided as described above and placed in a two-dimensional array on the host machine. Each row of the array received the pieces that had sizes within a specified interval. These intervals are determined by an analysis of the piece size distribution.

The packing loop is performed in parallel by all active processors. The size of each bin filler piece is broadcast in turn to all processors containing bin starter pieces. From the set of all processors that can pack the piece, the processor with the maximum self-address is selected to pack it. When all bin filler pieces have been processed, the well-traveled pieces are combined into new bins using an iterative next fit packing procedure.

Figure 2 is an example of the piece input for a set of one hundred input pieces. In our sample data the piece sizes were randomly generated and are evenly distributed between 0 and 1.

Initially, the 50 bin starter pieces were packed into 50 bins. Forty-seven of the bin filler pieces were added to these bins and one new bin was required to pack the three well-traveled pieces for a total of 51 bins. The optimal packing of this piece set required 49 bins.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 04 | 04 | 02 | 02 | 08 | 08 | 04 | 01 | 02 | 08 |
| 12 | 18 | 15 | 18 | 16 | 10 | 10 | 13 | 14 | 18 |
| 21 | 23 | 21 | 26 | 26 | 28 | 21 | 21 | 22 | 24 |
| 31 | 31 | 30 | 31 | 30 | 37 | 32 | 32 | 37 | 37 |
| 41 | 42 | 42 | 40 | 42 | 40 | 41 | 44 | 41 | 43 |
| 58 | 58 | 53 | 55 | 58 | 54 | 56 | 55 | 56 | 54 |
| 60 | 62 | 60 | 63 | 64 | 64 | 65 | 61 | 60 | 61 |
| 77 | 77 | 76 | 77 | 70 | 70 | 70 | 75 | 74 | 75 |
| 80 | 81 | 80 | 87 | 86 | 81 | 86 | 88 | 84 | 86 |
| 96 | 92 | 92 | 95 | 92 | 98 | 93 | 93 | 92 | 93 |

**Figure 2. Input Pieces**

## RESULTS

The algorithm was tested with sets of data of size 100 and 1000. The results presented in Table 1 are the average of 5 runs for a data set of size 1000. It compares the results obtained by the Grid Pack algorithm to those obtained by using parallel versions of the traditional Next Fit and First Fit bin packing heuristics. The Parallel Next Fit and Parallel First Fit algorithms that were used for comparison are adaptations of those presented in Reference 1. When the input was unsorted, or sorted into nondecreasing sequence, the Grid Pack algorithms resulted in a packing that was much better than the packing obtained with either Parallel Next Fit or Parallel First Fit. In the case where the pieces are sorted into nonincreasing order, the packing from Grid Pack was still much better than the Parallel Next Fit packing and was comparable to the Parallel First Fit packing.

| Algorithm | # bins used |
|---|---|
| Grid Pack | 501 |
| Next Fit unsorted | 676 |
| First Fit unsorted | 535 |
| Next Fit increasing | 639 |
| First Fit increasing | 627 |
| Next Fit decreasing | 640 |
| First Fit decreasing | 497 |

**Table 1. Packing Results**

## CONCLUSIONS

Bin packing algorithms can be efficiently implemented in a SIMD processing environment. The use of data partitioning to initialize the packing appears to be a practical method of allocating the packing workload to a set of parallel processors while maintaining the integrity of the packing algorithm.

This study supports our contention that parallelism offers opportunity for improvement in the efficiency of packing algorithms and that the SIMD model of computation with its data parallel programming approach is appropriate for bin packing problems.

## ACKNOWLEDGEMENTS

The author would like to thank the Naval Research Laboratory in Washington, D.C., for providing access to the Connection Machine 2 System.

## REFERENCES

1. Berkey, J.O., and Wang, P.W. "An Initial Study of Some Parallel Bin Packing Algorithms," to be published, 1989.

2. Coffman, E.G., Jr., Garey, M.R., and Johnson, D.S., "Approximation Algorithms for Bin-Packing—An Updated Survey," in *Algorithm Design for Computer System Design,* G. Ausiello, M. Lucertini, P. Serafini, eds., Springer, New York, 1984.

3. Horowitz, Ellis, and Sahni, Sartaj, *Fundamentals of Computer Algorithms,* Computer Science Press, Inc., Rockville, MD, 1978.

4. Quinn, Michael J., *Designing Efficient Algorithms for Parallel Computers,* McGraw-Hill Book Company, New York, 1987.

# SURFACE MODELING ALGORITHM FOR PYRAMID ARCHITECTURES

D. Britton

George Mason University
Trident Systems Inc.
Fairfax, VA

## ABSTRACT

To accurately model a surface, such as a landscape, a high density of three dimensional locations must be found. This data is not always available or easily extracted from a surface. Given a few critical locations on the surface, level of detail (surface density), and a roughness factor the random midpoint displacement algorithm produces a realistic looking approximation of the surface. A parallel version of this algorithm has been developed to take advantage of the similarities between the structure of the algorithm and massively parallel machines with pyramid (quadtree) architectures.

Keywords: Pyramid Architecture, Massively Parallel, Surface Modeling, Random Midpoint Displacement.

## INTRODUCTION

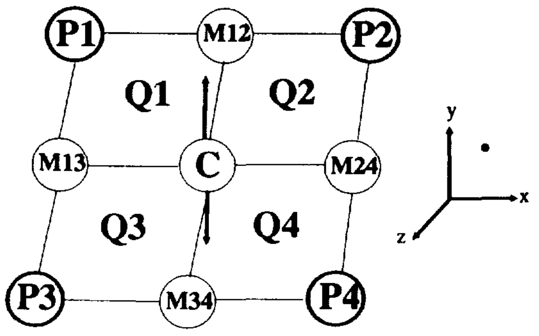
The proposed random fractal surface algorithm, associated with Fournier, Fussell, and Carpenter (Ref. 1), is designed to be used with massively parallel (greater than 1000 processing elements) pyramid architectures in a three dimensional environment. In following sections the general algorithm is explained, followed by an explanation and comparison of the sequential and parallel algorithms.

### The General Algorithm

Given four three dimensional locations forming a convex (in the horizontal plane) polygon, the midpoints along each of the lines forming the polygon and the center of the polygon are found. The center is randomly displaced in the vertical direction with a Gaussian distribution of mean-square variance proportional to the scale, determined by the size of the region and depth of recursion, and factors which characterize of the surface (Ref. 2). Each of the quadrants, formed by the subdivision of the four locations, are subdivided in the same way. This subdivision continues until the specified depth of recursion has been reached. The area subdivision and midpoint displacement are illustrated in Figure 1.

The characteristics of the surface can be specified using the 'H' (roughness) factor and the 'DEPTH' (the depth of recursion/amount of detail). The 'H' factor determines how much each location can be randomly displaced in a vertical direction from its center location between the surrounding points. The 'H' factor, therefore, determines how randomly high or low the surface may rise or dip. Equation 1 defines the vertical displacement at the depth of recursion 'depth' with roughness factor 'H', and 'RND' a gaussian variable with mean = 0 and variance = 1.

$$\text{displacement} = 100 * \text{RND} * e^{(2 * H * \ln(1/(2 * \text{depth})))} \qquad (1)$$



| | |
|---|---|
| Q.......................... | Quadrants |
| M.......................... | Boundary Midpoints |
| C.......................... | Center Midpoint |
| P.......................... | Points on Surface |

Figure 1. Area Subdivision and Midpoint Displacement.

### Single Processor Algorithm

To implement this algorithm on a machine with a single processor, much overhead is involved in the recursion (four calls to itself). In addition, the single processor solution is inefficient for this type of algorithm since it has to perform each calculation for each area subdivision and displacement sequentially. This inefficiency becomes exponentially worse as the depth of recursion increases. The number of iterations of the algorithm on a machine with a single processor grows exponentially with depth.

$$\#\ \text{iterations} = \sum_{k=0}^{n-1} 4^k$$

$$= (4^{\text{depth}} - 1)/3 \qquad (2)$$

where 'depth' is the maximum depth of recursion and 'k' is the curent level.

### The Relationship of the Pyramid Architecture to the Algorithm

The pyramid structure has a direct relationship to the structure of the algorithm. Each processing element has a parent processor (except the one at the apex) and four children processors (except the ones at the base). Since the algorithm calls itself four times within itself, each of the four calls (branches) can be handled simultaneously by the children processors. This match between the pyramid structure and the structure of the algorithm greatly improves the performance.

### The Parallel Algorithm for Pyramid Architectures

A patch (defined by four three dimensional locations on the surface) is entered into the processor at the apex of the pyramid and processed downward towards the base of the structure. During this processing, each processor calculates the midpoints and the center (as described by the general algorithm) of the patch passed from the parent processor on the level above and vertically displaces the midpoint randomly with a mean square variance proportional to the roughness factor and inversely proportional to the current level of detail, thus subdividing its patch into four smaller patches. The processor passes each of the smaller patches to the corresponding child processor on the level below. This process continues until the desired level of detail is reached. The randomness among the processors on a particular level is achieved by performing a random number algorithm simultaneously on a plane of seeds residing in the processors at that level. The number of iterations of this algorithm is

$$\#\ \text{iterations} = \text{depth} \tag{3}$$

where 'depth' is the depth of recursion (level of detail).

However, if the depth of recursion is greater than the number of levels of the pyramid structure, then the equation defining the number of iterations gets complicated. Each time the base of the pyramid structure is reached, the surface at the base must be divided into quadrants, and each quadrant fed back to one level above the base. Using this technique

$$\begin{aligned}\#\ \text{iterations} &= \text{LP} + (\text{depth-LP})^{*} \sum_{k=\text{LP}}^{n-l} 4^{k}/4^{(\text{LP-1})} \\ &= \text{LP} + (\text{depth-LP})^{*}(4^{(\text{depth-LP}+1)}-1)/3 \qquad (4)\end{aligned}$$

where 'n' is | depth/LP |, 'k' is the current level, and 'LP' is the number of levels in the pyramid structure.

### Comparison Between the Sequential and Parallel Algorithms

Since the processors at each level of the pyramid architecture perform the same calculations concurrently on individual patches, a level of detail can be added to the entire surface during each iteration of the parallel algorithm. A sequential machine requires $2^{n}x2^{n}$ iterations at each level of detail for a total of $(4^{n}-1)/3$ iterations for a surface of 'n' levels of detail.

If the depth of recursion is less than or equal to the number of levels in the pyramid structure, then the number of iterations using this parallel algorithm grows directly with depth. If the massively parallel pyramid machine uses a bit serial processing element, it operates on one bit of the numbers at a time. Most single processor machines can work on at least eight bits at a time. However, in comparison to an eight bit machine, the pyramid architecture outperforms it at a depth of recursion of three (which is not very deep) and greater. After this point, the number of iterations increases linearly with a slope of eight, whereas the single processor machine increases exponentially.

However, if the depth is greater than the number of levels, the performance depends upon the difference between the maximum depth of recursion and the number of levels in the structure. The number of iterations using the algorithm on a pyramid architecture grows exponentially with the truncated ratio of the depth to the number of levels. Therefore, as the ratio of the number of levels in the structure to the depth of recursion increases, the efficiency increases. If this ratio is greater than or equal to one, maximum efficiency is achieved. Also, if the number of levels in the structure were only one, the performance of this algorithm would be equal to that of the sequential algorithm, independent of the depth of recursion. Therefore, the depth of recursion must be greater than two and the number of levels must be at least as great as the depth of recursion to take advantage of the parallel algorithm on a massively parallel machine with a pyramid architecture.

### ᴾ.EFERENCES

1. Fournier, A., D. Fussell, and L. Carpenter, "Computer Rendering of Stochastic Models," Communications of the ACM, vol. 25, no. 6, June 1982, pp.371-384.

2. T. Jeffery, "Mimicking Mountains," BYTE, vol. 12, no. 14, December 1987, pp.337-344.

# A HOMOGENEOUS COMPUTATIONAL MODEL FOR SPATIAL INFERENCE ON MASSIVELY-PARALLEL ARCHITECTURES

Mark J. Carlotto
The Analytic Sciences Corporation (TASC)
55 Walkers Brook Dr.
Reading, MA 01867

## ABSTRACT

A computational model for 2-d spatial inference on massively parallel SIMD architectures is described. In the model, spatial information is represented by three basic types of parallel variables or pvars: label maps which assign unique numbers to sets of related processors (e.g., the largest cube address of the set of processors representing a connected region), feature maps which contain the property values of related sets of processors, and hypothesis maps which indicate the probability, membership, belief, etc. that a processor set belongs to a particular class. Spatial inference involves the application of parallel operators to pvars, e.g., labeling operators to assign unique labels to related groups of processors that belong to the same class, spatial operators to compute features of connected regions, and inference operators to assign classes to regions based on their properties. The application of the model to a geographic information retrieval problem is described.

## 1. INTRODUCTION

In geographic information systems, image understanding systems, and other systems that reason about spatial data, a variety of representations are used. Many employ some form of iconic representation (label maps, spatial occupancy arrays, quad trees, etc.) to explicitly delimit the spatial extent of regions in the image space. Iconic representations are usually complemented by some type of non-spatial or symbolic representation such as an attributed graph where the nodes in the graph correspond to regions in the image.The symbolic representation describes properties of and between regions, and provides a place to store hypotheses, and other summary information about the regions. Traditionally, spatial reasoning has been viewed as a process that involves the repeated transfer of information between spatial and symbolic representations. An alternate computational model is described here that is based on a uniform representation for all spatial information (iconic and symbolic) using parallel variables organized in a 2-d grid.

The organization of the paper is as follows. Section 2 summarizes the salient features of the Connection Machine and the *Lisp programming language. Section 3 presents a data-parallel model for spatial reasoning. Section 4 describes some of the operators that have been implemented to date. Application of the model to a geographic information retrieval problem is presented in Section 5.

## 2. THE CONNECTION MACHINE AND *LISP

The Connection Machine (CM) is a data-parallel computing system containing up to 64K physical processors which can act like millions of virtual processors. The CM, originally conceived by Hillis (Ref. 1), is built by Thinking Machines Corporation (TMC). A description of the CM system can be found in Ref, 2. The CM-2 contains 64K bits per physical processor and can perform 32 bit arithmetic at a rate of 2500 MIPs for a 64K system. The current system configuration at TASC is a 8096 processor CM-2 system with a Symbolics front-end processor and a frame buffer that allows the contents of the CM to be viewed at rates up to a gigabit per second.

*Lisp, a parallel dialect of Common Lisp, and PARIS, the assembly language of the CM are provided within the Symbolics software environment. *Lisp (Ref. 3) is based on objects known as parallel variables or pvars which we shall denote in uppercase Greek letters, e.g., A. Elements of pvars are processors that may be accessed by their cube address (i.e., relative to the hypercube) or their grid address, $\alpha(x,y)$. Elements of pvars may be signed and unsigned integers, variable precision floating point numbers, and booleans. The operation (!! $\alpha$ ) returns a pvar in which the value of $\alpha$ has been broadcast to all processors in the currently selected set. Macros such as *when, *cond, and *if select subsets of processors. For example the form (*if (=!! A B) (!! 1) (!! 0)) returns a pvar that contains ones in those elements in which A and B are equal and zeros elsewhere. Functions and macros that operate on all selected processors in parallel are identified by !! suffixes, e.g., (+!! A B). Reducing operations are denoted by a * prefix and return a value from the currently selected set, e.g., (*min A). Relative addressing in the grid is also provided. The form (pref-grid-relative!! A (!! -1) (!! 0)) returns a pvar that is equal to A shifted one position to the left. The reader is referred to Ref. 3 for additional information on *Lisp.

## 3. DATA-PARALLEL MODEL FOR SPATIAL REASONING

Fig. 1 is the proposed computational model for spatial reasoning in 2-d domains that contain objects that may belong to K possible classes. Such a model is appropriate for many image understanding and geographic information processing applications. It involves 1), representing spatial data (label maps, features, and hypotheses) by 2-d pvars and 2), viewing the processes of labeling, segmentation, feature extraction, and spatial inference as data-parallel transformations between pvars.

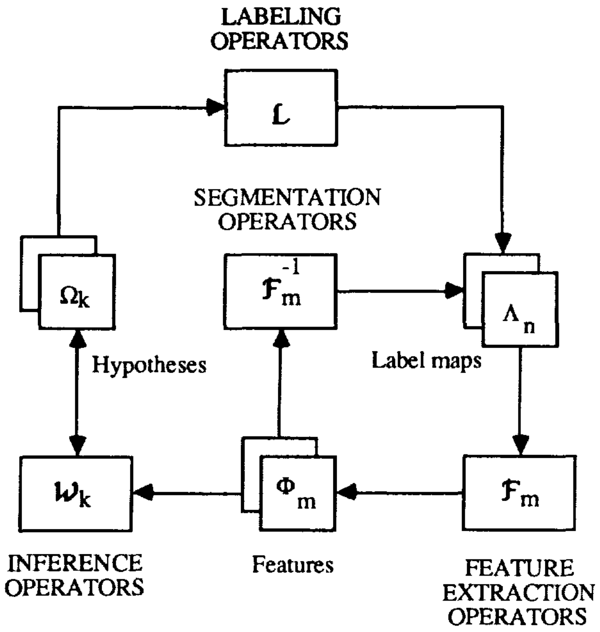


Fig. 1 Computational model

The organization and representation of spatial data using pvars is shown in Fig. 2. Label maps $\Lambda$ are pvars that are used to explicitly delimit the spatial extent of regions (or edges) that may or may not be spatially connected. Labeling operators, $\mathcal{L}(\Omega) \rightarrow \Lambda$ assign unique numbers to sets of related processors (e.g., the largest cube address of the set of processors representing a connected region). Features $\Phi$ are pvars that are used in conjunction with label maps to store properties of regions (e.g., area, distance from) or simply by themselves to store properties derived from other properties (e.g., the local mean of an image). In pvars that describe region properties, the value for each region is replicated in all the processors that belong to the region. Feature extraction operators $\mathcal{F}(\Lambda, \Phi') \rightarrow \Phi$ compute features from label maps and/or other features. Conversely, segmentation operators $\mathcal{F}^{-1}(\Phi) \rightarrow \Lambda$ can be viewed as inverse operations that compute label maps from features. Hypotheses $\Omega$ describe the degree to which regions belong to various classes. Inference operators $\mathcal{W}(\Phi) \rightarrow \Omega$ compute hypothesis maps from features.

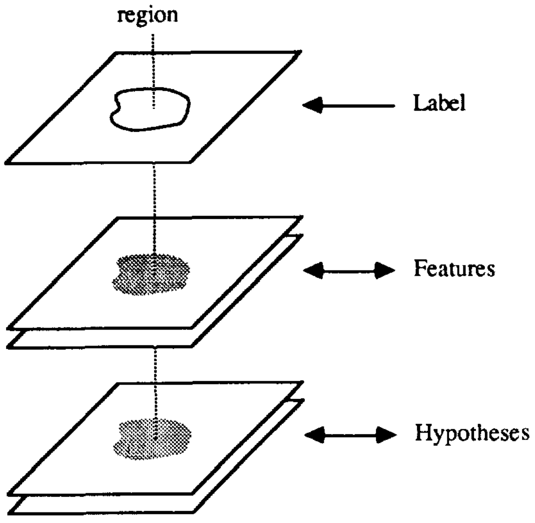


Fig. 2 Organization and representation of spatial data using parallel variables

Labeling and segmentation operators assign unique numbers in the range from one to the number of processors to each region in the image. All of the pixels in a region with label $\lambda$ can then be easily accessed in unit time within forms such as

(*when (=!! $\Lambda$ (!! $\lambda$)) &body).

Feature extraction and inference operators access regions by the label map. For example, the area $\phi$ of a region with label $\lambda$ can be computed as

(*when (=!! $\Lambda$ (!! $\lambda$)) (*set $\Phi$ (!! (*sum (!! 1))))

where the result is broadcast to each processor in the currently selected set. Inference operators can then compute hypotheses in parallel, e.g.,

(*set $\Omega$ (+!! $\Omega$ (abs!! (-!! ($\Phi$ (!! $\phi_0$)))

in an amount of time proportional to the number of features.

To see how the above model applies to spatial reasoning consider the following examples. In a black-and-white image interpretation application, the input $\Phi$ is segmented into homogeneous connected regions by some type of region grower, $\mathcal{F}^{-1}(\Phi) \rightarrow \Lambda$. Various features related to the average brightness, texture, size, and shape of the regions and the spatial relationships between regions are computed from the label map and image, $\mathcal{F}_m(\Lambda, \Phi) \rightarrow \Phi_m$. These features are then evaluated against a set of constraints in order to accumulate evidence for candidate object categories such as buildings, roads, etc., $\mathcal{W}_k(\Phi_m) \rightarrow \Omega_k$.

In a geographic information system application, the input might be a database that provides certain kinds of information (surface material type, soil drainage characteristics, slope, etc.). The objective is to infer other kinds of spatial information, e.g., likely locations for a nuclear waste site based on constraints such as soil drainage characteristics, distance from populated areas or bodies of water, etc. The inputs, represented by a set of spatial occupancy arrays $\{\Omega_k\}$ are labeled $\mathcal{L}(\Omega_k) \rightarrow \Lambda_k$ and are used to compute properties such as the area, compactness, containment, and distance between regions, $\mathcal{F}_m(\Lambda_k) \rightarrow \Phi_{mk}$. A measure of the suitability of various areas $\mathcal{W}_{k'}(\Phi_{mk'}) \rightarrow \Omega_{k'}$ can be used to determine the best places (if any) to put the nuclear waste site. This second type of application is pursued further in Section 5.

## 4. DATA-PARALLEL OPERATORS

This section discusses some basic operators that have been implemented to date for the purpose of developing the ideas introduced in this paper. Additional operators are currently under development and improved algorithms, e.g., based on scanning (Refs. 4 and 5), will be added in the future.

A connected components labeler based on the "brush fire" algorithm was implemented. Initially, each processor in the output pvar $\Lambda$ is assigned its cube address, i.e., a number between one and the number of processors, $N^2$. Then, for all processors whose input pvar $\Omega = \{\omega(x,y)\}$ is non-zero, if $\omega(x,y) = \omega(x+u,y+v)$ where (x+u, y+v) are the addresses of the 4- or 8-nearest neighbors of (x, y), the output is updated as $\lambda(x,y) = \max \{ \lambda(x,y), \lambda(x+u,y+v)\}$. The process is repeated until $\Lambda^{t+1} = \Lambda^t$. The run time is proportional to the size of the largest connected region, which for small connected regions or large highly irregular regions is comparable to the more complex scan-based algorithms described in Refs. 4 and 5. For large regularly shaped regions, significant improvement can be obtained using scanning to propagate the maximum label up, down, left, right, and along diagonal connected segments.

Spatial operators include those that compute properties of individual regions (unary operators) and those that compute information about relationships between two or more regions (n-ary operators). Unary operators that compute geometrical properties such as the area, perimeter, and centroid of connected regions have been implemented using a counting approach. As an example, the area is computed by stepping through each unique label and adding up the number of processors in the currently selected set as described earlier.The complexity is thus of the order of the number of regions. Relational operations such as the minimum distance between two sets of connected regions are performed by computing the distance from any point in one set to all image pixels. The method involves propagating the label with the minimum distance and either the minimum distance or the address of the nearest processor. The minimum distance to each connected region in the other set is obtained by stepping through all labels and executing a *min over the minimum distances within the currently selected set. The complexity is of the order of the number of regions and the size of the largest region.

The set of feature pvars can be viewed as an image of feature vectors. This motivates an inference strategy based on a decision theoretic pattern classification approach. The inference operator implemented computes a similarity measure between a feature value or constraint and a feature pvar, and accumulates the similarity measure across all features. Constraints have the form ($\Phi \lozenge \phi_0$ w) where $\Phi$ is a feature pvar, $\lozenge$ is a parallel version of the standard Common Lisp predicates, $\phi_0$ is a number, and w is a weighting factor. A constraint returns a pvar that contains zeros in those processors that satisfy the predicate and $w \mid \phi(x,y) - \phi_0 \mid$ in the others. The resultant pvar can be added to the results from other constraints to produce a score for the $k^{th}$ class $\Omega_k$. This is accomplished in an amount of time that is proportional to the number of features or constraints. For K classes, the process is repeated for each set of feature prototypes or constraints. The $\{\Omega_k\}$ can then be used as the basis for assigning the "best" class, in some sense, to each region.

## 5. CASE STUDY: GEOGRAPHIC INFORMATION SYSTEM

An example illustrating the application of our model to geographic information processing is shown in Fig. 3. The objective is to find regions that satisfy certain terrain constraints. The area of interest (a) is 512x512 pixels in size and contains the following categories: water, wetlands, coniferous and deciduous trees, bare soil, grass, agriculture, main, and secondary roads. The CM is configured as a 512x512 grid with a virtual/physical processor ratio of 32:1 thus providing up to 2048 bits per processor. First, coniferous and deciduous tree categories are merged and intersected with regions that are not main roads (b) and a label map computed. Information about tree regions such as the area, compactness, and distance between groups of trees can then be computed (c). Information about trees relative to other categories (e.g., distance from, containment, intersection, adjacency, etc.) is determined by marking those categories in working memory (d), computing a label map to uniquely identify each connected region, and applying the appropriate spatial operator, e.g., (e) is the minimum distance to main roads.

The result in (f) shows the five best forested areas given the constraints:

(area > 10000 0.5)
(compactness > 0.05 0.25)
(distance-from-water-or-wetlands > 5 0.75)
(distance-from-main-roads < 1 1.0)
(distance-from-secondary-roads < 1 0.5)

The result in (f) was obtained by ranking scores and selecting the top five areas (i.e., the five "closest" areas with respect to the decision region defined by the constraints).

## 6. SUMMARY

Massively parallel architectures motivate new approaches to old problems. While parallel processing solutions are almost always faster (they'd better be), in some cases they may even be simpler than those originally developed on serial machines. A homogeneous data-parallel model for 2-d spatial inference has been described that represents spatial information in a uniform manner by parallel variables organized in a 2-d grid. The model is simpler since it relies on a single representation as opposed to the heterogeneous (iconic and symbolic) representations used in more conventional systems. An initial application of the model to a geographic information processing problem was presented.

Future efforts will address other spatial reasoning tasks such as image interpretation and will involve developing additional data-parallel operators. We also plan to investigate the problem of handling images whose size exceeds the maximum number of virtual processors available in a given system.

## REFERENCES

(1) W.D.Hillis, The Connection Machine, MIT Press, Cambridge, MA, 1985.

(2) L.W.Tucker and G.G.Robertson, "Architecture and applications of the Connection Machine," Computer, Vol. 21, No. 8, August, 1988.

(3) *Lisp Reference Manual (Version 4.0), Thinking Machines Corporation, Cambridge MA, 1987.

(4) J.Little, G.E.Blelloch, and T.Cass, "Parallel Algorithms for computer vision on the Connection Machine," Proc. International Conference on Computer Vision, pp 587-591, 1987

(5) G.E.Blelloch, "Scans as primitive parallel operations," Proc. International Conference on Parallel Processing, pp 355-362, August, 1986

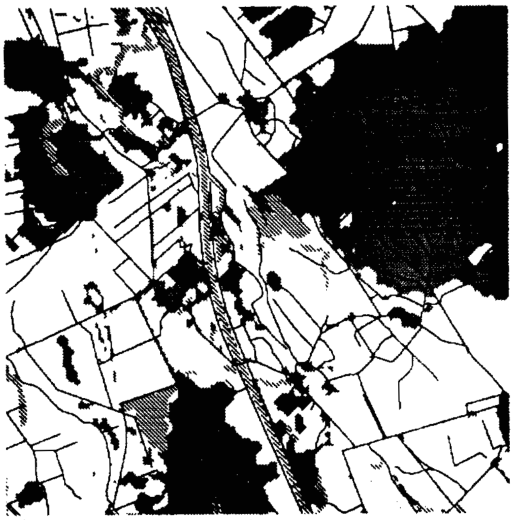


Fig. 3a Thematic map showing trees, water and wetlands, roads, and open areas (bare soil, grass, and agriculture)

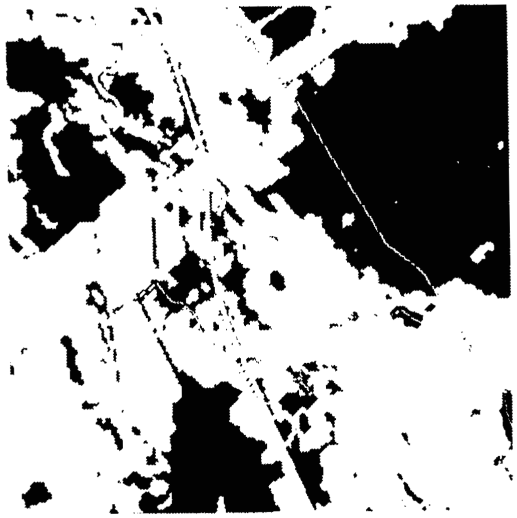


Fig. 3b Forested regions partitioned by main roads

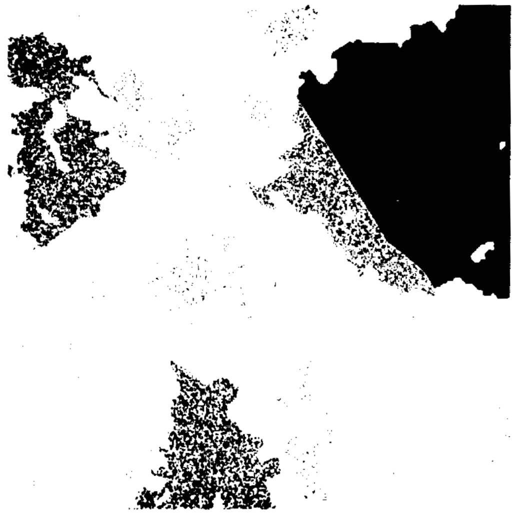


Fig. 3c Halftone rendition of the area of forested regions

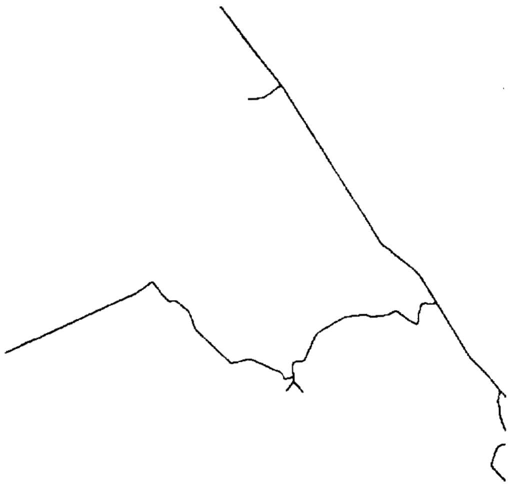


Fig. 3d Main roads

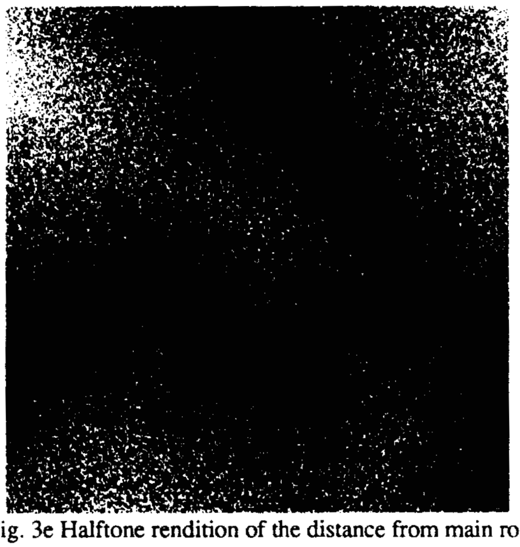


Fig. 3e Halftone rendition of the distance from main roads

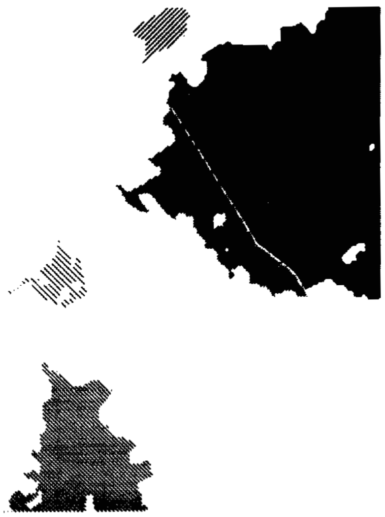


Fig. 3f Five best forested regions for given constraints

# SCAN LINE GRAPHICS GENERATION ON THE MASSIVELY PARALLEL PROCESSOR

John E. Dorband

NASA/Goddard Space Flight Center/635
Greenbelt, MD 20771

## ABSTRACT

This paper describes how we have implemented a scan line graphics generation algorithm on the Massively Parallel Processor (MPP). Pixels are compute in parallel and their results are applied to the Z buffer in large groups. To perform pixel value calculations, facilitate load balancing across the processors and apply the results to the Z buffer efficiently in parallel requires special virtual routing (sort computation[1,2]) techniques developed by the author especially for use on single-instruction multiple-data (SIMD) architectures.

Keywords: Graphics, scanline, Z-buffer, sorting, sort computation, SIMD, massively parallel, MPP, load balancing.

## INTRODUCTION

A scan line graphics generation algorithm basically determines the brightness of pixels in a simulated 3-D scene a scan line at a time. The brightness value of a pixel is based on the surface brightness of simulated polygon which would be seen through the pixel. Triangles are the polygons used here. Only the triangle nearest the pixel on the simulated viewing screen will be seen through the pixel. Therefore a Z buffer is setup to accumulate the values of the closest polygons to the viewing screen. It is actually not necessary to only process one scan line at a time. On the MPP, a subset of triangles at a time are processed for all scan lines that these triangles cover. This is done by projecting each triangle onto the viewing screen and determining which scan lines it covers. Then the pixels of each scan line that the triangle covers is determined. This results in pixels of different values and distances from the viewing screen which are loaded into the Z buffer. When all triangles are processed the Z buffer can be displayed as an image.

To efficiently compute pixel values in parallel an efficient load balancing method was developed so as many processors as possible could be kept busy. This is of importance when greater parallelism can be realize by duplicating data into more processors. This is made complex when it is determined that data in certain processors is of no more computational use, randomly leaving processors without work to do. Therefore the data must be moved in such a way that it is known that when the data is slid to new processors it will not be written over useful data. This movement or compression is done by sorting. Although efficiency of processor usage is of primary interest here, efficiency of data movement is also of importance. Therefore the inefficiencies in the use of sorting are also considered. This has prompted the modification of the sorts used. This involves a preprocessing (scout) step which determines how much of the sort is necessary to provide sufficient contiguous space to duplicate the data. Once this has been determined a sort is used to compress the data which can be terminated early based on the information derived by the scout step. This then allows one the ability to reasonably efficiently keep as many processors as possible busy.

## PROJECTION CALCULATION

The projection calculation converts the three coordinates of the three corners of a triangle in a 3 dimensional viewing space into two coordinates on the viewing screen and a range from the view point. Given the coordinates of the triangle ( $X_1, Y_1, Z_1, X_2, Y_2, Z_2, X_3, Y_3, Z_3$ ), the coordinates of the view point ($X_v, Y_v, Z_v$), and the projected coordinates ( $X'_1, Y'_1, R_1, X'_2, Y'_2, R_2, X'_3, Y'_3, R_3$ ) the following equations do the conversion from 3-D coordinates to 2-D projected coordinates. The first set of equations rotates the triangles in space so that the viewing axis lines up with the Z axis. Thus the view point will lie along the Z axis.

$$X''_i = X_i * x_v + Z_i * z_v , \quad Y''_i = Y_i ,$$

$$Z''_i = X_i * z_v - Z_i * x_v ,$$

$$X''_v = 0, \ Y''_v = Y_v, \text{ and } Z''_v = \sqrt{Y_v^2 + Z_v^2},$$

where $x_v$, $y_v$, and $z_v$ are normalized values of $X_v$, $Y_v$, and $Z_v$.

$$X'''_i = X''_i , \quad Y'''_i = Y''_i * y''_v + Z''_i * z''_v ,$$

$$Z'''_i = Y''_i * z''_v - Z''_i * y''_v ,$$

where $y''_v$ and $z''_v$ are normalized values of $Y''_v$, and $Z''_v$.

Thus the rotated coordinates of a triangle is $X'''_1$, $Y'''_1$, $Z'''_1$, $X'''_2$, $Y'''_2$, $Z'''_2$, $X'''_3$, $Y'''_3$, and $Z'''_3$. The rotated triangles are projected on to the screen which is the distance $R_s$ from the view point. The following equations give the values for X', Y', and R' for each corner of a triangle.

$$R_v=\sqrt{X_v^2+Y_v^2+Z_v^2}\ ,\quad X'=X'''\frac{R_s}{R_v},\quad Y'=Y'''\frac{R_s}{R_v},$$

$$\text{and}\quad R'=\sqrt{(X'''-X_v)^2+(Y'''-Y_v)^2+(Z'''-Z_v)^2}\ .$$

A brightness value (B) is also calculated for each triangle. The actual means of calculating it is not important, only that it exists and must be included with the rest of the information for each triangle.

## SCAN LINE DETERMINATION

Once the projection calculations have been performed each triangle will be described by an X and Y coordinate and a range for each corner and a brightness for the entire triangle. This information will make up a triangle description record. These records will be duplicated so that there exists one copy of a triangle's description record for each scan line that intersects the triangle's projection onto the screen.

Assume that scan lines are parallel to the X axis. Then the corners of a triangle with the largest and smallest Y values define the range of scan lines that the triangle intersects. By recursively dividing this range in half and making records corresponding to the two halves, we will eventually have a record for each scan line in the range. The difficulty arises when this has to be done in parallel, especially when it is done on a large array of processors, like the MPP. The number of scan lines that a triangle overlaps is not the same for all triangles. This means that the rate of creation of new records is uneven across the processors and some sort of load balancing must be performed if one is to efficiently utilize large arrays of processors.

## LOAD BALANCING

Load balancing consists of redistributing records across the processors when some processors contain more than one record. This is caused by creating more records in one area of the array of processors than in others. One can do this by moving all the records to one end of the array of processors, only one record per processor. Any left over records, if all processors have at least one record, can be saved in a stack. There several means by which the records can be moved(compressed) to one end of the array, but we have found that parallel bitonic sort is very efficient at doing this on the MPP. So the use of sort to load balance is what will be discussed here.

Actually the records are sorted to one end of the array so that there are two records per processor. Therefore if only half of the processors have any record in them, then half must have none. The final step of the load balancing is to move one record from each processor that has two to a processor that has none by sliding them halfway across the array. This means that a complete sort has to be done and the data moved halfway across the array. Though the sort is efficient, there is no sense in doing a complete one if one doesn't have to.

Therefore, a scouting step was developed to determine how much of the sort needs to be performed so that records can be simply moved to empty processors. Simply implies moving one record form each processor that has two to a processor that has none by moving them all the same number of processors away form their original processor.

For an incomplete sort to be useful at least the following condition must be true. That for every group of processors, at least half of the processors must be empty. These groups must contain the same number of processors and all records within each group must be compressed to the same side of the array of processors of the group. The scout routine determines the shortest sort necessary to meet these conditions by performing a sort on a set of flags that represent where the records exist within the array. The difference from the sort being that after every merge step it checks to see if the required conditions have been meet.

Scan line determination is merely duplication of records, modification so that they represent different ranges of scan lines, and redistribution of records (load balancing). This is repeated until each record represents only on scan line.

## PIXEL DETERMINATION

At this point each record represents a triangle and one scan line that it intersects. The range along the scan line which represents the part of the scan line that is covered by the triangle is determined. Then in the same way that scan lines ranges were reduced to individual scan lines, so pixel ranges are reduced to individual pixels. Analogous to scan line determination, pixel determination involves duplication of records, modification so that they represent different ranges of pixels, and redistribution of records (load balancing). Thus, each record will represent a triangle and a pixel that it covers. From each of these records a pixel record is created that contains the pixels location on the screen, the bright of the triangle, and the distance to the triangle as seen through the pixel.

## Z BUFFERING

Many of the pixel records will represent the same pixel, but with different range and brightness and range values. The Z buffer is merely a collection of the records for which duplicate pixel records are eliminated. They are eliminated based on there range value. Only the pixel record with the smallest range is kept for each pixel. This is done using a sort computation function, sort minimum, which will flag the minimum range record for each pixel during the sort. All unflagged records can be mark as deleted.

## IMAGE ASSEMBLY

The records in the Z buffer are then used to form a final image. Techniques for assembling data points into an image were developed previously in the process of developing algorithms of point plotting and raytracing on the MPP[3].

Since there may not be a Z buffer record for every pixel in the image, a template image must be created. This consists of a group of pixel records that contain a record for every pixel in the image. Image assembly is a two step operation, pixel value distribution and image organization. Both of these operation can be done with sort computation functions. Pixel value distribution is done with sort distribution. Z buffer records are flagged as containing valid data and image template records are not. Sort distribution copies data from Z buffer records to image template records. This however leaves Z buffer records interspersed with image template records. Thus the image can not be displayed in this form as is. Since image records are flagged as belonging to the image template and Z buffer records are not, the records can be sort with the image flag as the major key. This will separate the Z buffer records from the template records. At the same time the pixel location can be used as the minor key, which will order the pixels so that they can be displayed as a raster scan image.

## CONCLUSION

This technique is in use on the MPP, which is a 2-D grid of 128 by 128 processors. We are generating 3-D renderings of elevation data. The data consists of a 512 by 512 grid of points which is converted into 524,288 triangles (see Color Plate II, p. 694). These triangles take from 45 seconds to 75 seconds to render, which is from 6 to 12 thousand triangles a second. Currently we are working on more efficient means of data movement and organization to increase its speed.

## REFERENCES

1 Dorband, John E., *Sort Computation*, Frontiers 88 Conference Proceedings, September 1988.

2 Dorband, John E., *Sort Computation and Conservative Image Registration*, Ph.D. thesis, Pennsylvania State Univ., December 1985.

3 Dorband, John E., *3-D Graphic Generation on the MPP*, Proceedings of the 2nd International Conference on Supercomputing, Vol. II, pg 305-309, 1987.

# FINGERPRINT IDENTIFICATION ON A MASSIVELY PARALLEL ARCHITECTURE

George Mason University

Gowrishankar T. R.

## ABSTRACT

A feature-based recognition scheme for fingerprint identification on a massively parallel system is presented. The algorithm provides an orientation-independent recognition system that utilizes the features offered by massively parallel architectures. Implementation of this algorithm on the GAM I pyramid to extract the different features is discussed. The use of adder pyramid that is incorporated in GAM I architecture in determining the Euler count that is used to recognize the loops in the pattern is also highlighted. The results of identification of simulated patterns on GAM I pyramid is presented.

Keywords: GAM I pyramid, Feature vector, Euler count, adder pyramid, isolated point, loop, merge point, termination, expand, mask.

## INTRODUCTION

The increasing volume of fingerprints collected and the need for faster identification of patterns have generated a great deal of interest in improving the existing automatic fingerprint identification systems.

In (Ref. 1-2), a set of fingerprint impressions were partitioned into sampling squares which were preprocessed for feature extraction. A class of context-free languages described the fingerprint patterns and the recognition was accomplished using a sequential parsing technique. The same set of features were used to further classify the fingerprint patterns using a class of stochastic context-free languages. This approach was further improved in (Ref. 2) by the use of tree grammars to identify the features. This system has only provided a first step in automation of fingerprint identification.

The advent of massively parallel architectures (architectures with 1,000 or more processors) has provided a whole new perspective for image processing. Such highly parallel machines provide an excellent infrastructure for the analysis of highly complex fingerprint patterns. This paper demonstrates the use of massively parallel architectures for the extraction of features that uniquely identify each fingerprint impression.

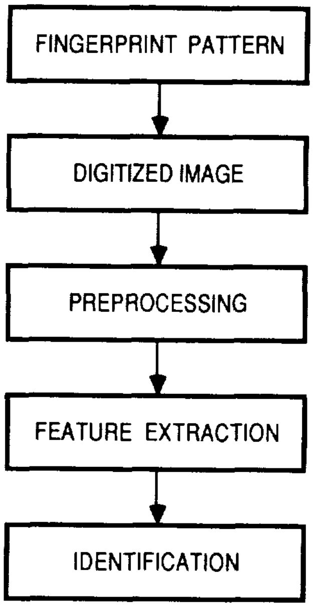


Figure 1. Automatic Fingerprint Identification System

These characteristics or features, referred to as minutiae, are essentially interruptions to the normal flow of ridges (Ref. 2), such as abrupt ridge ending (terminations), dots (isolated points), segments, branches or mergers and loops (i.e., holes). These minutiae (over 100 in each fingerprint) and their relative locations are considered sufficient to identify a given pattern.

The proposed fingerprint recognition system (Figure. 1) utilizes the salient features of GAM I pyramid architecture (Ref. 3), which belongs to the domain of massively parallel systems. The recognition scheme also utilizes the unique adder pyramid which is incorporated on the GAM pyramid to determine such useful parameters as the Euler count of the pattern. The slider component and Sum-OR circuitry of GAM I pyramid architecture provides an added advantage in processing the single-bit wide binary image representing a fingerprint pattern. The SIMD mode of operation of massively parallel architectures facilitate simultaneous identification of all minutiae of a particular type spread throughout the pattern.

CH2649-2/89/0000/0331$01.00 © 1988 IEEE

This SIMD operation can be restricted to a subset of processing elements by employing the '*masking*' feature available on the GAM I pyramid, as on other massively parallel systems. This provides a way to inhibit a selected set of processing elements from executing an instruction.

## THE RECOGNITION SYSTEM

The recognition system requires that the fingerprint image be preprocessed and input to the GAM I pyramid as a digitized single-pixel wide image. Proper representation of the image is necessary for correct extraction of features and accurate identification.

### Pattern preprocessing

The video camera interface of the GAM I pyramid enables the fingerprint pattern to be input directly to the base level of the pyramid. The pattern is then digitized to obtain a binary representation of the image. This image is subjected to preprocessing to get a single-bit wide image. The algorithm was tested on the GAM I pyramid using simulated patterns input directly as single-bit wide binary images using the keyboard.

### Feature vector description:

The identification system is built on a feature-based approach. The numerous ridge lines that run from one end of the pattern to the other do not form the characteristic features of a fingerprint. Only the interruptions to the flow of these ridge lines are considered fit for identification of fingerprint patterns.

This automatic fingerprint identification system considers five features as primary elements of a feature vector that uniquely identify a given fingerprint pattern. These five features are isolated points, terminating points, merge/branch points, segments and loops. These fingerprint primitives, shown in Figure. 2, referred to as minutiae have two characteristics, they form a finite set and their distribution throughout a pattern is so unique that no two patterns are alike (Ref. 2). Each fingerprint has fifty to hundred such minutiae (Ref. 4).

*Isolated points*: These are dots in the original fingerprint patterns. These points appear in the digitized image as pixels in state '*1*' having eight '*0*' state neighbors.

*Terminating points*: These are the points where the ridge lines that run from the fingerprint periphery end abruptly within the pattern. These points appear as pixels with only one neighbor in state '*1*' in the digitized image.

*Merge/branch points*: These are the points where two ridge lines merge into a single ridge line or the points where a single ridge line branches into two. These can be identified as the pixels with neighbors that form one of the junction patterns.

*Segments*: These are ridge lines that start and end within the pattern without touching the periphery of the pattern unlike the terminating ridges that have one of their endpoints at the edge of the pattern.

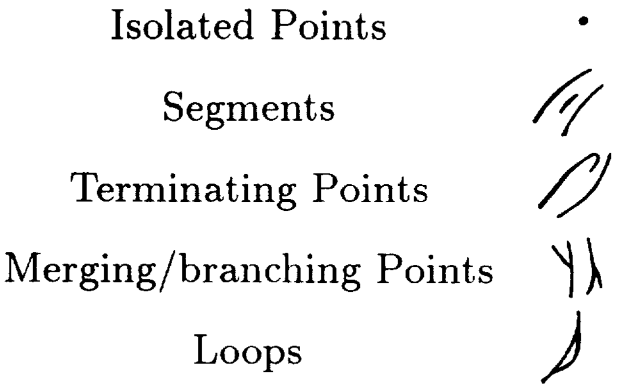


Figure 2. Fingerprint Primitives

*Loops*: As the name indicates, these minutiae are ridge lines that form loops within the pattern with the tangential pixel having three neighboring pixels in state '*1*'.

The bifurcation and termination minutiae are considered to carry high information associated with the identification of a fingerprint (Ref. 4).

### Feature extraction:

As mentioned earlier, the aim of this fingerprint identification system is to form a unique feature vector of minutiae extracted from the fingerprint pattern. In this section, extraction of each of the five features is described.

*Isolated points*: This feature can be identified as those pixels that appear with neighboring elements in all eight directions in '*0*' state. The N, S, E and W slider connections provided on the GAM I pyramid help achieve this with relative ease.

This is achieved by ANDing each pixel of the input image with the inverse image of OR of all neighbors. Since the minutiae are required to be present within the periphery of the pattern, the border elements of the image are inhibited from the operation by '*masking*' the border processing elements.

*Merging/branching points*: Since, the algorithm treats the input image as direction-independent, the merging and branching minutiae are considered as a single feature element. The merging points are identified as those pixels at the junction of three ridge lines. These are recognized by extracting those pixels with two neighbors separated from each other by at least one pixel in '*0*' state. The algorithm accounts for all possible combinations of such occurrences. The algorithm also confirms that the merging pixel does have three neighbors in state '*1*' indicating the merging or branching of three ridge lines.

*Terminating points*: As previously mentioned, these are the endpoints of ridge lines that terminate abruptly within the age for the endpoint patterns. The endpoints are essentially pixels with only one neighboring element in state '*1*'. The determination of merge points, too, require that the border pixels be transparent to the operation.

*Segments*: Segments are identified as the ridge lines connecting two terminating points within the periphery of the image. Thus, the algorithm initiates the segment search procedure with the end point image. The endpoint image is developed along the ridges containing the endpoints. The endpoint image is expanded successively and the seed pixel is excluded from the expanded image. This image is added to the processed image of the previous stage to get a new intermediate image. Alongside, the expanded image is also ANDed with the input image to obtain the seed pixels for further expansion. The procedure is repeated till the intermediate image formed in two successive stages are identical. Number of disconnected regions in the segment image thus formed gives the number of segments in the pattern.

*Loops*: Extraction of this feature utilizes the Euler characteristic number. The Euler number characterizes any polygon and is a function of the number of vertices V, edges E and faces F (Ref. 5). The Euler number C given by

$$C = V - E + F$$

represents the number of objects in the image (such as dots, merge points, segments, etc.) less the number of holes or loops.

The Euler Characteristic equation is applied for this blob counting on GAM I pyramid by defining the different variables as (Ref. 5):

**Vertex** - Each object pixel

**Edge** - A pair of adjacent horizontal or vertical object pixels, as well as the object pixels diagonally adjacent without any horizontal or vertical connections.

**Face** - Any *2x2* object pixel square. V, E and F are calculated by searching the input image for the specified patterns (Ref. 6).

V is calculated by counting the total number of object pixels while E is obtained by calculating the total number of occurrences of the various edge patterns. This can be represented as follows (Ref. 3):

$$E = 1\,1 + \begin{array}{c}1\\1\end{array} + \begin{array}{cc}1&0\\0&1\end{array} + \begin{array}{cc}0&1\\1&0\end{array}$$

where each pattern represents the number of occurrences of that particular pattern and "+" stands for addition. F is obtained by counting the number of patterns of the form:

$$\begin{array}{cc}1&1\\1&1\end{array}$$

The calculation of Euler count can be further simplified as given below:

$$C = \begin{array}{cc}1&0\\0&0\end{array} - \begin{array}{cc}1&1\\1&0\end{array} - \begin{array}{cc}0&1\\1&0\end{array} = \begin{array}{cc}1&0\\0&0\end{array} - \begin{array}{cc}x&1\\1&0\end{array}$$

where "x" stands for *don't care state.*

The number of objects in the image is determined by the sum of all the object pixel patterns discussed thus far, viz., isolated points, merge points, segments, endpoints, border segments (segments that run along the periphery which are not included in the feature vector set), etc. Hence, the Euler characteristic number of the image with the said features masked gives the number of loops in the input image. The adder pyramid on GAM I proves distinctly efficient in the application of Euler counting algorithm to determine the number of loops in the image.

The number of minutiae of each of the five types extracted from an image form the elements of a feature vector associated with the image. This feature vector serves to identify the fingerprint that corresponds to the digitized image.

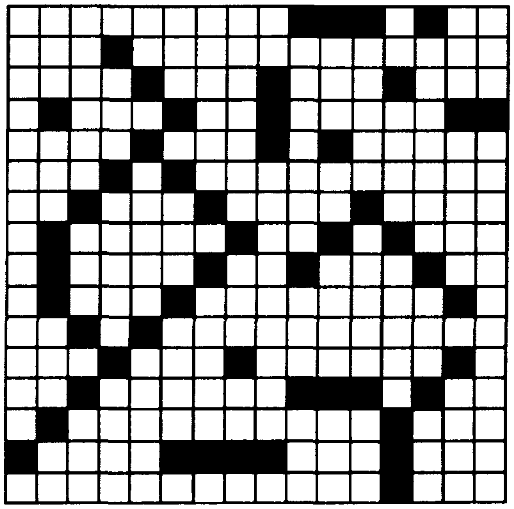


Figure 3. Digitized Input Image

## SIMULATION ON GAM I PYRAMID

The algorithm for feature-extraction in fingerprint patterns just described was tested on GAM I pyramid using simulated images. The programs for implementation of the algorithm were written in Function IV, a structured interpreted functional programming language (Ref. 7). Simulated images (as in Figure. 3) containing all or few of the features discussed thus far were input as binary images using the keyboard input option.

The exclusion of the border feature elements from the feature vector determination was achieved by utilizing the '*masking*' operation. Direct masking of border elements before subjecting the image to the extraction algorithm will introduce additional endpoints and thus will affect the determination of other feature vectors that are dependent on the number of endpoints in the image. Thus the different procedures for extraction of each of the five features were implemented twice. once on the border masked image and once on the unmaskeo image. The difference between the two values thus obtained determined the particular feature element.

The Sum-OR operation was widely used to detect the presence of image pixel(s) at various intermeditate stages of identification. Other salient features of GAM I pyramid such as the slider and other components are used for eight-directional expansion of the image and for shifting the image one pixel position in any of the eight directions. The nearest neighbor communication provides a distinct advantage in processing the image in parallel.

All the features of the simulated images were correctly identified. The different images were so constructed to include 'arious combinations of features and their locations that are realizable in actual fingerprint patterns. The implementation of the algorithm also generates various feature images that contain only the corresponding feature elements present in the input pattern. This will prove useful in analyzing the different features individually as opposed to the pattern in its entiretv

## CONCLUSIONS

A feature-based system for automatic fingerprint identification on massively parallel architectures is presented. The features selected associate every fingerprint pattern with a unique feature vector. The algorithm to extract these features fully utilize the parallel operations offered on the GAM I pyramid. Spatially parallel architectures, such as the GAM I pyramid, with their largely SIMD mode of operation are highly suitable for processing binary images representing fingerprint patterns.

Various simulated images (*16x16*) containing all or few of the features mentioned were used to test the identification system. The present dimensions of base level of GAM I pyramid *16*x*16* limit the algorithm to only simulated images. The successful identification of the features in all simulated images provide a strong motivation for implementing the algorithm on actual fingerprint patterns.

Use of massively parallel systems of dimensions commensurate with the normal size of fingerprints will aid in extracting the spatial parallelism to the fullest extent. A normal fingerprint pattern requires a dimension of 256 x 256 pixels for fine grain representation. Secondary features such as the distance between the center of loops, segment lengths, etc. can be used to improve the performance of the system. Existing serial algorithms use other features like whorls, archs, etc., in addition to the primary features mentioned for more efficient recognition.

The other main advantage of using massively parallel architectures is the parallel input/output capabilities that result in significant increase in the efficiency of I/O handling. This feature is of particular importance for this application due to the high order of input data.

## References

[1] B.Moayer and K.S.Fu, "*A syntactic approach to fignerprint pattern recognition*", Pattern recognition, Vol. 6, Dec. 1974.

[2] B.Moayer and K.S.Fu, "*A tree system approach for fingerprint pattern recognition*", IEEE transactions on Pattern analysis and Machine intelligence, Vol. PAMI-8, No.3, May 1986.

[3] D.H.Schaefer, et.al., "*The MPP pyramid computer*", In L.Uhr (Ed.): *Parallel Computer Vision*, Academic Press, Boston, 1987, pp. 15-42. IEEE International conference on Systems, Man and Cybernetics, Tucson, Arizona, Nov. 1985.

[4] M.Eleccion, "*Automatic fingerprint identification*", IEEE spectrum, Vol. 10, Sept. 1973.

[5] P.Ho, "*Object Identification and segmentation utlizing hierarchical structures of computing elements*", Master's thesis, Department of Electrical and Computer Engineering, George Mason University, Fairfax, VA, 1986.

[6] M.B.Chu, "*Object recognition utilizing combined general purpose and special purpose hierarchical structures*", Master's thesis, Department of Electrical and Computer Engineering, George Mason University, Fairfax, VA, 1987.

[7] J.Boyd, "*Function IV User's manual*", George Mason University, Fairfax, VA, 1987.

# GRAY SCALE ADJUSTMENT ALGORITHMS ON SIMD ARCHITECTURES

Christopher Lee Kuszmaul

MRJ, Inc.

## ABSTRACT

A large number of methods of gray scale adjustment are invoked in image processing. Substantial gains in time performance of such algorithms can be garnered by implementing them on highly parallel single instruction multiple data (SIMD) architectures. There are some gray scale adjustment algorithms that appear to be inherently serial, and thus seem unable to benefit from a SIMD architecture. But for one such algorithm (gray scale adjustment by histogram equalization) that falls in this category, an efficient parallel implementation is described that uses three fundamental operations: Sort, Send, and Scan. In this paper, several gray scale adjustment algorithms' SIMD implementations and order of growth performances are discussed.

## Linear Gray Scale Adjustment

As in all the algorithms described in this paper, the imagery is represented with one pixel per processor. To perform linear gray scale adjustment is then very easy. Simply find the maximum and minimum gray scale levels using the global reduction operators *MAX and *MIN, broadcast these values to every processor, along with the desired final dynamic range, and compute a new value:

NEW = (DYNMAX–DYNMIN)(OLD–MIN)/(MAX–MIN)

where MIN is the minimum gray scale value of the original image, MAX is the corresponding maximum value, OLD is the original gray scale value of the pixel in question, DYNMAX is the top of the desired dynamic range, and DYNMIN is the bottom of that range. NEW is then the new value for the pixel in question.

The global reduction, and broadcast functions take O(log$n$) time, where $n$ is the number of processors. The recomputation formula takes O($k$) time since every processor can compute independently of the others. The entire computation takes O(log$n$) time on $n$ processors for an image with $n$ gray scale values.

## Tabular Gray Scale Adjustment

This problem is trivial if the dynamic range of the input image is comparable to the number of pixels, in which case a single global communication (send) can perform the necessary computation. However, it is typical for the number of pixels to be roughly the square of the dynamic range. In general, the relationship between dynamic range and number of processors cannot be known, and thus there are several possible algorithms, each optimal under the proper conditions.

The most obvious choice besides performing the aforementioned send is to store the entire table in each processor with each pixel value, and execute an indirect local memory access into the table in each processor simultaneously. This will take order constant time, but require order $nm$ memory over $n$ processors, where $n$ is the number of pixels in the image and $m$ is the dynamic range.

A choice that is interesting, but fails to gain any advantage, involves an "interleaved scan" in which the number of each kind of pixel can be counted in a single scanning operation by setting the $kr$th bit of the memory location to be scanned, where $r$ is the input gray scale value and $k$ is the log of the

CH2649-2/89/0000/0335$01.00 © 1988 IEEE

maximum number of instances of a given value. Unfortunately, $k$ is too large to be of any use in almost any practical application.

Finally, one can perform the standard many-to-one sending operation that sorts the input data into some location A, compares each element of A with its right neighbor, storing a 1 into memory location B if the comparison returns an inequality, and 0 otherwise, subselect processors in which B is 1, in a sending operation retrieve the tabular lookup value for each subselected pixel/processor into C, perform a segmented copy scan on C, with segmentation memory location = B, and unsort C into the output memory location. This takes $O[(logn)^2]$ time on $n$ processors, with $n$ pixels, and constant memory per processor.

## Linear Local Neighborhood Gray Scale Adjustment

This algorithm performs precisely the same as the normal linear gray scale adjustment, except different values for MAX and MIN are computed for each pixel. This algorithm takes different time, because it finds the MAX and MIN for neighborhoods around each processor, which can be found in order $m$ time, where $m$ is the linear size of the neighborhood. To find MAX, each processor examines the processor in its own column, $m$ rows up, then $m$–1, then $m$–2, etc., to find the maximum value along that line. Then each processor does the same thing, with rows and columns switched, upon the maximum values so far found. Now each processor sends this result to the processor $m/2$ right, and $m/2$ up. Now each processor holds its appropriate MAX. MIN can be computed similarly.

## Median Filtering

This algorithm could be simple with indirect addressing. Each processor performs the standard median taking algorithm as done in serial. This takes order $q$ time, where $q$ is the number of elements in the immediate neighborhood. Indeed, for small immediate neighborhoods, that is the proper algorithm to use, but for larger neighborhoods, there is an algorithm that takes $O[(logq)^2]$ time. First, segment the image into blocks of $q$ pixels. Sort each of these blocks into memory location A. Now it is immediately possible to get the median of every $q$th pixel. In fact, in O(log$q$) steps we can find the median of every $q/2$nd pixel by combining adjacent sorted lists of pixels. This can be done recursively so that in O(log$q$) steps every pixel has the median of its neighborhood. Thus, this algorithm takes $O[(logq)^2]$ time.

## Histogram Equalization

The objective in this algorithm is to cause the integral of the histogram of the output image to be as nearly linear as possible, while following the constraint that any two pixels, if equal in shading in the input image, have the same shading in the output image.

A very simple sequence of parallel operations accomplishes this. First, sort the pixels in terms of intensity, and store the result into memory location A. Preserve the unsort information for later use. Have each processor look to its right neighbor (in a linear array), and if the value is different from its own, in A, then store a 1 into B, otherwise store a 0 into B.

Perform a min-scan on the processor index (self address), segmented by B, storing the result into C.

Unsort C using the preserved unsorting information saved from above. That is all. The unsorted C now contains the new values for the output image. At least one pixel of a given gray scale value in the output image has the same gray scale value as processor index. Any given pair of pixels with the same value on input has the same value on output.

The image can now be scaled linearly to match the dynamic range desired. This algorithm takes $O[(logq)^2]$ time, due to the sort.

One can also perform a local neighborhood histogram equalization in much the same way that the median filter works. In this case, every $q$th pixel ($q$ is used the same way here as above), determines its rank in its neighborhood by having its neighborhood sorted. This rank serves as an unscaled new value. Now every $q/2$nd pixel can determine its rank in O(log$q$) steps using the two sorted lists of numbers that already have been generated by its neighbors of index a multiple of $q$. This can be done recursively, so that the total time is $O[(logq)^2]$.

## Conclusion

A number of gray scale adjustment algorithms' descriptions and order of growth performances have been discussed. There is substantial reason to believe that since every algorithm attempted ran quickly (order polynomial in log of the size of the problem at worst), gray scale adjustment algorithms' performance, and their cousins in statistics, graphics, and image processing, will benefit from SIMD architectures.

# A PARALLEL PARTICLE-IN-CELL MODEL FOR THE MASSIVELY PARALLEL PROCESSOR

C. S. Lin A. L. Thring J. Koga

Southwest Research Institute
San Antonio, TX 78284

## ABSTRACT

The Particle-in-cell (PIC) model, which has been used extensively on supercomputers to simulate fusion and space plasma phenomena, is difficult to develop for the Massively Parallel Processor (MPP) because the model requires indirect indexing in computing electric fields. To overcome the difficulties, a parallel PIC algorithm is developed for MPP by mapping particles in a cell randomly to a row of processors. Because of this mapping, the algorithm needs only the nearest neighbor communication to sort particles and to collect charge density for each cell. From the cell charge density, this algorithm then calculates electric fields at the cell by Fast Fourier Transform. The developed PIC code has a speed comparable to that of the vectorized PIC code on CRAY X-MP. The results from simulating the plasma instabilities of a cold electron beam in a hot electron background are presented.

Keywords: particle-in-cell model, plasma simulation, beam plasma instability

## INTRODUCTION

This paper reports a parallel particle-in-cell (PIC) algorithm developed for the Massively Parallel Processor (MPP) at Goddard Space Flight Center. With the nearest-neighbor communication, the MPP consists of 16,384 processors configured in a 128 × 128 array. Based on the developed algorithm, the MPP PIC code called MPPPIC simulates the electrostatic interactions among 524,000 charged particles in a two dimensional plane with 128 by 128 cells. Although the MPP has a slow clock and I/O speeds, MPPPIC is as fast as the corresponding PIC code on CRAY X-MP.

The particle-in-cell model, which has been used extensively for studying wave-particle interactions in fusion and space plasmas, is difficult to develop on parallel computers because the model needs indirect indexing to compute charge density at the cell from particles' positions and electric forces on the particles from the cell charge density. Recently some success has been obtained with a one dimensional electrostatic PIC simulation developed for the Hypercube parallel computer with 32 processors where the simulation box has been divided into sub-domains containing an equal number of particles (Ref. 1). Two studies have attempted to develop two-dimensional plasma simulation codes on the MPP (Ref. 2-3). In one study, the model maps the simulation domain directly to the processor array and sorts particles according to their cells every time step (Ref. 2). The algorithm turns out to be inefficient on the MPP because it needs to extensively use I/O between the array unit and the staging memory. Furthermore, the processors do not evenly share the computation load because fluctuations in electric forces cause particles to distribute nonuniformly over the processors. In the other study, we developed a gridless model, which maps particles randomly to processors (Ref. 3). Avoiding the charge collection, the gridless model computes electric forces directly using discrete Fourier Transform. Although the algorithm is fully parallelized and particles are uniformly distribued over the processors, the gridless code is still seven times slower than the PIC code on the CRAY X-MP because the gridless model has much more computation than the PIC model (Ref. 4).

Learning from the previous two studies, we developed a parallel PIC algorithm that maps particles in a cell to a row of processors. This parallel algorithm can efficiently sort particles and easily collect charge density for each cell. The next section presents the key ingredient of this algorithm, particle mapping and sorting. Section 3 describes the numerical method, the flow chart and the timing of the MPPPIC code. Section 4 shows some results from simulating a plasma instability of a cold electron beam in a hot plasma. Finally, section 5 summarizes the results.

## PARTICLE MAPPING AND SORTING

Figure 1 illustrates the scheme that maps particles in a cell to a row of processors. For example, in a two dimensional $x-y$ plane, all the processors in a $n$th plane have the $x$ coordinate within the $n$th column $x_n < x < x_n + \Delta x$, where $x_n$ is the $x$ coordinate of the column and $\Delta x$ is the column width. This mapping scheme implies that a plane of array processors has particles with the same $x$ cell coordinate, that is, in the same column of cells. In each particle plane, particles in $j$th row of processors have the $y$ position as $y_j < y < y_j + \Delta y$, where $y_j$ is the coordinate of the $j$th cell in the $y$ direction.

For each particle plane, we use six array planes to specify the two spatial coordinates $x$ and $y$, the three components of velocities $V_x$, $V_y$ and $V_z$, and the species index $k$. For a simulation domain with 128 by 128 cells, the model ideally would have 128 particle planes to store particle coordinates. However, because the MPP's 64 Mbyte staging memory can store 64 but not 128 particle planes, we pack 2 columns of cells in a particle plane to simulate a simulation domain of 128 by 128 cells. Therefore, in the MPPPIC code, a row of processors actually contains particles in two cells. Moreover, MPPPIC partially fills the row of processors with particles and lets the sequence of particles in a row of processors be random (Figure 1, lower panel). As explained below, the vacant processors and the random sequence simplify the communication among the processors, resulting in a more efficient sorting procedure.

At every time step, MPPPIC sorts particles according to their cells as electric forces move some particles to the neighboring cells. The sorting procedure involves shuffling particles to the neighboring rows and merging particles into the neighboring particle planes. For particles moving to the adjacent cells in the $y$ direction, we rotate particles to the neighboring rows in the north-south direction. For particles moving to the neighboring cells in the $x$ direction, we merge them into their new particle planes. When particles are shuffled or merged to the occupied processors, we rotate the processors in the east-west direction until vacant processors are found. Searching of vacant processors in a row is possible because of the random sequence of particles in a row. If the row of processors has sufficient vacancy, the shuffling and merging of particles will not overfill the processors, thus greatly simplifying the

CH2649-2/89/0000/0339$01.00 © 1988 IEEE

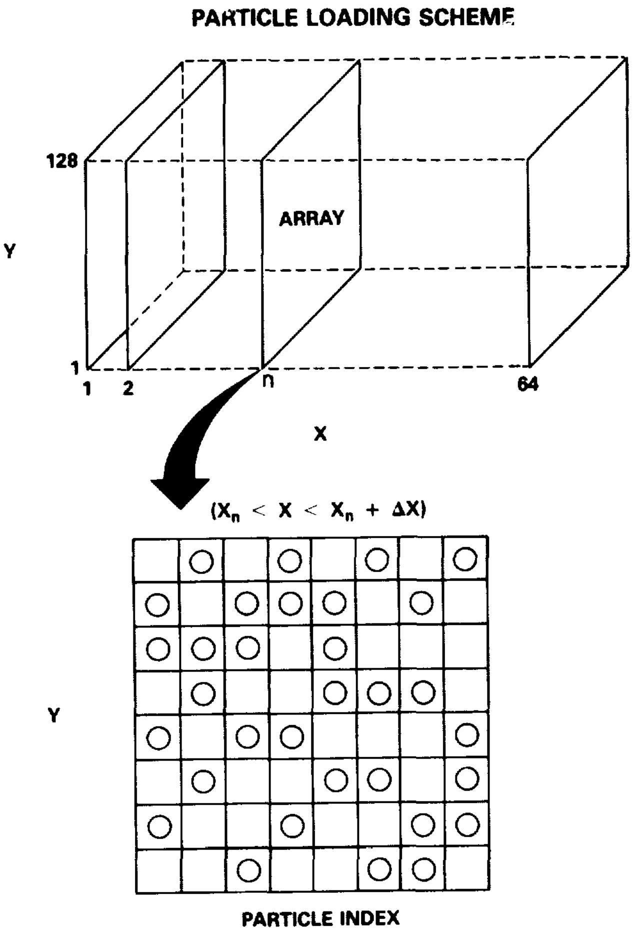


Figure 1. Schematic of mapping particles to the processors

sorting algorithm. However, too many vacant processors decreases the algorithm efficiency. We thus typically load 50% of the processors with particles by choosing the average number of particles per particle plane to be 64.

## NUMERICAL METHOD AND FLOW CHART

The particle-in-cell simulation code usually represents the plasma as a large number of finite-size particles moving according to classical mechanics in the self-consistent electromagnetic fields. The two-dimensional spatial system is then divided into fixed spatial cells or grids on which charge densities, potentials, and fields are defined. For the purposes of illustration, we will discuss only the electrostatic model which has no electric current density. Details of the PIC model are given in the textbooks of plasma simulations (Ref. 5-6)

A charged particle is assumed to have a rectangular shape with a width $\Delta$ comparable to the cell width. The charge density of each cell $\rho(x_g, y_g)$ is calculated by accumulating each particle's contribution according to its occupied area in the cell, where $x_g$ and $y_g$ are the coordinates of the cell. The charge density defines the electric potential $\Phi$ according to Poisson's equation

$$\nabla^2\phi(x_g, y_g) = -4\pi\rho(x_g, y_g) \tag{1}$$

We use Fast Fourier Transform to solve the electric field at the center of the cell $E(x_g, y_g)$ from Poisson's equation and the definition

$$\vec{E}(x_g, y_g) = -\vec{\nabla}\phi \tag{2}$$

From the electric fields at the nearest grid points $\vec{E}(x_g, y_g)$, the code then interpolates the electric forces on the particles. Finally, we use a leap-frog scheme to solve the equations of motion in a uniform magnetic field $\vec{B}$

$$\begin{aligned} d\vec{v}/dt &= (q/m)(\vec{E} + \frac{\vec{v}}{c} \times \vec{B}) && (3) \\ d\vec{r}/dt &= \vec{v} && (4) \end{aligned}$$

where $m$ and $q$ are the particle mass and charge, and $\vec{r}$ is the particle's radius vector.

Figure 2 gives the flow chart of the MPPPIC code. The first procedure INITPARTICLE initializes particles' positions at the zero time step and velocities at $-1/2$ time step according to the mapping scheme described in Section 2. Procedure PUSH then obtains particles' new positions and velocities $r^n$ and $v^{n-1/2}$ using Equations 3 and 4. The next procedure SORT contains two procedures SHUFFLE_Y to shuffle particles in the $y$ direction and MERGE_X to merge particle planes in the $x$ direction as described in Section 2. Procedure CHARGE collects the cell charge density $\rho$, which is the input to the procedure FIELD.

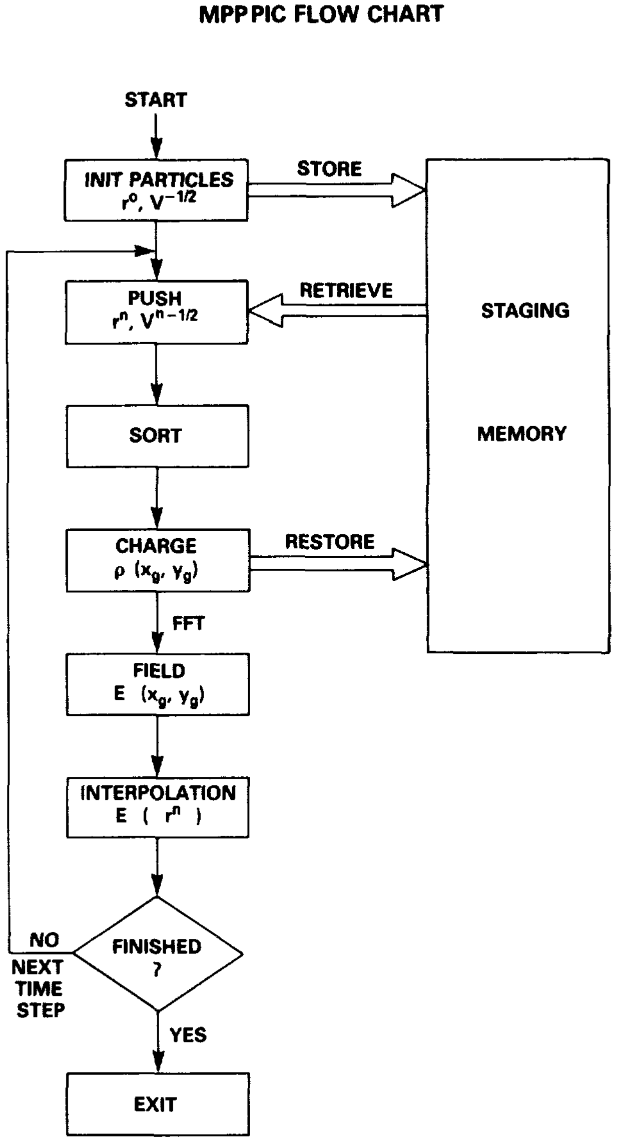


Figure 2. Flow Chart of the MPPPIC code.

Solving Equations 1 and 2 for the electric field at the cell $E(x_g, y_g)$, the FIELD procedure uses the two-dimensional Fast Fourier Transform developed for the MPP by Ref. 7. Finally, the INTERPOLATION procedure interpolates the electric fields $E(r^n)$ at the particles' position from $E(x_g, y_g)$ before returning to PUSH for the next time step.

Since the I/O between the staging memory and the array processors is slow, the MPPPIC code minimizes the number of I/O by retrieving each particle plane once during PUSH and restoring the particle plane in the staging memory after CHARGE. Furthermore, the code outputs diagnostic quantities such as particles' positions and velocities, charge density, and electric fields after FIELD.

Because MPPPIC loads particles randomly on half of the processors for 64 particle planes, the code can simulate up to 524,288 particles. For simulations with 524,288 particles, MPPPIC runs about 4.7 seconds per time step, which is six times faster than the vectorized PIC code on our CONVEX C-1 computer. On the CRAY X-MP at San Diego Supercomputer Center, we used to run the PIC code for 32,768 particles on a 32 x 32 grids for 0.25 seconds per time step. Since we no longer have access to CRAY X-MP, we cannot compare the performance of MPPPIC with the corresponding PIC code on CRAY X-MP. However, from our previous jobs, we estimate that MPPPIC is slightly slower than the vectorized PIC code on CRAY X-MP.

Table 1 lists the percentage of timings for each procedure during a time step, indicating that PUSH and FIELD procedures use very little time, 7% and 3% of the total time, respectively. MPPPIC spends about 22% of the time on transferring 64 particle planes between the array unit and the staging memory. On other parallel computers with large array memory, this code would increase its speed by simply eliminating the I/O for transferring particle arrays. Sorting particles according to their cells also takes a significant 41% percentage of the MPPPIC time. The sorting algorithm essentially plays the role of communications among the MPP processors, which have only nearest-neighbor communication. It is unclear how much improvement MPPPIC would make on other massively parallel computers with sophisticated communication.

TABLE 1. Percentage of Timings for MPPPIC Procedures

| Procedure | Timing Percentage (%) |
|---|---|
| PUSH | 7 |
| SORT | 41 |
| CHARGE | 15 |
| FIELD | 3 |
| INTERP | 12 |
| I/O | 22 |
| Total | 100 |

## SIMULATION RESULTS

To study beam plasma interactions in the earth's magnetosphere, we use MPPPIC to simulate the electrostatic interactions between a cold electron beam and a hot electron background. At altitudes of 2-3 earth radii in the auroral zone, the electron background is usually a mixture of cold electrons with a temperature of less than 1 eV and hot electrons with a temperature of about 1 keV. During auroral activities, keV electrons precipitate into the upper atmosphere to produce auroras because of electric fields along the geometric field lines. In the region where parallel electric fields accelerate precipitating electrons to form a cold beam, parallel electric fields also deplete the cold electron background. Assuming a neutralizing ion background, we can therefore simulate high frequency wave instabilities by using only the two component plasma, a cold electron beam and a hot electron background. In a separate paper, we reported the results of simulations of such a beam plasma instability in one dimension on the CONVEX C1 computer to explain broadband electrostatic waves observed by satellites in the earth's auroral zone (Ref. 8). The two dimensional simulation results from the MPP agree with those obtained from the CONVEX computer. Below we briefly describe the simulation model and present the MPP results.

We initially loaded the beam and background electrons uniformly in the simulation system with a uniform magnetic field in the $y$ direction. The background electrons had Maxwellian velocity distributions with the thermal velocity $a_h$, whereas the beam electrons had a shifted Maxwellian distribution with the beam velocity $v_b$ and the thermal velocity $a_b$. The simulation had the same number of particles for the background electrons and the cold beam. We chose $a_h = 0.1\,c$, $a_b = 0.001\,c$, and $v_b = 0.15\,c$, where $c$ is the speed of light, a unit of the simulation. The grid size, $\Delta$, was half of the Debye length of the hot electrons defined as $\lambda_d = a_h/\omega_{pe}$, where $\omega_{pe}$ is the hot electron plasma frequency. The electron gyrofrequency $\omega_{ce}$ in the simulation was $4.0\omega_{pe}$, which is typical at high altitude auroral zone. For the time increment per step, the simulations used $\Delta t = 0.05\omega_{pe}^{-1}$.

Figure 3 shows the phase space $v_y - y$ of the beam electrons (left panels) and the hot electrons (right panels), which are a small sample of the total population. The top two panels show the initial distribution for the beam component (Panel 1a) and the hot electrons (Panel 1b). Panel 1a indicates that the electron beam initially has a beam velocity $v_b = 1.5v_h$ in the $y$ direction. At $N = 200$ step, the cold electron beam began to oscillate in the phase space (Panel 2a). The phase space plot at $N = 600$ time steps shows that the electron beam forms the vortex pattern in the phase space, suggesting electron trapping by large amplitude electrostatic waves (Panel 3a). Although the vortex pattern is less clear, hot electrons also oscillate in phase space (Panel 3b). About this time step, the electrostatic wave grows from very small fluctuation level to a maximum amplitudes. At the end of simulation ($N = 1200$) the electron trapping motion is been destroyed (not shown).

The two dimensional contour plots of electric potential at $N = 600$ show horizontal wave structures (Figure 4), which is better illustrated in color plate III, p. 695 (Figure 5). The horizontal wave structures in potential suggest an electric field in the vertical direction or along the magnetic field. From Figure 4, we deduce the dominant wave mode to to have a wavelength of about $40\Delta$, or about $4\pi v_b/\omega_{pe}$ in the physical unit. The frequency spectrum of potential reported in Ref. 8 shows a broad power law spectrum up to $2\omega_{pe}$. The wave instability produced by a cold electron beam in a hot electron background therefore can produce broadband electrostatic waves above the electron plasma frequency. In contrast, the well known Buneman instability due to the interactions of a cold beam with a cold plasma occurs at frequencies less than the electron plasma frequency.

## SUMMARY

The objective of this paper is to demonstrate that a parallel particle-in-cell (PIC) algorithm is feasible for the Massively Parallel Processor. Currently, our parallel PIC code MPPPIC performs as fast as CRAY X-MP, a very good performance since the MPP has a slow arithmetic operational speed. In principle, the proposed algorithm also works for other single-instruction-multiple-data (SIMD) computers like the DAP and the Connection Machine. With faster I/O or more array memory, the parallel PIC algorithm has potential to simulate large-scale research problems in plasma physics and astrophysics, which have used the PIC model extensively. The efficiency of this algorithm mainly depends on the percentage of the occupied processors. In our example, electrostatic waves cause the localized distribution of particles in space, resulting in inefficient use of processors. Future studies will need to improve the algorithms in order to use the processors more efficiently.

*Acknowledgments.* We thank E. Seiler for providing the MPP FFT routines, which are critical for this study. This work was supported by NASA grant NAGW-1231, NSF grant ATM-8713225 and SwRI Internal Research Program.

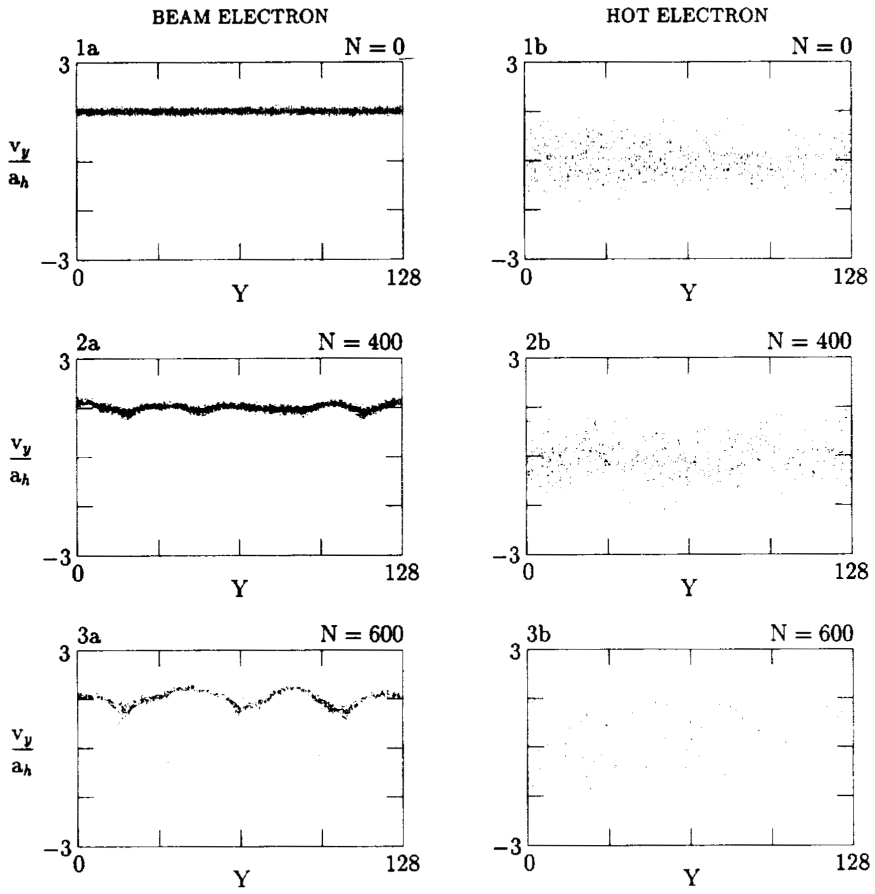


Figure 3. Phase space plot of the electron beam and hot electron background at 0, 400, and 600 time steps.

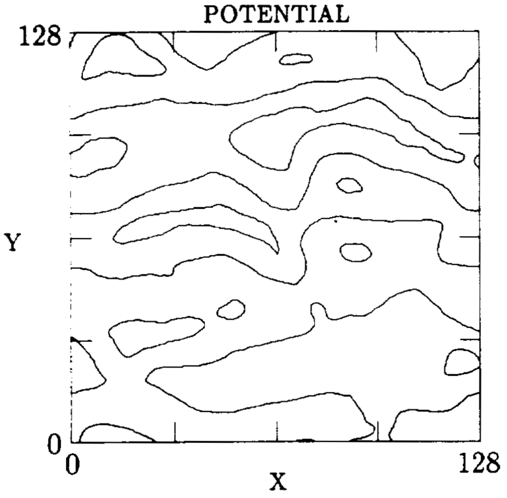


Figure 4. Electric potential contour plot at 600 time steps.

## REFERENCES

1. Liewer, P. C. and V. K. Decyk, "A Universal Concurrent Algorithm for Plasma Particle-in-Cell Simulation Codes," *UCLA Center for Plasma Physics and Fusion Engineering Report*, PPG-1164, 1988.

2. Gledhill, I. M. A., and L. R. O. Storey , "Particle simulation of plasmas on the Massively parallel processor," *(Proceedings of the First Symposium on the Frontier of Massively Parallel Scientific Computation)*, Goddard Space Flight Center, 1987, p. 37–46.

3. Lin, C. S., A. L. Thring and J. Koga, "Gridless particle simulation using the Massively Parallel Processor," *Computer Physics Communications*, 1988, 48, p. 149–154.

4. Lin, C. S. , A. L. Thring, J. Koga and R. W. Janetzke, "Plasma simulations using the Massively Parallel Processor," *(Proceedings of the First Symposium on the Frontier of Massively Parallel Scientific Computation)*, Goddard Space Flight Center, 1987, p. 185–191.

5. Hockney, R. W. and J. W. Eastwood, *Computer Simulation Using Particles*, McGraw-Hill, New York, 1981.

6. Birdsall, C. K. and H. B. Langdon, *Plasma Physics via Computer Simulation*, McGraw-Hill, New York, 1985.

7. Ramapriyan, H. K. and E. J. Seiler, "Synthetic Aperature Radar Signal Processing on the MPP," *Proceedings of the First Symposium on the Frontier of Massively Parallel Scientific Computation*, Goddard Space Flight Center, 1987, p. 177–181.

8. Wong, H. K., C. S. Lin and J. Koga, "Excitation of high frequency broadband electrostatic waves in the auroral zone," submitted to *Geophys. Res. Lett.*, 1988.

# PARALLEL ALGORITHM FOR DETERMINING MOTION VECTORS IN ICE FLOE IMAGES BY MATCHING EDGE FEATURES

M. Manohar, H. K. Ramapriyan and J. P. Strong

NASA/ Goddard Space Flight Center Space Data and Computing Division, Information Systems Facility, Code 636, Greenbelt, Maryland 20771

## ABSTRACT

A parallel algorithm is described to determine motion vectors of ice floes using time sequences of images of the Arctic ocean obtained from the Synthetic Aperture Radar (SAR) instrument flown on-board the Seasat spacecraft. Time intervals between two successive images of a given region can be as much as three days. During this period, large translations and rotations of ice floes can occur. Therefore, conventional local correlation techniques which perform searches in a small neighborhood to detect translated features have a very small chance of success. To account for large. translations and rotations, it is necessary to perform large area searches in a three dimensional space (two translational and one rotational). This makes conventional correlation techniques computationally intensive even on a high-speed parallel computer such as the Massively Parallel Processor (MPP). In this paper we describe a parallel algorithm which is implemented on the MPP for locating corresponding objects based on their translationally and rotationally invariant features. The algorithm first approximates the edges in the images by polygons or sets of connected straight-line segments. Each such "edge structure" is then reduced to a "seed point". Associated with each seed point are the descriptions (lengths, orientations and sequence numbers) of the lines constituting the corresponding edge structure. A parallel matching algorithm is used to match packed arrays of such descriptions to identify corresponding seed points in the two images. The matching algorithm is designed such that fragmentation and merging of ice floes are taken into account by accepting partial matches. The technique has been demonstrated to work on synthetic test patterns and real image pairs from Seasat in times ranging from .5 to 0.7 seconds for 128 x 128 images.

## INTRODUCTION

Sequential images of ice floes in the Arctic ocean were obtained from the Synthetic Aperture Radar (SAR) flown on-board the Seasat spacecraft in 1978. Using time sequences of these images, it has been shown in the literature that it is possible to map ice motion. The approach taken is to match recognizable features in the ice field which are imaged from two successive orbits. The matching procedures have been traditionally manual and time consuming. In order to perform this task routinely on a large number of images, it is necessary to develop automated analysis techniques. Recently, several automated techniques of estimating ice motion using cross correlations have been proposed (for example, [1-3]). Collins [3] posed the problem of finding a field of displacements between two successive images as an estimation problem. The typical time interval between two images of a given region is of the order of three days. During this period, large differential translations and rotations of ice floes can occur. Therefore, conventional local correlation techniques which perform searches in small local neighborhoods for displaced features have a very small chance of success. To account for large translations and rotations, it is necessary to perform large area searches in a three dimensional space (one rotational and two translational dimensions). These factors make correlation techniques computationally intensive even on a high-speed parallel computer such as the Massively Parallel Processor (MPP). Additional problems specific to ice floe images are fragmentation and merging during the time interval between the images. This requires approximate matches of ice floes from one image to the parts of larger ice floes from the other images. This problem is referred to as segment matching in the literature [4].

The work reported in this paper is an effort to automate ice floe matching in computationally feasible times (a few seconds for a pair of 512 x 512 images) using the MPP. In our approach the images are abstracted as line models of the boundaries of dominant objects (ice floes) in the image, and these models are matched using parallel matching techniques. The boundaries of the dominant objects are extracted by edge detection algorithms and the edges are segmented into set of lines by fitting polygons or connected sets of straight lines to the edge data. Such polygons or connected sets of straight lines will be referred to as *edge structures*. Now the problem is to match corresponding edge structures in the two images. Edge structure models have been used in the literature for matching cloud images [5] and terrain scenes [6]. Both these techniques are essentially sequential and are not considered segment matching. Davis [4] uses a relaxation technique for segment matching of edge structures. Initially, figures of merit are assigned to the matches between pairs of angles on two edge structures. Relaxation methods are then used to find acceptable combinations of these matches. This method is also sequential and is not a practical solution for images containing thousands of edge structures.

## THE ALGORITHM

The algorithm presented in this paper consists of the following steps, each of which is implemented in parallel. The edges of ice floe images are obtained by a suitable edge detection algorithm. The edges obtained are further subjected to some preprocessing such as thinning and eliminating isolated edge points. A connected component labeling algorithm [7] is applied to the edge map to obtain a label array, LBL. This algorithm locates a seed point in each connected set of edge points in the edge map and assigns the address of the seed point to all edge points in the connected set. Next, each edge is decomposed into a set of straight line segments. This is accomplished by detecting the corner points in the edge map. The corner point detection algorithm examines a local window (typically 7 x 7) and fits a straight line passing through its center. The fitting error, equal to the sum of the perpendicular distances from the edge points of the window to the fitted line is computed. The locations at which this error function has local maxima are identified as corner points. By assigning 0's to all corner points in the edge map the connected sets of edge points are separated into straight line segments to obtain a segmented edge map. Now, the length of each straight line segment is computed by shrinking the segmented edge map and counting the number of shrink operations each segment undergoes. The lengths so computed are stored in an array LEN, with the mid-points of each line segment containing its length and all other points containing 0. The orientation of each line segment is computed by applying the Hough transform to local windows (typically 5 x 5) surrounding its midpoint. This information is stored in an array, DIR, at addresses corresponding to the mid-points of the line segments. Next, sequence numbers are assigned to all line segments within each connected component indicating the order in which they are connected. This algorithm processes the edge map and LBL arrays to produce sequence numbers. This information is stored in an array, SEQ. Now, the quadruple (LBL,SEQ,DIR, LEN) provides a complete description of the edge structures in the image. These quadruples are sorted with LBL and SEQ as the primary and secondary keys, respectively, using a bitonic sorting algorithm [12]. This brings all data values of the arrays together into adjacent processing elements. Note that the arrays of sorted attributes is significantly smaller than the original images. This can make the matching more efficient, especially for images significantly larger than the n x n processing array if attribute records are packed by processing n x n segments of the images. These sorted arrays obtained from two images are then matched using a combination of global and local correlations. In the following sections we describe each of these steps in more detail.

### Preprocessing:

The preprocessing step consists of obtaining the boundary of the dominant objects of the ice floe images and applying thinning and eliminating isolated edge points. In the present work, various edge detection techniques such as Marr-Hildreth's zero crossing detection [8], Canny's algorithm [9], and Spatially Constrained Clustering (SCC) [Tilton, 10] were applied to the ice floe images. Both Marr-Hildreth's and Canny's algorithms gave unacceptable results. The zero crossings obtained by convolving the Laplacian of Guassian with the image combined the boundaries of the dominant objects with several other details and so it was difficult to delineate the boundaries from the zero crossings. Larger filter sizes could solve this problem to some extent but the edge location accuracy becomes poor. The Canny's algorithm also has the same problem for the ice floe images. The smaller filter gives unnecessary edges (due to noise and fine texture) and large filter sizes affect the shape of the boundary. Tilton's SCC algorithm [10], which grows regions based on a "best pair merging" criterion, performed better than the other two in delineating boundaries of dominant objects. The edge focussing algorithm by Bergholm [11] performed best for these images. This edge focussing algorithm is as follows.

1. Initialize a mask (of the size of the image) to 1's.

2. Detect major edges using Canny's algorithm with large size Guassian filters (e.g., $\sigma = 7.0$) and an appropriate threshold (e.g., 0.1) for the image. Accept edges only at locations where mask has values 1 as true edge.

3. Dilate the edges obtained in step 2 by one pixel in each direction and generate a mask which is 1 at all the dilated edge locations.

4. Now decrease sigma of the filter by 0.5. (It is shown analytically in [11] that change of sigma by 0.5 can displace an edge location by at most one pixel on either side of its previous location). Repeat steps 2 and 3 until the filter size is 0.5.

In order to speed up convolution the filter size also can be reduced in steps of two for every iteration starting with a size of 15 x 15.

This algorithm takes about 200 ms for 128 x 128 images on the MPP to perform 7 iterations with window sizes ranging from 15 x 15 to 5 x 5. The results of the edge detection algorithm are shown in Fig. 2(a - b) for both the images of the given pair Fig.1(a-b).

The edges obtained using the above algorithm are one pixel wide, but they are 4-connected for inclined (other than horizontal and vertical) lines. The subsequent processes involved require inclined edges to be 8-connected. A thinning algorithm is used to reduce such inclined 4-connected edges to 8-connected edges.

The thinning algorithm examines a 3 x 3 neighborhood and replaces the central pixel by 0 under the following conditions:

$$\begin{matrix}0&0&1\\0&1&1\\0&0&1\end{matrix}\qquad\begin{matrix}x&1&x\\0&1&1\\x&1&x\end{matrix}\qquad\begin{matrix}0&0&x\\0&1&1\\x&1&x\end{matrix}$$

In the above configurations 0 denotes the absence of an edge point, 1, the presence and x, a don't care condition. The other configuration of masks can be obtained by rotating the above masks by 90 degrees. The elimination of isolated edge points is straightforward.

## Connected Component Labeling:

The connected component labeling algorithm labels each of the connected edges with a label that is equal to the array address of the seed point. The algorithm is discussed in detail in [7]. This algorithm is based on parallel shrinking and expansion of binary patterns and requires about 30 ms of MPP time for labeling the edges. The time required is data dependent and is proportional to the length of the within-component-path of the longest component. The connected component labelling reduces all connected patterns, both open and closed, to single points called seed points. The connected component labels obtained from the edge maps are shown in Fig. 3(a-b). The value of the label at a pixel is coded as its grey level so that all pixels belonging to a given connected component are displayed with the same grey level value.

## Edge Decomposition:

Each connected component identified above is decomposed into a set of straight line segments by first identifying the corner points. The edge points between a pair of corner points are assumed to form a straight line. Corner points are edge points where the line direction changes significantly. The algorithm fits a straight line at every edge point to the connected set of edge points in a local neighborhood of size w x w (w = 7, typically). The fitting error (which is sum of perpendiculars from the edge points onto the fitted line) is computed. The local peaks in the fitting error function correspond to the corner pixels. The fitting error function is derived as follows.

Let $\alpha x + \beta = 0$ be the equation of the straight line passing through the origin of a local coordinate system to be fitted to the connected set of edge points in a w x w local neighborhood.

The error term, ε, is sum of perpendiculars from all connected edges of the local window which is given by

$$\varepsilon = \sum_i (\alpha x_i + y_i)^2 \qquad (1)$$

The value of a resulting in minimum ε is obtained by differentiating error, ε with respect to α and equating it to zero.

$$\alpha = -\sum_i x_i y_i \;/\; x_i^2 \qquad (2)$$

The minimum error for the best fit, $\varepsilon_m$ can be obtained by substituting a from equation 2 in equation 1.

$$\varepsilon_m = \sum_i y_i^2 - (\sum_i x_i y_i)^2 / \sum_i x_i^2 \qquad (3)$$

High values of the error term $\varepsilon_m$ indicate corners because a single line cannot be fitted to the given set of edge points. Therefore, local peaks of $\varepsilon_m$ correspond to the corner points. The corner points detected in both images are shown in Fig. 4(a - b) superimposed over the corresponding edge images.

## Length Computation :

The line segment lengths are computed by applying an 8-connected shrinking algorithm [7] repeatedly on the edge map wherein corner points are replaced with 0's. The number of shrinking operations required to reduce each line segment to a single point corresponds to the length of the line segment. Thus the lengths of the line segments are stored at their mid- points (called *line-seeds*). These points are the locations where all information needed for matching (such as label, direction, and length) about the line segments is stored. The length of the different edge segments are shown in Fig. 5(a-b) for two images. For displaying length as gray level image its value is propagated throughout the edge segment. The algorithm however, does not require this propagation.

## Direction Computation :

The direction of each line segment is computed at the line-seeds using a localized version of the Hough transform. At every line-seed in the edge map a local neighborhood of size w x w is examined. For every point in the neighborhood with the same label as the line-seed, the angle subtended by the line joining it and the line-seed is computed. A 32-bin histogram of these angles over the w x w window is computed. This corresponds to an angular resolution of less than 6 degrees. The slope corresponding to the peak of the histogram is the direction of the edge segment. The direction of different edge segments are shown in Fig. 6(a-b) for two images.

## Sequence Computation :

An ordered set of direction and length measurements corresponding to a sequence of connected line segments constituting an edge structure describes the edge structure completely. The order of occurrence of line segments is essential for this description to be unique. Thus attaching sequence numbers to all line segments of the edge structures is an important step of the algorithm. The sequence numbers are attached to all edge points constituting the edge structure and the numbers at the location of line-seeds are retained as line attributes. The sequence numbers are attached to all points of the polygon boundary starting from seed points for closed polygons. The process is slightly different for open edge structures and will be discussed later. Initially a sequence number of 1 is attached to all seed points (starting points) of the polygons. Now a 5 x 5 neighborhood centered at one of the immediate neighbors of starting point is considered. This point is given the next sequence number provided none of the other seven neighbors of the starting point has already been given this number. Then the position of the starting point is shifted to the current pixel where sequence number is assigned. This is repeated until no more assignments are possible. This algorithm is sequential along perimeter of a given polygon, but operates in parallel on all polygons.

For open edge structures the seed points are the mid-points (rather than end points), so they are not suitable as starting points for sequence generation. One of the two end points should be considered as a starting point. To locate the starting point for all open edge structures in parallel, we proceed as follows. The line ends are detected by examining 3x3 windows and counting number of edge pixels surrounding the central edge pixel. If this number is equal to 1 then the central pixel is an end point. This is valid for thin edges where inclined edges are 8-connected (but not 4-connected). Then the sequencing algorithm is applied from the seed points as in the case of closed polygons. The sequencing algorithm terminates at one of the end points. Now, it is not difficult to locate unique end points in parallel. The points where the sequence array has a value greater than 1 and the line-end array has a value of 1 are the starting points for open edge structures.

Thus the sequence computation algorithm treats closed and open edge structures separately. The open edge structures can easily be separated from closed edge structures (polygons) by applying an 8-connected shrinking algorithm. The open edge structures shrink to isolated points and closed ones are not affected. By eliminating isolated points after shrinking, the array will have only closed polygons. To obtain only open edge structures, the array containing closed polygons is subtracted from the array of all edge structures. The sequence numbers obtained by this algorithm have been coded as gray values and shown in Fig. 7 (a-b). The sequence information, label, length, and direction are retained only at the line-seed locations for further processing.

## Matching of edge structures:

The four edge structure attributes, namely, label of each edge structure, sequence numbers of line segments, direction value of each line segment, and length of each line segment, are stored in arrays LBL, SEQ, DIR, LEN. In these arrays, all locations except the line-seed points (defined above) contain 0's. Since this data is quite sparse the matching can be significantly improved by packing the measurements in the adjacent Processor Elements (PE) of the MPP. This is accomplished by sorting the edge structure attribute quadruplets using LBL, and SEQ as primary and secondary keys, respectively. A parallel bitonic sort algorithm [12] is used for this purpose. The sorting brings each edge structure attributes into the adjacent PE locations. The quadruplet (LBL, SEQ, DIR, LEN) completely characterizes the edge structures. Since DIR, LEN are sorted using LBL and SEQ as primary and secondary keys, respectively, the order in which they occur in adjacent PE's is a complete description of the edge structures. The matching algorithm essentially looks for similar list of attributes, DIR and LEN. The array, DIR contains the directions of line segments. The angle between the adjacent sides is used for matching. This can be obtained by a single absolute difference operation of DIR values contained in adjacent PE's. If DIR alone is used, the polygon matching is not affected by rotation as well as scale changes. It is necessary to use both DIR and LEN to ensure that significantly different scales of similar objects are not considered identical.

The sorted arrays (LBL,SEQ,DIR,LEN) for each of the two images are treated as one dimensional vectors for subsequent matching. (The *snake-shift* feature on the MPP is extensively used for this purpose). To permit matching of open edge structures in one of the images with closed polygons in the other without being sensitive to the (arbitrary) starting segments in the polygons and to avoid sensitivity to reversal of the sequencing of segments in the edge structures, the quadruplets for each edge structure are duplicated in the forward and reverse direction respectively, for the first and second images. Thus for example, the directional attributes for an n-sided edge structure in the first image are stored as D1, D2, .., Dn, D1, D2, ..., Dn and for an m-sided edge structure in the second image as d1, d2, ..., dm, dm, dm-1, ..., d1. The parallel matching of polygons proceeds as follows.

1. Normalize DIR and LEN features of each polygon

2. Perform global correlation of DIR features and LEN features separately for both images. That is compute $C_d$ = DIR1⊗DIR2 and $C_l$ = LEN1⊗LEN2, where ⊗ denotes correlation. This is done efficiently using Fast Fourier Transform.

3. Locate local peaks in $C_d$ and $C_l$.

4. Shift DIR2 by an amount equal to the peak coordinate and subtract from DIR1. Find sum of absolute differences within a local window of size 5 around each point.

5. Where the sum of absolute differences is less than a predetermined threshold, load the corresponding labels in

the output array.

6. Set DIR1 and DIR2 at locations of matched labels to 0.

7. Repeat steps 2 to 6 until no more matches are possible or DIR1 or DIR2 contains all 0's.

## RESULTS AND DISCUSSIONS

Using the matching technique described in this paper we are able to match ice floes which have undergone significant translations and rotations during the time interval between two images (Fig. 8). The edge extraction significantly affects the results of the algorithm. We have demonstrated using test patterns that our algorithm is able to match polygons accurately which have undergone significant translations, rotations, fragmentation, and merging. These synthetic images are shown in Fig. 9(a-b). The corresponding edge structures are shown in Fig. 10. The algorithm has been tested with subimages of ice floe images and found to yield satisfactory results.

In the present work, we have established correspondence among objects contained in the images by matching the sides having same subtended angles and lengths. Using the information so derived, it is possible to establish correspondence among the pixels and thus compute optical flow.

The computation times are data dependent. For the 128 x 128 test image, where the edge detection step was not needed the algorithm took approximately 500 msec on the MPP. For ice floe images, 700 msec of the MPP time was required including the time needed for the edge focussing algorithm. However, it is to be noted that for larger images than 128 x 128, one would process all 128 x 128 segments to obtain the edge structures and pack the attribute quadruples and then perform the matching.

## REFERENCES

1. Fily M., and Rothrock D. A., 'Sea Ice Tracking by nested correlations', IEEE Trans on Geoscience and Remote Sensing,' Vol. GE-25, pp. 570-580, Sept 1987.
2. Vesecky J. F., Smandani, R., Smith M.P., Daida, J. M., and Bracewell R. N., 'Observation of Sea-Ice Dynamics Using Synthetic Aperture Radar Images : Automated Analysis,' IEEE Trans on Geoscience and Remote Sensing, Vol. 26, pp. 38-48, Jan 1988.
3. Collins M. J., Emery W. J., ' A Computational Method for Estimating Sea Ice Motion in Sequential Seasat Synthetic Aperture Radar Imagery by Matched Filtering,' J. of Geophysical Research, Vol. 93, pp. 9241-9251, Aug, 1988.
4. Davis L. S., 'Shape Matching Using Relaxation Techniques' IEEE Trans. on Pattern Anal. and Mach. Intell. Vol. PAMI-1, pp. 60- 72, Jan 1979.
5. Aggarwal J. K., and Duda, R. O., 'Computer Analysis of Moving Polygonal Images,' IEEE Trans. on Computers, Vol. C-24, pp. 966-976, Oct 1975.
6. Clark C. S., Conti, D. K. Eckardt, W. O., McCulloh, T.A., Nevatia, R., and Tseng, D. Y.,'Matching of Natural Terrain Scenes,' Poc of the IEEE Conf on Pattern Recognition and Image Processing, pp. 217-222, 1980.
7. Manohar M., Ramapriyan H. K., ' Connected Component Labeling of Binary Images on Mesh Connected Massively Parallel Processor,', to Appear in Computer Vision Graphics and Image Processing', Jan 1989.
8. Marr D. C., Hildreth E., ' Theory of Edge Detection,' Proc R. Soc. London, B 207, pp. 187-217, 1980.
9. Canny J., 'Computational Approach to Edge Detection,' IEEE Trans on Patten Anal. Mach. Intell. Vol. PAMI-8, pp. 679-698, Nov 1986.
10. Tilton J., 'Image Segmentation by Parallel Region Growing with Application to Data Compression and Image Analysis, Proc of 2nd Symposium on the Frontiers of Massively Parallel Computation, Oct 10-12, 1988.
11. Bergholm F., 'Edge Focussing', IEEE Trans. on Pattern Anal. and Mach. Intell. Vol. PAMI-9, pp. 726-741, 1987.
12. Dorband J. E., 'Sort Computation', Proc of 2nd Symposium on the Frontiers of Massively Parallel Computation, Oct 10-12, 1988.

**NOTE:** All figures appear as Color Plates IV, V, and VI, on pp. 696–698 of these *Proceedings*.

# SYMBOLIC SOLUTION OF SIMULTANEOUS LINEAR ALGEBRAIC EQUATIONS VIA PARALLEL NUMERICAL COMPUTING

I.D. Mayergoyz and F.P. Emad
Electrical Engineering Department
and Institute for Advanced Computer Studies
University of Maryland

## ABSTRACT

Symbolic solutions of linear algebraic equations are important in many applications. Usually, special languages for symbolic manipulations (e.g. MACSYMA) are used for handling problems like these. However, pure symbolic computations become very time- and memory-consuming when the number of simultaneous equations increases. In this paper we describe a new approach to the symbolic solution of simultaneous linear algebraic equations via numerical computing. This approach has the following attractive features: (i) computations are inherently parallel and can be implemented on parallel processors with a fine-grain architecture, (ii) calculations are reduced to two well-studied problems: numerical solution of simultaneous linear equations and the fast Fourier transform (FFT).

## I. INTRODUCTION

This paper describes a new method for the symbolic solution of simultaneous linear algebraic equations via parallel numerical computing. Symbolic solutions of linear algebraic equations are important in many applications. One particular application, which we shall constantly keep in mind, is the computation of analytical expressions for transfer functions of multivariable control systems. Usually, special languages for symbolic manipulations (e.g. MACSYMA [1]) are used for handling problems like these. However, pure symbolic computations become very time- and memory-consuming when the number of simultaneous equations increases. In this paper we describe a new approach to the symbolic solution of simultaneous linear algebraic equations via numerical computing. This approach has the following attractive features: (i) computations are inherently parallel and can be implemented on parallel processors with a fine-grain architecture, (ii) calculations are reduced to two well-studied problems: numerical solution of simultaneous linear equations and the fast Fourier transform (FFT).

The paper is organized as follows. The description and mathematical substantiation of the method are given in section II. Section III provides some numerical examples illustrating the potential of the method.

## II. DESCRIPTION OF THE METHOD

Consider the following simultaneous algebraic equations

$$\hat{A}(s,\lambda)\vec{X}(s,\lambda) = \hat{B}(s,\lambda)\vec{U}(s,\lambda), \tag{1}$$

where $\hat{A}(s,\lambda)$ and $\hat{B}(s,\lambda)$ are matrices whose entries are polynomials with respect to $s$ and $\lambda$.

We are concerned with the computation of the matrix $\hat{T}(s,\lambda)$ such that:

$$\vec{X}(s,\lambda) = \hat{T}(s,\lambda)\vec{U}(s,\lambda). \tag{2}$$

The matrix $\hat{T}$ is known in control theory as a transfer function matrix. We restrict our discussion to the case of two variables $s$ and $\lambda$ only for the sake of notational simplicity. The technique itself is directly applicable to any number of variables. The condition that matrix entries are polynomials is not very restrictive either. By introducing new variables, we can always satisfy this condition.

It is easy to see that matrix entries of $\hat{T}(s,\lambda)$ can be represented as follows:

$$T_{nm}(s,\lambda) = \frac{d_{nm}(s,\lambda)}{\Delta(s,\lambda)} = \frac{\sum\limits_{\nu}\sum\limits_{k} d_{nm}^{k\nu} s^k \lambda^\nu}{\sum\limits_{\nu}\sum\limits_{k} c_{k\nu} s^k \lambda^\nu}, \tag{3}$$

where $\Delta(s,\lambda)$ is the determinant of $\hat{A}(s,\lambda)$. Thus, the problem of computing symbolic expressions for $T_{nm}(s,\lambda)$ is tantamount to the problem of numerical computing of polynomial coefficients $d_{nm}^{k\nu}$ and $c_{k\nu}$.

To solve the last problem, we shall use the following change of variables:

$$s = \exp(i\theta), \qquad \lambda = \exp(i\phi), \tag{4}$$

$$0 \le \theta \le 2\pi, \qquad 0 \le \phi \le 2\pi, \qquad i = \sqrt{-1}. \tag{5}$$

Then the expression (3) can be represented as follows:

$$T_{nm}(\theta,\phi) = \frac{d_{nm}(\theta,\phi)}{\Delta(\theta,\phi)} = \frac{\sum\limits_{\nu}\sum\limits_{k} d_{nm}^{k\nu} e^{ik\theta} e^{i\nu\phi}}{\sum\limits_{\nu}\sum\limits_{k} c_{k\nu} e^{ik\theta} e^{i\nu\phi}}. \tag{6}$$

Thus, $d_{nm}(\theta,\phi)$ and $\Delta(\theta,\phi)$ can be construed as finite two-dimensional Fourier series and the unknown coefficients $d^{k\nu}_{nm}$ and $c_{k\nu}$ can be interpreted as Fourier coefficients. Consequently, the values of these coefficients can be found through FFT if we somehow generate the values of $d_{nm}(s,\lambda)$ and $\Delta(s,\lambda)$ in discrete mesh points:

$$\begin{aligned} s_\gamma &= \exp(i\theta_\gamma), \qquad (\gamma = 1,2,\cdots\Gamma), \\ \lambda_\omega &= \exp(i\phi_\omega), \qquad (\omega = 1,2,\cdots\Omega), \end{aligned} \tag{7}$$

where

$$\theta_\gamma = 2\pi\frac{\gamma}{\Gamma}, \qquad \phi_\omega = 2\pi\frac{\omega}{\Omega}. \tag{8}$$

The last problem can be attacked as follows [2]:

Consider the basis

$$\vec{u}^{(1)} = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \vec{u}^{(2)} = \begin{pmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{pmatrix}, \ldots, \vec{u}^{(M)} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{pmatrix} \tag{9}$$

in the linear vector space $R^M$ of input values. Substituting these basis vectors into (1), we obtain:

$$\hat{A}(s,\lambda)\vec{X}^{(m)}(s,\lambda) = \hat{B}(s,\lambda)\vec{u}^{(m)}, \tag{10}$$

where $\vec{X}^{(m)}(s,\lambda)$ is a solution of (10) corresponding to $\vec{u}^{(m)}, m = 1,2,\ldots M$ . We will solve linear simultaneous equations (10) for each mesh point $(s_\gamma,\lambda_\omega)$ :

$$\hat{A}(s_\gamma,\lambda_\omega)\vec{X}^{(m)}(s_\gamma,\lambda_\omega) = \hat{B}(s_\gamma,\lambda_\omega)\vec{u}^{(m)}. \tag{11}$$

According to (9), we have:

$$\hat{B}(s_\gamma,\lambda_\omega)\vec{u}^{(m)} = \vec{b}^{(m)}(s_\gamma,\lambda_\omega), \tag{12}$$

where $\vec{b}^{(m)}(s_\gamma,\lambda_\omega)$ is the vector which coincides with the m-th column of matrix $\hat{B}$ computed at $(s_\gamma,\lambda_\omega)$ .

Using (12), simultaneous equations (10) can be represented in the form:

$$\hat{A}(s_\gamma,\lambda_\omega)\vec{X}^{(m)}(s_\gamma,\lambda_\omega) = \vec{b}^{(m)}(s_\gamma,\lambda_\omega). \tag{13}$$

If triangularization is used for the solution of (13), then both the determinants $\Delta(s_\gamma,\lambda_\omega)$ and the solution $\vec{X}^{(m)}(s_\gamma,\lambda_\omega)$ can be found for each mesh-point $(s_\gamma,\lambda_\omega)$ . Knowing these, the mesh-point values of $d_{nm}(s_\gamma,\lambda_\omega)$ are determined as follows:

$$d_{nm}(s_\gamma,\lambda_\omega) = X^{(m)}_n(s_\gamma,\lambda_\omega)\Delta(s_\gamma,\lambda_\omega). \tag{14}$$

This formula can be substantiated as follows.

According to (2), we have

$$\vec{X}^{(m)}(s,\lambda) = \hat{T}(s,\lambda)\vec{u}^{(m)}. \tag{15}$$

Taking into account the structure of the basis vectors $\vec{u}^{(m)}$ , we find

$$X^{(m)}_n(s,\lambda) = T_{nm}(s,\lambda) = \frac{d_{nm}(s,\lambda)}{\Delta(s,\lambda)}. \tag{16}$$

Consequently,

$$D_{nm}(s,\lambda) = X^{(m)}_n(s,\lambda)\Delta(s,\lambda). \tag{17}$$

Now, the validity of (14) follows from (17).

By using the mesh-point values of $\Delta(s_\gamma,\lambda_\omega)$ and $d_{nm}(s_\gamma,\lambda_\omega)$ found as described before, the polynomial coefficients $d^{kv}_{nm}$ and $c_{kv}$ are computed by employing the two dimensional discrete Fourier transform:

$$c_{kv} = \frac{1}{\Gamma\Omega}\sum_\omega\sum_\gamma \Delta(s_\gamma,\lambda_\omega)e^{-ik\theta_\gamma}e^{-iv\phi_\omega}, \tag{18}$$

$$d^{kv}_{nm} = \frac{1}{\Gamma\Omega}\sum_\omega\sum_\gamma d_{nm}(s_\gamma,\lambda_\omega)e^{-ik\theta_\gamma}e^{-iv\phi_\omega}. \tag{19}$$

Thus, the numerical algorithm can be summarized as follows.

On the first step, the linear simultaneous equations (13) are solved for each mesh point. Employing triangularizations of matrices $\hat{A}$ , the determinants $\Delta$ and the solutions $\vec{X}^{(m)}$ are found. Using them and formula (14), $d_{nm}$ are determined. It is worthwhile noting that it is desirable to solve the linear equations (13) *simultaneously* for all different right-hand-side vectors $\vec{b}^{(m)}, m = 1,2,\ldots,M$ at each mesh point. By this way, the triangularization of matrix $A$ (which is the most time-consuming part of the solution of linear equations (13)) need only be performed once.

It is apparent from the given description that the method is conceptually simple and very general in nature. The most computationally laborious part of the method is the solution of the linear simultaneous equations (13) at the mesh points. But these computations can be performed in parallel and hardly any communication is required between different processors (or different groups of processors) assigned for the solution of linear simultaneous equations (13) at different mesh points. For this reason, the numerical implementation of the method on parallel computers with a fine-grain architecture (such as MPPs) is very attractive. However, the above method can also be competitive with other methods in the case of implementation on sequential computers. This is particularly true for problems with large degrees of polynomial entries of matrices $\hat{A}$ and $\hat{B}$ and relatively lower orders of these matrices. For such problems, the numerical realization of the above method on sequential computers will require sequential solutions of simultaneous equations (13) of relatively low order in many mesh points.

## III. NUMERICAL EXAMPLES

Even when implemented on a serial computer, the method presented above proved to be very fast. As an example, the transfer functions for an n-output, 2-input system were computed using a SUN3/260 computer. The results of these computations are the $2n$ transfer functions between the 2 inputs and the $n$ outputs. The computation was performed via two methods: MACSYMA and the new FFT method presented in this paper. The time in seconds per transfer function is shown in Figure 1 for values of $n$ ranging from 3 to 6. Above $n = 6$ MACSYMA would not operate due to stack limitations. It is noted that the computation time for MACSYMA increased fast, while the time for the new FFT method was very small and did not increase appreciably.

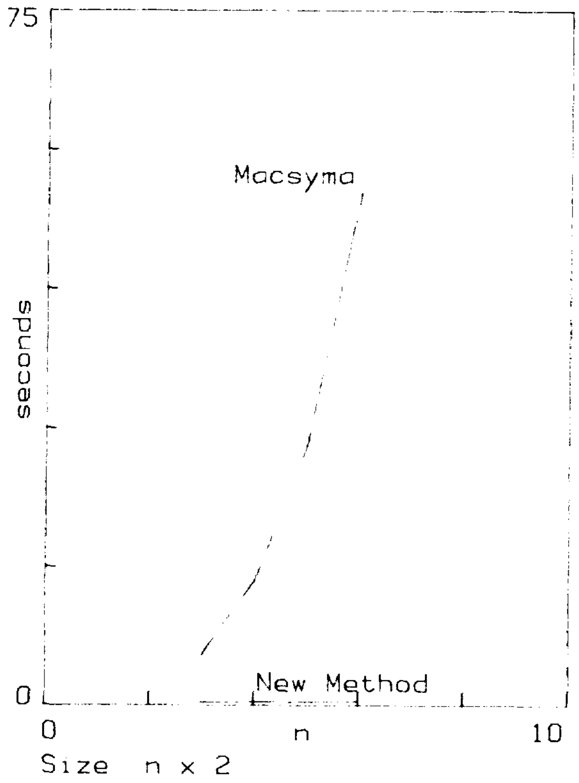


Figure 1. Time per transfer function.

To illustrate the accuracy of the new method, one of the transfer functions computed above is detailed below and compared with the exact results from MACSYMA. The matrices $\hat{A}$ and $\hat{B}$ had entries made of second order polynomials in $s$ with random integer coefficients in the range [0,99]. Integer coefficients were chosen because MACSYMA operates faster with integers. Two entries of $\hat{A}$ had $\lambda$ added to them, these were in row 1 column 2 and in row 2 column 3. The size of the matrix $\hat{A}$ was $6 \times 6$ while the size of the $\hat{B}$ matrix was $6 \times 2$. For brevity, only the numerator of the first of the twelve transfer functions is described below, first using MACSYMA and then using the new method:

Using MACSYMA:

$$\begin{aligned}
&21932609902s^{12} + 131917336967s^{11} \\
&+ (413599055\lambda + 215922330025)s^{10} \\
&+ (2354561915\lambda + 356798599743)s^{9} \\
&+ (542728\lambda^2 + 2802355102\lambda + 144001382381)s^{8} \\
&+ (5134725\lambda^2 + 3520281393\lambda + 23626820611)s^{7} \\
&+ (14494201\lambda^2 + 1288205855\lambda - 726612572275)s^{6} \\
&+ (21713039\lambda^2 + 1577795555\lambda - 657405288782)s^{5} \\
&+ (10367073\lambda^2 - 4613069249\lambda - 408739454002)s^{4} \\
&+ (118576\lambda^2 - 3978678622\lambda - 98433602039)s^{3} \\
&+ (-29024444\lambda^2 - 5576569512\lambda + 473611280028)s^{2} \\
&+ (-3377269\lambda^2 - 1691134813\lambda + 126088102484)s \\
&- 22221022\lambda^2 + 1564822806\lambda + 20522603760).
\end{aligned}$$

And using the new FFT method (five significant figures shown, only the coefficients are printed, in the same positions as they appear above):

$$\begin{array}{lll}
0.21932E+11 & 0.13192E+12 & \\
0.41358E+09 & 0.21592E+12 & \\
0.23546E+10 & 0.35680E+12 & \\
0.54334E+06 & 0.28024E+10 & 0.14400E+12 \\
0.51435E+07 & 0.35203E+10 & 0.23628E+11 \\
0.14490E+08 & 0.12882E+10 & -0.72661E+12 \\
0.21693E+08 & 0.15778E+10 & -0.65741E+12 \\
0.10345E+08 & -0.46130E+10 & -0.40874E+12 \\
0.97177E+05 & -0.39787E+10 & -0.98434E+11 \\
-0.29018E+08 & -0.55766E+10 & 0.47361E+12 \\
-0.33440E+07 & -0.16912E+10 & 0.12609E+12 \\
-0.22188E+08 & 0.15648E+10 & 0.20523E+11.
\end{array}$$

The new FFT method was tested for larger systems (MACSYMA could not be used for $n > 6$ due to stack size limitations). The results are shown in Figure 2 for $n$ up to 15. The irregularity of this curve is due to the fact that the mesh points used were a power of two, while the order of the transfer functions was not exactly equal to a power of two, thus for some sizes, more mesh points were used than the minimum number (in order to simplify the FFT algorithm). It is also noted in Figure 2 that the rise in time to about 2 seconds per transfer function is mostly due to the use of a serial computer. Had the computation been done on a parallel computer, due to the highly parallel nature of the algorithm, the time per transfer function

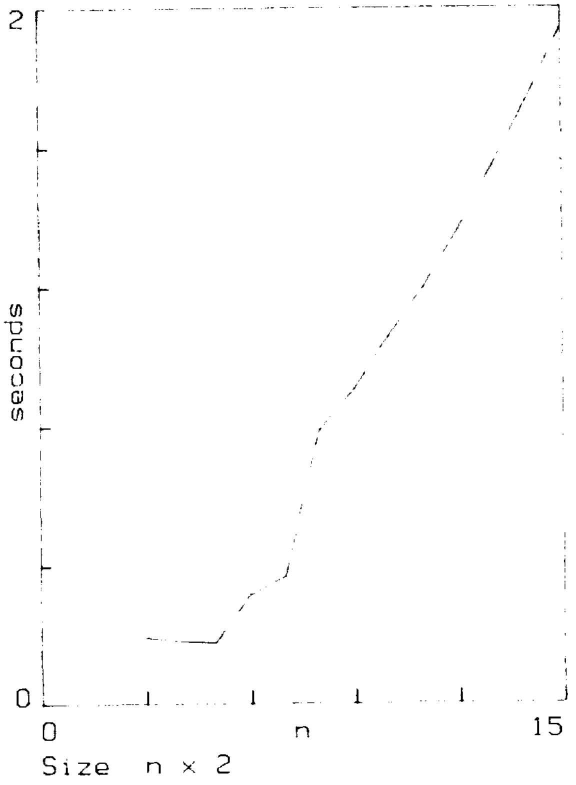


Figure 2. Timing for the new FFT method.

would not have risen as much as seen in Figure 2 which was derived using a serial computer.

To further explain the significance of the times shown in Figure 2, it is noted that when $n = 15$, there are 30 transfer functions, each of them a multinomial in $s$ and $\lambda$ with powers of $s$ up to 30 and powers of $\lambda$ up to 2. Each of these multinomials took less than 2 seconds to compute using a *serial* computer (SUN 3/260). The coefficients of the nemerator of the first transfer function are shown in Table I. The $\hat{A}$ matrix was of size $15 \times 15$ with entries polynomials in $s$ of second order whose coefficients were random real numbers over [0, 1.5]. The $\hat{B}$ matrix was similar but of size $15 \times 2$. The $\hat{A}$ matrix had $\lambda$ added to two of its entries (row 1 column 2 and row 2 column 3).

## IV. REFERENCES

[1] MACSYMA, a language for symbolic manipulation developed at Project MAC at MIT. MACSYMA is a trademark of Symbolics, Inc., a Delaware corporation.

[2] I.D. Mayergoyz, F.P. Emad, International Journal of Control, vol. 46, No. 6, pp. 1935-1945, 1987.

TABLE I. The numerator of the first of 30 transfer functions using the new FFT method detailed in the paper. Entries for the coefficients of $s$ and $\lambda$ in the $\hat{A}$ and $\hat{B}$ matrices were in the interval [0, 1.5]. (Note that each line in the Table corresponds to a term $(a_{2k}\lambda^2 + a_{1k}\lambda + a_{0k})s^k$, $k = 0, 1, \ldots, 30$).

```
Numerator of first term:
 0.17142e+02  (k=30 term)
-0.45880e+02  (k=29 term)
 0.58635e+02 -0.14387e+02  (k=28 term, etc...)
 0.36528e+03  0.42288e+02
-0.46595e+03 -0.58699e+02  0.61924e+01
-0.21756e+04 -0.43130e+03 -0.14208e+02
-0.96628e+03  0.48333e+03  0.13680e+02
 0.29870e+04  0.18400e+04  0.13935e+03
 0.50037e+04  0.10554e+04 -0.17604e+03
 0.50059e+04 -0.17465e+04 -0.36679e+03
 0.50602e+04 -0.60018e+04 -0.54296e+03
-0.31939e+04 -0.40111e+04  0.80154e+03
-0.41172e+04  0.12517e+04  0.16603e+04
 0.13213e+04  0.95373e+04  0.13955e+04
-0.43065e+03  0.32556e+03 -0.11903e+04
 0.25146e+05  0.44178e+03 -0.32236e+04
 0.18089e+05  0.15231e+04 -0.23648e+04
-0.36137e+04 -0.26542e+04  0.34657e+03
-0.30316e+05 -0.75001e+03  0.24571e+04
-0.31543e+05 -0.21666e+04  0.30358e+04
-0.13684e+05  0.11334e+05  0.16189e+04
-0.77809e+04  0.11057e+05 -0.43347e+03
-0.10435e+04  0.40239e+04 -0.43849e+04
 0.38144e+04 -0.19618e+04 -0.31910e+04
 0.44378e+04 -0.27993e+04 -0.71390e+03
 0.51358e+04  0.11420e+04  0.10551e+04
 0.66292e+03  0.10322e+04  0.77673e+03
-0.11991e+04 -0.60690e+03 -0.11806e+03
-0.44176e+03 -0.74645e+02 -0.65851e+02
 0.95457e+02  0.83314e+02  0.18320e+02
 0.54301e+02  0.23279e+02  0.62786e+01
```

# PARALLEL IMPLEMENTATION CONSIDERATIONS FOR A CLASS OF SIGNAL PROCESSING ALGORITHMS

Nidal M. Sammur and Martin T. Hagan
Electrical and Comp:uter Engineering Department
Oklahoma State University
Stillwater, OK 74078

## ABSTRACT

The Burg filter is a signal processing algorithm which is widely used in such areas as geophysical data analysis, speech processing and spectral analysis. The structure of this algorithm is typical of a class of important signal processing algorithms. The Burg algorithm has been investigated with the objective of determining its suitability for implementation on parallel processing architectures. The algorithm has been implemented on three different machines, representing a variety of parallel architectures: the Denelcor HEP, the Intel iPSC/2 hypercube and the NASA/Goodyear MPP. It is concluded that the algorithm is especially suited for implementation on massively parallel architectures, such as the MPP.

## INTRODUCTION

This paper discusses an investigation of parallel processing implementations of an important type of signal processing algorithm. The Burg algorithm is commonly used in such areas as seismic data processing, spectral estimation and speech signal analysis. It has a structure which is similar to several other signal processing algorithms, and the results of this study should be relevant to these algorithms as well.

During the course of this study the Burg algorithm has been implemented on three different parallel processing architectures: the Denelcor HEP, a tightly coupled MIMD machine; the Intel iPSC/2 hypercube, a loosely coupled MIMD machine; and the NASA/Goodyear MPP, a massively parallel SIMD machine. It was found that the algorithm is especially well suited to the massive parallelism of the MPP. This was somewhat surprising since the MPP, with its mesh architecture, was designed (in part) to process two-dimensional images; and the Burg filter is a one-dimensional signal processing algorithm.

The next section of this paper will introduce the Burg algorithm. This will be followed by a description of the parallel implementation of the algorithm, with special emphasis on the MPP, and a comparison of the performances of the three machines. Finally, there will be a discussion of the results.

## ALGORITHM DEVELOPMENT

The Burg filter is a method for fitting an autoregressive time series model. An autoregressive model of order m is given by:

AR PROCESS $X_n+a_1^mX_{n-1}+\dots+a_m^mX_{n-m}=e_n^m$ (1)

where $X_n$ is the autoregressive process, $a_1^m$ thru $a_m^m$ are the process parameters, and $e_n^m$ is white noise. This model can be implemented by the digital filter shown in Figure 1. This figure illustrates the direct form II implementation [1].

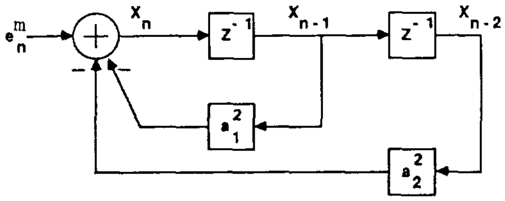


Figure 1. Direct Form II Implementation of Eq. (1).

Another way of realizing this process is by means of a lattice structure, as shown in Figure 2.

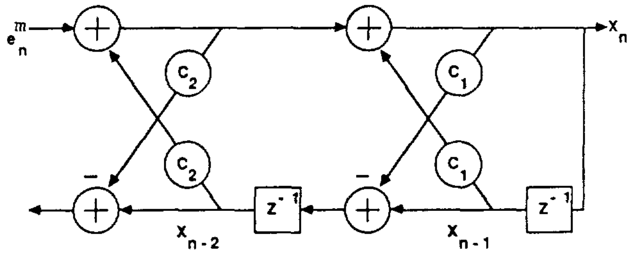


Figure 2. Lattice Implementation of Eq. (1)

In most applications the order of the process (m in Eq. (1)) is not known a priori. Therefore the analysis begins with a first order model, and the order is increased one step at a time. It can be shown [2] that the autoregressive coefficients ($a_i$'s of Figure 1) are related to the reflection coefficients ($c_i$'s of Figure 2) by the following equation:

$a_i^{m+1} = a_i^m + c_{m+1}\, a_{m-i+1}^m \quad i=1,\dots,m$ (2)

$a_{m+1}^{m+1} = c_{m+1}$

CH2649-2/89/0000/0353$01.00 © 1988 IEEE

The objective of the Burg algorithm is to estimate the reflection coefficients, and in turn the autoregressive coefficients, so as to best fit the data sequence $\{x_1, x_2, \ldots, x_n\}$.

To derive the Burg algorithm, first consider the forward prediction error of the m+1th order autoregressive model:

$$e_n^{m+1} = x_n + a_1^{m+1} x_{n-1} + \ldots + a_{m+1}^{m+1} x_{n-m-1} \qquad (3)$$

If equation (2) is used to obtain the coefficients, (3) can be written:

$$e_n^{m+1} = x_n + (a_1^m + c_{m+1} a_m^m) x_{n-1} + \ldots$$
$$+ (a_m^m + c_{m+1} a_1^m) x_{n-m} + c_{m+1} x_{n-m-1} \qquad (4)$$

Now consider the forward prediction error of the mth order model:

$$e_n^m = x_n + a_1^m x_{n-1} + \ldots + a_m^m x_{n-m} \qquad (5)$$

There is an equivalent backward prediction model:

$$b_{n-m-1}^m = x_{n-m-1} + a_1^m x_{n-m} + \ldots + a_m^m x_{n-1} \qquad (6)$$

It can be shown that the statistics of this model are equivalent to those of the forward prediction model.

By comparing equation (4) with equations (5) and (6), we can see that:

$$e_n^{m+1} = c_{m+1} b_{n-m-1}^m + e_n^m \qquad (7)$$

Likewise we could show:

$$b_{n-m-1}^{m+1} = b_{n-m-1}^m + c_{m+1} e_n^m \qquad (8)$$

The Burg algorithm chooses $c_{m+1}$ so as to minimize the sum of squares of the forward and backward prediction errors:

$$J = \sum_{n=m+2}^{M} (e_n^{m+1})^2 + (b_{n-m-1}^{m+1})^2 \qquad (9)$$

It can be shown [3] that the optimal choice of reflection coefficient is:

$$c_{m+1} = \frac{-2 \sum_{n=m+2}^{M} e_n^m b_{n-m-1}^m}{\sum_{n=m+2}^{M} (e_n^m)^2 + (b_{n-m-1}^m)^2} \qquad (10)$$

To summarize, the Burg algorithm consists of two steps: 1) update the forward and backward prediction errors using (7) and (8); 2) calculate the reflection coefficient using (10) and then repeat step 1). If the autoregressive coefficients $(a_1^{(m+1)}, \ldots, a_{m+1}^{(m+1)})$ are desired they are calculated using equation (2).

## IMPLEMENTATION ON THE MPP

A standard sequential implementation of the Burg algorithm is shown in Figure 3. In this figure the individual computations, or tasks, are labeled: $T_{in}(1)$, $T_n(1)$, $T_{in}(2)$, $T_{nn}(2)$, $T_{in}(3)$.

```
1. INITIALIZATION
   FOR i = 1 TO M DO
       e(i) = x(i)
       b(i) = x(i)

2. THE MAIN LOOP
   FOR n = 1 TO MAX DO
       s1 = 0.0, s2 = 0.0
       FOR i = n+1 TO M DO
           s1 = s1 + e(i)*b(i-n)
           s2 = s2 + e(i)$^2$ + b(i-n)$^2$                  $T_{in}(1)$
       c(n) = -2*s1/s2                                $T_n(1)$

       IF n>1 THEN DO
           FOR i = 1 TO n-1 DO
               a1(i) = a(i) + c(n) * a(n-i)
           FOR I = 1 TO n-1 DO
               a(i) = a1(i)                           $T_{in}(2)$
       a(n) = c(n)                                    $T_{nn}(2)$
       FOR I = n+1 TO M DO
           temp = e(i) + c(n)*b(I-n)
           b(i-n) = b(i-n) + c(n)*e(i)                $T_{in}(3)$
           e(i) = temp
```

Figure 3. Sequential Implementation of the Burg Algorithm

To implement the Burg algorithm using parallel techniques we need to determine which tasks can be performed in parallel. Figure 4 illustrates the relationship between the various tasks for the case where there are 5 data points (M=5) and 3 coefficients to be calculated (MAX=3). Any tasks

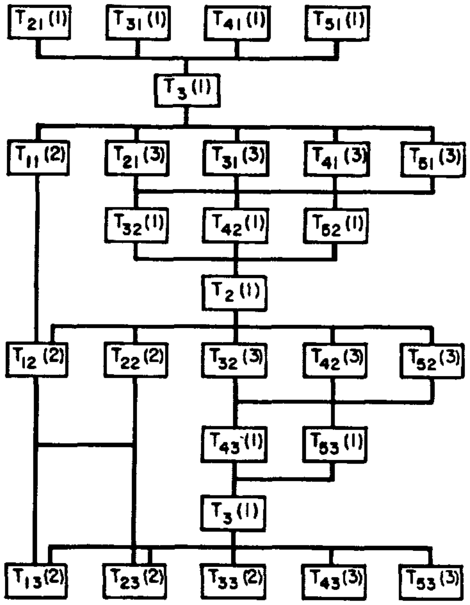


Figure 4. Maximally Parallel Graph for M=5 and MAX=3

which are on the same level can be performed at the same time. As illustrated by the maximally parallel graph we would need to have M processes (where M is the number of data values) to take full advantage of the parallel nature of the algorithm. This is clearly infeasible on machines like the Denelcor HEP or the Intel iPSC/2 hypercube computer but feasible on the NASA/Goodyear MPP due to its massive number of processors (16834). In practice it is unlikely to get a time series made of more than this number of observations.

The MPP is a two-dimensional mesh type architecture with nearest neighbor communication between the processing elements in the array unit (ARU). This type of architecture is most suitable for the processing of two-dimensional images; the Burg filter is a one-dimensional signal processing algorithm. The implementation problem reduces to finding a way to map the one-dimensional structure inherent in the Burg filter onto the two-dimensional architecture of the MPP. A miniature (16 elements) ARU is shown in Figure 5 with arrows representing the required connections or communication channels needed to view the mesh architecture as a linear array of processors, which would be most suitable for the implementation of the Burg filter.

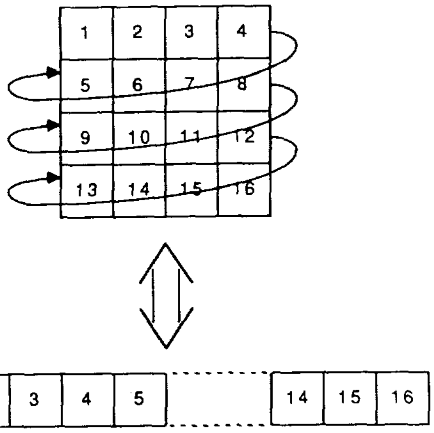
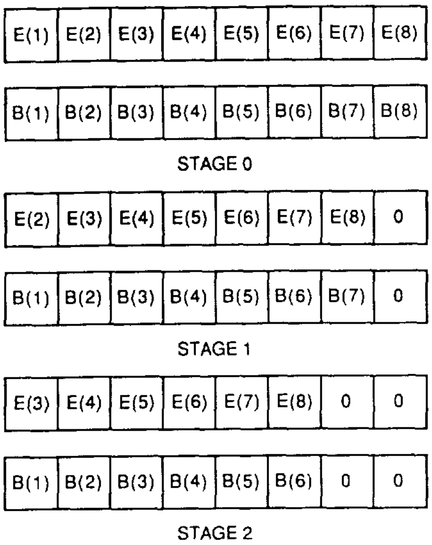


Figure 5. Mapping a Linear Array on a Mesh

Figure 6 illustrates the data movement for the Burg filter in a linear array of eight processing elements. In stage 0 the linear array is loaded with both the forward and backward prediction errors, actually the observed time series. To calculate the first reflection coefficient the forward prediction errors are shifted to the left by one as shown in stage 1. Now the reflection coefficients can be determined by forming the two sums: the first is the sum of the products of the two elements in each processing element, the second is the sum of the squares of the two elements in each processing element. Equation (10) can then be used to calculate the reflection coefficient that will be broadcast to all the processing elements, where it will be used to update the forward and backward prediction errors. To calculate the second reflection coefficient the updated forward prediction errors are shifted to the left by one, as shown in stage 2. The above sequence of operations will be repeated until all reflection coefficients are computed.



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| E(1) | E(2) | E(3) | E(4) | E(5) | E(6) | E(7) | E(8) |
| B(1) | B(2) | B(3) | B(4) | B(5) | B(6) | B(7) | B(8) |

STAGE 0

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| E(2) | E(3) | E(4) | E(5) | E(6) | E(7) | E(8) | 0 |
| B(1) | B(2) | B(3) | B(4) | B(5) | B(6) | B(7) | 0 |

STAGE 1

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| E(3) | E(4) | E(5) | E(6) | E(7) | E(8) | 0 | 0 |
| B(1) | B(2) | B(3) | B(4) | B(5) | B(6) | 0 | 0 |

STAGE 2

Figure 6. Data Movement For the Burg Filter in a Linear Array

The MPP Pascal code used to implement the described sequences is:

```
for n:=1 to max do begin
    e: = snake-shift(e);
    where (col_index = 127) do
       where (row_index = 128-n) do
          b: = 0.0;
    s1: = e*b;
    s2: = sqr(e)+sqr(b);
    sum1: = sum(s1,1,2);
    sum2: = sum(s2,1,2);
    c[n]: = -2.0*sum1/sum2;
    temp: = c[n]*b+e;
    b:    = c[n]*e+b;
    e:    = temp;
end;
```

The procedure snake-shift is used to simulate the effect of shifting the forward prediction errors to the left by one in the linear array mapped onto the MPP. The MPP Pascal code for the main part of snake-shift is:

```
r2: = shift(x,0,1);
r1: = rotate(x,1,1);
where (col_index=127) do
   where (row_index<127) do
      r2: = r1;
snake-shift: = r2;
```

Figure 7 illustrates the procedure.

The parallel Burg algorithm was implemented on the MPP as described above. The number of data points was 16384 and the number of reflection coefficients to be calculated was varied from one to 100. The results are shown in Table 1. Notice the linear relationship illustrated here.

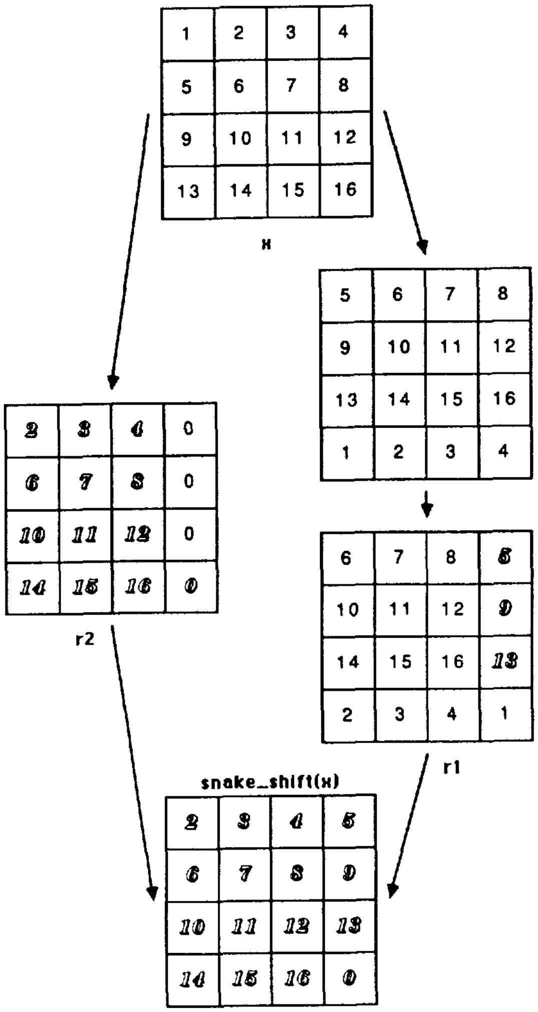


Figure 7. Procedure Snake-shift (x)

| MAX | TIME(msec) |
|---|---|
| 1 | 5.5392 |
| 2 | 11.0590 |
| 5 | 27.6197 |
| 10 | 55.2215 |
| 40 | 220.8270 |
| 80 | 441.6460 |
| 100 | 552.0589 |

Table 1. Summary of Parallel Burg Algorithm

The Burg filter was also implemented on the Denelcor HEP (a tightly coupled MIMD machine) and the Intel iPSC/2 hypercube (a loosely coupled MIMD or SPMD machine). The mapping techniques used for those machines is discussed in [4], [5] and [6]. A comparison of the performances of those machines with the MPP is given in Table 2, which compares the execution times to compute 10 reflection coefficients for 16,384 data points. Clearly the Burg algorithm takes full advantage of the massive parallelism of the MPP.

| Machine | Execution Time |
|---|---|
| Denelcor HEP | 1.679 sec |
| Intel iPSC/2 | 0.6121 sec |
| MPP | 0.05522 sec |

Table 2 Comparison of Burg Execution Time (16384 Data Points, 10 Reflection Coefficients)

## SUMMARY AND CONCLUSIONS

The Burg filter was implemented on three different computers: the Denelcor HEP (a tightly coupled MIMD machine), the Intel iPSC/2 hypercube (a loosely coupled MIMD or SPMD machine); and the NASA/ Goodyear MPP (an SIMD machine). The choice of these three architectures provides a variety of mapping possibilities for the algorithm.

This study showed that the Burg algorithm has a high degree of parallelism, which can be fully exploited only if the number of processors is equivalent to the number of data points. This is only feasible if a massively parallel machine like the MPP is used.

It should be emphasized that the Burg filter was used in this study because its structure is typical of a class of important one-dimensional signal processing algorithms. The results obtained here should be relevant to these other algorithms as well.

## REFERENCES

[1] W.D. Stanley, G.R. Daughterty, R.D. Daugherty, Digital Signal Processing, Restor Publishing Company, Reston, Virginia, 1984.

[2] John Makhoul, "Stable and efficient lattice methods for linear prediction," IEEE Trans. Acoust., Speech, Signal Process., Vol. ASSP-25, No. 5, pp. 423-428, Oct. 1977.

[3] J. Burg, "Maximum entropy spectral analysis," Ph.D. dissertation, Stanford University, Stanford, CA, May 1975.

[4] M. Hagan, H. Demuth and P. Singgih, "Parallel Signal Processing Research on the HEP,"Proceedings of the 1985 International Conference on Parallel Processing, St. Charles, ILL., pp. 599-606, August 20-23, 1985.

[5] H. Demuth and M. Hagan, "A Parallel Burg Algorithm and other HEP Research," Proceedings of the Workshop on Parallel Processing Using the Heterogeneous Element Processor (HEP), Norman, OK, pp. 349-361, March 20-21, 1985.

[6] N.M. Sammur, M.T. Hagan, "Mapping signal processing algorithms on the hypercube," To be submitted.

# IMAGE SEGMENTATION BY ITERATIVE PARALLEL REGION GROWING WITH APPLICATIONS TO DATA COMPRESSION AND IMAGE ANALYSIS

James C. Tilton

Mail Code 636
NASA Goddard Space Flight Center
Greenbelt, MD 20771

## ABSTRACT

Image segmentation can be a key step in data compression and image analysis. However, the segmentation results produced by most previous approaches to region growing are suspect because they depend on the order in which portions of the image are processed. An iterative parallel segmentation algorithm avoids this problem by performing the globally best merges first. After a background section, this paper describes such a segmentation approach, and two implementations of the approach on NASA's Massively Parallel Processor (MPP). Application of the segmentation approach to data compression and image analysis is then described, and results of such application are given for a Landsat Thematic Mapper image.

Keywords: Image segmentation, Image Analysis, Data compression, Data parallel analysis.

## BACKGROUND

Segmentation is the process of partitioning images into constituent parts called regions using image attributes such as pixel intensity, spectral values, and textural properties. Image segmentation produces an image representation in terms edges and regions of various shapes and interrelationships.

Image segmentation is a key step in many approaches to data compression and image analysis. An optimal coding of an image segmentation, such as through a region label map and region feature file, can be used to effect data compression (see *Ref. 3*). Image analysis can be performed on an image segmentation by using the shape, texture, spectrum, etc. of the regions found by the image segmentation and interrelationships between the regions. This region based analysis of imagery is potentially more effective than pixel based analysis, because region based analysis exploits spatial information whereas pixel based analysis does not.

Most image segmentation approaches can be placed in one of three classes: (i) characteristic feature thresholding or clustering, (ii) boundary detection, and (iii) region extraction. Characteristic feature thresholding or clustering is often ineffective because it does not exploit spatial information. Boundary detection does exploit spatial information through examining local edges found throughout the image. For simple noise-free images, detection of edges results in straightforward boundary delineation. However, edge detection on noisy, complex images often produces missing edges and extra edges which cause the detected boundaries to not necessarily form a set of closed connected curves that surround connected regions. One way to overcome this problem is to combine region extraction and boundary detection. Ref. 2, reports on some experiments in combining boundary detection approaches with the iterative parallel region growing approach discuss here.

Early approaches to region extraction (usually by region growing) had the disadvantage that the regions produced depended on the order in which portions of the image are processed. But Schachter, *et al* (Ref. 1) suggest that implementing region growing as "an iterative parallel process" would overcome the order dependent problem. This is the approach taken by the iterative parallel image segmentation algorithm presented here.

## ITERATIVE PARALLEL REGION GROWING

The basic concept behind our iterative parallel segmentation approach is to perform the globally best merges first. With this approach, the whole image is processed in parallel, eliminating the order dependence problem that troubled earlier approaches to region extraction by region growing.

*The globally best merge is defined as follows.* A similarity criterion is calculated for all pairs of spatially adjacent regions in the image. The globally best merge is the merge of the pair of spatially adjacent regions with the best similarity criterion value over the entire image (*i. e.*, the most similar pair of regions). (NOTE: For convenience, we assume from this point that the best similarity criterion value is the ***minimum*** similarity criterion value.)

Since only spatially adjacent regions can group together in this approach, we call our approach the Spatially Constrained Clustering (SCC) algorithm. The basic SCC algorithm is as follows:

i. Initialize the segmentation process by labeling each pixel as a separate region.
ii. Calculate a similarity criterion between each pair of spatially adjacent regions.
iii. Find the minimum similarity criterion measure value for the entire image.
iv. Check for convergence by projecting if the proposed merge would produce an error larger than the error threshold. If converged, stop. Otherwise continue on to step v.
v. Merge pairs of regions with the minimum similarity criterion measure value.
iv. If the number of regions remaining in the image is less than the preset minimum, stop. Otherwise return to step ii.

Two different versions of the SCC algorithm have been implemented (see next section) that differ only in how step v is handled. The serial merge version is:

$v^s$. Merge a single pair of regions with the minimum similarity criterion measure value (break ties arbitrarily).

The parallel merge version of the SCC algorithm implements step v as:

$v^p$. Merge all pairs of regions with similarity criterion equal to or less than 1 + $\delta$ times the minimum similarity criterion.

When $\delta = 0$, the parallel merge version is still an exact implementation of the basic SCC algorithm. It is only different from the serial merge implementation in that ties are not broken when more than one pair of regions have the minimum similarity criterion value. All such regions are merged (in parallel). For $\delta > 0$, the parallel merge version becomes an approximation of the basic SCC algorithm. Using $\delta > 0$ speeds convergence with the cost of finding a less optimal segmentation.

For either the serial or parallel merge version, the algorithm is considered to have converged when either a desired number of regions remain, or when no pair of adjacent regions is similar enough to be merged according to a predefined bound on the similarity criterion.

A key aspect of any region growing approach is the similarity criterion employed. The optimum similarity criterion depends upon the application. To fully explore the utility of the general SCC approach, we will need to devise and test several different similarity criteria for different types of image data and for various analysis procedures performed on each type of image data. In the experiments reported here, the similarity criterion used is based on minimizing variance normalized mean squared error.

The Mean Square Error (MSE) of band "i" of a multiband image is defined as

$$\mathrm{MSE}_i = E[(D_i - D_i^r)^2] \cong \frac{1}{N-1} \sum_{p=1}^{N} (D_{ip} - D_{ip}^r)^2 \qquad (1)$$

where $D_i$ and $D_i^r$ are the data values of the $i^{th}$ band of the original and reconstructed images, respectively; $D_{ip}$ and $D_{ip}^r$ are the values of the $p^{th}$ pixel of the $i^{th}$ band of the original and reconstructed images, respectively; E denotes the expected value; and N is the total number of pixels in the image.

The variance normalized mean squared error for band "i" ($\mathrm{NMSE}_i$) is defined as

$$\mathrm{NMSE}_i = \frac{\mathrm{MSE}_i}{\mathrm{VAR}_i} \qquad (2)$$

where $\mathrm{VAR}_i$ is the variance of band "i". The similarity criterion used in our tests is the $\mathrm{MAX}_i(\Delta\mathrm{NMSE}_i)$ for each pair of spatially adjacent regions, where the maximum is taken over all bands ($1 \leq i \leq m$). (Optionally, the similarity criterion can be taken as $\sum_{i=1}^{m}(\Delta\mathrm{NMSE}_i)$.) For a particular pair of spatially adjacent regions, $\Delta\mathrm{NMSE}_i$ is the change in $\mathrm{NMSE}_i$ when the pair of regions is merged and the reconstructed image is formed by substituting the mean vector of each region for the multispectral radiance values of each pixel in the region.

The change in $\mathrm{NMSE}_i$, or $\Delta\mathrm{NMSE}_i$, is calculated as follows:

$$\Delta\mathrm{NMSE}_i = \frac{\mathrm{MSE}_i^c - \mathrm{MSE}_i}{\mathrm{VAR}_i} \qquad (3)$$

where $\mathrm{MSE}_i^c$ is the mean squared error when regions j and k are merged, while $\mathrm{MSE}_i$ is the mean squared error before regions j and k are merged. Using the definitions of $\mathrm{MSE}_i$ and the region mean, it is easy to derive a more fundamental version of equation (3), *viz*

$$\Delta\mathrm{NMSE}_i = \frac{n_j(\bar{D}_{ij} - \bar{D}_{ijk})^2 + n_k(\bar{D}_{ik} - \bar{D}_{ijk})^2}{(N-1)\mathrm{VAR}_i} \qquad (4)$$

where $n_j$ and $n_k$ are the number of points in regions j and k, respectively, before combining, and N is the number of points in the image. $\bar{D}_{ij}$ and $\bar{D}_{ik}$ are the mean values of band i for regions j and k, respectively, before combining, and $\bar{D}_{ijk}$ is the mean value of band i for the region that would result from combining regions j and k.

## IMPLEMENTATION ON THE MPP

We have implemented the serial and parallel merge versions of the SCC algorithm on the Massively Parallel Processor (MPP) at the NASA Goddard Space Flight Center. For a description of the MPP see Ref. 4. Both implementations use the staging memory extensively to allow the processing of multispectral images of up to 512-by-512 pixels and up to 12 bands. Without the staging memory, either implementation would be restricted to a 128-by-128 4-band image, or a 128-by-256 2-band image or a 128-by-384 single band image because of the local array memory limitations of the MPP. While the use of the staging memory makes possible the processing of reasonably large multispectral images, this use does extract a penalty in the terms of processing time for the data transfers between the staging memory and array memory. We estimate that for a 7-band, 256-by-256 pixel image, the parallel merge version of the SCC algorithm would execute 10 times faster on an MPP with sufficient local array memory to eliminate the need for extensive stager-array data movements.

The implementation of the serial merge version of the SCC algorithm (using step $v^s$) on the MPP is extremely straightforward. The initialization is trivial, and local neighborhood data movements are used in step ii to calculate in parallel the similarity criterion for spatially adjacent regions. (For images larger than 128-by-128 pixels, a virtual MPP of up to 512-by-512 processors is emulated by data rotates across the edges of the 128-by-128 array and masked assignments.) In step $v^s$, a single pair of regions is identified for merging. (When more than one pair of regions has similarity function value equal to the minimum, the pair of regions with a minimum region label value is chosen.) The feature values (number of pixels and mean vector) for this pair of regions is extracted from the array, and new feature values are calculated in scalar mode for the new region. The merged region is given a new region label equal to the minimum of the two region labels, and the feature values are assigned to the merged region using a masked assignment.

The implementation of the parallel merge version of the SCC algorithm (using step vP) on the MPP is more complicated than the serial version. In order to merge more than one pair of regions in parallel in step vP, we need to resort to more than just local neighborhood data movements and masked assignments. The method we chose is as follows. First perform all the merging on the region label level. This is done through parallel region label propagation keyed on the similarity criterion function values. Once the new region label map is established, the new region feature values (number of pixels and mean vector) need to be calculated. In order to do this in parallel we grow a tree from a single pixel (seed pixel) in each region until it covers every region completely. (A unique seed pixel can be identified in region by comparing the current region label map with the initial region label map.) Then the number of pixels and sum of the data values at each pixel in each region are accumulated by tracing back up each tree. All region means are then calculated at each seed pixel, and the feature values for each region are broadcast out to each pixel in each region by traveling back down each tree, and depositing the feature values at each node of each tree.

## APPLICATION TO DATA COMPRESSION AND IMAGE ANALYSIS

An image segmentation can be a key step in a lossy data compression process. This type of data compression is a variant upon an image data compression process often referred to as vector quantization. In this form of data compression, each region in an image segmentation is given a unique label, and a list is generated of feature values corresponding to each region. This region label map and feature list is then encoded by a lossless compression scheme. For a more detailed discussion of this process, see Ref. 3.

The amount of information lost by this lossy data compression process is determined by how well the segmented image represents the original image. If the key region feature is taken to be the multispectral mean vector for each region, the effect of this data compression an image can measured by calculating the Root Normalized Mean Squared Error (RNMSE), which we define as follows:

$$\mathrm{RNMSE} = \frac{1}{m} \sum_{i=1}^{m} \sqrt{\mathrm{NMSE}_i} \; . \qquad (5)$$

The Normalized Mean Squared Error of band "i", $\mathrm{NMSE}_i$, was defined in equation (2). The RNMSE carries the following intuitive interpretation: The RNMSE is the band average of the single-band RNMSE, which can be regarded as the mean deviation of a reconstructed image pixel value from the corresponding original image pixel value per standard deviation of the band.

An image segmentation can also be used as a first step in an image analysis scheme. As mentioned before, image analysis can be performed on an image segmentation by using the shape, texture, spectrum, etc. of the regions found by the image segmentation, and by the interrelationships between the regions. Whereas the more complicated shape, texture and interrelationship analysis have the greatest analysis potential, we will demonstrate here how even a simple analysis approach using spectral information alone - the Maximum Likelihood Classifier - can be improved by proceeding it with an image segmentation step.

## EXPERIMENTAL RESULTS

A 256-by-256, 7-band subset of a Landsat Thematic Mapper (TM) image over Ridgely, Maryland was used as a test data set for this study. For this test, we processed the TM image with the parallel merge SCC algorithm. We first used a value 0.5 for $\delta$ and stopped the segmentation process when the total remaining number of regions was $\leq$ 2.5% of the number of pixels in the original image (1486 regions). Then we restarted the algorithm and processed from that point with a $\delta$ value of 0.1 until the number of regions was $\leq$ 2.0% of the number of pixels in the original image (1299 regions). (This produced better results than processing all the way down to 2.0% with a $\delta$ of 0.5.)

Figure 1 (color plate VII, p. 699) shows the original and segmented images, along with the difference image (plus a bias) between the original and segmented images (bands 2, 4 and 5 of the 7-band image are displayed). A subjective evaluation of the segmented image reveals that areas in the original image that are relatively homogeneous, but not necessarily uniform, become completely uniform in the segmented image. Low contrast spatial features are often lost in the segmented image, but higher contrast spatial features, such as edges of regions, are retained very precisely. Even very small spatial features are retained if they have sufficient contrast relative to the surrounding area.

The RNMSE image quality measure for segmented image in figure 1 is 0.33. That is, the mean deviation of an image pixel value in the segmented image from the corresponding original image pixel value per standard deviation of each band is 0.33.

The segmented image was encoded into region label map and a region feature files, and the region label map was losslessly compressed using run-length encoding. This segmentation/run-length encoding combination produced a data compression ratio of 13.1 to 1. (A optimal lossless compression technique may produce an even higher compression ratio). Optimal lossless encoding of the original TM image data typically produces a compression ratio of 3 to 1 or less (see Ref. 5).

We tested an image analysis approach where the segmented image was classified by a simple Maximum Likelihood Classifier. This analysis result was compared with the result obtained by using the same classifier on the original image. (For a more detailed description of the test setup see Ref. 3.)

The classification results for the original and segmented image are given in figure 2 (color plate VII, p. 699) and Table 1. The classification accuracies are consistently better for the segmented image than they were for the original data! We hypothesize that the segmentations produced by the SCC algorithm encode information from the surrounding regions of the image in each pixel. The MLC classification results are improved because each pixel has knowledge of its spatial surroundings in the segmented image.

Table 1. Accuracy comparison (% correct classification) between classifications of the original and segmented TM images.

| | Classification | |
|---|---|---|
| Class | Original Image | Segmented Image |
| Water/Marsh | 73.7% | 79.3% |
| Forest | 74.8% | 75.6% |
| Residential | 54.4% | 64.9% |
| Ag./Dom. Grass | 81.9% | 83.4% |
| OVERALL | 79.2% | 80.9% |

The first ten iterations of the parallel merge version took 118 seconds to perform 6192 merges. The serial merge version would need 6192 iterations to perform 6192 merges. In an actual test, the serial merge version took 2913 seconds to perform 6200 merges. This means that the parallel merge version performed the first 6192 merges nearly 25 times faster than the serial merge version. The last ten iterations of the parallel merge version took 2174 seconds to perform 164 merges. We estimate that the serial merge version would take roughly 250 seconds to perform those 164 merges. Thus, the serial merge version would have performed those last 164 merges better than 6 times faster than the parallel merge version did them. For this data set, it would have been most efficient to use step vP for 138 iterations (resulting in 60,037 merges), and switch to step $v^s$ for the remainder of the processing (to do the last 4,013 merges at one merge per iteration).

The parallel merge version took 4.6 hours to produce the segmentation shown in Figure 1. The serial merge version would have taken an estimated 8.4 hours to do the same number of merges. An optimal parallel merge/serial merge combination would have taken an estimated 2.4 hours. Further, such a combined implementation on an MPP-like machine with sufficient local array memory for all data and variables would take roughly 15 minutes (assuming the estimated 10 times speed-up mentioned earlier.) Clearly, the best way to implement this iterative parallel region growing approach is a parallel merge/serial merge combination on an MPP-like machine with significantly more local array memory. Within the coming year, we hope to have made such an implementation on AMT's DAP 610.

An ultimate segmentation goal would be to find the globally best image segmentation for a given similarity criterion and number of regions. Our iterative parallel region growing approach can only approximate this desired result. Fortunately, for many applications an approximate result may be sufficient. Nevertheless, we are seeking improvements to our SCC algorithm. One such improvement would be to allow pixels split out of regions when appropriate. We eventually plan to explore neural network optimization as an approach that could actually produce the globally best image segmentation.

## REFERENCES

1. B. J. Schachter, L. S. Davis and A. Rosenfeld, "Some Experiments in Image Segmentation by Clustering of Local Feature Values," Pattern Recognition, Vol. 11, No. 1, pp. 19-28, 1979 .
2. M. Manohar, H. K. Ramapriyan and J. P. Strong, "Parallel Algorithms for Determining Motion Vectors in Ice Floe Images by Matching Edge Features," Proc. of the 2nd Symposium on the Frontiers of Massively Parallel Computation, Fairfax, VA, Oct. 10-12, 1988.
3. J. C. Tilton and H. K. Ramapriyan, "Data Compression Experiments with Landsat Thematic Mapper and Nimbus-7 Coastal Zone Color Scanner Data," Proc. of the Scientific Data Compression Workshop, Snowbird, UT, May 3-5, 1988.
4. K. E. Batcher, "Design of a Massively Parallel Processor," IEEE Trans. on Computers, Vol. C-29, No. 3, pp. 836-840, 1980.
5. T. M. Chen, D. H. Staelin and R. B. Arps, "Information Content Analysis of Landsat Image Data for Compression," IEEE Trans. on Geoscience and Remote Sensing, Vol. GE-25, No. 4, pp. 499-501, 1987.

# THE PERFORMANCE OF THE IMAGE UNDERSTANDING ARCHITECTURE ON THE DARPA INTEGRATED IMAGE UNDERSTANDING BENCHMARK

Dr. Charles C. Weems

Computer and Information Science Department
University of Massachusetts
Amherst, MA 01003

## ABSTRACT

This paper will present an overview of the DARPA Integrated Image Understanding Benchmark and of the Image Understanding Architecture. It will then discuss the performance of the architecture on various portions of the benchmark. The article will conclude with a discussion of lessons that have been learned as a result of the experience, both in terms of the design of the architecture and of the benchmark.

## THE DARPA INTEGRATED IMAGE UNDERSTANDING BENCHMARK

The need for a computer vision benchmark for parallel architectures has become apparent as researchers from the fields of computer vision and computer architecture have had increasing contact over the last several years. Motion sequences at moderate resolution (512 × 512) and typical frame rate (30 frames/sec) in color (3 bytes) involves about 23.5 Mbytes of data per second. The amount of computation required for dynamic scene interpretation including the labeling of objects, surface/volume reconstruction and motion analysis is difficult to estimate; however, for many applications computational power in the range of 100 billion instructions per second, plus or minus two orders of magnitude, is probably required. Thus, vision has become a subject of major interest to computer architects.

Unfortunately, the evaluation of progress in vision architectures has been difficult [Duff, 1986]. There are now quite a few interesting machines, both existing and proposed, that may be effective for at least part of the vision problem. However, computer vision transcends a wide range of representations and forms of processing. In addition, despite exciting advances in many of the subtopics of computer vision, there is currently no consensus in the research community on a unified approach to vision. There are many competing approaches and a great deal of debate has persisted. Nonetheless, it is clear that there is a need to address some of the vision/architecture issues in a form that will allow scientific insight and progress in hardware development.

Recent attempts at defining a vision benchmark include the Abingdon Cross problem [Preston, 1986], defined at the 1982 Multicomputer Workshop in Abingdon England, and the Tanque Verde benchmark suite [Uhr,1986] defined at the 1984 Multicomputer Workshop in Tucson Arizona. The most recent attempt at constructing a benchmark for vision emerged from the DARPA Image Understanding community, where a set of ten vision tasks were defined. These were: Gaussian convolution, zero crossing detection and output of border lists, connected components labeling, Hough transform, convex hull, Voronoi diagram, minimal spanning tree, visibility of vertices in a 3-D model, minimum cost path, and subgraph isomorphism. A meeting was held in November, 1986, in Washington to compare the results of programming, simulating, or estimating the performance of a number of machines on the individual benchmark tasks. The results [Rosenfeld, 1987] were both interesting and thoroughly confusing. The data sets were only loosely specified, leading some to groups to report average performance while others reported worst case performance; different groups used different algorithms; some used 32-bit floating point arithmetic while others used 16-bit integer arithmetic, etc. These results must be interpreted with extreme care.

The DARPA Integrated Image Understanding Benchmark Exercise is an outgrowth of the first DARPA Image Understanding Benchmark Workshop. The new benchmark has been designed by the University of Massachusetts and the University of Maryland to address the need for an integrated vision benchmark that transcends several different representations and forms of processing that are typical of complex vision applications. The specification for the benchmark is now being widely distributed [Weems, 1988a], and the results from participating groups are to be gathered at a workshop in October of 1988.

This benchmark task suite involves model-based object recognition given images from intensity and range sensors. It is our intention that the test images be designed so that neither, by itself, is sufficient to form a complete match. The object to be recognized is a collection of rectangles of various sizes, brightnesses, two-dimensional orientations and depths. It can be thought of as a semi-rigid mobile consisting of suspended rectangles floating in space with fixed

CH2649-2/89/0000/0361$01.00 © 1988 IEEE

spatial relationships. To simplify the task, each rectangle is oriented normal to the Z axis (the viewing axis) and the image is constructed under orthographic projection. A set of models is provided, from which the best matching model must be selected. The models are approximate in the sense that the sizes, orientations, and depths of the rectangles as well as their spatial relationships may vary and are constrained to within some tolerances.

The rectangles that make up the object are interspersed with additional extraneous rectangles in the scene from which the two images are taken. These additional rectangles may occlude portions of the mobile object, and some of the adjacent rectangles in the scene may have very similar brightnesses. The image from the range sensor is further degraded by the addition of Gaussian noise.

The low-level operations on the intensity image consist of identifying connected components and finding the corners of each connected component based on a K-curvature operation. The initial processing of the intensity image also includes an intermediate level grouping operation that consists of creating good hypotheses for rectangles from the lists of corners around connected components in the image. The result of the initial intensity image processing is thus a set of connected component tokens. The only feature that is extracted from each connected component as a whole is its intensity. However, each component region has associated with it a list of the corners that were extracted from its boundary.

The low-level operations on the depth image consist of smoothing via median filtering, computing the magnitude of the gradient, and thresholding the gradient magnitude. The result of the initial depth image processing is an image array that represents points in the depth data that have large gradient magnitudes. The smoothed depth image is also used in later stages of processing.

Intermediate level processing starts with bottom-up grouping of right-angle corners, in component tokens, in order to generate rectangle hypotheses. The resulting candidate rectangles form the basis of the initial model graph matching operation. The intermediate level operation on the depth image is a top-down directed search for expected rectangles, incorporating a spatially local Hough transform with model-constrained ranges on the parameters for each rectangle. The high level operations are first, constrained subgraph-to-subgraph matching to choose and orient the models to be matched; and second, top-down control of probes into the depth and intensity images to find and fix the parameters of the rectangles that are required to fill out the chosen models. As a concluding step, an image is produced that represents the single best model match as an overlay with the original intensity image.

The goal of the first graph match step is to attempt to establish the most likely positions and orientations of the modeled objects in the image and possibly to eliminate some of the models from further consideration for matching. Since only some of the model rectangles will have been extracted from the intensity image, portions of each model will have no match. The unmatched portions of a graph model are used to focus attention in the depth image so that additional localized features can be extracted and the model can be extended through the use of context. This match extension step is further divided into three parts that are repeated for each model rectangle: model directed rectangle detection, rectangle depth and intensity verification, and model update.

The scientific gain that should result from this exercise is a better understanding of vision architecture requirements, and the performance bottlenecks in different classes of machines, so that the needs of vision processing can be better addressed in the next generation of architectures. We also expect to learn even more about the design of benchmarks for computer vision.

## THE IMAGE UNDERSTANDING ARCHITECTURE

The Image Understanding Architecture (IUA) is a massively parallel, multi-level system for supporting real-time image understanding applications and research in knowledge-based computer vision. The design of the IUA is motivated by the architectural requirements for integrated real-time vision in terms of the type of processing element, control of processing, and communication between processing elements. (Figure 1)

The IUA integrates parallel processors operating simultaneously at three levels of computational granularity in a tightly-coupled architecture. It consists of three different, tightly coupled parallel processors. These are the Content Addressable Array Parallel Processor (CAAPP)[1] at the low level, the Intermediate Communications Associative Processor (ICAP) at the intermediate level, and the Symbolic Processing Array (SPA) at the high level (Figure 2). The CAAPP and ICAP levels are controlled by a dedicated Array Control Unit (ACU) that takes its directions from the SPA level. Each level of the IUA is a parallel processor that is distinctly different from the other two levels, in order to best meet the processing needs at each of the corresponding levels of abstraction in the interpretation process. Communication between levels takes place via parallel data and control paths. The processing elements within each level can also communicate with each other in parallel, via a different mechanism at each level that is designed to meet the specific communication needs of each level of abstraction.

An associative processing paradigm has been utilized as the principle control mechanism at the low and intermediate levels. It provides a simple yet general means of

[1] The term "content-addressable" is a synonym for "associative" and is an alternate term that now is not as widely used as it was when some of our work began [Foster, 1976, Weems, 1984a.]

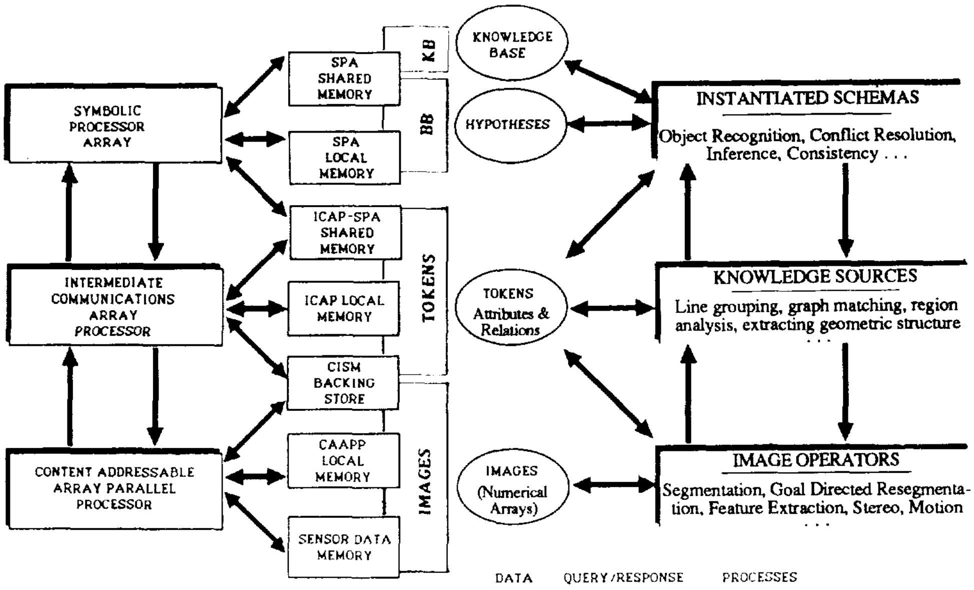

SYMBOLIC PROCESSOR ARRAY
INTERMEDIATE COMMUNICATIONS ARRAY PROCESSOR
CONTENT ADDRESSABLE ARRAY PARALLEL PROCESSOR
SPA SHARED MEMORY
SPA LOCAL MEMORY
ICAP-SPA SHARED MEMORY
ICAP LOCAL MEMORY
CISM BACKING STORE
CAAPP LOCAL MEMORY
SENSOR DATA MEMORY
KB
BB
TOKENS
IMAGES
KNOWLEDGE BASE
HYPOTHESES
TOKENS Attributes & Relations
IMAGES (Numerical Arrays)
INSTANTIATED SCHEMAS
Object Recognition, Conflict Resolution, Inference, Consistency . . .
KNOWLEDGE SOURCES
Line grouping, graph matching, region analysis, extracting geometric structure . . .
IMAGE OPERATORS
Segmentation, Goal Directed Resegmentation, Feature Extraction, Stereo, Motion . . .
DATA
QUERY/RESPONSE
PROCESSES

Figure 1: IUA Compared to UMass VISIONS System

managing massive parallelism, through rapid responses to queries involving partial matches of processor memory to broadcast values. This has been enhanced with hardware operations that provide for global broadcast, local compare, Some/None response, responder count, and single responder select. A 1/64th scale proof-of-concept prototype is currently under construction by the University of Massachusetts and Hughes Research Laboratories. The prototype will contain 4096 1-bit SIMD processors at the low level, 64 16-bit microprocessors at the intermediate level, and a single symbolic processor at the high level. It is scheduled for completion late in 1988. [Weems, 1988b] provides a more extensive discussion of the architecture.

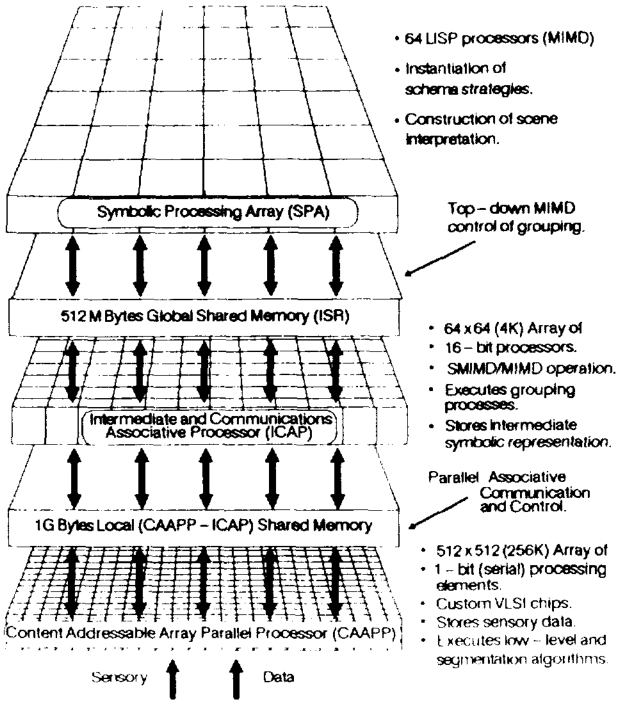

• 64 LISP processors (MIMD)
• Instantiation of schema strategies.
• Construction of scene interpretation.
Symbolic Processing Array (SPA)
Top-down MIMD control of grouping.
512 M Bytes Global Shared Memory (ISR)
• 64 x 64 (4K) Array of
• 16-bit processors.
• SMIMD/MIMD operation.
• Executes grouping processes.
• Stores intermediate symbolic representation.
Intermediate and Communications Associative Processor (ICAP)
Parallel Associative Communication and Control.
1 G Bytes Local (CAAPP - ICAP) Shared Memory
• 512 x 512 (256K) Array of
• 1-bit (serial) processing elements.
• Custom VLSI chips.
• Stores sensory data.
• Executes low-level and segmentation algorithms.
Content Addressable Array Parallel Processor (CAAPP)
Sensory
Data

Figure 2: IUA Overview

## IUA BENCHMARK PERFORMANCE

As of this writing, only part of the benchmark has been programmed on the IUA simulators. The deadline for completion of the benchmark, for all participating groups, is October of 1988. Preliminary results for the IUA are listed in Table 1. For comparison purposes, the table also shows the results for a Sun-3/160 workstation running the benchmark sequentially.

| Task | IUA | Sun 3/160 |
|---|---|---|
| Overhead | – | 6.28 |
| Connected Components | 0.0000500 | 28.26 |
| Find Rectangles | 0.0070163 | 6.28 |
| Median Filter | 0.0005625 | 247.90 |
| Gradient Magnitude | 0.0026000 | 136.68 |
| Initial Match | 0.0076055 | 24.40 |
| Complete Match | 0.0488545 | 326.04 |
| Output Result | – | 24.80 |
| Total Time | 0.0727513 | 800.64 |

–depends on execution of entire benchmark

Table 1:
Integrated IU Benchmark Results
(All times in seconds)

## LESSONS LEARNED

With regard to the IUA itself, these preliminary results indicate outstanding performance on the lower-level tasks, especially with the 8-bit intensity data. Floating point performance, while good, could be improved with additional hardware support in the processing elements of the CAAPP. Of greater interest will be the performance of the IUA on the intermediate level processing and top-down control aspects of the benchmark (which are currently being programmed), as these portions of the architecture have not been as extensively exercised as the low-level processor. Our development to date of the graph matching portion of the task indicates that it greatly underutilizs the ICAP. We estimate that the size of the model-base could be increased by a factor of 400 without signficantly increasing the execution time. This is quite a reasonable result, since a real vision task is likely to have a much larger and more complicated set of models to work with.

In terms of the benchmark, we have found that solving even a task as simple as this can present serious difficulties. The benchmark code is significantly larger and more complex than that of any preceding vision benchmark, which was the inevitable result of attempting to capture some of the complexity of a real vision task. Unfortunately, even this complex benchmark does little more than touch upon true high level processing, because there has yet to be any real consensus among the members of the image understanding research community as to what constitutes typical high level tasks. Once some consensus is achieved, it is likely that this benchmark will be extended to address further high level processing, so that participants can make use of the large body of code already developed. Our initial sequential testing of the benchmark has also revealed that just three of the tasks account for eighty-nine percent of the total time. The match extension process (which includes all of the top-down probing of the image data) is the most time consuming. The second most costly task is the median filter that is applied to the floating-point depth image. The computation of the gradient magnitude in the depth image is also a significant contributor to the total time. The other tasks are minor in comparison, which serves to emphasize that the total execution time is just one part of the benchmark result, and must be considered carefully in the context of all the other data that are obtained.

## ACKNOWLEDGEMENTS

This work was supported in part by the Advanced Research Projects Agency of the Department of Defense under contract number DACA76-86-C-0015, monitored by the Engineer Topographic Laboratory.

The author would like to thank Edward Riseman, Allen Hanson, Azriel Rosenfeld, Larry Davis, and Todd Kushner for their help in developing the IU Benchmark; Poornima Balasubramaniam, Sunit Bhalla, Chris Connoly, John Dolan, Martin Herbordt, Michael Scudder, Lance Williams, and James Burrill for their efforts in designing, programming, and debugging solutions to the benchmark; Steven Levitan, Deepak Rana, David Shu, Gregory Nash, and Caxton Foster for their many contributions to the development of the IUA.

## REFERENCES

[Draper, 1988] Draper, B.A., Collins, R.T., Brolio, J., Griffith, J., Hanson, A.R., Riseman, E.M., The Schema System: Knowledge Based Vision (in preparation).

[Duff, 1986] Duff, M.J.B., "How Not to Benchmark Image Processors", in Evaluation of Multicomputers for Image Processing (Proceedings of the 1984 Multicomputer Workshop, Tucson, AZ), L. Uhr, K. Preston Jr., S. Levialdi, M.J.B. Duff editors, Academic Press, Orlando, FL, 1986, pp. 3-12.

[Foster, 1976] Foster, C.C., "Content Addressable Parallel Processors", New York: Van Nostrand Reinhold, 1976.

[Preston, 1986] Preston, Jr., K., "Benchmark Results: The Abingdon Cross", in Evaluation of Multicompuers for Image Processing (Proceedings of the 1984 Multicomputer Workshop, Tucson, AZ), L. Uhr., K. Preston, Jr., S. Levialdi, M.J.B. Duff editors, Academic Press, Orlando, FL, 1986, pp. 23-54.

[Rosenfeld, 1987] Rosenfeld, A.R., "A Report on the DARPA Image Understanding Architectures Workshop", Proceedings of the 1987 DARPA Image Understanding Workshop, February 1987, Los Angeles, CA, pp. 298-302, Morgan Kaufmann Publishers, Los Altos, CA, 1987.

[Uhr, 1986] Uhr, L., Preston Jr., K., Levialdi, S., Duff, M.J.B., "Preface", in Evaluation of Multicomputers for Image Processing", in Evaluation of Multicomputers for Image Processing (Proceedings of the 1984 Multicomputer Workshop, Tucson, AZ), L. Uhr, K. Preston Jr., S. Levialdi, M.J.B. Duff editors, Academic Press, Orlando, FL, 1986, pp. ix-xiv.

[Weems, 1984a] Weems, C.C., "Image Processing on a Content Addressable Array Parallel Processor", Ph.D. Dissertation Computer and Information Science Department, also, COINS Technical Report 84-14, University of Massachusetts at Amherst, September, 1984.

[Weems, 1988a] Weems, C.C., Riseman, E., Hanson, A., "A Computer Vision Benchmark for Parallel Processing Systems", Proceedings of the Third International Conference on Supercomputing, Volume III: Supercomputing Design: Hardware & Software, Boston, MA, 1988.

[Weems, 1988b] Weems, C.C., "Some Example Algorithms for the CAAPP and ICAP Levels of the Image Understanding Architecture", Proceedings of the Third International Conference on Supercomputing, Volume III: Supercomputing Design: Hardware & Software, Boston, MA, 1988.

[Weems, 1988] Weiss, R., "On the Organization and Application of Shape in Recognition", AAAI Symposium, Stanford University, March 22-24, 1988.

# SECTION III: ARCHITECTURES

## Part 1: Oral Presentations

# THE MARTIN MARIETTA ADVANCED SYSTOLIC ARRAY PROCESSOR

A. Haug R. Graybill

Martin Marietta Aero & Naval Systems
Baltimore, Maryland

## ABSTRACT

An overview of the Martin Marietta Advanced Systolic Array Processor (ASAP$^{TM}$) will be presented. The modular design of the ASAP$^{TM}$ allows a user to build as large a processor as desired by concatenating multiple, independently programmable unit arrays. Each unit array, housed on a single VME card, is composed of a two-dimensional grid of 256 bit serial nearest-neighbor connected processors (SIMD), a local controller, and all I/O support. Multiple unit arrays can be configured in any configuration desired; all SIMD, groups of SIMD arrays with MIMD between groups or all MIMD.

Keywords: ASAP$^{TM}$, Massively Parallel Processor, Beamforming, Data Fusion, Associative Memory, Image Processing, Neural Processing, Signal Processing.

## INTRODUCTION

Martin Marietta has a corporate-wide commitment to develop and use advanced computer architectures for military systems. At Aero & Naval Systems, in Baltimore, our efforts have concentrated on coarse and fine grain systolic architectures. To this end we have put in place a real-time signal and data processing laboratory that serves as an architectural testbed for all ongoing research in this area. This laboratory contains a number of high-speed devices for I/O, memory, and computation. The computational devices include a vector processor, a 1 Gigaflop linear coarse grain systolic processor, and several Martin Marietta-developed two-dimensional fine grain systolic processors. Significant expansion of this laboratory is planned for 1989.

Corporate commitment to the development of a fine grain bit serial massively parallel processor began in 1981 when the Geometric Arithmetic Parallel Processor (GAPP) was designed for image processing applications. At Baltimore, the GAPP technology was transitioned to a new application area with the development of a Programmable Sonar Beamformer (PSB). Based on the experience gained on the PSB, it was realized that a processor with more powerful computational capabilities would be useful for beamforming, so a new VLSI chip, the Advanced Systolic Array Processor (ASAP$^{TM}$) was developed in 1986. In 1987 the ASAP$^{TM}$ chip was redesigned to increase capabilities still further, resulting in our current fourth generation chip, ASAP$^{TM}$II.

To meet a diverse set a potential customer requirements, we designed a unit array processor that was completely modular at the card level. To achieve the desired packaging requirements, three additional VLSI chips were designed and developed. To demonstrate our processor capabilities, several generations of full-up processors that use multiple unit arrays have been built. A variety of applications are being demonstrated including conventional and adaptive beamforming, signal processing, high level image processing, data fusion, parallel inference engines, and neural net simulators.

## PRODUCT DESCRIPTION

### ASAP$^{TM}$ Chip Significant Features

The significant features of the current generation ASAP$^{TM}$ chip are shown in Table 1. The processing cells are connected via nearest neighbor paths with a broadcast bus for each north-south column. Our processor design provides for much improved add times, an equalization of multiply times relative to adds, and a significant floating point capability. All of these capabilities were traded-off against chip real estate resulting in an ASAP$^{TM}$ chip with 16 processing cells per chip. Each processing cell has a 1200 bit local memory with additional external memory capability.

Table 1. Simplified features of the ASAP$^{TM}$ II VLSI device

- Complex, High-Performance Cells
- Low to Medium Chip Density
- Balanced Ratio of Computational Power to Memory
- External Memory and I/O Cell Ports
- Floating Point
- Multiplier Accelerator (10X)
- MIL-STD Ada
- VLSI Support Logic
- Software Development Tools

### Unit Array Features

To achieve the desired single card packaging for the ASAP$^{TM}$ unit array, three additional VLSI chips were developed. In order to understand the function of these chips, we first define the unit array concept, shown in Figure 1. As shown, a 4 x 4 array of ASAP$^{TM}$ chips provide a unit array of 256 processing cells. High speed data is normally passed into the array from external parallel buses via the serial-to-parallel I/O (SPIO) interface. This interface converts external 32 bit-parallel words to bit serial data and visa-versa. Data then passes through a 16 x 16 cross point switch (XPOINT) into a

CH2649-2/89/0000/0367$01.00 © 1988 IEEE

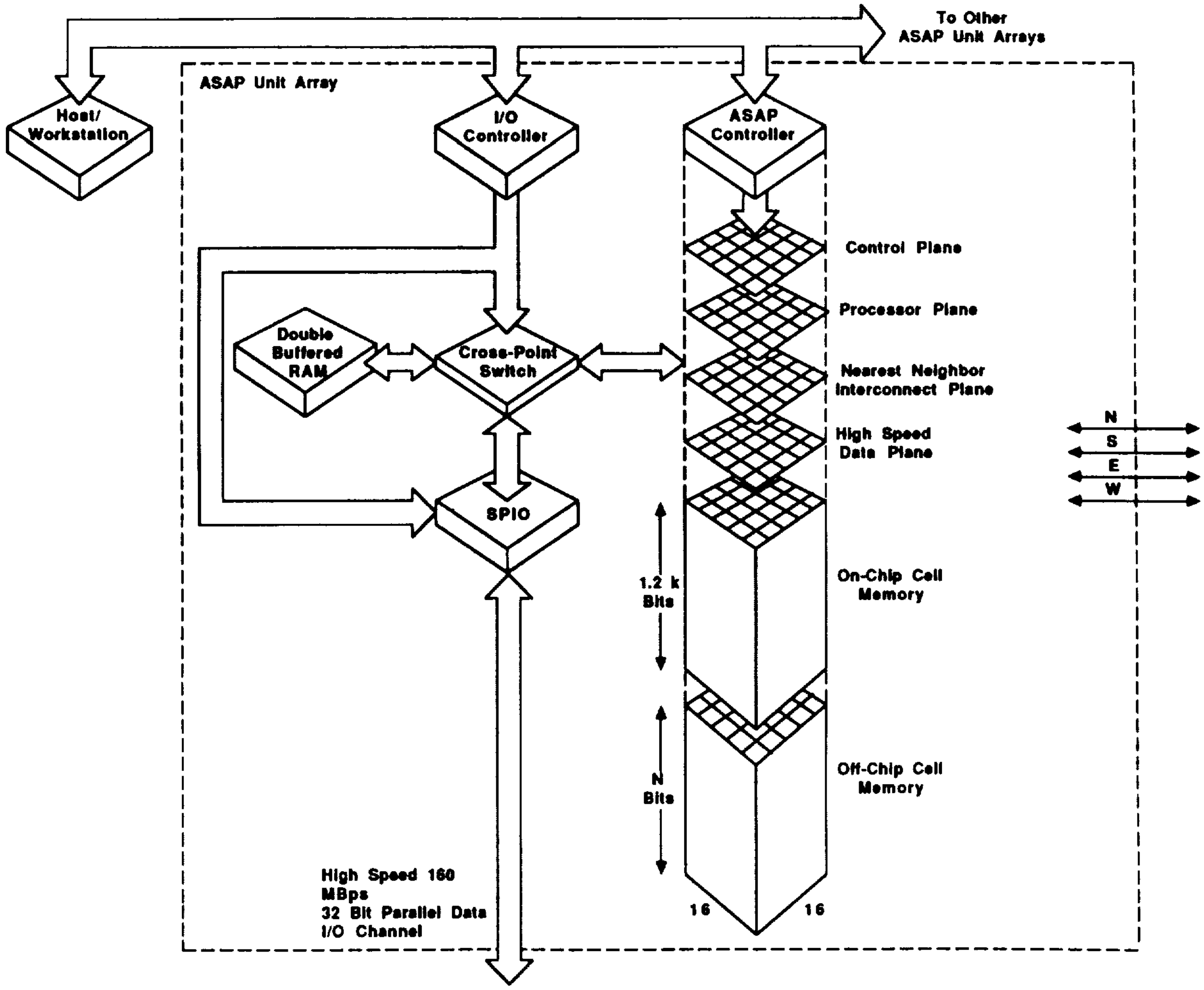


Figure 1. ASAP$^{TM}$ unit array functional design

double buffered RAM. Simultaneously, data can be passed from the RAM into the ASAP$^{TM}$ cells through the cross point switch. An alternate data path allows data to be passed to adjacent unit arrays through the north/south/east/west serial buses.

I/O and ASAP$^{TM}$ array operations are controlled by the I/O controllers and the ASAP$^{TM}$ controller, respectively. A simple AMD 2910 based processor chip (IOCTLR) is used for both the I/O and ASAP$^{TM}$ controllers. For compactness and reliability, the SPIO, XPOINT, and IOCTLR were developed as VLSI devices. In addition to the above functions, the XPOINT device provides for a unique software controlled arbitrary interconnection among the ASAP$^{TM}$ processing cells.

The unit array has been functionally implemented with a VME interface, as shown in Figure 2. Figure 3 shows the card layout for a 9 U VME card. This VME unit array design provides the following features:

- 16 x 16 array of processing cells (16 ASAP$^{TM}$ chips)
- Dual I/O controllers
- ASAP$^{TM}$ controller
- High speed data and control interfaces
- 20 MHz operation
- Independent I/O speed operation
- A 24, D32, VME interface.

## Multiple Unit Array Configuration

The ASAP$^{TM}$ unit array has been designed to allow multiple unit arrays to be interconnected, as shown in Figure 4. Since each unit array has only a single ASAP$^{TM}$ controller, all processing cells on the array operate in a SIMD fashion. However, because each unit array has its own controller, operation from unit array to unit array can be MIMD or SIMD.

ORIGINAL PAGE IS
OF POOR QUALITY

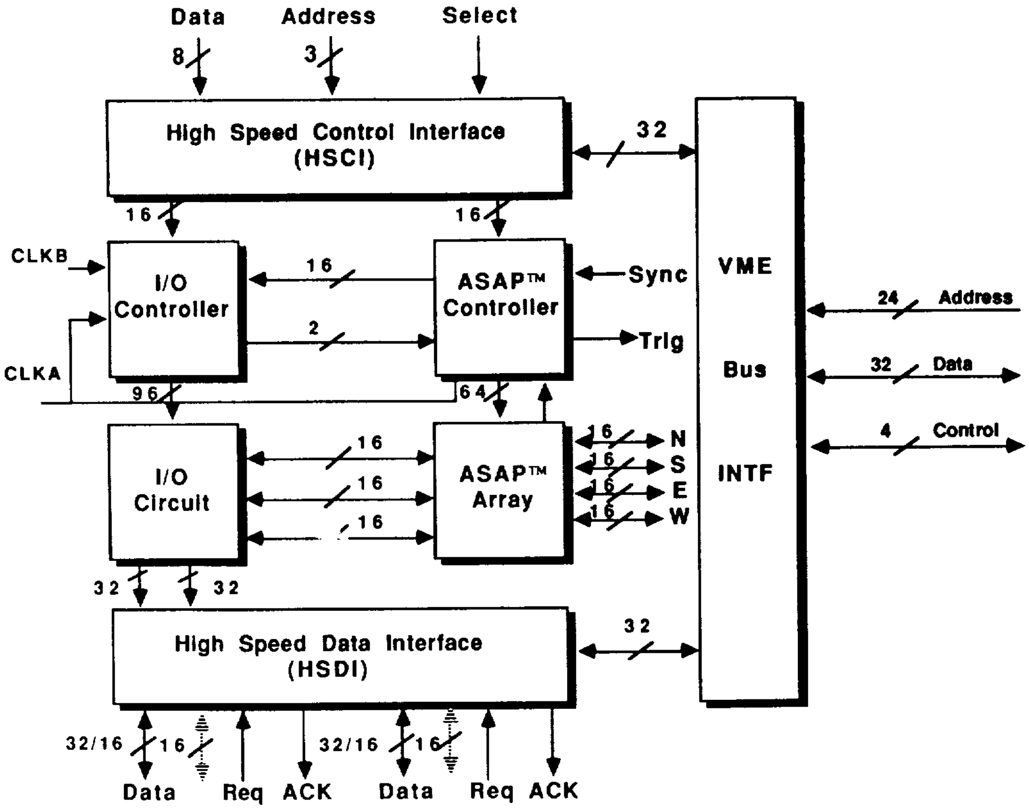


Figure 2. VME based unit array functional design

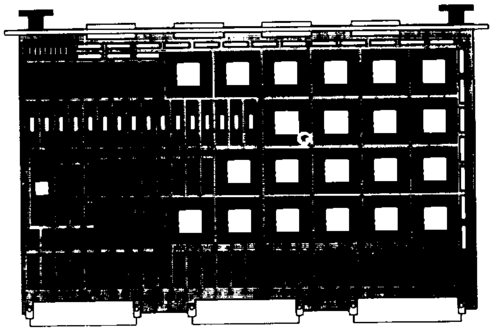


Figure 3. Card layout for VME based ASAP™ unit array module

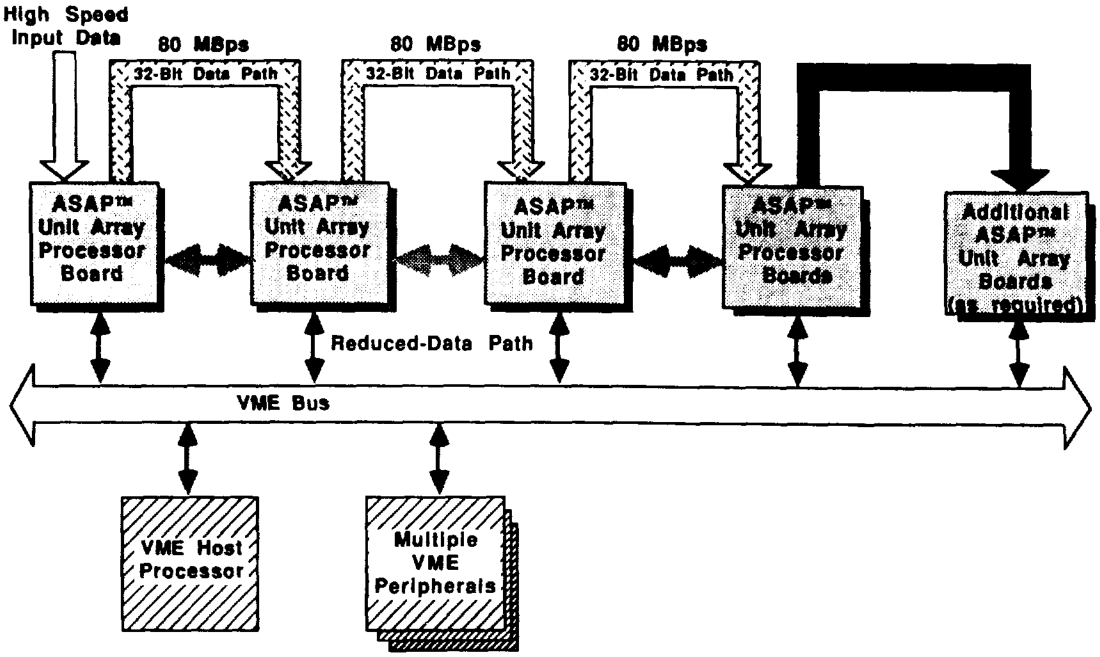

High Speed Input Data
80 MBps
32-Bit Data Path
80 MBps
32-Bit Data Path
80 MBps
32-Bit Data Path
ASAP™ Unit Array Processor Board
ASAP™ Unit Array Processor Board
ASAP™ Unit Array Processor Board
ASAP™ Unit Array Processor Boards
Additional ASAP™ Unit Array Boards (as required)
Reduced-Data Path
VME Bus
VME Host Processor
Multiple VME Peripherals

Figure 4. Multiple unit array interconnectability

### Software

Since our primary customers are within the DoD, we have chosen Ada as our initial high-level language. Figure 5 shows the support software developed to date. All of the software is currently hosted on VAX computers, but is portable to other systems since this is designed into Ada. The procedure for applications software development is as follows. After an Ada program is developed it is executed within the Ada Code Generator. This generates the microcode that can then be downloaded to an Ada ASAP™ simulator or the target hardware. The Ada ASAP™ Run-Time Debugger provides a complete software debugging capability with both the simulator and target hardware. The Hardware Test Software provides a complete set of test vectors down to the device level. This test software was used to certify all VLSI devices as they arrived from the vendor showing first pass success on each device with a very short turn-around for each certification. In addition to the above software, a set of mathematical and engineering library functions, as well as numerous applications programs and a graphic/software development tool, are under development.

### ASAP™ Products

To date we have developed several products which include:

- SMSP-20M Single Module Systolic Processor, 20MHz, Multibus
- SMSP-20V Single Module Systolic processor, 20MHz, VME
- SMSP-20S Single Module Systolic Processor, 20MHz, VME/SUN.

The baseline versions of each of these products contain the equivalent of 4 unit arrays (1024 processors) and can be expanded with additional unit arrays. The existing units are being used for a number of capability demonstrations and extensive library and applications software development is planned for 1989.

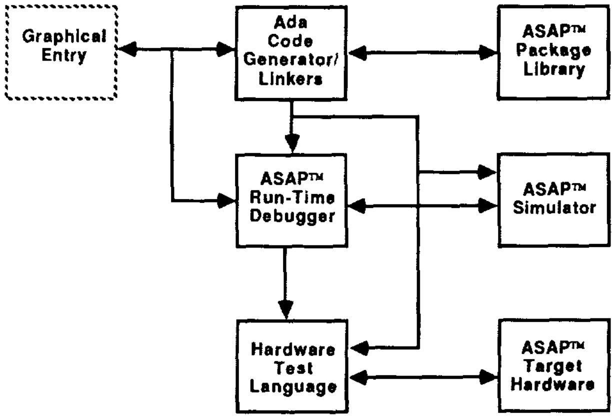


Figure 5. ASAP™ Ada software development tools

### PERFORMANCE COMPARISONS

A comparison of peak performance for five massively parallel bit serial machines are shown in Figure 6 and 7 and in Table 2, for add, multiply, and 32 bit floating point operations, respectively. Data for the MPP, DAP, and CM-2 were derived from data found in References 1, 2, and 3, respectively. In all cases, performance has been normalized to that of 1024 processors. It is obvious from this data that the ASAP™ processor provides a significant performance improvement over all others. In fact, in a package approximately the size of the current MPP, we could provide in excess of 1.3 Terra-ops for single bit operations.

Table 2. Performance comparison for 32 bit floating point operations normalized to 1024 processors (Mflops)

| | ASAP™ | MPP | DAP | GAPP | CM-2 |
|---|---|---|---|---|---|
| Adds | 91 | 29.3 | 12 | 0.54 | 0.07 |
| Multiplies | 170 | 18.2 | 7 | - | 0.07 |

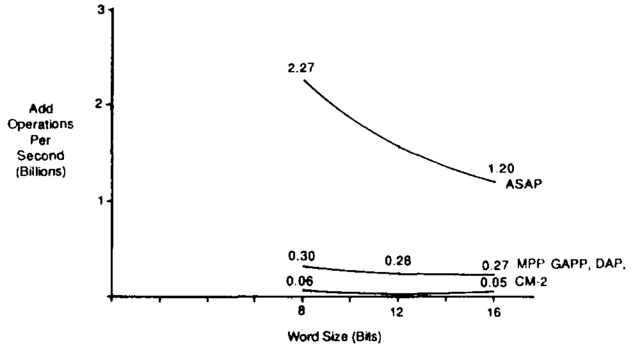


Figure 6. Performance comparison for add operations normalized to 1024 processors

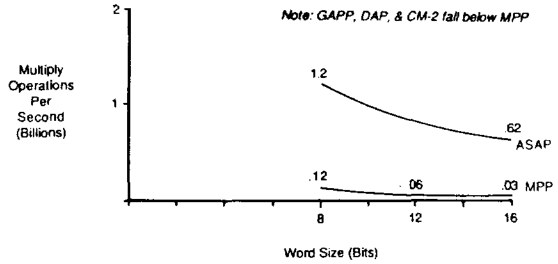


Figure 7. Performance comparison for multiply operations normalized to 1024 processors

## APPLICATIONS

Several sonar signal processing applications have been demonstrated to date. The first of these is time domain delay and sum beamforming for linear, planar, spherical, and volumetric receive arrays with data words ranging from one to twenty bits fixed point, and twenty four and thirty two bits floating point. In 1989 we will extend this application to time and frequency domain adaptive beamforming. Conventional digital signal processing has also been demonstrated in 1988. Using ASAP$^{TM}$ to generate simulated hydrophone data for linear and spherical sonar arrays is planned for early 1989.

A multitude of additional military applications are scheduled for demonstration in 1989. In the sensor data fusion area we are planning a Data Fusion Workbench. This workbench will contain multiple groups of unit arrays with different memory configurations on which we will demonstrate data association performed on a parallel processor, a parallel implementation of Kalman filtering, adaptive hypothesis scoring based on changing sensor modes and environment, and an intelligent "rule-based" hypothesis pruning parallel inference engine. Each of these will be implemented in a separate "processor" or group of unit arrays, with overall control residing in a SUN workstation.

Also in 1989, we are planning several demonstrations in the high-level 3-D image understanding arena by using an ASAP$^{TM}$ workstation as a Conformal Image Fusion Processor (CIP$^{TM}$). Finally, we hope to demonstrate a sonar classifier using an ASAP$^{TM}$ workstation as a Systolic Neural-type Array Processor (SNAP$^{TM}$). Table 3 presents a comparison of the performance potential of an ASAP$^{TM}$ workstation configured as a SNAP$^{TM}$ relative to existing neural array processors.

## PLANNED ASAP$^{TM}$ ENHANCEMENTS

Over the next few years several improvements are planned for the ASAP$^{TM}$ chip. ASAP$^{TM}$III should provide a 400 percent increase in performance relative to ASAP$^{TM}$II, and ASAP$^{TM}$IV should provide a 1500 percent increase.

Extensive development and applications software is also planned for the next few years. This includes libraries of mathematical and, signal processing functions as well as software for the specific applications mentioned above.

Planned improvements to the total system include the development of MIL-SPEC VME based processor, as well as Navy Standard Electronics Module (SEM) based processors. Preliminary designs for these products have been developed and will be implemented as soon as funding permits.

## SUMMARY

Martin Marietta has introduced the most powerful bit serial massively parallel processor available to date. It provides significant computational power in a SUN based workstation environment with larger machines available. Extensive software is under development, with a substantial software environment already available. And although the current version of ASAP$^{TM}$ can be configured to provide a terra-op of operation, within a few years this performance will be available in an ASAP$^{TM}$ processor packaged in a relatively small volume. As the ASAP$^{TM}$ products evolve, upward compatibility will be maintained for all software, as was the case in our transition from ASAP$^{TM}$I to ASAP$^{TM}$II.

## REFERENCES

1. Batcher, K.E., "The Massively Parallel Processor System Overview" *The Massively Parallel Processor*, edited by J.L. Potter, MIT Press., 1985.

2. Reddaway, S.F., "Signal Processing on a Processor Array". Course 14, ICL, Defense Technology Center, Eskdale Road, Winnersh, Workingham, Berk, RG11 5TT, UK, 1987.

3. "Connection Machine Model CM-2 Technical Summary", *Thinking Machines Technical Report*, HA 87-4, April 1987.

Table 3. Neurocomputer performance parameters comparison (preliminary)

| Company Corporation | Nestor, Inc. | SAIC | TRW | HNC | Martin Marietta Aero & Naval Systems |
|---|---|---|---|---|---|
| Machine | Sun/ Appollo/ PC-AT | $\Sigma$- 1™ NC W/S | $\mu$ VAX | PC-AT The HNC/ ANZA | (SNAP™) Concept Development Stage |
| Capacity<br>Processing Element | 150,000 (Simulated) | 1,000,000 (Simulated) | 250,000 (Simulated) | 30,000 (Simulated) | 1024 Dedicated (Non-Simulated-H/W) |
| Interconnects, Connections | 15,000,000 | 1,000,000 | 5,500,000 | 300,000 | 33,000,000 |
| Layers | ? | ? | 8 | ? | 32 & Expandable to (1024) |
| Speed<br>Connections/ Sec | 500,000 | 10,000,000 | 5,000,000 | 25,000 | 200,000,000 |
| CUPS* | ? | ? | 250,000 | ? | 1,333,333 |
| Utility | 2-D Pattern Match | | 2-D Pattern Match | | 3-D Pattern Match Conformal Object Manipulation |

* CUPS = Connection Updates Per Second

# THE GEOMETRIC ARITHMETIC PARALLEL PROCESSOR

Eugene L. Cloud

Martin Marietta Electronic Systems
Orlando, FL 32862–8007

## ABSTRACT

In the last decade the demand for parallel processors, particularly for image processing, has grown rapidly. Single instruction multiple data (SIMD) computers provide very high throughput, can be directly scaled to computational problems, and exhibit a tractable, readily solvable control problem.

The Martin Marietta Geometric Arithmetric Parallel Processor (GAPP™) is a massively parallel processor that contains over 10,000 processing elements (PEs) in one or more arrays of PEs. Developed to meet the requirements for installation into tactical military units, the GAPP-based processor is based on principles which result in small size, light weight, and low power.

This paper describes the GAPP processing element, the array of processors and its control, the system into which an array is embedded, the interface to external data sources and data sinks, and the software development environment. Typical applications are discussed.

KEYWORDS: massively parallel processors, high throughput computers, image processing, Geometric Arithmetic Parallel Processor (GAPP), single instruction multiple data (SIMD), full adder/subtractor (FAS).

## INTRODUCTION

Massively parallel processors provide unique, high-performance solutions to a large class of problems. In the fall of 1981, Dr. Wlodzimierz Holsztynski applied his mathematical expertise to the research and development of solutions to Martin Marietta's image processing problems.

This research resulted in the invention of the Geometric Arithmetic Parallel Processor (GAPP) and a family of derivatives. The fundamental processing requirement for this system is to provide flexible processing power despite environmental size constraints.

## BACKGROUND

The GAPP concept was first implemented as a medium scale integration (MSI) breadboard in 1982. This first system emulated a 6 x 12 cell array using programmable logic and discrete memory components to mechanize the cells. Later that year, this system demonstrated the execution of a simple pattern matching algorithm. The development of GAPP technology continued into 1983 with the commitment to develop a GAPP-based custom integrated circuit. NCR Corporation of Fort Collins, Colorado, was licensed by Martin Marietta to design and build GAPP chips. The first design was a PLA-based approach that resulted in 3 x 6 cell chips. Prior to the completion of these parts, known as GAPP I, we began to improve the design of the basic cell toward higher cell density per chip (6 x 12 cells, or 72 cells per chip). The first of these new chips, GAPP II, was delivered to Martin Marietta in lage 1984. Two chip design and fabrication cycles were completed within 2 years. These chips were fabricated in 3-micron complementary metal oxide semiconductors (CMOS) using a double metal process. NCR has continued the process improvement and is now delivering parts from a 2-micron double metal CMOS process.

During chip design and development, Martin Marietta designed and built a prototype system on company funds. This system was designed to perform real-time (30 frames per second) video processing. The video source of primary interst and focus was a forward looking infrared (FLIR) sensor. This application was chosen because the FLIR is a major product line of Martin Marietta Electronics and Missiles Group in Orlando.

The desired result from this processor was the extraction of targets from each image and the rejection of all nontargets (clutter) in a tactical military scenario. In addition, the targets were classified by type. This GAPP system was designed to be both a research tool and an example of an automatic high-speed processor. Packaged in a standard laboratory rack with considerable room for additional experimental pieces, the design included two GAPP arrays: the main array containing 41,472 processing cells or elements and the target array containing 4,608 cells. The system also contained two 29,116 micro controllers, two 68,000 single board microcomputers, and an extended MIL-STD-1750A instruction set processor designed by Martin Marietta. The GAPP II chips and the system design came together in 1985 and has since been in continuous operation.

The merit of the GAPP computation approach was well recognized within the company and additional versions of GAPP processors were authorized prior to completion of the first processor. A flightworthy helicopter system whose main array contained 51,840 processing cells was built and flown in 1986 and 1988. This system continues to serve as a test bed for various programs. We also recognized that algorithm development for GAPP-based systems would be dramatically improved by installing GAPP processors as peripherals to our VAX™-based image processing laboratories. There are currently three GAPP systems attached to VAX-based systems in support of programs at Martin Marietta. The largest GAPP systems' main array contains 82,944 processing elements. This is probably the largest array of processors ever constructed.

VAX™ is a registered trademark of Digital Equipment Corp.

Martin Marietta and NCR jointly developed a peripheral processor for the NCR Tower™ Computers (also compatible with IBM PC-ATs™ and other compatible computers). Martin Marietta expanded the development so that the basic system is also compatible with SUN 3™ systems. NCR continued its development system for their customers.

## CELL DESCRIPTION

### General

We intentionally keep the GAPP cell as simple as possible. This simplicity is the driving principle resulting from Dr. Holsztynski's work. Nothing should be included in the cell that is not involved in the computation clock cycle. This requirement keeps the cell structure small, allowing a large number of cells per chip. As shown in Figure 1, the cell consists of six active components: four 1-bit registers, a 1-bit full adder/subtractor (FAS), and 128 bits of memory. Additionally, multiplexers and data paths permit the movement of signals within the cell.

### Registers

Three of the 1-bit registers, labeled North-South (NS), East-West (EW), and carry borrow (C) are connected to the inputs of the 1-bit FAS. Additionally, the NS register output is connected as an alternate input to the NS registers in the cells that exist geometrically to the north and south of this cell. Likewise, the EW register is connected as an alternate input to the EW registers in the cells that exist geometrically to the east and west of this cell. This is the nearest neighbor orthogonal connection of the fine grid array of GAPP cells. The contents of the C register are not available outside the cell in which it exists without passing through some other register.

### Full Adder/Subtractor

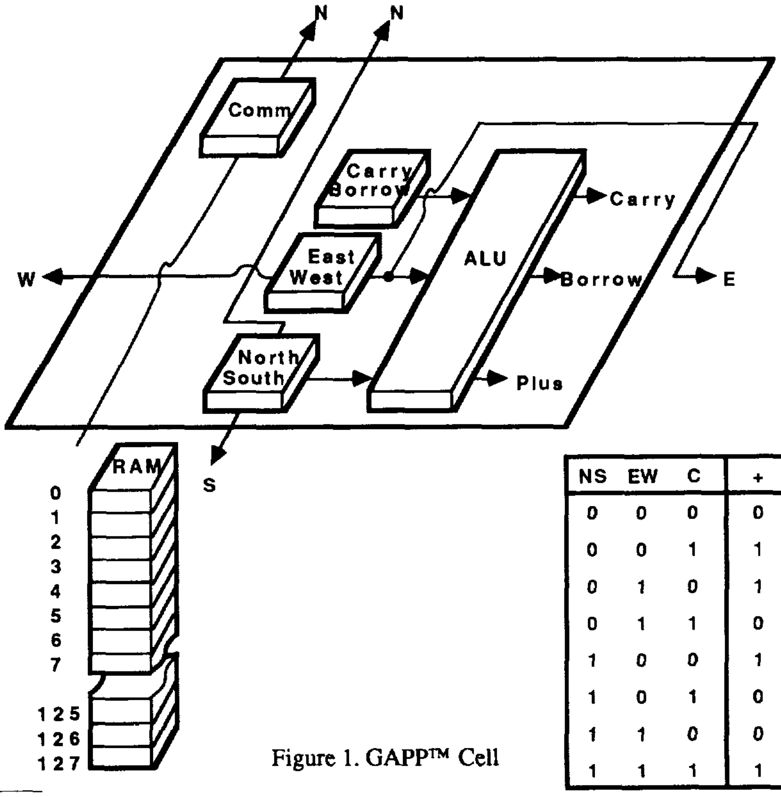
The 1-bit full adder/subtractor (FAS) is the computational element of the cell. It implements the truth table shown in Figure 1. The three 1-bit inputs come from the three previously mentioned registers, NS, EW, and C. On every clock cycle the FAS automatically produces the result prescribed by the truth table. This truth table allows the construction of arithmetic and logical results, in a bit serial fashion, that are completely general. In principle, one can perform all arithmetic and logical operations with this element. The output labels represent respectively sum, carry, and borrow (SM, CY, BW). The SM output is the exclusive OR of the three inputs. The CY output is the majority of the three inputs. The BW output is the majority of the three inputs with the NS value as the negated value in the table.

### RAM

The memory bit in each cell is organized as a 1 x 128 bit static RAM. When a 7-bit address is supplied, along with read-write signals, 1 bit of data may be read from or written into the addressed memory location in each cell. The cell RAM is only used to store data.

### Control/Clock

Each cell requires 20 bits of control/address information defining the activity required of the cell. The control section consists of 13 signals (the other 7 are associated with the RAM addressing) which primarily select data paths within the cell. There are



| NS | EW | C | + | CY | BW |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 |

Figure 1. GAPP™ Cell

NCR Tower™ Computer is a registered trademark of NCR Corp.
IBM PC-AT™ is a registered trademark of International Business Machines Corp.
Sun 3™ is a registered trademark of Sun Microsystems, Inc.

five independent parallel groupings of data paths: one associated with each portion of the cell that can store data. Thus, the RAM and registers NS. EW, CM, and C can be manipulated in parallel. Additionally, each cell must receive a clock signal. All changes of state within the cell occur synchronously with the clock.

## GAPP CHIP DESCRIPTION

### Control

The control, address, and clock signals are common among all cells on the chip. Thus, every cell performs exactly as its neighboring cell. The only difference between activities are a function of the data content within each cell's registers and RAM. These data differences are crucial because a cell or group of cells, through the proper use of algorithms, can appear to be "turned off." The cells' ability to perform logical operations makes individual cell operations practical even in an SIMD control strategy.

### Shift Register Groups

If the NS registers are pictured as in Figure 2a, they form bidirectional shift registers in the north-south direction. On a single GAPP chip, there are six sets of 12-bit NS shift registers. If the EW registers are likewise pictured as in Figure 2b, they form bidirectional shift registers in the EW direction. On a single chip there are 12 sets of 6-bit EW shift registers. Since every processing element contains one each NS register and one each EW register, then these groups of NS and EW shift registers form a geometric orthogonal arrangement, as shown in Figure 3, across groups of cells. Means are provided for the contents of the EW registers to be transferred  to the NS registers and vice versa.

Similarly, the CM registers are organized as a group of shift registers, geometrically placed in parallel with the NS shift registers. The CM registers are unidirectional (from south to north).

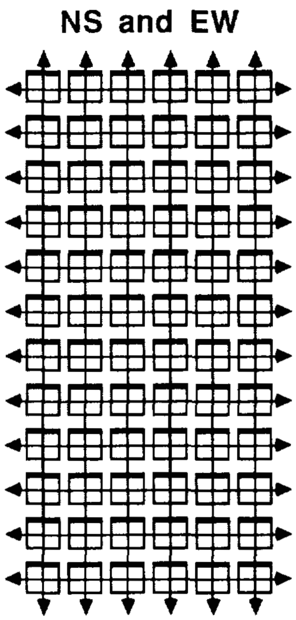


Figure 3. Orthogonal GAPP™ Register Pairs.

### Chip Performance/Mechanics

Each grouping of like-named registers, such as outputs from the FAS and RAM locations at the same address, can be thought of as planes of data (Figure 4). When an instruction is executed, every cell in the chip reacts in exactly the same way. Since each chip contains 72 cells, this has the effect of operating on a 72-bit "word" within the chip for up to five planes (instructions involving NS, EW, C, CM, and RAM) in one clock time. Usually one to three planes are moved at once.

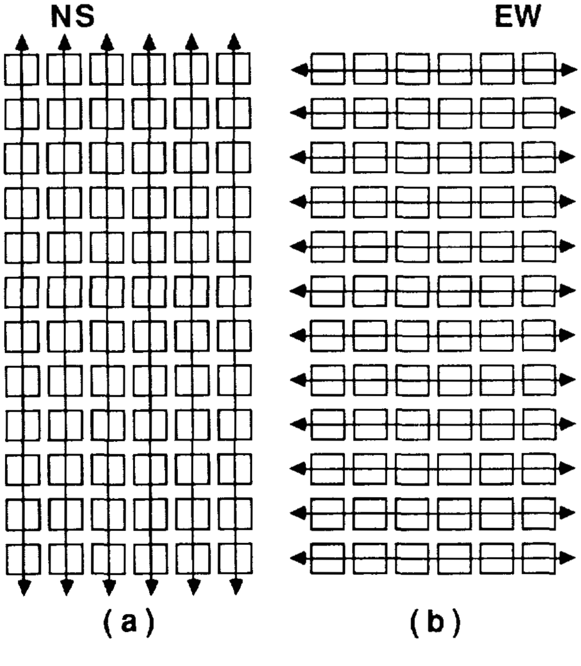


Figure 2. GAPP™ One-Bit Latches Viewed as Shift Registers

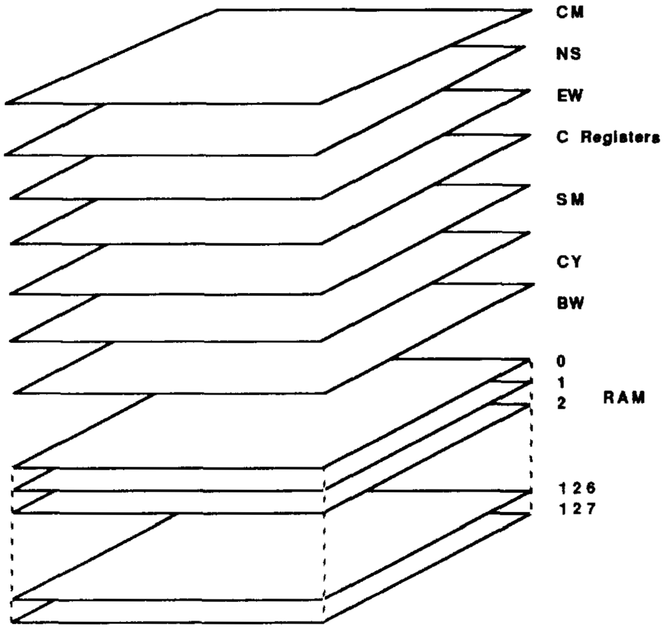


Figure 4. GAPP™ Chip Plane Names

### Chip Edges

The ends of each of the three groupings of shift registers (CM, NS, and EW) come to the edge of the chip. Both the CM and NS groups exit at the north and south edges while the EW group exits at the east and west edges. Each of these data groups may be thought of as input/output ports to the chip. In that sense, each chip has 6 ports; 2 bidirectional 6-bit ports (one at the northern andddd one at the southern edge for NS; 2 unidirectional 6-bit ports (one for output at the northern edge of CM and one for input at the southern edge of CM; and 2 bidirectional 12-bit ports (one at the western and one at the eastern edge for EW). Further, the system designer may choose to provide three simultaneously input (CMS, and E or W and N or S) and three simultaneously output (CMN, and E or W and N or S) paths on a given clock cycle. At a 10 MHz clock, each chip has an input/output bandwidth of 60 MBytes/second, 30 MBytes/second input, and 30 MBytes/second output.

The data signals are deliberately pinned out of the chip package at four mechanical edges, providing relatively easy printed circuit board layout.

### Chip Performance/Mechanics

Consider the following: a cell requires 25 clocks to perform an 8-bit add ($3n+1$, where $n$ is the number of bits in each operand to be added). Each chip can be clocked at 10 MHz frequency. At this 100 nanosecond rate, each cell can perform 400,000 8-bit adds per second. Each chip contains 72 cells each performing their add: thus the chip throughput is equivalent to 28.8 million 8-bit adds per second. The 8-bit add executes in 2.5 microseconds. As an example of an elementary image processing operation, a 3 x 3 neighborhood Sobel operator takes 54.6 microseconds or 18,315 Sobels per second per cell.

Each chip contains 84 pins for power, ground, clock, control, address, and data exchange. Each chip occupies about one square inch of board space and dissipates about one-half watt.

## GAPP ARRAY

### Assembly

The assembly of an array is simple; each chip is connected to its logical neighbor (east connects to west and north to south). Clocks and control are distributed to every chip in the array. Practical limits exist and most are imposed by the choice of board housing, backplanes, bus standards, or system architecture. In the current design, arrays are modularized as 48 x 132 (6,336) cells on a single 9u board (15 x 17 inches). See Color Plate VIII, page 700.

### Input/Output

In standard systems, input of data occurs via the CM south port and output occurs from the CM north port. This arrangement takes advantage of CM plane, allowing for simultaneous input and output during computation. To obtain simultaneous input and output, three conditions must exist. First, a result must be available at the start of the input/output operation. Second, a plane of input data must be available in the external world. Third, the algorithm currently running must require at least N clocks, where N is the size of the GAPP array in the north-south direction. To obtain free input/output, a result is loaded from RAM or registers into the CM plane in one clock cycle. Data in the CM register plane are shifted north one position for each clock. Simultaneously, a row of data is output into an appropriate buffer on the northern edge of CM. On the same clock, a row of data is input into the southern edge of CM. This operation continues for N clocks. During the N clocks, any other operation can occur withint the array as long as it does not involve the CM plane. At the end of N clocks, data are written from CM into RAM or registers as dictated by the program.

### Sizes

The smallest size array is one chip. The required array size is tailored to the system problem. In real-time image processing, the major parameters then determine size including input data rate, algorithm length (execution time), and array clock speed. For example, assume a 10 MHz clock speed for the array. Further assume that the data are arriving at 12 megapixels per second and that the algorithm requires 50,000 instructions or clocks (the equivalent of 2,000 8-bit adds for every pizel in the array).

The algorithm will require 5 ms to execute (50,000/10,000,000). The array must contain at least 60,000 cells (5ms x 12,000) to maintain real-time rates without missing any data; this equates to about 833 chips. Using the 48 x 132 cell GAPP modules previously mentioned, a system containing ten modules will suffice (880 GAPP chips or 63,360 cells). The modular design approach can accommodate up to 24 GAPP modules, 2,112 chips, or 152,064 cells as shown in Figure 5. At a 10 MHz clock frequency, a 24-module system would exhibit a computational throughput of 60 giga 8-bit adds per second. The largest GAPP system to date contains 1,152 chips or 82,944 cells.

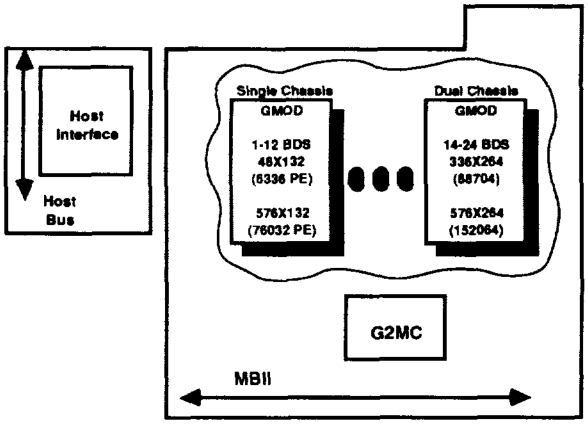


Figure 5. GAPP™ Chip Modules—Optional Configurations

## GAPP SYSTEMS

### Composition

Every GAPP-based image processing system must interface with the outside world, as shown in Figure 6a. At the next level of detail, the systems must include elements unique to GAPP-based architecture such as the interface to data sources and sinks appropriate to the array as well as program store and control for SIMD style processors, as depicted in Figure 6b.

All GAPP systems have these functions but unique implementations are too numerous to mention in detail. The range of system applications has spanned NASA Mars Lander studies to Strategic Defense Initiative (SDI) applications, from laboratory systems to tactical military hardware. GAPP array designs range from small arrays (12 x 18 cells) to very large arrays. GAPP systems have handled frame rates from seconds per frame to more than 100 frames per second.

The remainder of this paper discusses systems composed of a standard set of modules. The system component overview is shown in Figure 7. We assume that these systems will be connected to a host computer, although the majority of our GAPP systems are stand alone ROM or disk-based machines. The required modules (solid outline) are described and available optional modules (dashed outline) are discussed.

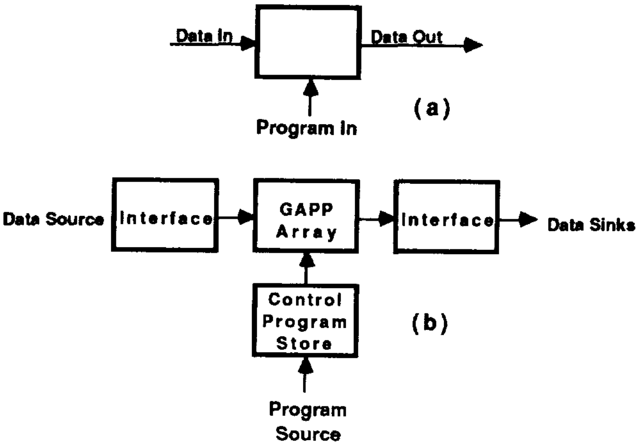


Figure 6. GAPP™ Chip System View

## GAPP Module

The GAPP module, pictured in Color Plate VIII, page 700, and diagrammed in Figure 8, is the heart of the system. The design philosophy applied to this module drives the overall system definition. The module contains three major elements, the array of GAPP chips, an input buffer memory and an output buffer memory. The GAPP array contains 88 chips arranged as a 48 x 132 cell group. CMS lines are used as the input means to the input buffer section. CMN lines are used as the output means to the output buffer section.

The input buffer is composed of standard memory chips, either with 16K or 64K parts, depending on the application. These parts are arranged to form a 48-bit wide memory that matches the GAPP array size. Both data to be stored in the input buffer

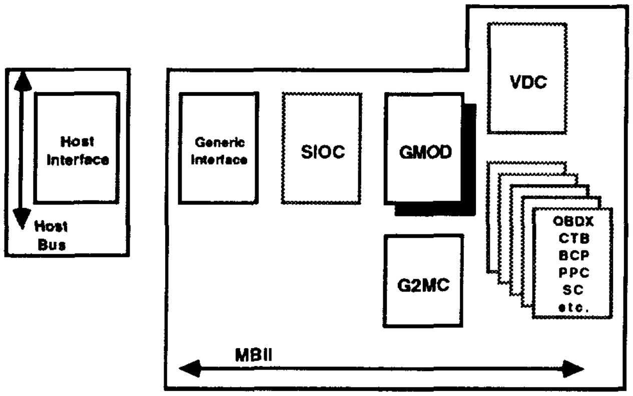


Figure 7. GAPP™ Chip-Based Optional Elements

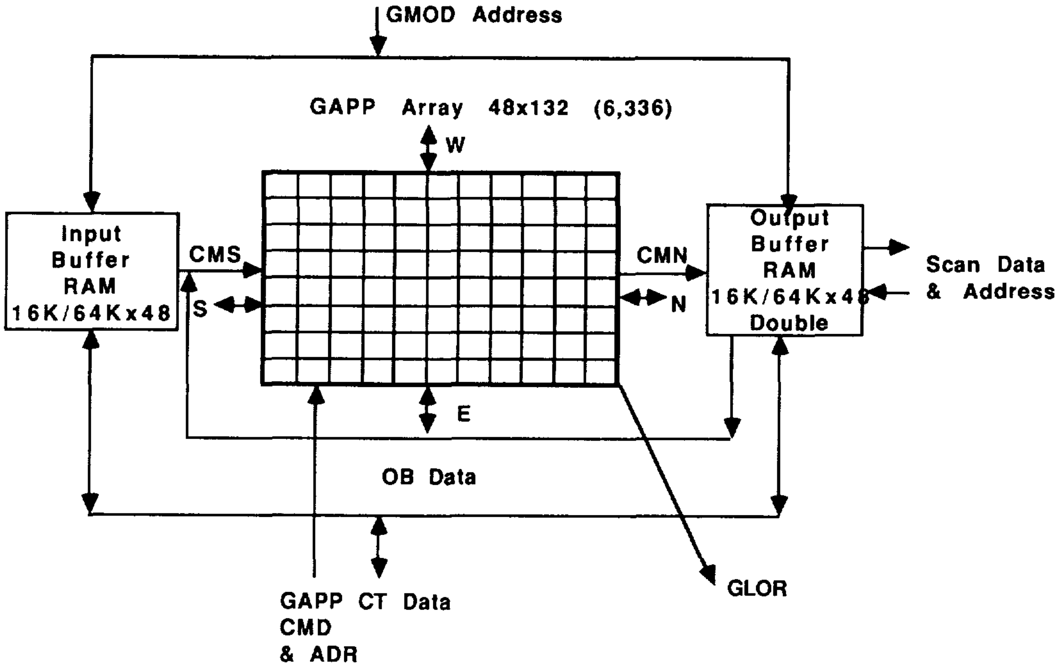

GMOD Address
GAPP Array 48x132 (6,336)
W
Input Buffer RAM 16K/64Kx48
CMS
S
CMN
N
Output Buffer RAM 16K/64Kx48 Double
Scan Data & Address
E
OB Data
GLOR
GAPP CT Data
CMD
& ADR

Figure 8. GAPP™ Chip Module GMOD

and its addresses may come from two sources: the GAPP controller (G2MC) and (if one is included in the system) the SIOC. These sources control and write 48-bit wide data into the memories. The G2MC has priority over the input corner turn (CT) data and address bus.

The output buffer is composed of the same type of chips as the input buffer, but is double buffered. This double buffering allows the simultaneous transfer of results to the data sink during GAPP computations involving the other section of the output buffer. Depending on the system configuration, the scan bus output from the output buffer is the data and control path by which real-time results and display transfers are made to the remainder of the system. If few results are desired, or if the data rates are low, then results can be extracted from the output buffer via the corner turn bus through the G2MC to the host.

Data may be input into the GAPP array from two sources via the CMS port: the input buffer and the output buffer. This arrangement allows continuous real-time sensor input while enabling the output buffer to be used as virtual memory to augment the in-cell GAPP data memory. Since every module has its own output buffer, the time required to save and recover data to virtual memory is independent of array size or organization.

The impact of this approach reflects into all areas of the system design. First, consider the input side of the module. Since the input buffer is not double-buffered, it is possible for the GAPP program to need input from the input buffer while a real-time sensor is inputting data. This eventuality is handled via control from the GMC in conjunction with FIFO storage on the SIOC.

Second, on the output side, there are currently two system components that might require simultaneous access to the scan bus: the video display controller (VDC) and the post-GAPP processing chain [entered via the output buffer data extractor (OBDX)]. Because of this possibility, an arbitration function is included on each board.

Third, from a system standpoint, each GAPP module has its own unique relationship with respect to the other modules and to the mapping of data into the modules. When the data frame size is established (512 x 512 or 100 x 100) the relationship between that frame and the physical processor size must be determined. This reflects into GAPP module and input buffer memory addressing. Further, GAPP modules do not reside on the system bus, MBII. Rather, they are accessed via the G2MC on MBII.

### GAPP Module Controller

The purpose of the GMC shown in Figure 9 is to provide system synchronization and program storage and broadcast to the GAPP modules. It is the primary interface to the GAPP modules and consists of two major sections plus the MBII interface. The first section is the control and GAPP program storage section and the second is the data input/output section.

All data and control commands are provided to the G2MC via MBII from the host, including download of GAPP algorithms and the control sequence to be executed. The controller can be configured to hold as many as 1 mega word of GAPP instructions. The minimum configuration is 256K words. Board control starts the sequence of GAPP execution via the system control and sequencer. This in turn establishes system synchronization as a function of embedded commands within the GAPP code. Both GAPP module input and output buffer address sequences are established from the controller for normal operational interface with the GAPP array. A priori knowledge of the algorithm is used by the system to precompute addresses to maintain system throughput. These addresses are known as either compile or load time.

The second section, data input/output, is used for low volume real-time, high volume nonreal-time, or setup data transfers. As previously noted, results may be transferred via this path to the host. Some applications require masks to be available as a bit

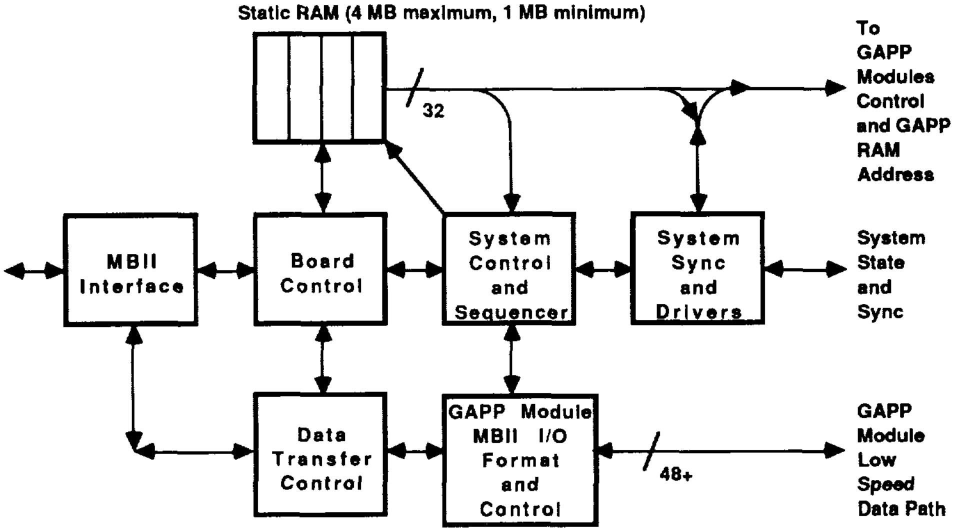

Static RAM (4 MB maximum, 1 MB minimum)
32
To GAPP Modules Control and GAPP RAM Address
MBII Interface
Board Control
System Control and Sequencer
System Sync and Drivers
System State and Sync
Data Transfer Control
GAPP Module MBII I/O Format and Control
48+
GAPP Module Low Speed Data Path

Figure 9. GAPP™ Chip Module Controller (G2MC)

plane while others assign a unique array cell address to each cell loaded from the output buffer. These data are downloaded via this path, which sets up data and extracts test algorithm execution results to determine the integrity of the GAPP array.

### Generic Interface

The generic interface (GI) module, shown in Figure 10, provides a standard GAPP system interface to a host system. This smart switch routes programs and data to the appropriate modules within the GAPP-based system via the highest speed path available. A host unique interface is required in conjunction with the GI. Currently, three host interfaces are planned: IBM PC-AT, SUN, and VAX/Aptec Systems™.

The GI passes GAPP program data, lookup data, execution orders, and the like to respective bus modules via the MBII message passing protocol. Modules that can reside on the bus have a predefined list of recognizable primitives that enable new capabilities to be added to the system equipment list.

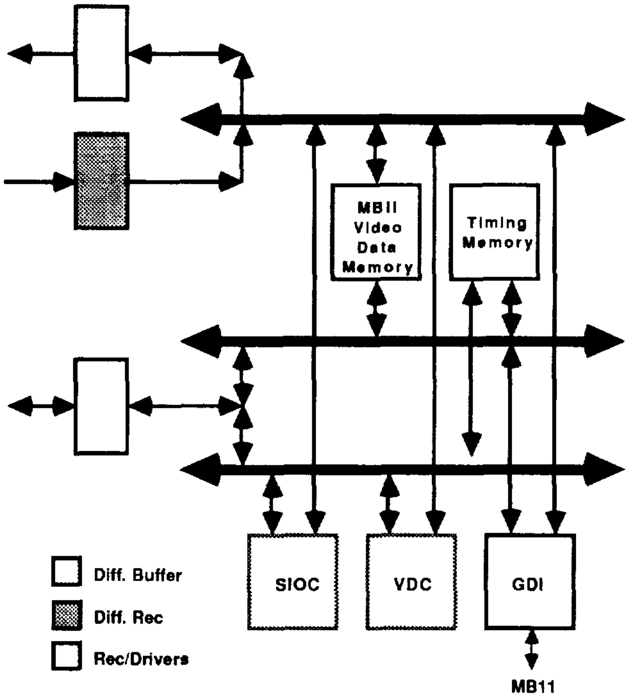


Figure 10. Generic Interface

### Minimum System

With these three modules and the host interface, a minimum system can be configured (Figure 11). GAPP modules (1 minimum), a controller, and the GI perform algorithm development and numeric, geometric, and logical computations. This configuration is useful in a personal workstation with the IBM PC-AT or the SUN as a host. The data transfer rates of these hosts prevent gross inefficiencies for a 2-to-6 GAPP module system.

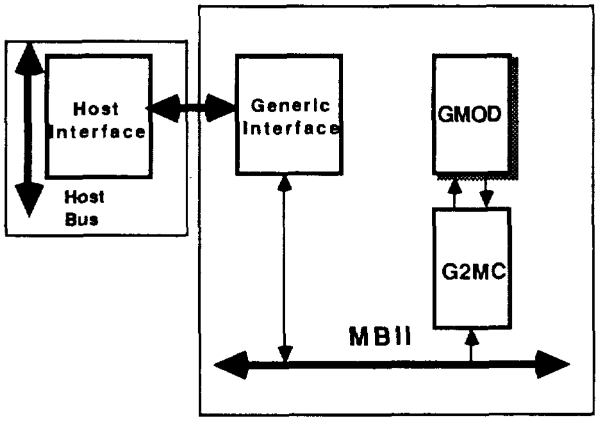


Figure 11. GAPP™ Chip-Based Minimum System

### High Speed System

Figure 12 represents a high-speed real-time display and processing system. If connected to a VAX/APTEC system with an IBIS disk system, this system would provide powerful processing capabilities for a multiple user environment. With a minimum of eight GAPP modules, this system would avoid input/output overload. Additional equipment would include video disks and tapes for real-time data input.

The video display controller (Figure 13) provides for symbology overlay over output video derived from GAPP output or symbology overlay over digitized analog video with the optional video digitizer. The VDC provides a digital output port to supply digitized video as an input to the GAPP system from standard analog inputs (RS170 or RS343). The analog video output is provided in RGB format. Lookup tables establish color parameters. The VDC may be used alone, for output only, or in conjunction with the sensor input/output controller (SIOC).

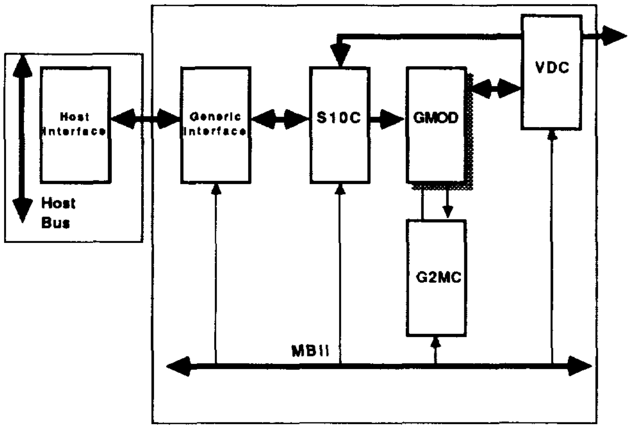


Figure 12. GAPP™ High-Speed System/Multiuser Environment

Aptec Systems™ is a registered trademark of Aptec Computer Systems, Inc.

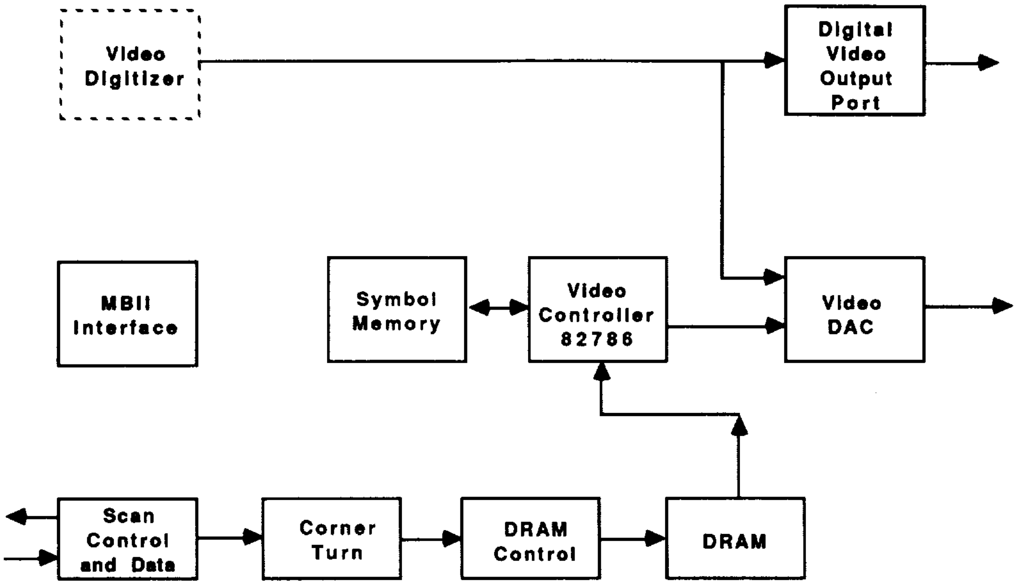


Figure 13. Video display controller

The SIOC, shown in Figure 14, provides the interface between digital video and the input buffer section of the GAPP modules. The SIOC accommodates a quantity of GAPP modules and can select from one of two video sources, A or B. With optional adapters, Emitter-Coupled Logic (ECL) or double-ended Transister Transister Logic (TTL) digital inputs can be used. In the high-speed system configuration, one of these inputs is the generic interface, while the other is the digitized video port of the VDC. Thirty frames per second video can be received and processed from either port. In some sensors, the order of pixel presentation is scrambled. The DSC block is a programmable window function. Windows of different sizes may be selected from each frame of a continuous stream of images. The CMP or compressor function is a programmable down-sampler.

The statistics box represents a custom statistic chip that accumulates image statistics as the image is input to the system. These data may be used to adjust the algorithm in real time. The lookup table (LUT) can remap the image if desired. Some sensors are scanned vertically while others are scanned horizontally. This box remaps vertical scan to horizontal scan. The system can accommodate direct vertical scanned input if desired. The remainder of the SIOC interfaces the digital video stream to the GAPP modules in conjunction with the GAPP controller module.

## Utilities–Algorithms–Software

The software development environment for GAPP-based systems is continuously improving and expanding. Our first software system was based on the STOIC threaded language. While this approach produced a complete software development environment in about 4 weeks of elapsed time, it was not attractive to outside customers. To overcome this aversion to STOIC, an Ada-like language compiler was developed.

In conjunction with NCR, a C language-based sofware development system was created. This system has been ported to the NCR Tower computers, IBM PC-AT and its close compatibles, and the SUN computer.

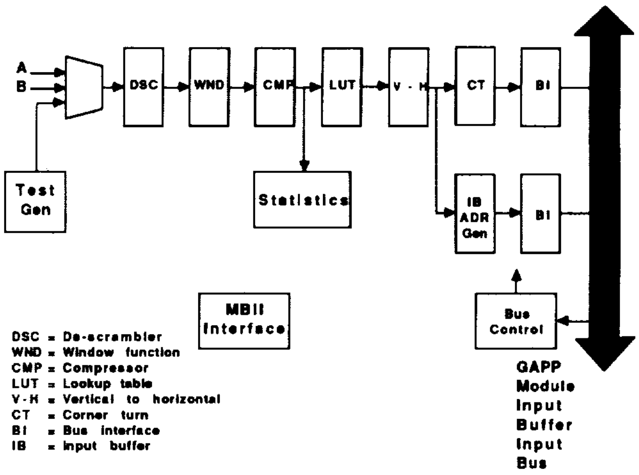


Figure 14. Sensor input/output controller (SIOC)

A number of utilities functions and algorithmic approaches have been developed between 1986 and 1988. These include computational and representational approaches to Hough Transforms, artificial neural networks, 3-dimensional image manipulation and rotation, multiple simultaneous discrete Fourier transforms, and floating point representations.

Many basic level operations exist as libraries. These libraries are available for addition, subtraction, multiplication, division, absolute values, thresholding, erosion, dilation, general convolution, spatial filtering, down sampling, histogramming, and logical operations on both single and multiple bit functions. The "ADD" library, for example, must handle a significant diversity of operand locations and operand precision for bit serial geometric processors. Geometrically, operands may be in the same cells as data memory or in neighboring cells. The operands may be located at the same GAPP RAM address or at different addresses, and the number of bits in the operands need not match. All of these considerations must be implemented in a general purpose library function for each of the standard elemental computer operations.

### Mars Rover Sample Return (MRSR)

Martin Marietta Astronautics Group uses a GAPP-based system to investigate complex autonomous spacecraft landing. Color Plate IX (p. 701) shows the marrain (Martian terrain) board used to simulate the surface and the TV probe that captures the pictures to be processed by the 100 x 100 GAPP cell process. This processor is coupled to a Gould SEL™ computer via RS422 link.

## SUMMARY

The GAPP-based processor is a flexible and affordable system architecture based on simple fine-grain bit serial processing cells. This system has broad application, including use in laboratory, tactical military, and space-based systems.

Gould SEL™ is a registered trademark of Gould, Inc.

# NAP (NO ALU PROCESSOR) THE GREAT COMMUNICATOR

§†Jeff Fried and §‡Bradley C. Kuszmaul

§MIT Laboratory for Computer Science
Cambridge, MA 02139

†GTE Laboratories
Waltham, MA 02254

‡Thinking Machines Corporation
Cambridge, MA 02139

## ABSTRACT

Message routing networks are acknowledged to be one of the most critical portions of massively parallel computers. This paper presents a processor chip for use in massively parallel computer. The programmable approach used in this processor provides enough flexibility to make it a "universal" part for building a wide variety of interconnection networks and routing algorithms. A SIMD control scheme is used to make programming and synchronizing large numbers of processors simple.

In the course of designing this processor, we were faced with the decision of which logic operations to implement in an ALU; informal design studies showed that it was best to provide none. The processor performs all computations by a sophisticated table lookup mechanism, and has no ALU; it is thus called the No ALU Processor (NAP). Using tables rather than an ALU provides a very flexible instruction set, and in real programs often allows more than one "operation" to be done in one cycle.

Benchmarks written for the NAP show that indirect addressing mechanisms can speed many common operations by a factor of about $\log N$. We have therefore provided hardware to support indirect addressing, or Multiple Address Multiple Data (MAMD) operation. In addition, the NAP contains local storage used for flexible instruction decoding: the same instruction can result in different operations on different chips. These two mechanisms allow programmers to write programs for NAP machines easily using SIMD style, and also provide the power of different computations happening simultaneously in different parts of the machine.

Keywords: Universal, Table lookup, ALU, Parallel, Processor, Network, VLSI

## INTRODUCTION

Message routing networks for parallel supercomputers occupy a unique place in the spectrum from specialized to general-purpose machines. Although these routing networks can used to build general-purpose parallel computers (as well as specialized computers), they themselves are usually built out of very specialized hardware. This paper presents a single processor design which is useful for building a variety of different networks; in this sense it is a general-purpose element within the specialty of interconnection networks. This processor is an experimental design incorporating several novel architectural features which make it simple to program, general purpose, and efficient. Specifically, no ALU is provided in the processor. The arithmetic functions normally performed by an ALU are instead performed by table lookups into memory. In addition, a very flexible programming model is provided, which supports indirect addressing and multiple concurrent instructions while operating in a SIMD or Multiple SIMD (MSIMD) mode.

The NAP chip described in this paper is the result of a design experiment which explores architectures for communication network support. The experiment has three main design goals:

- Act as a "universal" element for routing networks. By universal we mean both general purpose and efficient. The performance of the NAP when used as a node within a network should be as close as possible to the performance of a special purpose chip designed especially for that network.

- Provide communications control which is as flexible as possible.

- Keep the processor's I/O pins (which connect to other NAP chips) and memory as busy as possible performing useful work.

In the course of designing the NAP, we were faced with the decision of which logic operations to implement in an ALU; informal design studies showed that it was best to provide none. Using tables rather than an ALU provides a very flexible instruction set, and in real programs often allows more than one "operation" to be done in one cycle. One of the most interesting lessons from the design of the NAP was that table lookup is a very powerful mechanism.

A collection of NAP chips can be wired together and can be programmed to simulate many things. We have programmed our simulators to perform several important parallel algorithms, including reduction and parallel prefix in a tree network (Ref. 1), connection-machine style routing on a cube connected cycle (Ref. 3), cellular automata programs (such as Conway's game of Life) (Ref. 7). We are able to support any network with a large number of nodes (up to about $2^{32}$ nodes) of constant degree, including fat-trees (Refs. 4, 2), butterfly networks (Refs. 8, 5, 6), cube connected cycles, trees, and meshes.

Section of this paper describes the instruction set architecture of the NAP. Section discusses the processor design and the implementation of the NAP chip. Finally, Section evaluates the NAP in the light of our design goals, and summarizes the lessons learned from this project.

## INSTRUCTION SET ARCHITECTURE

We adopt the (M)SIMD model of one or more controllers broadcasting microinstructions to sets of processors; each set of processors is controlled by one controller. The controller handles all instruction sequencing, like loops or branches. In this SIMD model all processors are globally synchronized at the instruction level. Each processor communicates with other processors through eight bidirectional wires. The bidirectional wires may be connected in any fashion to form an interconnection network; the NAP chips form the nodes of that network, and may do computations in parallel to perform routing, do actual computing for the system, or both. A system-level view of the NAP is shown in Figure 1. Examples of networks which can be built using NAPs are Butterfly or Fat-Tree networks, Banyan or Flip-type networks, Hypercubes (more than $2^8$ processors require multiple NAPs per node), Cube-Connected-Cycles, Shuffle-Exchange networks, Torus and Mesh networks, restructurable networks, and Trees. An important restriction is that the networks are regular enough to have fewer than 16 distinct types of nodes; most practical networks have one or two.

### Indirect Addressing and MIMD

One very important mechanism provided by the NAP which is not found in conventional SIMD computers is indirect addressing. We support indirect addressing because of the wave nature of the computations performed by many routing networks. Consider for example parallel prefix (Ref. 1), which is a class of parallel algorithms which use a tree interconnection structure between processors to perform many operations (such as addition) in logN time. At any stage of a parallel prefix computation, each level of the tree may be accessing a bit at a different address than other levels of the tree are accessing. Conventionally, this would be handled by enabling or disabling the processors at different levels of the tree, and running the computation on different levels at different times, thus slowing down the overall computation. Indirect addressing provides a mechanism for different processors to access different memory addresses at the same time under SIMD control. The result is that parallel programming can be done more flexibly and more efficiently.

In addition to indirect addressing, there are three means of differentiating processors within the SIMD control structure and hence making programming more flexible and efficient.

1. Conditional execution: the instructions broadcast on the SIMD bus can conditionally load a local instruction store called the nanostore, conditionally load the memory, conditionally load configuration bits within the NAP (called I/O-or-State-Select or ISS bits), and conditionally execute sequences of instructions. An instruction may be conditioned on any of the 16 bits of state within the NAP.

2. The instructions stored within the nanostore of each NAP may be different, so that different processors may perform totally different operations in response to the same broadcast instruction.

3. Processors can have different tables at the same address in local memory, and thus perform different functions even while they are accessing the same address.

These three mechanisms, which are explained in more detail below, provide a large degree of flexibility to NAP programmers.

### Instruction Philosophy

We assume that off-chip wire delays are slow compared to on-chip cycle times and local memory access time, since we are implementing systems with long wires.

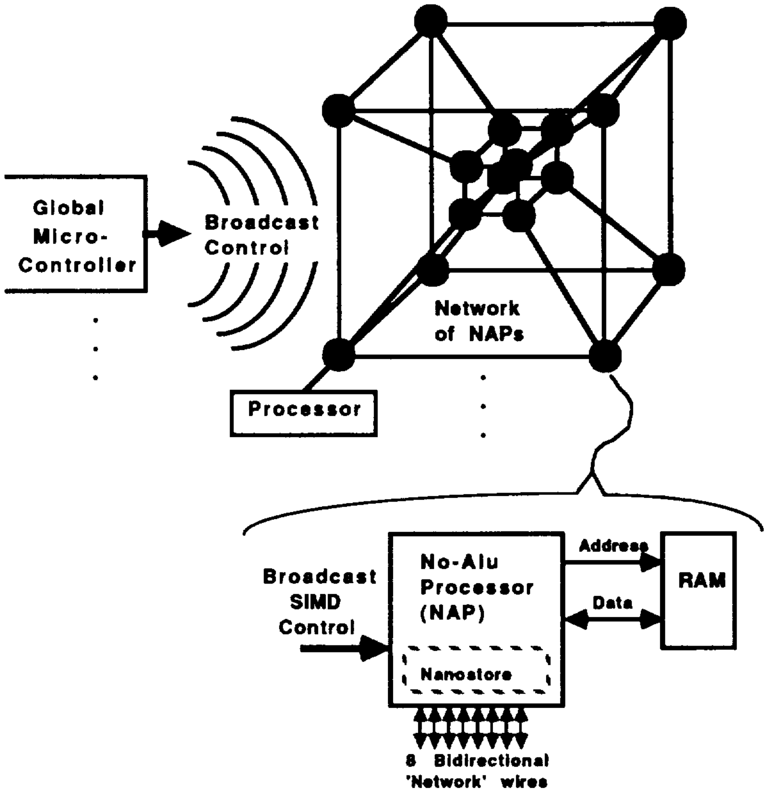


Figure 1. System-level view of a NAP-based computer. Global microcontrollers broadcast instructions to sets of NAP chips. Each NAP chip is connected to an off-chip RAM, a SIMD instruction broadcast bus, and 8 bidirectional network lines.

Therefore, we chose a microcycle/nanocycle timing approach. At each microcycle, the controller broadcasts a global microinstruction, and each processor can read or write from each of its eight pins. Within each microcycle, there are four minor cycles called nanocycles. During each nanocycle, a nanoinstruction is executed which nearly always references the external memory twice (one read and one write or write-back). Thus, the NAP uses a two-phase timing methodology internally, and the memory may be accessed during each phase. Two phases make a nanocycle, and four nanocycles make a microcycle. The memory address may be changed once a nanocycle.

The NAP is heavily memory based. As we have seen, each phase of a nanocycle may involve a memory access, so that the performance of the NAP is driven by memory performance. Most programs written for the NAP are also very memory-oriented. Operations are performed using tables in memory under the control of broadcast microinstructions. Typically, these table-based operations take as operands an arbitrary combination of state and input wire values, an integer, or an address. Each table (called a function table) requires 256 words (8 bits each). Our prototype supports up to 2K words of external RAM, so that up to eight different tables can be stored in memory at once; additional tables are downloaded as needed. Tables may be accessed using either direct or indirect addressing.

## The NAP microword

Figure 2 shows the format of the NAP microword. This word is the instruction broadcast from a controller to a number of NAP chips in (M)SIMD fashion each microcycle. The 39 bits of the microword are common to all the NAPs in a set. Each microword contains distinct operation codes for every nanocycle, as well as condition codes, a direct memory address, and two table offsets used for indirect addressing or table-based logical operations. The microword is also very memory-oriented; 17 of its 39 bits are used for memory addressing.

The microword does not contain the actual nanoin-

| Name | Function | Width |
|---|---|---|
| INIT | initialization and download control | 1 bit |
| OP0 | four-bit indexes into the nanostore | 4 bits |
| OP1 | which specify which nanoinstruction | 4 bits |
| OP2 | to perform in each nanocycle | 4 bits |
| OP3 | OPs share one address and condition code | 4 bits |
| CC | Condition code; this decodes to 16 conditions | 4 bits |
| MIP | Memory address (for direct addressing) | 11 bits |
| F0 | Function table offsets (for indirect addressing) | 3 bits |
| F1 | normally contains the start address of a table | 3 bits |
|  | total number of microword bits | 39 |

Figure 2. The microinstruction word format shows the mnemonics, functions, and width of each instruction field.

structions executed each nanocycle by the NAPs. Rather, it contains four four-bit OP codes which specify an address in an on-chip memory called the nanostore. The nanostore contains the nanoinstructions in the form of a bit for every control line needed by the NAP hardware. The OP fields give the 'address' of the nanoinstruction within the nanostore. This approach reduces the number of bits broadcast to the processors and thus economizes on chip pins. In addition, it provides a mechanism for different processors to perform different work under the control of the same microinstruction, since different processors may have different nanoinstructions loaded into the same address in the nanostore.

### Memory Addressing Modes

A number of memory addressing modes are supported by the NAP. Bit-read, bit-write, word-read, and word-write modes are supported, and each of these may be addressed using any combination of bits available to the address multiplexors. A memory address is built as shown in Figure 3. Bits are multiplexed onto the SRAM address pins from the microinstruction (the MIP, F0, and F1 fields), or from internal state bits. There are sixteen bits of state in the NAP: eight bits from the external SRAM held in a Memory Data Latch (MDL), and eight bits which can be configured as any arbitrary combination of I/O bits or additional State bits (called IS bits). All of the these state bits may control the memory address.

A memory address specifies an eight-bit word. Within that word, the low order three bits of the MDL specify a bit in that word. A memory address is 11 bits (providing 8K bits of address space) in the NAP chip. Each memory address is used for one nanocycle only, although the memory addressing fields are held constant for a whole microcycle.

### Providing a Global OR-tree

A global-or line to the microcontroller (the computer which broadcasts the SIMD instruction stream) can be derived from any of the I/O/State bits by ORing the external wires together. This capability is extremely useful. For example, when checking for a condition (e.g. does any processor contain zero, or does any processor's memory contain a pattern which matches the broadcast pattern), the result can be returned to the microcontroller within a microcycle. The distance from the microcontroller to the NAP chips through the SIMD bus and back through the global-or tree might be more than 200 ns, so that programmers using the global-or mechanism might have to take account of the pipelining effect. Any bidirectional communications pin on the NAP may be used to construct a wired-OR tree.

## PROCESSOR DESIGN

A block diagram of the NAP processor is shown in Figure 4. The major subcircuits are a set of datapath circuitry, a Nanocode store consisting of 16 by 28 bits of static RAM, Nanosequencer logic to control the execution of instructions, and a set of Instruction pipeline registers. The NAP uses a three-phase (1.5 nanocycle) pipeline internally: operation lookup, nanocode access, and datapath operations happen sequentially in every nanoinstruction.

The NAP is designed to work with 35-ns external Static RAMs. These are expensive. It would make sense to move this memory on-chip.

Sixteen words by 28 bits of nanocode store are provided which are addressed in sequence by each of four opcode fields in the microinstruction. These nanoinstructions are downloadable and may differ for different processors. The outputs of the nanostore are the con-

| Address bits | bit source 0 | bit source 1 | bit source 2 |
|---|---|---|---|
| Bits 0:4 | MIP[0:4] | MDL[3:7] | IS[0:4] |
| Bits 5:7 | MIP[5:7] | IS[5:7] | |
| Bits 8:10 | MIP[8:10] | F0[0:2] | F1[0:2] |

Figure 3. The memory address is constructed from combinations of the Memory Immediate Pointer (MIP), the Memory Data Latch (MDL), the I/O-State bits (IS), and the Function table pointers (F0 and F1).

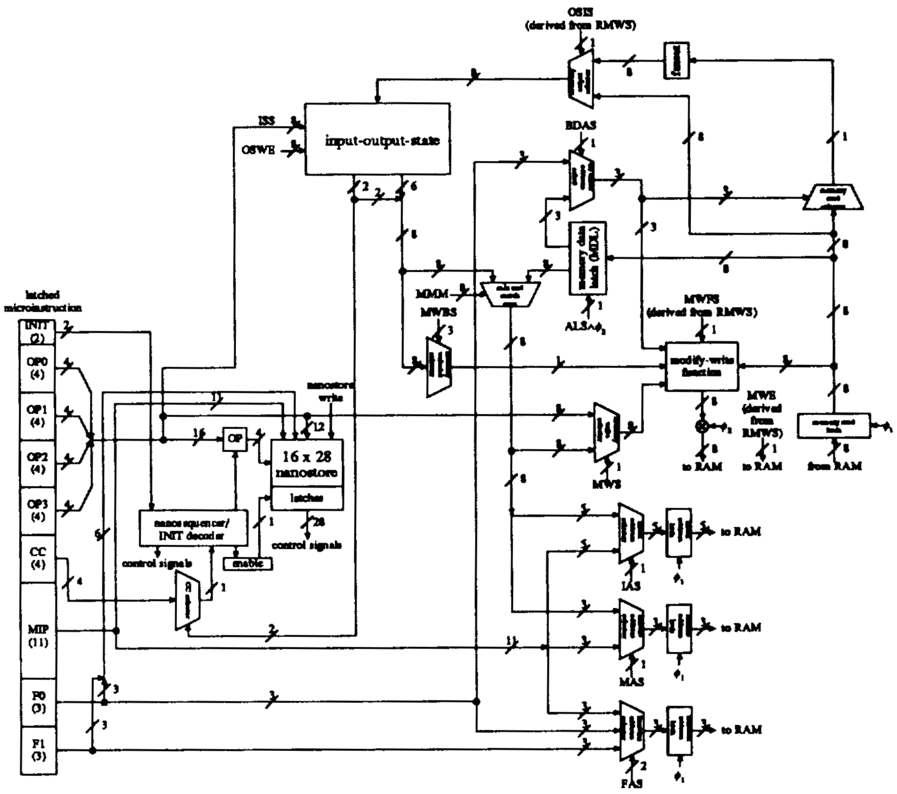


Figure 4. The block diagram of the NAP chip shows the SIMD instruction latches (left), the nanosequencer (lower left), the state and I/O circuitry (top center), the data-path (right), and the RAM interface (lower right).

trol bits used directly by the logic in the processor; the nanostore itself is a static RAM with decoders, write amps, and sense amps. This RAM has a access time goal of 25 ns, and is 1974 by 1620 microns in area using a 3 micron CMOS technology.

Conditional instruction execution is provided in the nanosequencer via an enable control which may disable all outputs of the nanostore. This disabling happens if the bit in the microinstruction condition code field selected by the state bits is high. This mechanism allows up to 16 different classes of processors.

The NAP is designed using a fully static CMOS circuit methodology in MOSIS scalable CMOS design rules. A two phase non-overlapping clocking approach is used; Approximately half of the circuitry on the chip (and exactly half of the control lines) are 'active' on phase 1, while the other half is active on phase 2. The MAGIC layout system was used for the layout of the chip. Each chip contains four NAPs, although only one of these processors is fully connected to the pins of the chip. The other three processors are accessible through scan path circuitry. The overall circuit is 7900 by 9200 microns in a 3 micron CMOS process.

## EVALUATION AND CONCLUSIONS

We have shown that it is feasible to design a processor chip which supports a variety of bit-serial routing networks efficiently. This type of chip is a step towards understanding how to build and operate interconnection networks for massively parallel computers. The NAP chip we have designed provides very flexible addressing mechanisms, and allows indirect addressing so that MAMD operation is possible. This chip also supports three distinct means of multithread operation, so that different processors operating off the same instruction stream can do different things. Finally, this processor chip has no ALU; table lookup is used for all operations. We have found all of these mechanisms useful in writing example programs, and believe that the NAP approach can teach designers about how to provide addressing and processor selection mechanisms in SIMD processors, and about the issues involved in providing flexible and high-performance interconnection networks.

How well has the NAP design stood up to its original design goals? Let us examine those goals one by one:

- **Provide communications control which is as flexible as possible.** The operation of the processor is completely programmable at both the microinstruction and nanoinstruction levels. Processors have considerable flexibility in addressing modes, and indirect addressing at both the bit and word level is well supported. In addition, there are three distinct means for processors operating from the same instruction stream to do different things: in addition to the standard conditional execution (which is made very general in the NAP), they can have different nanoinstructions in their nanostore, or use different operation tables in their memory. In practice, this allows programmers to write programs with the simplicity implicit in SIMD control and synchronization, yet keep processors efficiently utilized doing different things at the same time. Essentially, one can program a machine built of NAPs as sets of processors, even if those processors share the same controller.

- **Keep the I/O pins and memory as busy as possible performing useful work.** Each microinstruction may make up to four memory references, each of a read-modify-write nature. Every microinstruction executed by the processor can be able to read from and write to up to eight I/O pins on the processor. All of the programs written on NAP to date have been able to keep the I/O pins active at at least one bit per microcycle, which corresponds to our assumptions about wire latency. Similarly, most of these programs use most of the nanocycles in a microcycle to perform useful work, so that memory is well utilized. The cycle time of the NAP is also in good agreement with the speed available from state-of-the-art commercial SRAMs or on-chip dRAM.

- **The NAP should serve as a 'universal' element for routing networks.** To date, we have written NAP programs for message routing using algorithms designed for butterfly networks (Ref. 6) using the same number of cycles as a node designed specifically for that purpose. We have also written NAP programs for parallel prefix (Ref. 1) which execute in one microcycle per bit. Although these examples are not sufficient evidence to prove that NAP is in fact a universal communication element, they do indicate that NAPs would be useful in a number of different networks.

- **Experiment with an ALU-less processor.** Our experience in writing NAP programs using tables for operations is that 'compressed tables', which do more than one thing in one operation, are immediately of use. For example, one portion of the table might be used to increment a pointer while another part might perform a boolean operation on a few I/O bits. We had hoped that experimenting with table-based operations might lead us to a choice of which operations to put into an ALU; instead, we discovered that the generality offered by these tables was just the right thing for programming.

We hope that the NAP chip will eventually serve as a testbed for experimentation with new interconnection networks and parallel algorithms. We plan to test the NAP design using a variety of 'benchmark' programs and networks to test its utility as a general-purpose network element. Measurement of effect of indirect addressing and our processor differentiation mechanisms on processor utilization will tell us something about the efficiency of our approach. Finally, using these mechanism to write programs may lead to future insights about what programming constructs are useful for writing effective parallel programs for communication networks.

## ACKNOWLEDGEMENTS

The authors would like to thank Tom Cormen and Elliot Kolodner for their hard work on the design, layout, and programming of the NAP, Bill Dally for his direction and feedback during the course of the NAP project, and Charles Leiserson for his insightful comments, direction, and consultation in the early stages of the NAP design.

## REFERENCES

1. A. Borodin and J. E. Hopcroft. Routing, merging, and sorting on parallel models of computation. In *Proceedings of the 14th Annual ACM Symposium on the Theory of Computing*, pages 338–344, 1982.

2. Ronald I. Greenberg and Charles E. Leiserson. Randomized routing on fat-trees. In *Proceedings of the 26th Annual IEEE Symposium on the Foundations of Computer Science*, November 1985.

3. W. Daniel. Hillis. *The Connection Machine.* MIT Press, Cambridge, MA, 1985.

4. Charles E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, C-34(10), October 1985.

5. Nicholas Pippenger. Parallel communication with limited buffers. In *Proceedings of the 25th Annual IEEE Symposium on the Foundations of Computer Science*, October 1984.

6. Abhiram G. Ranade. How to emulate shared memory. In *Proceedings of the 28th Annual IEEE Symposium on the Foundations of Computer Science*, pages 185–194, October 1987.

7. Tommaso Toffoli and Norman Margolus. *Cellular Autonoma Machines.* MIT Press, Cambridge, MA, 1987.

8. L. G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *Proceedings of the 13th Annual ACM Symposium on the Theory of Computing*, May 1981.

# Generalization of Orthogonal Multiprocessor for Massively Parallel Computation*

*Kai Hwang* and *Dongseung Kim*

Dept. of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089-0781

### Abstract

An *orthogonal multiprocessor* (OMP) was recently proposed by the authors. This paper presents multidimensional OMP which is conceived for massively parallel computation. The OMP compares very favorably over two known architectures, namely ***generalized hypercube*** and ***spanning-bus hypercube***. All three architectures extend the original concept of binary *n*-cube. Among the three, the OMP requires equal or less hardware and yet presents some interesting application potentials in massively parallel computation. The orthogonal architecture is a viable alternative to the conventional SIMD array computers which use distributed local memories.

## 1 Introduction

An *orthogonal multiprocessor* (OMP) architecture has been recently conceived at University of Southern California [4] and independently at Princeton University [6]. A similar architecture, called EMPRESS, has been built by ETH (Swiss Federal Institute of Technology) for solving PDE problems [3].

An OMP has $p$ processors and $p^2$ memory modules interconnected by $p$ dedicated memory buses in two orthogonal directions (Fig.1). The details of the OMP architecture can be found in [4]. Each white circle represents a memory module and each shaded circle is regarded as a computer node consisting of a processor, its local memory and a bus selector. Each bus is used by only one processor. The unique feature of the OMP rests on conflict-free memory access operations: *column access* and *row access*. The two accesses are mutually exclusive, which is termed as the *orthogonality principle*. In a row access, all processors access the *row memories*. Likewise, in a column access, all processors access the *column memories*. All processors can communicate with each other in two memory accesses. By the orthogonality principle, parallel computation and communication can be achieved among multiple processors. High memory bandwidth and efficient computation are provided. A rich class of parallel algorithms has been developed for the OMP [4,5,6]. Parallel programming is easy due to easy partitioning of the tasks and simple allocation of resources. However, a major problem in implementing a large OMP comes from the fact that it requires $O(p^2)$ memory modules and their interconnections.

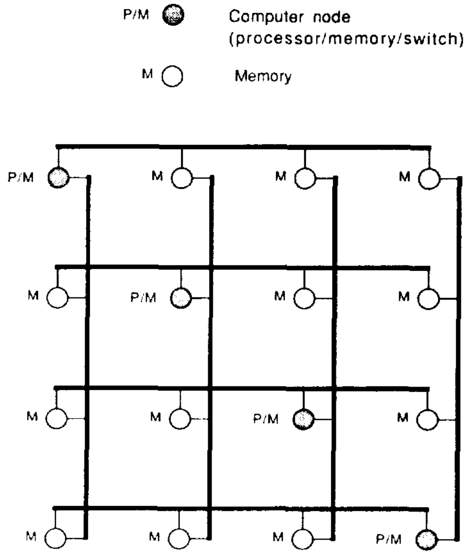


**Fig. 1:** OMP(4,2), a two-dimensional orthogonal multiprocessor with radix 4.

*This research was supported in part by the ONR grant N00014-86-K-0559 and in part by the NSF grand DMC-84-21022.

CH2649-2/89/0000/0391$01.00 © 1988 IEEE

In this paper, the OMP is extended to multidimension for massively parallel computation. The OMP compares very favorably over two known architectures, namely *generalized hypercube* [1,2] and *spanning-bus hypercube* [1,7]. All three architectures generalize the original concept of binary $n$-cube. The OMP requires equal or less hardware among the three and yet presents some interesting application potentials in massively parallel computation.

The organization of the paper is as follows: Sections 2 and 3 describe detailed architecture of binary OMP and $k$-ary OMP; Section 4 deals with interprocessor communication strategy and network characteristics; Section 5 includes potential applications and conclusions.

## 2 Binary Orthogonal Multiprocessors

The original OMP (Fig.1) is regarded as a 2-dimensional architecture with radix 4 since the memories are organized into a 2-dimensional array. Before a general extension of the OMP (i.e. $k$-ary OMP), *binary orthogonal multiprocessors* are characterized below to clarify the idea.

A binary 3-dimensional OMP, denoted as OMP(2,3), is shown in Fig.2. It consists of $2^{3-1} = 4$ processors and $2^3 = 8$ memory modules. Each processor has a dedicated memory bus to shared memory array. Memory buses are switched to provide 3 directional accesses ($x, y$, and $z$). Figure 3 shows patterns of parallel memory access using the orthogonality principle. For the $x$-access, only those buses running in the $x$ direction are active. There is no contention between processors in memory accessing. $y$-access and $z$-access are performed in a similar fashion. Each memory bus is spread into $x, y$, and $z$ directions. There are two memory modules tied to a bus for each dimension. One of four memory modules connected to the same bus can be accessed in any direction. It is dedicated to the processor that owns the bus, called a *private memory*. Thus, private memories and associated processors are assumed to be at same nodes (i.e. shaded circles, which is termed *computer nodes* in the figures).

A binary 4-dimensional OMP (OMP(2,4)) has $2^{4-1} = 8$ processors and $2^4 = 16$ memory modules. The architecture is illustrated in Fig.4. There are 4 disjoint ways of memory accesses ($w, x, y$ and $z$ accesses). Figure 4b-c illustrates the memory accesses.

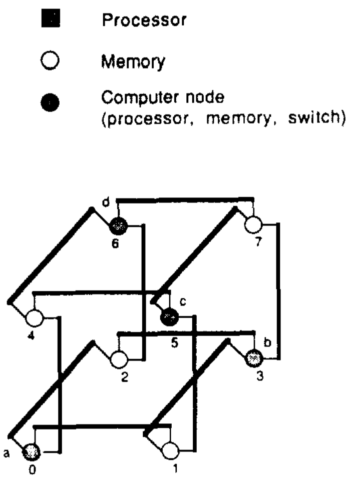
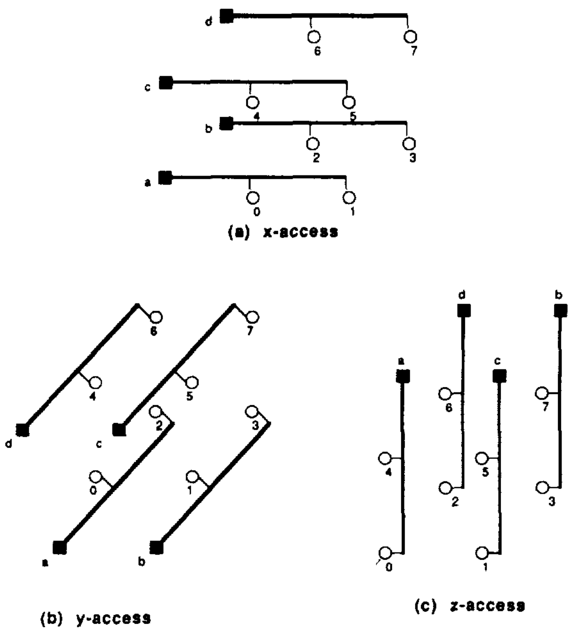


Fig.2 The multiple bus interconnection structure of an OMP(2,3)



(a) x-access

(b) y-access

(c) z-access

Fig.3 Three orthogonal directions for memory accesses in an OMP(2,3)

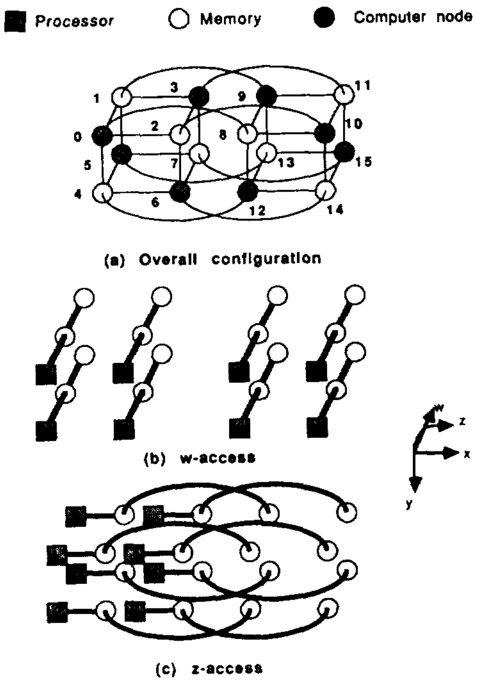
A binary $n$-dimensional OMP (OMP$(2,n)$) is constructed by $p = 2^{n-1}$ processors and $m = 2^n$ memory modules based on a binary $n$-cube. Memories are located at all vertexes of the cube. Edges of the cube form $2^{n-1}$ buses interconnecting processors and memory modules. $2^{n-1}$ processors exist every two links away. This corresponds to two-bit difference in the node index when an $n$-bit binary representation is used.

Binary OMPs are very similar to binary $n$-cube computers. Buses with shared memories are counterparts of the point-to-point links in the hypercube computer. An OMP uses equal number of memory modules and a half of processor nodes. The OMP needs $2^{n-1}$ buses, each of which runs into $n$ dimensions, while the hypercube requires $n \cdot 2^n$ point-to-point links. Thus, OMP demands fewer links (buses) to achieve massive parallelism. Orthogonality rule simplifies the control complexity in memory access.

A *network diameter* is defined as the maximum number of links of the shortest path between any two processors. In a hypercube topology, the network diameter is obtained by *Hamming distance.* It corresponds to the number of edges of the shortest path between the source and the destination. The communication cost in the OMP is measured by the number of *memory reads* or *memory writes* for a message to reach the destination via the shortest path. Message passing between processors $d$ edges apart in an OMP needs $d - 1$ memory reads and $d - 1$ memory writes. Thus, two memory references in the OMP are equivalent to the routing cost for one edge in the hypercube. We will call them a *routing unit.* By the above definition, the network diameter of a binary $n$-dimensional OMP is $n - 1$. Thus, OMP has slightly smaller network diameter. Comparisons of binary OMP with binary hypercube are summarized in Table 1.

Table 1: Comparison of a binary OMP with a binary hypercube

| | OMP(2,$n$) | Binary $n$-cube |
|---|---|---|
| # processors | $2^{n-1}$ | $2^n$ |
| # memory modules | $2^n$ | |
| # buses or links | $2^{n-1}$ | $n \cdot 2^{n-1}$ |
| Ports per node | $n$ | $n$ |
| Network diameter | $n - 1$ | $n$ |



(a) Overall configuration

(b) w-access

(c) z-access

Fig.4 The OMP(2,4) architecture with 4 orthogonal memory access directions.

# 3 $k$-ary Orthogonal Multiprocessors

In general, a $k$-ary $n$-dimensional OMP (OMP$(k,n)$) consists of $p = k^{n-1}$ processors, $m = k^n = kp$ memory modules, and $p$ dedicated memory buses. Memories and interconnecting buses form a $k$-ary $n$-dimensional hypercube ($(k,n)$ hypercube). The architecture may be called a *shared-memory hypercube.* Nodes on colinear edges are connected by a bus instead of point-to-point links. Figures 5 and 6 show a ternary 3-dimensional OMP (OMP$(3,3)$) and ternary 4-dimensional OMP (OMP(3,4)), respectively.

A memory bus spans to $n$ orthogonal directions (dimensions). $k$ memory modules are tied together per dimension in a bus. A private memory located at computer node is common to every direction. A total of $n(k-1)+1$ memory modules are accessible by each individual processor. A memory module is shared by $n$ different processors except private memories. Since $n \gg p$, memory is partially shared. The access is controlled by an $n$-to-1 switch in each memory module that allows only one bus (or a processor) be

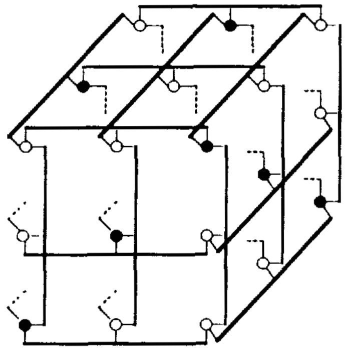


**Fig. 5: An OMP(3,3) architecture consisting of 9 processors and 27 memory modules.**
**(Hidden nodes and buses are not shown.)**

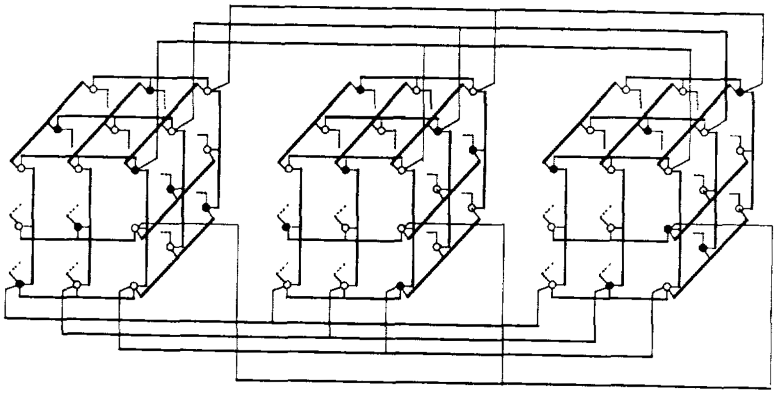
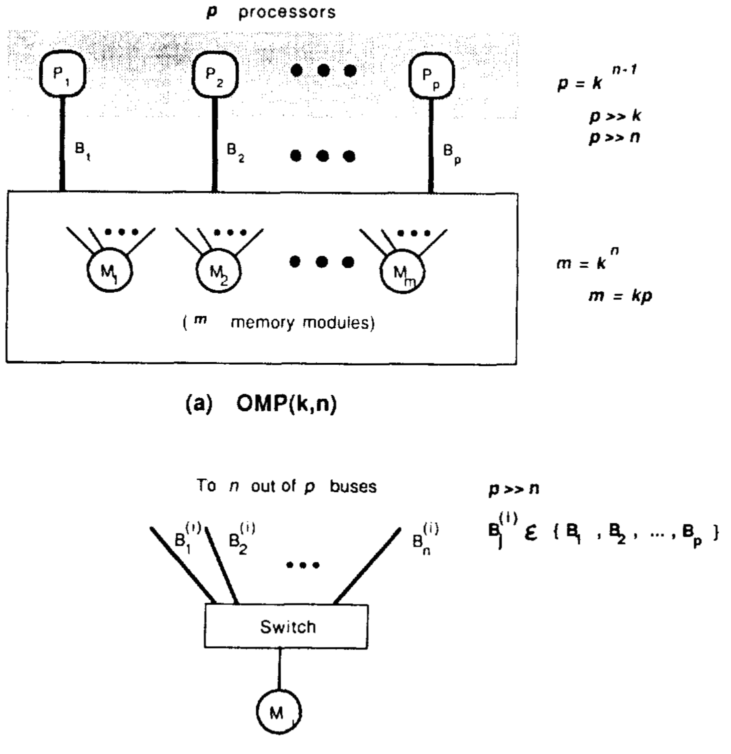


**Fig. 6: An OMP(3,4) architecture consisting of 27 processɔrs and 81 memory modules.**
**(Only partial interconnections are shown.)**



(a) OMP(k,n)

(b) A memory module

**Fig. 7:** An OMP(k,n) architecture consisting of p= $k^{n-1}$ processors and $k^n$ memory modules interconnected by p dedicated buses.

switched to at a time. The logical structure can be viewed as Fig.7. The orthogonal access rule is observed for memory read/write: all processors should access the memory in the same direction in a memory cycle. Processors which do not need memory accesses do not participate the memory access. There are $n$ different patterns of memory access by the rule. Due to the synchronous memory access, the SIMD operation is quite suitable for controlling high-dimensional OMPs.

The OMP architecture is very similar to *spanning bus hypercube* (SBH) [1,7] and *generalized hypercube structure* (GHC) [1,2] as shown in Fig.8. They are organized based on hypercube topology. Each node of both architectures represents a processor and its local memory. SBH and GHC are distributed-memory multiprocessors while the OMP is a shared-memory one. A bus connects multiple processor-memory nodes in SBH. There could be bus contentions between processors while the OMP does not. The GHC contains point-to-point links for interprocessor communication. Radixes of all dimensions of GHC are assumed same here. If the three systems have an equal number of nodes, OMP uses $k$ times less nodes as processors. Table 2 summarizes the comparisons of hardware complexity of OMP with GHC and SBH. The longest path between two nodes in $k$-ary $n$-dimensional hypercube has $n$ edges. Thus the network diameter of OMP$(k,n)$ is $n-1$. Section 4 deals with the issue in more detail. So the diameter of the OMP is a slightly smaller than others. The number of ports is the largest for GHC. Both OMP and SBH have $n$ ports per node. For massively parallelism, OMP is appealing since it has the least interconnection complexity for the interconnections. Figure 9 shows the interconnection complexities of the OMP compared with GHC architectures for various configurations for massively parallel computation. The number of independent buses and the number of links are considered in both OMP and SBH, and GHC, respectively.

For the OMPs with the same number of processors, there are various system configurations with different radixes and dimensions. The larger the dimensionality, the smaller the number of the memory modules required ( $p = k^{n-1}$). For example, a OMP(2,7) of 64 processors requires only 128 memory modules with a diameter of 6, while an OMP(8,3) needs 512 modules with a diameter 2 (Table 3). Similar situation can be observed for 64K processor systems. The tradeoff between the choice of radix $k$ and dimension $n$ lies in cost-effectiveness and communication speed (network diameter).

# 4 Orthogonal Computation and Communication

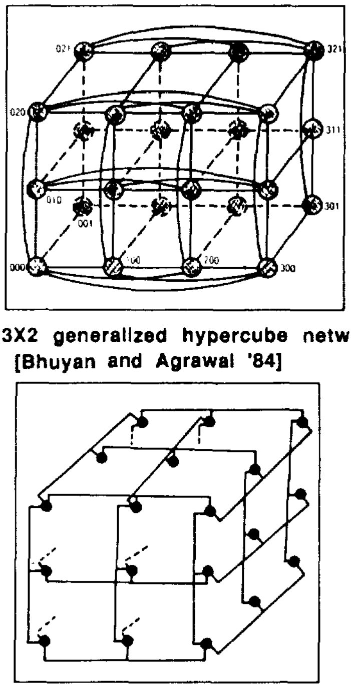
There are various mappings for $k^{n-1}$ computer nodes onto $(k,n)$ hypercube with $k^n$ vertice. The following sections deal with interprocessor communication after formalizing the node assignment of an OMP onto $(k,n)$ hypercube.

## 4.1 Processor/Memory Assignment

Processors in an OMP$(k,n)$ can be indexed by $(n-1)$ dimensions.

$$P_i = P(a_{n-2}, a_{n-3}, ..., a_1, a_0), \qquad (1)$$

$$(0 \leq a_p < k \quad \text{for all} \quad p = 0, 1, ..., n-2)$$



(a) A 4X3X2 generalized hypercube network [Bhuyan and Agrawal '84]

(b) A $3^3$ spanning bus hypercube network [Wittie '81]

Fig. 8: Two known generalizations of the hypercube architecture

Table 2: Comparisons of Orthogonal Multiprocessor (OMP) with Generalized Hypercube Computer (GHC) and Spanning-Bus Hypercube (SBH) architectures

| Architecture | OMP$(k,n)$ | GHC$(k,n)$ | SBH$(k,n)$ |
|---|---|---|---|
| # processors | $k^{n-1}$ | $k^n$ | $k^n$ |
| # memory modules | $k^n$ | | |
| # buses or links | $k^{n-1}$ | $nk^n(k-1)/2$ | $nk^{n-1}$ |
| Ports per node | $n$ | $n(k-1)$ | $n$ |
| Network diameter | $n-1$ | $n$ | $n$ |

Memory modules can be indexed similarly by $n$ dimensions as $M(b_{n-1}, b_{n-2}, ..., b_1, b_0)$ for $0 \leq b_q < k$, $q = 0, 1, \cdots, n-1$. One simple way to assign $k^{n-1}$ computer nodes on the vertices of $(k,n)$ hypercube as follows: $P_i$ is mapped to a node $j = (b_{n-1}, b_{n-2}, ..., b_1, b_0)$ such that

$$\left.\begin{aligned} b_k &= a_{k-1} \quad \text{for } k = 1, 2, \cdots, n-1 \\ b_0 &= \left[\sum_{k=0}^{n-2} a_k\right] \bmod k \end{aligned}\right\} (2)$$

Thus, only the last digit of $j$ needs to be determined in the mapping.

A processor located at $(b_{n-1}, \cdots, b_0)$ can directly access $n(k-1)+1$ memory modules whose indexes are the same as that of the processor node except in one dimension:

$$\begin{array}{ll} M(b_{n-1}, b_{n-2}, \cdots, b_2, b_1, x) & \\ M(b_{n-1}, b_{n-2}, \cdots, b_2, x, b_0) & \\ \vdots & (x = 0, 1, \cdots, k-1) \\ M(b_{n-1}, x, b_{n-3}, \cdots, b_1, b_0) & \\ M(x, b_{n-2}, b_{n-3}, \cdots, b_1, b_0) & \end{array}$$

For a non-private memory module $M(b_{n-1}, b_{n-2}, \cdots, b_1, b_0)$, the $n$ processors sharing the memory are those at $(b_{n-1}, b_{n-2}, \cdots, b_1, z_0)$, $(b_{n-1}, b_{n-2}, \cdots, z_1, b_0)$, $\cdots$, $(b_{n-1}, z_{n-2}, \cdots, b_1, b_0)$, and $(z_{n-1}, b_{n-2}, \cdots, b_1, b_0)$, where $z_0, z_1, \cdots, z_{n-1}$ are some fixed constants satisfying the relationship (2).

The OMP architecture can be conceptually simplified such that only the processors and their interconnections are considered. Hypothetical interconnections between processors, called *logical links*, are introduced. Two processors are assumed connected by a logical link if they share a memory module. The original OMP$(k,n)$ can be reduced to $(k, n-1)$ hypercube by removing all memory nodes and creating logical links. The $(k, n-1)$ hypercube consists of processors at each node and links between nodes. Such a structure may be called a *processor-hypercube*.

Table 3: Various OMP configurations for two given machine sizes

| # processors | p = 64 | | | p = 65,536 (= 64K) | | |
|---|---|---|---|---|---|---|
| Organization:<br>(k-ary, n-dim) | (2,7) | (4,4) | (8,3) | (2,17) | (4,9) | (16,5) |
| # memory modules | 128 | 256 | 512 | 128K | 256K | 1000K |
| Network diameter | 6 | 3 | 2 | 16 | 8 | 4 |

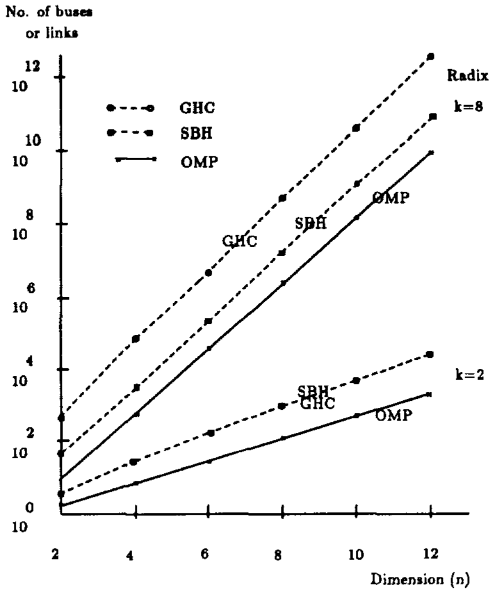


**Fig. 9: Comparisons of interconnection complexities of OMP with Generalized Hypercube (GHC) and Spanning Bus Hypercube (SBH) architectures.**

A $(k, n-1)$ processor-hypercube can be created by projecting the OMP$(k,n)$ to $n-1$ dimensional space. For example, the original OMP$(k,n)$ may be projected along the $n$th dimension. Because no two processors exists on the same line (bus) in the OMP$(k,n)$, there is no overlap of processors at the same node in the projection. A processor at $(b_{n-1}, b_{n-2}, \cdots, b_1, b_0)$ is mapped to $P(b_{n-1}, b_{n-2}, \cdots, b_1, b_0)$ in the processor-hypercube. The $(k, n-1)$ processor-hypercube preserves the interprocessor connection relationship *except* the $n$th dimension. In other words, the interconnection information for the $n$th dimension is lost in the processor-hypercube. This results in decreasing the physical proximity (edge count) by one between processors whose $n$th digits in their indices were different. Suppose distance in the processor-hypercube is computed using Hamming distance. It will overestimate the distances for processor pairs which had the same coordinates in $n$th dimension, since $d$-digit difference in their indices does not change after projection. However, the $d$-digit difference corresponded to the distance of $d-1$ from our original definition. Those with different coordinate values in the $n$th dimension will get correct distance. Thus, the routing distance measurement based on $(k, n-1)$ processor-hypercube gives an upper bound.

The *average internode distance* $\bar{d}$, determining the queueing delay in the communication network, is defined as

$$\bar{d} \equiv \sum_{d=1}^{n-1} dN_d/(p-1) \qquad (3)$$

where $p_d$ is the number of processors in the distance $d$ from a source node. It is computed as $(n-1)(k-1)k^{n-2}/(k^{n-1}-1) \approx n-1$, if $k \gg 1$. In binary OMP, $\bar{d} \approx (n-1)/2$. The average internode distances of generalized hypercube and spanning bus hypercube are $n$ [1]. Hence the network properties of the OMP are quite comparable to those two hypercube architectures.

## 4.2 Routing Algorithms

Data can be routed on the processor-hypercube using algorithms employed in hypercube computers. Two processors having only one-digit difference can communicate with the cost of one routing unit. Messages are sent toward the direction with unequal coordinate values. Processors with two-digit difference have a distance of two. For example, in the OMP(4,4), the processor $P(0,0,0)$ can send a message to the processor $P(0,1,3)$ in two routing units. There are two disjoint paths in the processor-hypercube.

Path 1 : $P(0,0,0) \rightarrow P(0,1,0) \rightarrow P(0,1,3)$

Path 2 : $P(0,0,0) \rightarrow P(0,0,3) \rightarrow P(0,1,3)$

Exact paths in original OMP can be derived using the same idea with computer node assignment defined in (2).

Data routing between two processors whose indices differ by multiple digits can be done using a successive routing to immediate neighbor processors whose indices differ only one digit. There are many alternative paths for a given pair of processors. Data routing can be simplified if the routing sequence follows the order of the index, i.e. if there are $d$-digit difference in the indices, we arbitrarily choose a dimension in which the digit is different. The data are sent along the direction, then along the next highest dimension with unequal indices, and so on. Once the lowest dimension is reached, the next dimension is determined in wraparound fashion. We can find $d$ disjoint paths to route data to the same destination due to $d$ different choices for the first routing dimension. Hamming distance is the routing distance required to reach the destination. The maximum difference in the indices between processors is $n-1$ for the $(k, n-1)$ processor-hypercube. Thus the network diameter is $n-1$ for OMP$(k,n)$. Simultaneous data movements by multiple processors via the same logical link causes no contention. It is because each logical link represents $n$ separate physical memory buses associated with the processors in the direction. The orthogonal access guarantees parallel data movement for all processors without conflict.

*Broadcasting* from a processor to all $p$ processors takes $n-1 = \log_k p$ steps. The data propagate along $j$th dimension at $j$th step $(j = 0, 1, \cdots, n-2)$. At one memory cycle, each processor simultaneously writes the data received onto all $k$ memory modules in the $j$th dimension. Each step the number of processors grows $k$ times which have already received the data. Thus, after $n-1$ steps all processors will get the data.

Below we show a parallel algorithm for computing a consensus function. Consensus functions include the maximum/minimum, sum, and/or of $p$ numbers. By recursive doubling, $k$ numbers stored in $k$ processors of OMP$(k,2)$ can be summed up in $O(\log_2 k)$ memory accesses [5]. This corresponds to the summing of $k$ numbers stored in $k$ processors which lie on a straight line in a $(k, n-1)$ processor-hypercube. There are $p/k$ such parallel lines along the direction. Thus, for each of those parallel lines, a summation of $k$ numbers is computed simultaneously. After one step of the concurrent sum, the number of intermediate sums to be further added is reduced to $1/k$ of previous one. By alternating the direction among the $n-1$ dimension, the total is found after $n-1$ summing operations since there are $k^{n-1}$ numbers. The algorithm is sketched below. The result (a sum of the $p$ numbers) will be obtained in processor $P(0, 0, \cdots, 0)$.

**Algorithm SUM**

```
for $j = 0$ to $n-2$ do (1) and (2):
    (1) Form groups of $k$ processors having same coordinate
          $(a_{n-1}, a_{n-2}, \cdots, a_{j+1}, a_j, 0, 0, \cdots, 0)$ except $a_j$.
        Each group of $k$ processors simultaneously computes a
          sum of $k$ data by recursive doubling using $k \times k$
          memory array in the group.
    (2) The results are shifted to the processors whose nodes
          have the same indices except a zero in $j$-th
          dimension.
```

The following computes the time complexity of Algorithm SUM. Step (1) can be done in $O(\log_2 k)$ time. Step (2) takes constant time. Hence the overall time complexity of the algorithm is $O\{(n-1)\log_2 k\} = O(\log_2 k^{n-1}) = O(\log_2 p)$.

## 5 Concluding Remarks

The OMP is a hybrid architecture which combines the advantages of both shared-memory and distributed-memory systems. It is generalized from the hypercube to use some of the nodes as processor nodes and the remaining as partially shared memories. Partial sharing of memories significantly reduces the required memory ports per module as compared with fully shared-memory multiprocessor. Logarithmic diameter provides fast and efficient communication. The orthogonality principle enables conflict-free, parallel memory access, and scalable performance, which are very desirable for either SIMD or MIMD operations. These properties support massively parallel computation for mechanics, physics, chemistry, vision, and etc. Potential applications are summarized in Table 4.

Table 4: Potential applications of the generalized OMP architecture

- Signal/image/speech processing
- Numerical modeling
- Neural network simulation
- Graphics for visualization
- Sensory fusion and robotics

The distinct characteristics of the generalized OMP are summarized below:

- By introducing high-dimensional configurations, the number of memory modules required is reduced significantly from the original OMP ($O(kp)$ vs. $O(p^2)$).
- The orthogonal memory access rule may be rigid for general-purpose applications. However, it contributes to the delivery of high memory bandwidth and simple memory access control.
- The performance of the OMP in interprocessor communication is comparable to other similar architectures like GHC and SBH in terms of network diameter and average internode distance. The OMP requires lower interconnection complexities.
- For fine-grain computation., the shared-memory organization performs better in communication than distributed one. It has less overhead in carrying routing information and does not need complex routing algorithms than the message-passing scheme.

Indeed, the orthogonal architecture becomes a viable alternative to the conventional SIMD array architecture which uses distributed local memories.

## References

[1] D.P. Agrawal, V.K. Janakiram, and G.C. Pathak, "Evaluating the Performance of Multicomputer Configurations", *IEEE Computer*, vol. 19(5):pp. 23–37, May 1986.

[2] L.N. Bhuyan and D.P. Agrawal, "Generalized Hypercube and Hyperbus Structures for a Computer Network", *IEEE Trans. Computers*, vol. C-33(4):pp. 323–333, April 1984.

[3] R.E. Buehrer, H.J. Brundiers, H. Benz, B. Bron, H. Friess, W. Haelg, H.J. Halin, A. Isacson, and M. Tadian, "The ETH-Multiprocessor EMPRESS: A Dynamically Configurable MIMD System", *IEEE Trans. Computers*, vol. C-31(11):pp. 1035–1044, Nov. 1982.

[4] K. Hwang, P.-S. Tseng, and D. Kim, "An Orthogonal Multiprocessor for Lager-Grain Scientific Computations", *IEEE Trans. Computers*, Jan. 1989.

[5] D. Kim and K. Hwang, "Parallel Image Processing and Pattern Analysis on Orthogonal Multiprocessors", submitted to *IEEE Trans. Pattern Analysis and Machine Intelligence*, July 1988.

[6] I.D. Scherson and Y. Ma, "Orthogonal Access Multiprocessing: An Architecture for Numerical Applications", *Journal of Parallel and Distributed Computing*, to appear 1989.

[7] L.D. Wittie, "Communication Structures for Large Networks of Microcomputers", *IEEE Trans. Computers*, vol. C-30(12):pp. 273–284, April 1984.

N90-16453

# BLITZEN:

# A HIGHLY INTEGRATED MASSIVELY PARALLEL MACHINE*

D. W. Blevins, E. W. Davis[1], R. A. Heaton, and J. H. Reif[2]

The Microelectronics Center of North Carolina
Research Triangle Park, NC 27709-2889

## ABSTRACT

The goal of the BLITZEN project is to construct a physically small, massively parallel machine. A highly integrated chip has been designed with 128 processing elements (PEs). A BLITZEN system consisting of 16,384 SIMD PEs will require only 128 PE array chips. This paper presents the PE architecture , the organization of PEs on the chip, and the feature set of the chip which has been custom designed and is being fabricated at the Microelectronics Center of North Carolina. Each PE has 1K bits of static RAM and performs bit-serial processing with functional elements for arithmetic, logic, and shifting. Unique local control features include modification of the global memory address by data local to each PE, and complementary operations based on a condition register. PEs on the chip are positioned in an 8 by 16 array. Data I/O is accomplished through a new method using a four-bit bus for each row of 16 PEs. The BLITZEN chip is one of the first to incorporate over 1.1 million transistors on a single die. It has been designed with MCNC's advanced 1.25 micron CMOS process to operate in excess of 20 MHz. A 16K PE system, operating at 20 MHz, can perform IEEE standard 32-bit floating point multiplication at a rate greater than 450 megaflops. Fixed point operations on 32 bit data can exceed the rate of one billion operations per second. Since the processors are bit-serial devices, performance rates improve with shorter word lengths. The bus oriented I/O scheme can transfer data at 10240 megabytes per second.

Keywords: massively parallel, custom VLSI, parallel processing, SIMD, MPP.

## OVERVIEW AND MOTIVATION

Parallel machines make use of multiple processing elements executing simultaneously to speed up computation. For the purposes of this paper, we will consider a *massively* parallel machine to be a parallel machine with at least 10,000 processors. A number of massively parallel machines have been constructed, including the Massively Parallel Processor (MPP) built for NASA Goddard Space Flight Center by Goodyear Aerospace Corporation (now Loral Systems Group), the Distributed Array Processor (DAP) built by the British firm ICL, and the Connection Machine (CM) built by Thinking Machines, Inc. (Refs. 1, 7, 8, and 11). These projects demonstrated the feasibility of constructing machines with massive parallelism. Nevertheless, only a relatively small number (a few dozen) of the machines have been built so far and they have been utilized almost exclusively by *research* branches of government agencies, academic, and industrial organizations.

### Miniaturization of Sequential Computing Machines

The situation now may be very similar to the development of the first mainframe computers in the late 40's: only a few general purpose computers existed. At that time, IBM made an early study which indicated that the worldwide use of computers would require only a few dozen mainframes (the rest of the computing equipment being calculators or special purpose machines). Nevertheless, a combination of advantageous engineering and economic factors resulted in the proliferation of computers. Central among these factors was the use of advanced electronic techniques to reduce the physical size, that is, to *miniaturize* computing machines. By miniaturization, we mean a high level of integration of the hardware onto VLSI components. Note that the process of miniaturizing sequential architectures has not necessarily at all degraded the computing power available to users. Miniaturization first allowed mainframe computing machines to be economically manufactured; and later, further improvements in integrated circuit technology allowed personal computing machines to be physically placed within the working environment of office workers, engineers, and scientists. In fact the development, for example, of miniaturized RISC architectures, has actually *improved* performance in many cases, by allowing higher execution rates.

### BLITZEN: A Miniaturized Massively Parallel Machine

The central goal of the BLITZEN project is to develop a miniaturized massively parallel machine. The machine will be physically small while providing the performance associated with massively parallel processing. We are convinced that the development of such a miniaturized machine will have the same benefits as discussed above for conventional sequential machines:

(1) These miniaturized machines should be much more economical, allowing a much larger market for massively parallel machines.

* This work was supported in part by NASA Goddard Space Flight Center under Contract Number NAG-5-966 to the Microelectronics Center of North Carolina.

[1] Dept. of Computer Science, North Carolina State Univ., Raleigh, NC 27695-8206.

[2] Dept. of Computer Science, Duke Univ., Durham, NC 27706. He is also supported by contracts: ONR #N00014-80-C-0647, Air Force #AFOSR-87-0386, ONR #N00014-87-K-0310, NSF #CCR-8696134, DARPA/ARO #DAAL03-88-K-0195, and DARPA/ISTO #N00014-88-K-0458.

(2) The miniaturized machines could be backplaned with conventional workstations, making the capabilities of massively parallel computation easily accessible to engineers and scientists.

(3) A miniaturized machine could potentially be used in environments that require very small size and power consumption, such as on space flights. For example, NASA plans to have such a machine as a component of the Space Station computing system.

This paper provides rationale for design decisions, many of which have the dual benefit of both insuring miniaturization and also improving performance.

### The Project Team

The BLITZEN project involves a number of institutions in the Research Triangle area of North Carolina, including Duke University, North Carolina State University (NCSU), and the Microelectronics Center of North Carolina (MCNC). Project personnel included John Reif, Jonathan Rosenberg, and graduate students Jonathan Becher, Nigel Hooke and Lars Nyland of the Computer Science Dept. of Duke, Edward Davis of the Computer Science Dept.of NCSU, and Don Blevins and Fred Heaton of MCNC. The BLITZEN project has received partial support under a grant from NASA Goddard Space Flight Center.

Team effort to date has resulted in development of the processing element architecture (Refs. 4 and 5), custom design for the PE array chip, development of a full scale PE array simulator (Ref. 10), microcode for selected arithmetic operations, and the specification of an assembler language and architecture for the BLITZEN controller (Ref. 9). We are in the process of developing a prototype system and a high level parallel programming language which is an extension of C++ for the BLITZEN machine.

### Organization of the Paper

In the next section, "Processing Element Architecture", we describe the bit serial processing element and provide some comparisons with the MPP and Connection Machine. Local control features and methods for memory access are emphasized. Following the discussion of individual PE architecture, we describe, in the section "PE Array Chip Architecture", the organization of PEs on the custom chip, with emphasis on our interconnection and I/O schemes. The section "Chip Feature Set", provides details of the custom chip design and instruction pipeline. An overview of system architecture concepts and software for BLITZEN is given in the final section, "BLITZEN Systems".

## PROCESSING ELEMENT ARCHITECTURE

Each processing element in BLITZEN is a bit serial processor, with a variable length shift register and random access memory. The BLITZEN design used the MPP PE architecture, described in Ref. 2., as a starting point.

The existence of the MPP has provided experience with massively parallel processing such as that reported by the MPP Working Group (Ref. 6) and by K. E. Batcher, the chief architect of the MPP, (Ref. 3).

Our group has designed various improvements on the MPP PE architecture into BLITZEN:

(1) Incorporation of RAM *on-chip* for each PE.

*Motivation*: This allows the PE to access memory without off-chip delays.

(2) *Bus oriented I/O* with a four bit path for each set of 16 PEs.

*Motivation* This gives BLITZEN a total I/O capability of 4,096 bits per cycle. (In comparison, the MPP has a total I/O capability of 256 bits per cycle, and the Connection Machine has an I/O capability of 1,024 bits per cycle.)

(3) *Local* modification of RAM addressing.

*Motivation*: This allows on-chip memory accesses to be determined by the contents of each PE's shift register.

(4) *Local* conditional control of arithmetic and logic functions.

*Motivation*: This improves the performance of various arithmetic operations.

(5) *Bidirectional* shift register.

*Motivation*: This allows more flexible data movement.

(6) An *X-grid interconnect*, allowing eight neighbors per PE.

*Motivation*: This gives a factor of two improvement (over the NEWS grid) in diagonal data movement.

Note that (3) and (4) give the BLITZEN PE a degree of MIMD control, which can improve the flexibility and efficiency of the machine.

Figure 1 presents the functional elements of one BLITZEN PE and shows a similarity to the PE in the MPP. Blocks with double line boundaries are storage devices. There are six single-bit registers labelled A, B, C, G, K, and P. Two devices hold multiple bits. One is a variable length shift register which, in conjunction with registers A and B, has a capacity of 32 bits. The remaining storage device is a 1024 bit random access memory (RAM). Arithmetic and logical operations are performed by a full adder and a logic block. The above elements communicate primarily over a single bit data bus. A four bit I/O bus provides a path to pads of the chip for connection to external storage devices. An I/O bus is shared among 16 PEs on a chip. Following paragraphs discuss features that represent significant departures of BLITZEN from the MPP.

### On-Chip Memory

An on-chip, static random access memory (RAM) is associated with each PE. From a processing point of view it is a 1024 by 1 bit RAM. A memory read operation reads the single bit specified by a ten bit address and places the value on the data bus. A memory write operation writes the value from the data bus into the location specified by a ten bit address.

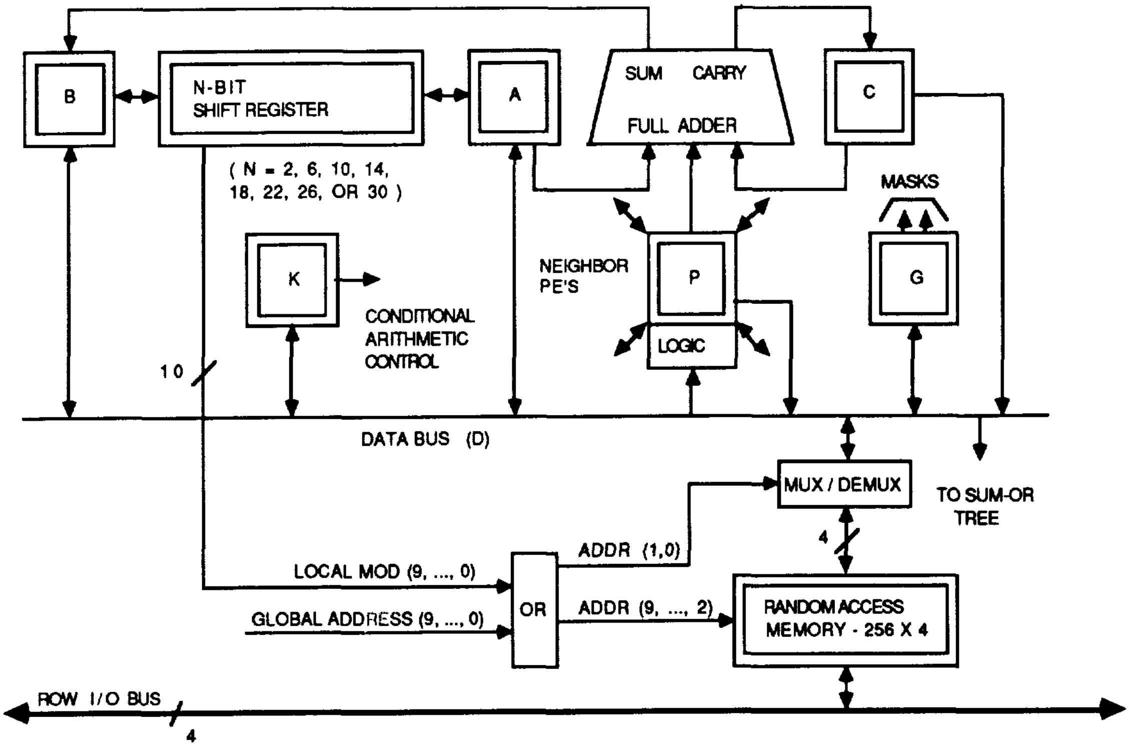


Figure 1. Functional elements of one BLITZEN PE.

Input/output operations view memory as a 256 by 4 bit RAM. I/O operations access memory using the eight most significant bits of the ten bit address, and transfer four bits between the I/O bus and memory.

Masking, the local control feature that can be used to enable or disable certain operations, is possible on all memory accesses.

### Local Address Modification

In a SIMD machine, the control unit issues an instruction to all PEs. If a memory operation is involved, one address is delivered to all PEs. In BLITZEN, the global address can be modified at each PE. Conventional processors generally modify an address that appears in an instruction by adding index or base register values, or extracting an address from some location for indirect use. In a SIMD machine, logic that handles local modification of addresses must appear at each PE and be locally decoded. That is, the logic must appear at each of the 128 PEs on this chip. To conserve chip area the modification chosen is the logical OR of the global address with ten bits from the shift register. This can simulate indexing when data structures begin on appropriate power of two boundaries where the least significant bits are zeroes. When normal (unmodified) memory operations are issued, the global address is unchanged.

Figure 1 shows a ten bit bundle of signals from the shift register labeled "local mod". The ten most significant bits of the 16 bit section of the shift register are used to provide local address modification.

We believe BLITZEN is the first massively parallel machine with the ability to modify the global SIMD memory address in every PE. BLITZEN has addressing logic with every PE. Previously, a SIMD machine developed by DEC, and the Connection Machine 2, allowed a large group of processors to share indirect addressing logic.

### Conditional Operations

BLITZEN provides additional new local control of PEs through the use of a programmable conditional operation test involving register K. When using the conditional feature, operations which are complements of each other can be performed at the same time in different PEs. The feature applies to operations involving logic at register P, or loading a value into register C. When a conditional operation is issued, processing is normal in all PEs where K = 0. In those PEs where K = 1 the results are complemented. Since both normal and complemented operations take place, based on testing a condition, this is like a restricted form of the high level IF-THEN-ELSE concept with both the THEN and ELSE clauses happening concurrently. When a conditional operation instruction is not used by the programmer, register K is available to hold a temporary value.

The conditional operation feature can be used to improve performance, by a factor near two, in non-restoring division algorithms where the next iterative step depends on the result of the current step. If the current step produces a negative partial remainder, the divisor is added at the next step. If the current step produces a positive partial remainder the divisor is subtracted at the next step. The approach to following both paths concurrently is to program the subtraction operation for conditional execution. By using the sign bit as the conditional flag in K, subtraction will take place in those PEs where K=0 and addition where K=1, as desired.

### Bidirectional Shift Register and Data Paths

The MPP shift register is unidirectional. In BLITZEN it has been made bidirectional. In the MPP all bits shift during a shift operation, even if they are not selected under the current length setting. Since BLITZEN uses a section of the shift register to hold local address bits, the register design has been changed such that bits do not shift if they are not selected. This also lets the shift register be used to hold temporary variables.

Several smaller changes have been made, as compared to the original MPP PE. Bidirectional paths are provided between the data bus and all registers except C. Since a masked write operation is possible, the equivalence function between registers P and G has been eliminated. For a more detailed description of the BLITZEN PE architecture, see Refs. 4 and 5.

## PE ARRAY CHIP ARCHITECTURE

### Organization of PEs and Functional Components

The above PE architecture is used as the basis for the BLITZEN VLSI processor array chip. A single chip contains 128 PEs, each with 1K bits of locally addressable memory.

By placing 128 PEs and their local memory on a single chip, we make a major step toward miniaturization of the BLITZEN machine. Only 128 of these PE array chips are required for an entire 16,384 PE BLITZEN machine (In comparison the MPP processing element array chip contains eight PEs, and the system requires a total of 2048 such chips. The Connection Machine has 16 PEs per chip.).

A single PE is a building block for the chip architecture. PEs are organized into an 8 by 16 array on the chip. They are interconnected with a two dimensional grid for communication between PEs, as discussed in the next section. Data is moved on and off the chip over a set of eight I/O buses, each with 16 PEs attached, as described in the section "BLITZEN I/O Scheme" Figure 2 shows the organization of PEs on the chip, including the X-grid interconnections, I/O buses, and some logic and control signals that are common to all PEs on the chip.

### Message Routing Capability on the BLITZEN Machine

**Why a Hypercube Interconnect Is Not Necessarily an Improvement Over a Grid** - One major design decision was *not* to use a logarithmic diameter interconnection network, such as the hypercube used by the Connection Machine. Instead we used a variant of the two dimensional grid, namely the X-grid (due to C. Fiduccia), with diameter 128, which is the square of the number of processors. In spite of our background in theoretical computer science, we concluded that a logarithmic diameter network would be impractical for our needs. The key problems with logarithmic diameter networks, such as the hypercube, are:

(1) The number (namely 896) of I/O pads that would be required for hypercube edges exiting a processing element chip with 128 PEs is impossibly large.

(2) The inter-PE wiring requires large amounts of area, both on-chip and between chips.

A decision to use a hypercube interconnection network would make it very difficult to highly integrate our machine. Because of pin count and network area requirements, we would have been limited to only 16 PEs per chip, and even then only have 1/16 of the I/O pins required for a full hypercube interconnect. The result would be an interconnect with perhaps no greater communication capabilities than a two dimensional grid.

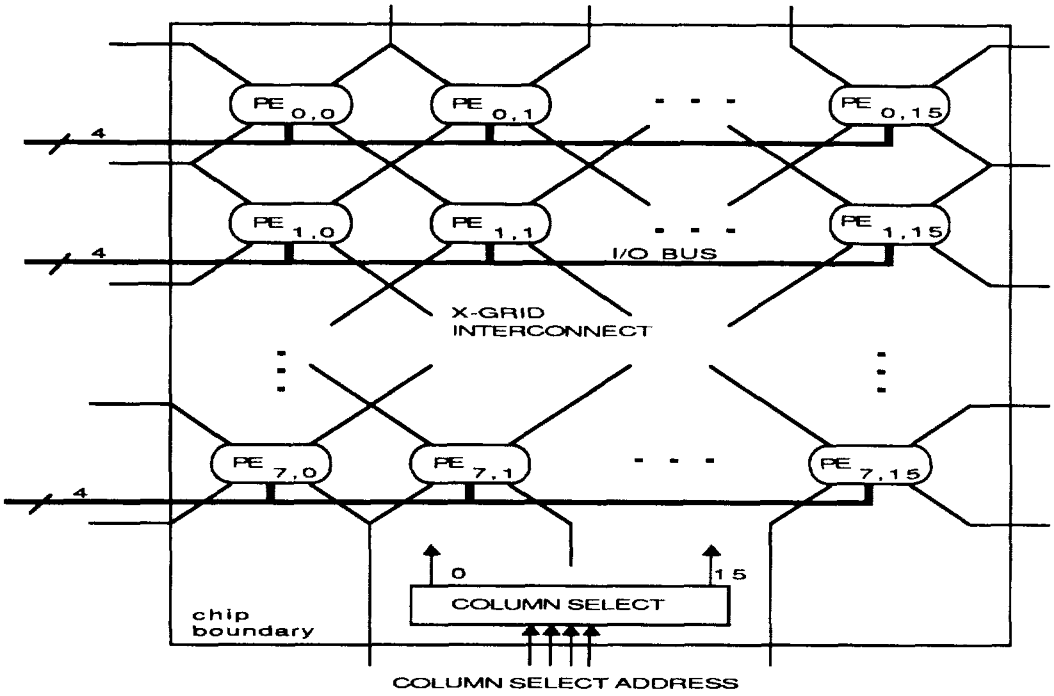


Figure 2. BLITZEN Chip Architecture

Another argument in favor of the grid interconnect is the empirical experience that a very large class of applications naturally require the grid interconnect.

The Connection Machine has some impressive built-in hardware for doing permutation message routing. Unfortunately, this routing circuitry uses a large fraction of their processor chip area and decreases the step rate of their machine. We decided that our need for a high performance, miniaturized architecture was more important than the need for message routing circuitry, (which can be replaced by software routing routines that are nearly as efficient.)

**X-Grid Interconnection** - Processing elements are interconnected in two ways on a chip: a grid interconnection for routing and a bus structure for I/O. Figure 2 shows the X-grid nearest neighbor routing network. PEs are arranged in a two dimensional grid with interconnection paths to neighbors in the eight compass directions N, NE, E, SE, S, SW, W, and NW. A routing operation transfers the state of P to the P register of a neighboring PE and accepts a new state from the PE in the opposite compass direction.

Four bidirectional routing connections are brought out of each PE from the four logical corners: NE, SE, SW, and NW. The connections intersect between PEs as shown in figure 2. A routing path is established by an operation which sends data out in one direction and accepts data in from one of the remaining directions. As an example, routing in the north direction can be achieved by sending P out to the NE and accepting P in from the SE. The data value on the SE input originated in the PE to the south. All PEs route the same direction in one processing cycle.

Eight paths can be established with four wires out of each PE by sending data on one wire, receiving data on one of the other three wires, and placing the remaining two wires in the high impedance state. This X-grid interconnects PEs on a chip and extends across chip boundaries so that an array of chips can be uniformly interconnected. Additional off-chip logic can provide various treatments of edges of the total array, as was done in the MPP system. The use of the X-grid allows a factor of two improvement in the frequently occurring case of diagonal data movement.

**BLITZEN I/O Scheme** - Data I/O is the critical path in any parallel machine. The MPP's I/O scheme is simple -- data is shifted in from the west edge of the array using the S–plane, and shifted out simultaneously along the east edge. In a BLITZEN system the array would be segmented along chip boundaries, so a natural extension to the MPP I/O scheme would be to have data flow in one side of a chip and out the other using the same S-plane idea. Thus BLITZEN would have data I/O occurring every 16 PEs, from west to east, using 32 pins.

At that time in the chip design activity, floorplanning predicted that the local static RAM should have a 256 by 4 aspect ratio. The RAM would have a four-bit interface, with further demultiplexing and multiplexing for the one-bit PE data bus. Since there were four data wires available per row of PEs on a chip, an alternative I/O approach was presented. The approach was to move, conceptually, the 16 output S-plane connections from the east edge to the west edge, and combine them with the 16 input S-plane connections to form eight bidirectional, four-bit I/O buses on each chip. Each four-bit bus is shared by the 16 PEs in a row. This scheme has several advantages, such as very high bandwidth, an easier interface for extending memory off-chip, the ability to broadcast data to all PEs simultaneously, fast data movement across the chip, and elimination of the S-plane.

Each chip has column select logic that is used in conjunction with the I/O buses. For normal I/O transfers, one PE in each row is active. The PE column index is the same for all rows and is given by a four bit address to the column select logic. In broadcast mode, data can be input to all PEs on a row, thus column selection is not used.

Video RAM (VRAM) chips are available with very high block data transfer rates, matching the rates of our PE I/O buses, and with four bit outputs, matching our four bit I/O buses. We plan to use one megabit VRAM chips, organized as 256K by 4, to augment the PE memory by 64K bits each. We will allow the 16 PEs along an I/O bus to share a vertically packaged VRAM chip.

## CHIP FEATURE SET

The BLITZEN PE array chip was designed by the Microelectronics Center of North Carolina (MCNC) with two orthogonal constraints: maximize both integration and speed. The chip incorporates over 1.1 million transistors on a die 11.0 by 11.7 mm. It was designed with MCNC's 1.25 micron, two level metal, CMOS process. It is packaged in a 168 pin pin grid array and is designed for the JEDEC 3.3 volt power supply standard. The operating frequency is 20 MHz worst case, and power dissipation is 1.0 watt.

The chip contains 128 PEs positioned in an 8 by 16 array. Internally, a three stage pipeline enables BLITZEN to execute an instruction every cycle, as shown in figure 3. During the first cycle a 23 bit SIMD instruction from the control unit is latched and decoded into a fully horizontal 59 bit microinstruction. During the second stage of the pipeline the microinstruction is broadcast to all 128 PEs. In the final stage the instruction is executed. By issuing a fully horizontal microinstruction, no additional decoding logic was needed in the PEs. The encoding of the 23 bit instruction was optimized to minimize the amount of internal decoding.

Data transfers on the I/O bus take place in a single cycle as shown in the timing diagram in figure 4. If the I/O buses are used as an interface to high density video RAMs, blocks of data can be transferred quickly to and from the chip. Routing communication on the X-grid also takes place in a single cycle.

Figure 5 is the floorplan of a single PE. Each PE has access to its own 1K bits of memory, which are internally organized as 32 by 32 bits. Multiplexing is provided to select four out of 32 bits for interfacing to that PE's I/O bus. When a PE accesses memory for an operand, further selection of one out of four bits is needed. Address calculation logic (predecode) is also needed at each PE to support the indirect addressing mode provided by local modification of the global address. The execution unit of a PE, including the shifter and ALU, contains approximately 1130 transistors.

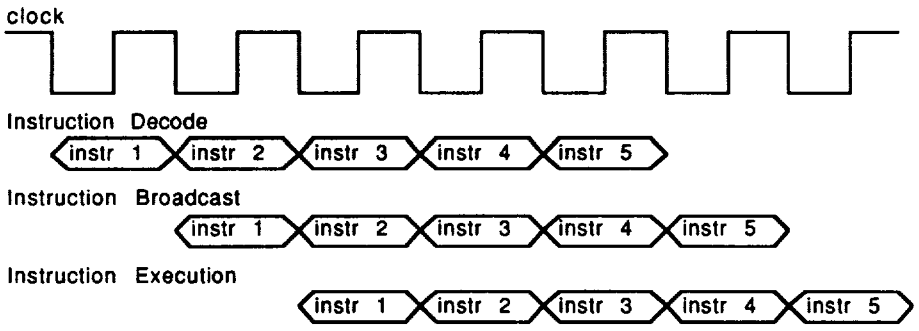


Figure 3. The instruction pipeline.

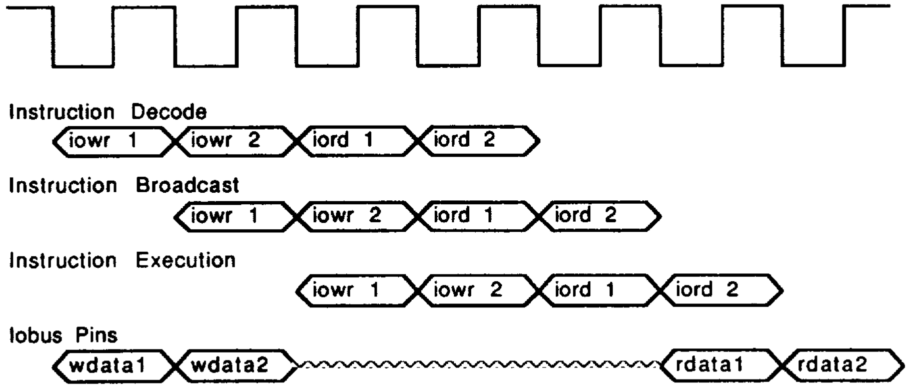


Figure 4. The instruction pipe for I/O bus transfers.

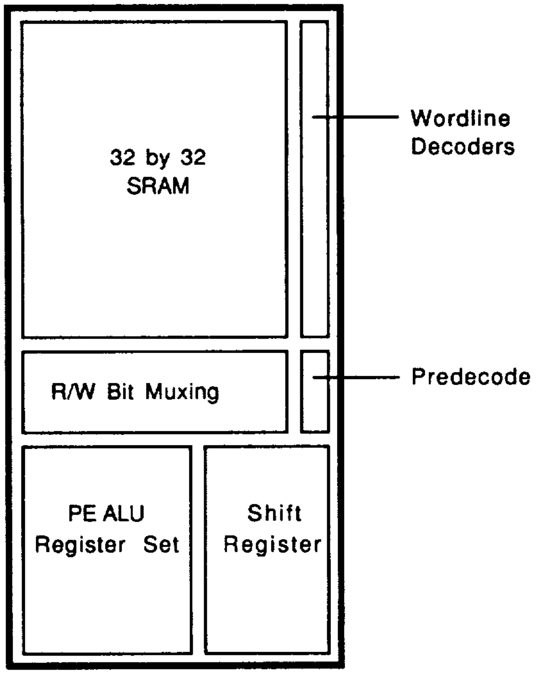


Figure 5. VLSI design floorplan for one PE.

## BLITZEN SYSTEMS

In a top level view of the system architecture, major components are organized around two buses. An internal bus supports data transfers between register and memory components. The second bus is used for transfers between BLITZEN and a host computer. Massive SIMD processing takes place in the processing array. Data in the on-chip local memory is supplied from off-chip, video RAM data memory, with the transfers considered as I/O operations with respect to the array.

Instructions are broadcast from the control unit to all PEs in the array. More specifically, operation codes originate in microcoded routines stored in control memory, and local memory addresses are generated from the register set. Together they form an array instruction. Control logic manages the register set and sequences the microinstructions. A scalar microprocessor can be included for use as the processor running an application program. It executes scalar instructions and sends calls for array instructions to the sequencing logic in the control unit.

Two external interfaces are planned. The host interface is a narrow path that matches the host wordlength. It is used for downloading programs (both application and microcode) and transferring data at low bandwidth between BLITZEN and the host with it's peripherals. High speed peripherals communicate with BLITZEN through custom peripheral interface logic. This path accesses the data memory and is potentially very wide for very high bandwidth.

### Data Memory

Each BLITZEN processing element has 1K bits of RAM on-chip for holding data. It is known that many applications can benefit from additional memory, but the 1K amount was governed by chip size and density limits. In BLITZEN, the memory limitation can be alleviated by off-chip data memory that is accessed across the I/O buses. The use of VRAM for this purpose was mentioned earlier. Data memory can be viewed as the primary data memory of the system with on-chip RAM treated as registers or data cache.

Using the high bandwidth I/O buses it is possible to change the content of all or part of the on-chip RAM very quickly. In one instruction cycle 32 bits (eight four-bit items) can be transferred between VRAM and each array chip. If the system is operating at 20 MHz, the total transfer rate is (4 bytes/chip)*(128 chips) per 50 nanoseconds, or 10.24 Gigabytes per second. In 128 instruction cycles, 32-bit data items can be transferred into (or out of) the on-chip RAM of each PE. In 4096 instruction cycles the entire 1K per PE RAM can be loaded. In 8192 cycles the content of RAM for the entire array can be swapped. Operating at 20 MHz, the time required to swap the total content is 409.6 microseconds.

### Holographic Routing

J. Reif, at Duke, has invented a holographic message routing system, using electro-optical components yielding very high routing rates. He is developing this device under DARPA/ARO contract. K. Johnson from the Electro-optical Computing Center at University of Colorado, Boulder, is constructing a prototype of this system. We are developing microcode to allow BLITZEN to use this electro-optical routing device.

### Programmer's Model

BLITZEN is a computing system whose primary computational resource is a single instruction stream, multiple data stream array processor with a massive number of processing elements. This massively parallel array operates in conjunction with several other major system components.

Programming BLITZEN takes place at several levels. At the lowest level is the machine language for the array. The hardware instruction set is specified in Ref. 4. Since the instruction set is concerned with single bit register transfers, it is not expected to be used by application programmers. Rather, it is the basis for a microcode development language, named BLITZ (Ref. 10), that couples array operations with control unit register transfers and sequencing operations. Commonly used routines corresponding to assembly language instructions such as load, store, add, floating point add, etc. are being written in BLITZ for inclusion in a microcode library whose routines can be called from a higher level language. An object oriented language based on C++ is being developed for application programming. High level language statements will be compiled into parallel assembly language statements that result in a calls to microcode routines which are executed on the array hardware.

### Parallel PE Array Simulator

Prior to the existence of hardware, a software behavioral simulator known as "Zyglotron" was developed (Ref. 10).It is a "full scale" simulator in that it can simulate the entire 16,384 PE array with very high performance. Zyglotron is being used for microcode development, and can allow the development of algorithms and high level software to proceed concurrently with hardware system development. As noted in the abstract of Ref 10, " The simulator has achieved such high performance by taking advantage of a natural mapping that exists between massively parallel bit-serial machines and the vector architecture used in many high performance scientific super-computers." The simulator runs on the CONVEX C-1 vector processing machine and is written in C and in the CONVEX C-1 assembly language.

## CONCLUSION

This paper has reported on the architecture and VLSI design of a new massively parallel processing array chip. The BLITZEN PE array chip, containing 1.1 million transistors, has been submitted to the Microelectronics Center of North Carolina for fabrication. The chips are the basis for a highly integrated, miniaturized, high performance, massively parallel machine that is currently under development.

The work reported in this paper resulted from the efforts of a group of researchers, mentioned in the overview section, participating in this project with the support of the Microelectronics Center of North Carolina. We also benefitted from discussions with Kenneth Batcher of Loral Systems Group concerning architecture of the MPP and local address modification schemes; with John Dorband of NASA Goddard SFC concerning conditional operations; and with Charles Fiduccia of General Electric who described their cross-omega machine with an eight neighbor grid interconnect. The interest and support of Milt Halem, NASA Goddard SFC, has been crucial to the success of this project.

## REFERENCES

1. Batcher, K. E., "Design of a Massively Parallel Processor", *IEEE Trans. on Computers*, C-29(9), p 836-840.

2. Batcher, K. E., "Array Unit", *The Massively Parallel Processor*, J. L. Potter, Editor, The MIT Press, 1985.

3. Batcher, K. E., "The Architecture of Tomorrow's Massively Parallel Computer", *Proc. of the First Symposium on the Frontiers of Massively Parallel Scientific Computation,* 1986.

4. Blevins, D. W., E. W. Davis, and J. H. Reif, "Processing Element and Custom Chip Architecture for the BLITZEN Massively Parallel Processor", *MCNC Technical Report TR87-22*, 1987.

5. Davis, E. W., and J. H. Reif, "Architecture and Operation of the BLITZEN Processing Element", *Proc. of the Third International Conference on Supercomputing*, Boston, MA, May 1988.

6. Fischer, J. R., et al, "Report from the MPP Working Group to the NASA Associate Administrator for Space and Science Applications",*NASA Technical Memorandum 87819,* November,1987.

7. Hillis, W. D., *The Connection Machine*, The MIT Press, 1985.

8. Potter, J. L., Ed., *The Massively Parallel Processor,* MIT Press, 1985.

9. Rosenberg, J. B., and E. W. Davis, "BLITZ: Blitzen's Microcode Assembly Language Design Document, *MCNC Technical Report TR88-14*, 1988.

10. Rosenberg, J. B., J. Becher, and N. Hooke, "Vectorization Enables Full Scale Simulation of Massively Parallel (SIMD) Architectures", *Proceeding of the Third International Conference on Supercomputing*, Boston, MA, 1988.

11. Sharp, J. A., *An Introduction to Distributed and Parallel Processing*, Blackwell Scientific Publications, 1987.

# MASSIVELY PARALLEL COMPUTING SYSTEM FOR RESEARCH AND DEVELOPMENT APPLICATIONS

W. Keith Johnson
Amber Engineering, Inc.
Goleta, CA

## ABSTRACT

A description of Amber Engineering's SIMD Processor Development System (SPDS) is presented. The SPDS is designed to provide a cost-effective, turn-key solution to users desiring to explore massively parallel computing applications. The SPDS, which is based upon the Geometric-Arithmetic Parallel Processor (GAPP) integrated circuit, contains a two-dimensional array of between 2,304 and 10,368 processing elements. This processing element array operates in a classical single-instruction/ multiple-data fashion. The SPDS processing electronics may be connected to any "AT-bus" compatible computer via an Amber provided interface card. The SPDS works in conjunction with an optional frame grabber card to acquire RS-170 imagery and to display processed results on a standard analog monitor. Software bundled with the SPDS runs under the MS-DOS operating system and includes a compiler/linker/ microcode generator for the GAPP array, a GAPP utility library in source code form, and a menu-driven user interface with interactive symbolic debugging capability.

Keywords: Massively Parallel Processing, Single-Instruction/Multiple-Data (SIMD), Geometric Arithmetic Parallel Processor (GAPP), Development System.

## INTRODUCTION

Massively parallel computing systems are emerging from highly-specialized laboratory and research environments into mainstream processing applications. Impeding the progress of this trend has been the lack of low-cost development system products designed to expedite the generation of application software for massively parallel computers. To date, development environments for massively parallel computers either emulate the target platform with a general-purpose machine (very slow) or reside upon the target platform itself (very expensive). Amber Engineering has recently released a development system for massively parallel computers that alleviates both of these problems.

### SIMD Processor Development System

Amber Engineering's SIMD Processor Development System (SPDS) provides a complete development environment for the Geometric-Arithmetic Parallel Processor (GAPP) integrated circuit (Ref. 1). The SPDS incorporates an array of GAPP integrated circuits, an array controller circuit card, and compiler/debugging software into an integrated development platform. Application code generation is expedited by the fact that execution is performed in hardware at the full rated speed of the GAPP integrated circuit. The cost of the SPDS is minimized by utilizing an "AT"-class personal computer as the host machine, by constraining the input/output operation of the SPDS to the DMA rate supported by the host computer, and by utilizing a moderately sized GAPP array as the execution engine. Although a moderately sized GAPP array is used (from 48-by-48 up to 108-by-96 processing elements), the SPDS software allows the GAPP array to process images of arbitrary size via successive subimage processing (windowing). Once application software has been developed and validated upon the SPDS, the resulting executable microcode may be directly ported to a user developed GAPP execution system. In many applications, the SPDS itself may be used as an execution system with the addition of a real-time input/output interface circuit card.

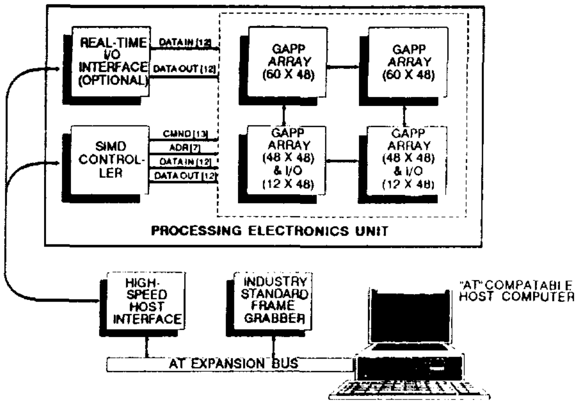

REAL-TIME I/O INTERFACE (OPTIONAL)
DATA IN [12]
DATA OUT [12]
GAPP ARRAY (60 X 48)
GAPP ARRAY (60 X 48)
SIMD CONTROL-LER
CMND [13]
ADR [7]
DATA IN [12]
DATA OUT [12]
GAPP ARRAY (48 X 48) & I/O (12 X 48)
GAPP ARRAY (48 X 48) & I/O (12 X 48)
PROCESSING ELECTRONICS UNIT
HIGH-SPEED HOST INTERFACE
INDUSTRY STANDARD FRAME GRABBER
"AT" COMPATABLE HOST COMPUTER
AT EXPANSION BUS

Figure 1. SPDS Block Diagram

## SPDS HARDWARE ARCHITECTURE

As depicted in Figure 1, the SPDS is comprised of a Processing Electronics Unit that is connected to a host computer via a High-Speed Host Interface. Contained in the Processing Electronics Unit are from one to four GAPP Array circuit cards, an SIMD Controller circuit card, and an optional Real-Time I/O Interface circuit card. The host computer is an "AT"-class personal computer (including 80386-based machines that support the "AT" bus). Installed on the AT expansion bus of the host computer are the High-Speed Host Interface circuit card and an optional Frame Grabber circuit card.

### Host Computer

The host computer facilitates the man-machine interface operations of the system, and is also responsible for the compile/link/download & control functions. In addition, the host computer's mass storage device is utilized to store source code, intermediate compiled code, executable microcode, and image data for use by the Processing Electronics Unit. If equipped with the optional Frame Grabber circuit card, the host computer may perform RS-170 image acquisition and display. The Frame Grabber incorporated into the SPDS is the Imaging Technologies model FG100-640/2 AT. All interactions between the host computer and the Processing Electronics Unit are accommodated via the High-Speed Host Interface circuit card, which is capable of transferring program and image data between the host computer and the Processing Electronics Unit at the DMA rate supported by the host computer. The DMA rate is between 500,000 and 1,000,000 bytes per second depending upon the manufacturer and model of the host computer.

### Processing Electronics Unit

The Processing Electronics Unit, shown in Figure 2, is packaged in a self-contained enclosure that houses the GAPP Array circuit cards, the SIMD Controller circuit card, and, optionally, a Real-Time I/O Interface circuit card. The Processing Electronics Unit is responsible for executing the program(s) supplied by the host computer upon image data supplied by the host computer (development mode) or upon image data supplied by the Real-Time I/O Interface (real-time execution mode). The Processing Electronics Unit has dimensions of 15 by 28 by 26 inches making it ideal for lab or office environments.

The SPDS can be configured with one, two, or four GAPP Array circuit cards, thus providing main array sizes of 48-by-48, 48-by-96, and 108-by-96 processing elements respectively. In addition to the main processing array, the one, two, or four GAPP Array circuit cards will provide corner-turn arrays of 12-by-48, 12-by-96, and 12-by-96 processing elements, respectively, to accommodate input/output reformatting for the main GAPP array. Each GAPP Array circuit card contains 40 GAPP integrated circuits as well as the necessary buffers, transceivers, and decoding logic necessary for the proper operation of the card. Jumpers installed on the GAPP Array circuit card allow the card to be configured as a 48-by-48 element main array with a 12-by-48 element corner-turn array, or as a 60-by-48 element main array. The SPDS has been designed for field expansion such that users may enlarge their GAPP arrays by installing additional GAPP Array circuit cards.

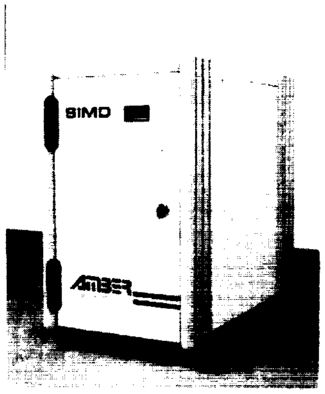


Figure 2. SPDS Processing Electronics Unit

The SIMD Controller circuit card broadcasts command and address information to the GAPP Array circuit cards during program execution. The SIMD Controller performs this operation at a 10 megahertz rate, the maximum rate allowed by the GAPP integrated circuit. Prior to program execution, microcode is downloaded from the host computer into the SIMD Controller's 64K deep by 48-bit wide writeable control store memory. During program execution, the microcode is read out from the control store memory, under the direction of a program sequencer, to cause the GAPP processing elements to perform the desired operations. Additionally contained on the SIMD Controller is a 256 bit dynamic constant memory. The dynamic constant memory allows the host computer to download run-time constants, such as thresholds or adaptive filter coefficients, for subsequent injection into the GAPP processing array.

Additionally contained on the SIMD Controller circuit card is a 256K byte input/output buffer memory. This buffer streamlines data transactions between the host computer and the GAPP array circuit cards. The input/output buffer memory is divided into two independent blocks. While one block is communicating with the host computer, the other block may send data into and receive data from the GAPP array, thereby overlapping host communications with processing. The roles of the two buffer blocks are reversed after each successive GAPP processing window, i.e. the block communicating with the host will then communicate with the GAPP array and the block communicating with the GAPP array will communicate with the host. Each of the two blocks is furthermore segmented into separate sections for input to the GAPP Array and output from the GAPP Array so that input and output may occur simultaneously.

Although designed as an interactive development system, the SPDS may be converted into a high-speed execution platform with the addition of a Real-Time I/O Interface circuit card. This circuit card will place the image acquisition and display functions (normally performed by the frame grabber and host computer via software drivers) into hardware for real-time performance. The Real-Time I/O Interface is responsible for partitioning the incoming video stream into GAPP array-sized regions of pixels for injection into the GAPP array. The Real-Time I/O Interface is also responsible for reintegrating the GAPP array output data into a contiguous image for display. In both the input and output functions, the Real-Time I/O Interface circuit card must accommodate overlap between adjacent processing windows in order not to introduce artificial edge effects during subimage processing.

## SPDS SOFTWARE ENVIRONMENT

The SPDS includes a software package that is combined with the hardware to provide a complete development environment. The software consists of four major elements - a user interface shell, a GAPP compiler, a GAPP applications/utility library, and embedded hardware drivers.

The user interface shell is a multiwindow, menu-driven environment in which all aspects of the SPDS system may be controlled. When entering the user interface shell, the user is prompted to select one of four modes of operation:

1. Algorithm Development Mode,
2. Image - No Corner Turn Mode,
3. Corner Turn Development Mode,
4. Image - with Corner Turn Mode.

These four modes of operation allow the user to tailor the operation of the SPDS to the task at hanu as the user is progressing with the development of a GAPP application. Extensive built-in help files are available to assist the user at any time via the user interface shell.

### Algorithm Development Mode

In the algorithm development mode, the SPDS works with an image that is identical to the size of the GAPP array. The GAPP-sized subimage may be selected from a 640 by 480 frame grabber image under software control. Once a subimage of data has been selected, the user is provided with a menu set that allows him to call up an editor to generate source code, to compile and link the source code into executable microcode, and to cause the SPDS system to execute the microcode. Additionally, a number of debug facilities are included in the algorithm development mode menu structure. The debug facilities include breakpoint and stepping functions as well as GAPP RAM and GAPP register upload/download utilities. GAPP RAM uploading/downloading functions may be performed symbolically (i.e. as GAPP variable names) or absolute (i.e. as GAPP RAM addresses). When uploading GAPP information, the data may be sent to the frame grabber for presentation as video information or to the host computer for presentation as numerical information. Figure 3 shows the top-level menu structure for the algorithm development mode along with a window of GAPP RAM information displayed in numerical format. After a user has completely debugged a GAPP program in the algorithm development mode on single subimage data, the user may move on to the image-no corner turn mode to test the algorithm on full images.

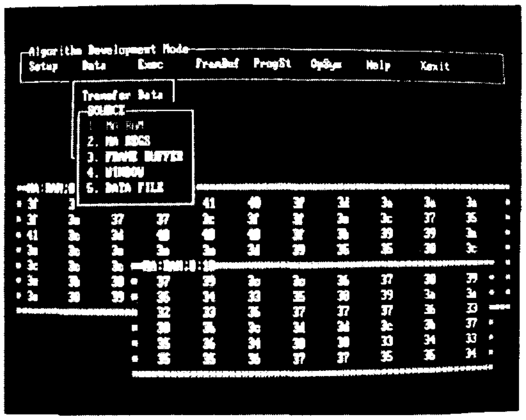


Figure 3. Algorithm Development Mode Menu

### Image-No Corner Turn Mode

In the image-no corner turn mode, the SPDS works with the full 640 by 480 frame grabber image (or other full image size as specified by the user). The GAPP array processes the larger full image by splitting the full image into GAPP-sized subimages and sequentially processing each subimage. To avoid artificial edge effects between adjacent subimages, the SPDS software overlaps adjacent subimages. The amount of overlap is a function of maximum algorithm kernel size, and as such, the overlap parameters may be arbitrarily set by the user. After a frame of data has been captured by the frame grabber or uploaded from the host computer's hard disk, the user may instruct the SPDS to execute a number of predefined programs on the data with the results automatically displayed. Menu options in the image-no corner turn mode include input/ output RAM location selection, kernel size selection, microcode file selection, image transfer functions, and

ORIGINAL PAGE
BLACK AND WHITE PHOTOGRAPH

execution control functions. Figure 4 shows the top-level menu structure for the input-no corner turn mode.

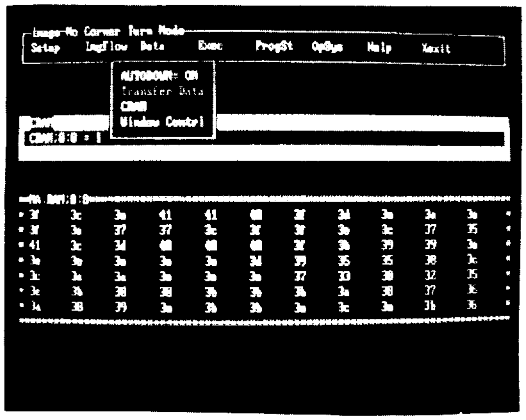


Figure 4. Input-No Corner Turn Mode Menu

In the image-no corner turn mode, all input/output functions are performed via standard, built-in SPDS hardware drivers. Additionally in this mode, input/output and processing occur sequentially instead of overlapped. To generate custom input/output code for the GAPP Corner Turn array and/or to overlap corner turn code with main array processing code, the last two SPDS modes of operation are provided.

### Corner Turn Development Mode and Image-With Corner Turn Mode

The final two modes of operation are reserved for constructing custom input/output routines (corner-turn programs) and for merging corner-turn programs with main processing programs. Custom corner-turn programs may be developed for applications that have input/output requirements that are not met by the standard corner-turn program supplied with the SPDS system. Furthermore, corner-turn programs may be merged with main array processing programs so that input/output and processing occur simultaneously in the GAPP array for maximum performance.

### NCR-GAL Compiler

The GAPP array within the SPDS utilizes NCR's GAPP Algorithm Language (GAL) as source code. NCR-GAL combines the structure and syntax of the "C" programming language with GAPP-specific assembly level mnemonics resulting in source code that is easy to develop and maintain yet compiles to extremely efficient object code. The NCR-GAL compiler included with the SPDS reduces the source code to an intermediate form. A linker and microcode generator, also included with the SPDS system, transforms this intermediate code to executable form which may be downloaded to the SIMD Controller.

## SPDS PERFORMANCE

The throughput of the SPDS is indicated in Table 1 for a variety of image processing tasks. As indicated in the table, the millions of operations per second rating for the SPDS is a function of the operation complexity, the size (number of bits) in the operands, and the size of the GAPP array.

| | MOPS PER ARRAY SIZE | | |
|---|---|---|---|
| OPERATION | 48x48 | 48x96 | 108x96 |
| BOOLEAN OPERATION (1-BIT) | 5760 | 11500 | 25920 |
| ADDITION (8-BIT) | 853 | 1706 | 3838 |
| MULTIPLICATION (8-BIT) | 65.0 | 130 | 292 |
| MULTIPLICATION (16-BIT) | 15.6 | 31.2 | 70.2 |
| 3x3 CONVOLUTION (8-BIT) | 16.7 | 33.4 | 75.2 |
| 5x5 CONVOLUTION (8-BIT) | 6.0 | 12.0 | 27.0 |
| 3x3 MEDIAN FILTER (8-BIT) | 8.0 | 16.0 | 36.0 |

Table 1. SPDS Performance

## SPDS APPLICATIONS

The SPDS is currently being utilized to perform front-end processing functions for a passive airborne warning application. As shown in Figure 5, a standard SPDS system is outfitted with an application-specific Real-Time Processor Interface circuit card to receive digitized video from an infrared-sensitive telescope. The telescope provides 360 degree azimuth coverage to detect incoming threat objects (helicopters, aircraft, and missiles) from all directions.

The custom Real-Time Processor Interface provides delay equalization, nonuniformity correction, intensity transformation, and frame buffering functions upon the incoming video. The buffered video data is then placed into the GAPP array contained in the SPDS under the direction of the SIMD Controller. The GAPP array performs spatial and spectral filtering on the video data and then prescreens the filtered image for candidate targets. The processed image, along with detection point indication flags, is passed back out from the GAPP array into the Real-Time Processor Interface where each detection point location and associated intensity information are recorded. The detection point data is then handed off to the Compaq 386/20 Host Computer for final classification and track file processing.

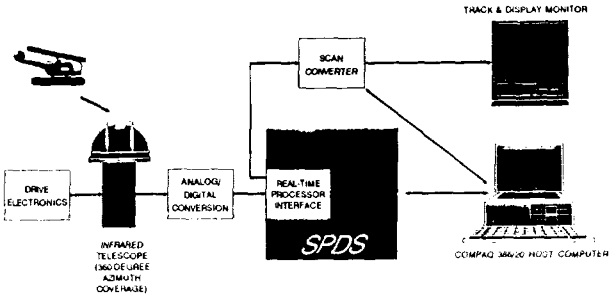

TRACK & DISPLAY MONITOR
SCAN CONVERTER
DRIVE ELECTRONICS
ANALOG/ DIGITAL CONVERSION
REAL-TIME PROCESSOR INTERFACE
SPDS
INFRARED TELESCOPE (360 DEGREE AZIMUTH COVERAGE)
COMPAQ 386/20 HOST COMPUTER

Figure 5. SPDS Applied to Passive Airborne Warning System

Additionally output from the Real-Time Processor Interface is a uniformity corrected video stream which is displayed upon a monitor after passing through a scan converter. Track and mission symbology generated by the Host Computer is also received by the scan converter and superimposed upon the video data for presentation to the system user.

## REFERENCES

1. "Geometric Arithmetic Parallel Processor Data Sheet," NCR Corporation, 1987.

# THE APx ACCELERATOR

E. Abreu, D. Jenkins, M. Hervin, D. Evans

Visionary Systems, Inc.
New Haven, CT 06511

## ABSTRACT

The APx Accelerator is an SIMD Parallel Processor system designed to provide very high computing power in a PC/workstation environment. The APx is an expandable system and provides from 64 to 256 16-bit processors which provide peak instruction rates from 800 to 3200 MIPs. The individual processors in the APx Accelerator are 16-bit RISC processors which are quite powerful and versatile. In addition, pairs of 16-bit processors can be configured to operate in 32-bit mode under software control. IEEE format single precision floating point operations are supported in 32-bit mode with peak ratings from 40 to 160 MFLOPs.

This presentation deals with the architectural and implementation features that work together in the APx Accelerator to achieve high sustained system performance for a significant set of compute-intensive functions. These features include VLSI integration, memory bandwidth, concurrency of operations, inter-processor communications, processor selection mechanisms, and I/O bandwidth.

Keywords: SIMD, RISC, Parallel Processors, Workstation, PC, Array Processors.

## INTRODUCTION

As processing needs of engineers, scientists and professionals continue to outpace the performance improvements of conventional machines, computer scientists have come to realize the advantages of parallel processing techniques to keep up with these demands. Parallel processors which contain thousands of processing elements have been developed to try to keep in pace with today's processing requirements. Examples of these machines include Thinking Machines' Connection Machine CM2 (Ref. 1), Goodyear's Massively Parallel Processor MPP (Ref. 2), ICL's Distributed Array Processor DAP (Ref. 3) and the NCUBE Ten (Ref. 4). Each of these machines has particular strengths and features that are not available with the others. These machines may also be characterized as being too large and expensive to be widely available in a desktop environment.

## The APx PROCESSOR SYSTEM

The APx Parallel Processor System from Visionary Systems Inc. (VSI) is a desktop supercomputer that offers the performance advantages of massively parallel processors in a desktop and interactive environment. The APx is a result of the research and development efforts performed at the ITT Advanced Technology Center on Cellular Array Processors (Ref. 6), and continued through product development at VSI. The major design goals of the APx are the following:

- True supercomputing performance: Floating point performance equivalent to supercomputers and integer and memory bandwidth approaching that of massively parallel processors.
- Price range equivalent to that of technical workstations. Substantial reduction in cost is afforded through the use of custom VLSI technology.
- Compact size that is expandable and scalable to fit the user's present and future needs.
- Architecture that provides uniform acceleration over a wide range of applications rather than very specific and limited sets of routines.

## THE APx ARCHITECTURE

The APx is a data parallel computing system based on the Single Instruction, Multiple Data (SIMD) architecture. In an SIMD machine, all of the processors execute the same instruction sequence from a common controller. Figure 1 shows a block diagram of the system. The block diagram reveals the following features:

- There is a single program Controller which contains the sequencer, program memory, scalar address generator and host (AT Bus) interface.
- The Processor Array contains from 64 to 256 powerful RISC-based, 16-bit processing elements. The Processor Array may be expanded in increments of 64 processors.

CH2649-2/89/0000/0413$01.00 © 1988 IEEE

- Both program memory and data memory are memory mapped into the host workstation's memory space via the AT Bus. The host workstation or PC then becomes the front end computer, allowing users to retain their user-friendly and familiar environments.

- I/O channels are provided for high-speed data transfers between the processor array and high-speed peripherals like high resolution monitors and image scanners for real time graphics and imaging.

Figure 2 shows a graphical presentation of the basic APx system, which consists of a controller board and from one to four processor boards each containing 64 processing elements. Each processor board contains four custom VLSI chips, each containing 16 powerful RISC-based 16-bit processors with substantial on-chip memory. These boards are AT-sized, and can plug directly into available slots of popular workstations and PCs. Larger systems are housed in their own self-contained units which can be connected to the host computer via an interconnect cable.

Programming the APx will be done in C, FORTRAN, or the PAC assembly language provided by VSI. PAC is a parallel programming language designed to take full advantage of the features of the APx hardware but simple enough to be easily understood and learned. By utilizing the host workstation/PC as a front-end, the user can retain major amounts of software, and simply replace compute-intensive routines and applications with routines that utilize the APx. These APx routines may have come from subroutine libraries supplied by VSI or may have been written by the user.

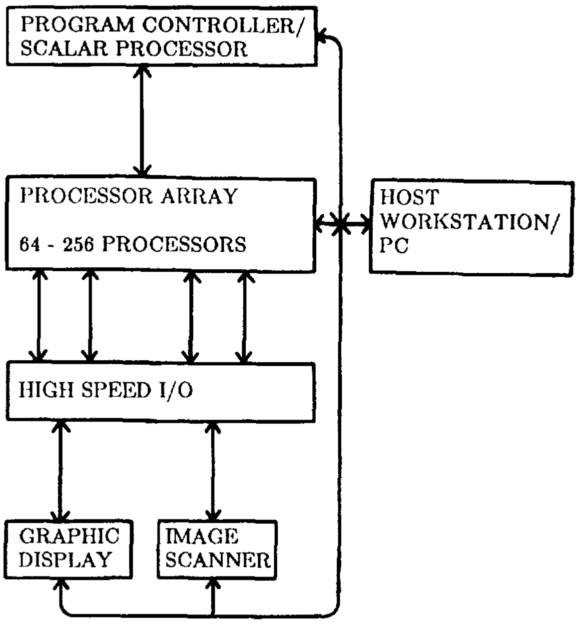


Figure 1. APx System.

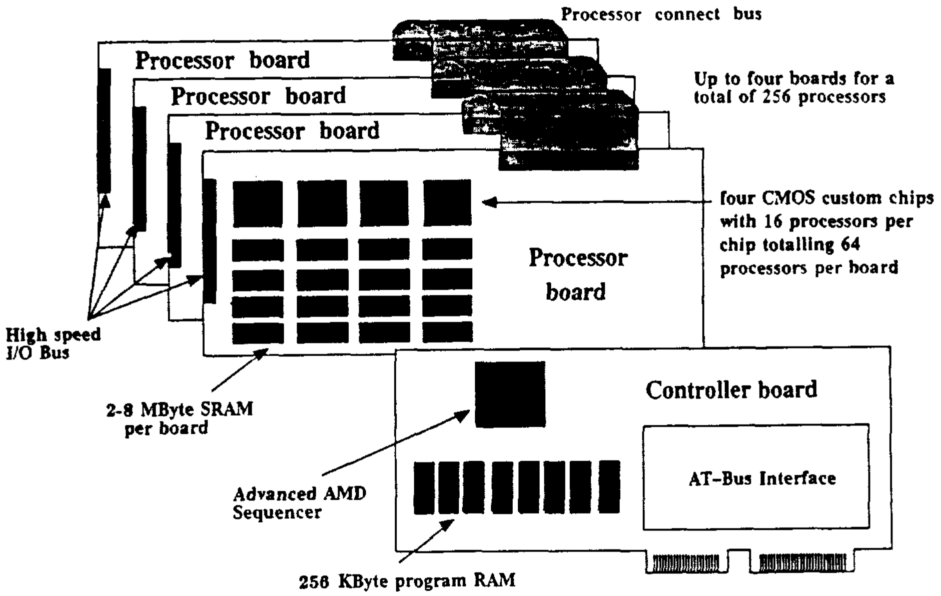


Figure 2. APx Hardware.

## PROCESSING ELEMENTS

At the heart of the APx system are the processing elements. Figure 3 shows a block diagram of an individual processing element, which is a powerful 16-bit processor as opposed to very simple 1-bit serial processors found in a number of massively parallel computers. The APx processing elements have several features to increase overall system performance. These are:

- Pairs of 16-bit processors can be configured under software control to act as single 32-bit processors. Each 16/32-bit processor has an integral multiplier and IEEE format single precision floating point unit. An integral multiplier/floating point unit eliminates the need to transfer data to a separate co-processor which could slow down overall performance.

- Each processor contains 16 general purpose registers and has access to a substantial amount of on-chip memory and a much larger external memory. Furthermore, the processors employ a Load/Store architecture that permits concurrency between arithmetic and memory operations.

- Addressing of memory can be done in two ways. The first is through the use of a scalar address generator which provides a single address to be used by all processors. The second approach is through each processor's register set wherein each processor supplies an independent address to its own memory. VSI's research into applications showed that both scalar and independent addressing modes contribute to the applicability of a parallel processor over a wide range of problems.

## APx COMMUNICATIONS

Efficient inter-processor communication was a major design implementation goal in the APx system. Because of VLSI technology and the ability to package an entire system in a compact enclosure, VSI was able to incorporate two necessary communication mechanisms in the APx. These are:

- A high speed mesh topology: Each processor in the Processor Array is connected with its left, right, up and down neighbors. The processors at the end of the array are connected to form a torus. What differentiates the APx implementation from other machines that use a mesh or a hypercube topology is that the APx uses 16-bit busses to connect the processors as opposed to 1-bit serial lines employed by other parallel machines. As a result, an APx processing element can transfer a 16-bit word of data in one cycle.

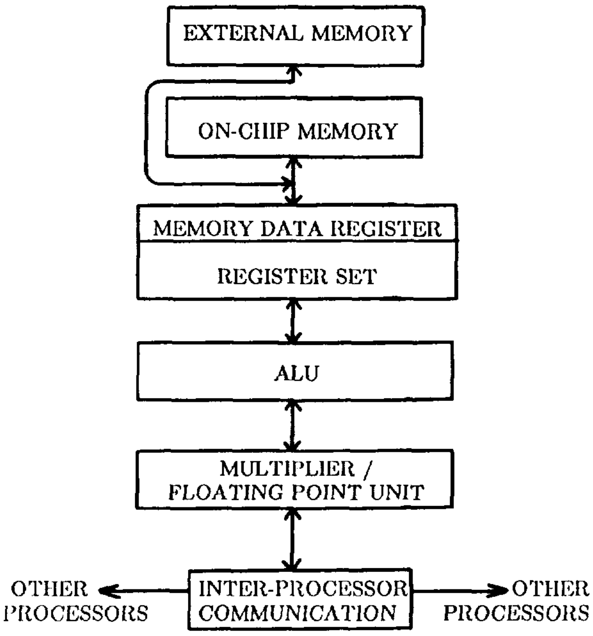


Figure 3. APx Processing Element.

- A high-speed **32-bit broadcast** bus: The broadcast bus allows the scalar controller or a selected processing element to send data to all or a subset of processing elements. Because of VLSI technology, a 32-bit number can be broadcast globally in a single cycle.

## APx PROCESSOR SELECTION

VSI realizes that it is not enough to have powerful processors with very high memory and communication bandwidths, but it is also necessary to be able to efficiently control and coordinate these processors. The APx is equipped with a number of processor selection mechanisms to enable efficient coordination between processors. These mechanisms can be classified into the following groups:

**Intra-processor selection** - In an SIMD machine, the instructions are broadcast to all processors. It is important that each processor be able to conditionally execute these instructions based on its own internal status and flags. In the APx, each processor contains a mechanism called the Vector If Else stack for this purpose. The result of a conditional test is pushed onto this stack. The status of the stack then determines whether the processor is activated/deactivated to execute the instructions being broadcast by the controller. The Vector If Else mechanism supports 16-deep nesting of conditional instructions and provides support for high-level constructs like If, Else, Endif, etc.

**Inter-processor selection -** Frequently, it is important to be able to single out a particular processor out of the entire Processor Array. This selected processor is often a source processor during broadcast operations as explained earlier. Two inter-processor selection mechanisms are provided for this purpose. The first mechanism enables the programmer to sequentially select processors in an ascending or descending manner, treating the processors in the array as a string of processors. The second mechanism is a Find/Drop capability wherein the first Find operation causes the system to scan all the processors and select the first active processor. Each succeeding Find then selects the next active processor. Hardware is provided such that the scanning and selecting activity is completed in one cycle.

**Global Status -** A status which indicates if "any" of the processors are active is available to the controller. The controller can then use this information to conditionally jump over program code. This feature is useful in implementing high-level constructs like WHILE ANY which is useful for sparse matrix operations.

## APx I/O CAPABILITY

The APx is designed to be ideal for applications such as high-speed graphics, for visualization of engineering and scientific analysis, and for advanced image processing. As such it is very important that the APx not only have superior computational capability, but it should also be complemented by an I/O system which can take advantage of the high processing bandwidths.

In addition to the industry standard AT Bus, separate I/O channels are provided in the APx which are geared for interface with high-speed peripherals for graphics and imaging. An I/O channel is a separate independently controlled and asynchronously timed bus available for each group of 64 processors. Double buffered I/O registers are provided in each processor. An external I/O controller can transfer data to and from these I/O registers concurrent with other operations in the APx. This is possible because the instructions to transfer data between the I/O registers and the I/O bus are independent of the other instruction fields. Interrupt logic is provided in the Controller such that when the I/O controller has filled the I/O registers in each processor, it can activate an interrupt routine which transfers I/O register data directly into general purpose registers, on-chip memory or external memory. An I/O board is being designed to interface the I/O channels directly to frame buffers for graphics/image controllers.

## APx SOFTWARE ENVIRONMENT

The APx system software environment is based upon the host workstation's operating system. This has the distinct advantage that users retain a familiar and friendly interface (i.e., UNIX, MS-DOS). Application programs are written using the host's existing environment and programming languages, but key compute intensive routines are run on the APx. These routines are either called from a set of libraries supplied by VSI or from user programmed routines. Data is shared between the host application program and the APx routines, since the APx program and data memories are mapped into the host computer's address space using the AT Bus.

Users can program their own routines in the PAC assembly language or in C or FORTRAN. At the lowest level, the PAC assembly language gives the user access to all the capabilities of the APx hardware. PAC instructions are very similar to those in RISC-based instruction sets, with the only difference being that data operations like ADD, MULT, LOAD, and STORE are really vector operations that are executed on all processors.

VSI expects to supplement PAC with two different types of C and FORTRAN subroutine compilers; those which compile code for a single processor and those which vectorize code to use the entire processor array. The first type of compiler will compile C or FORTRAN routines for a single processor, with interprocessor communication explicitly controlled by the program. This gives the user the benefits of programming in a high-level language while retaining the efficiency of explicitly managing the parallel execution and data storage mapping on individual processors. The second type of compiler will vectorize C or FORTRAN subroutines for users who want to automatically convert existing subroutines to use the APx hardware. In this case, the compiler maps the program data onto the processors and generates the necessary code for interprocessor communication if one processor needs data stored in another processor's registers or memory.

In addition to the compilers and assembler, VSI will provide a linker, loader, debugger, and simulator to aid in software development and debugging.

## SPECIFICATION AND PROJECTED PERFORMANCE

The APx specifications are shown in Table 1. It indicates that the APx performance indeed competes with supercomputers and massively parallel processors. Performance for various selected subroutines on a 256-processor system (AP256) using 16-bit precision are shown in Table 2. Again the projections compare favorably against very expensive supercomputers and massively parallel processors. We believe that performance like this that is widely available in a desktop environment will not only answer the present needs of engineers and scientists but will also open up markets for new capabilities in computing.

|  | AP64 | AP128 | AP256 |
|---|---|---|---|
| Number of Processors | 64/32 | 128/64 | 256/128 |
| Peak MIPS | 800/400 | 1600/800 | 3200/1600 |
| Peak MFLOPS (32-bit IEEE format) | 40 | 80 | 160 |
| Integer Add, Logical, Move, and Shift (MIPs) | 800/400 | 1600/800 | 3200/1600 |
| Integer Multiply (MIPS) | 160/40 | 320/80 | 640/160 |
| Floating Point Add (MFLOPS) | 40 | 80 | 160 |
| Floating Point Multiply (MFLOPS) | 40 | 80 | 160 |
| Total On-Chip RAM (Kbytes) | 32 | 64 | 128 |
| Total External Data RAM (Mbytes) | 2 to 8 | 4 to 16 | 8 to 32 |
| External Memory Bandwidth (Mbytes/s) | 200 | 400 | 800 |
| On-Chip Memory Bandwidth (Mbytes/s) | 800 | 1600 | 3200 |
| I/O Bus Bandwidth (Mbytes/s) | 25 | 50 | 100 |
| Processor to Processor Bandwidth (Mbytes/s) | 1600 | 3200 | 6400 |

Note: Where two numbers are shown, separated by a slash, the first is performance in 16-bit mode and the second is performance in 32-bit mode.

**Table 1. APx Specifications**

|  |  |
|---|---|
| 512x512 Image<br>3x3 Convolution | 5 msec |
| 512x512 Complex<br>2D FFT | 60 msec |
| 512x512 Complex<br>2D FFT (16x16 Blocks) | 20 msec |
| 256x256 Hough Transform<br>(10% density) | 20 msec |
| Maximum of a 64K Vector | 0.1 msec |
| Sort of a 64K Vector | 7 msec |

**Table 2. AP256 Projected Performance. 16-bit precision used.**

## References

1. Hillis, W.D., "The Connection Machine", *Scientific American*, Vol. 256, No. 6, June 1987, pp. 86-93.

2. Potter, J.P., editor, *The Massively Parallel Processor*, MIT Press, Cambridge, Massachusetts, 1985.

3. Hunt, J.D., "The ICL DAP and its Application to Image Processing", in *Languages and Architectures for Image Processing*, M.J.B. Duff and S. Levialdi, eds, Academic Press, London, 1981.

4. Hayes, J.P. et al, "A Microprocessor-based Hypercube Supercomputer", *IEEE Micro*, Vol. 6, No. 5, Oct. 1986, pp. 6-17.

5. Morton, S.G. et al, "ITT CAP - Toward a Personal Supercomputer", *IEEE Micro*, Vol. 5, No. 6, Dec. 1985, pp. 37-49.

# The Design of a Bit-Serial Coprocessor to Perform Multiplication and Division on a Massively Parallel Architecture

Robert E. Morley, Jr., Gary E. Christensen, Thomas J. Sullivan, Orly Kamin

Electronic Systems and Signals Research Laboratory
Department of Electrical Engineering
Washington University
St. Louis, Missouri 63130

***Abstract*** **- Many signal and image processing algorithms are being mapped onto SIMD massively parallel architectures which are often composed of single, bit-serial processor elements (PEs).These PEs are slow when performing the many multiplications and divisions required by FFTs, convolutions, and other common DSP operations. A custom VLSI coprocessor has been designed to greatly increase the speed of these operations. In this paper, the need for such a processor, the multiplication and division algorithms of the processor, and comparisons of these algorithms to those implemented on several machines is presented.**

## I. Introduction

CONVENTIONAL bit-serial processors such as the AMT DAP[1] and the NCR GAPP[2,3] execute the operations of multiplication and division in $O(N^2)$ cycles, where N represents the operand bit length. This corresponds to a design style that saves area at the expense of processing time and yields an area-delay product (ADP) of $O(N^2)$. By employing a parallel adder whose area is proportional to N, another design is obtained which retains an ADP of $O(N^2)$, but the delay is reduced to $O(N)$. Furthermore, if successive multiply-accumulate operations are to be executed, as is commonly the case in a wide class of digital signal processing algorithms, a series of M multiply-accumulate operations takes $O((N^2+N)M)$ cycles for the bit-serial processor and only $O(NM)$ cycles for the the parallel adder design.

We will present the design of a VLSI coprocessor element (COPE) that employs a parallel adder to achieve the increased speed described above. It accepts operands of variable length ($N \leq 16$), and produces a 2N bit product. The overall process takes 4N cycles, and if needed, the result may be accumulated for further processing without any processing time penalty. In addition, the COPE has been designed to perform division in $O(4N)$ cycles, as opposed to $O(N^2)$ for a conventional bit serial processor.

When fabricated in 1.2 µm CMOS technology, we estimate that 64 coprocessors will easily fit on a single die. This is equivalent to the number of AMT DAP processor elements on a chip. Hence, only a doubling of area is required to increase the multiplication and division performance by an order of magnitude for 16 bit operations.

A description of the simple hardware architecture is presented, followed by the algorithms used for multiplication and division. Finally, performance comparisons between the algorithms used in the COPE, DAP, and GAPP for multiplication, division, and multiply-accumulate operations are given.

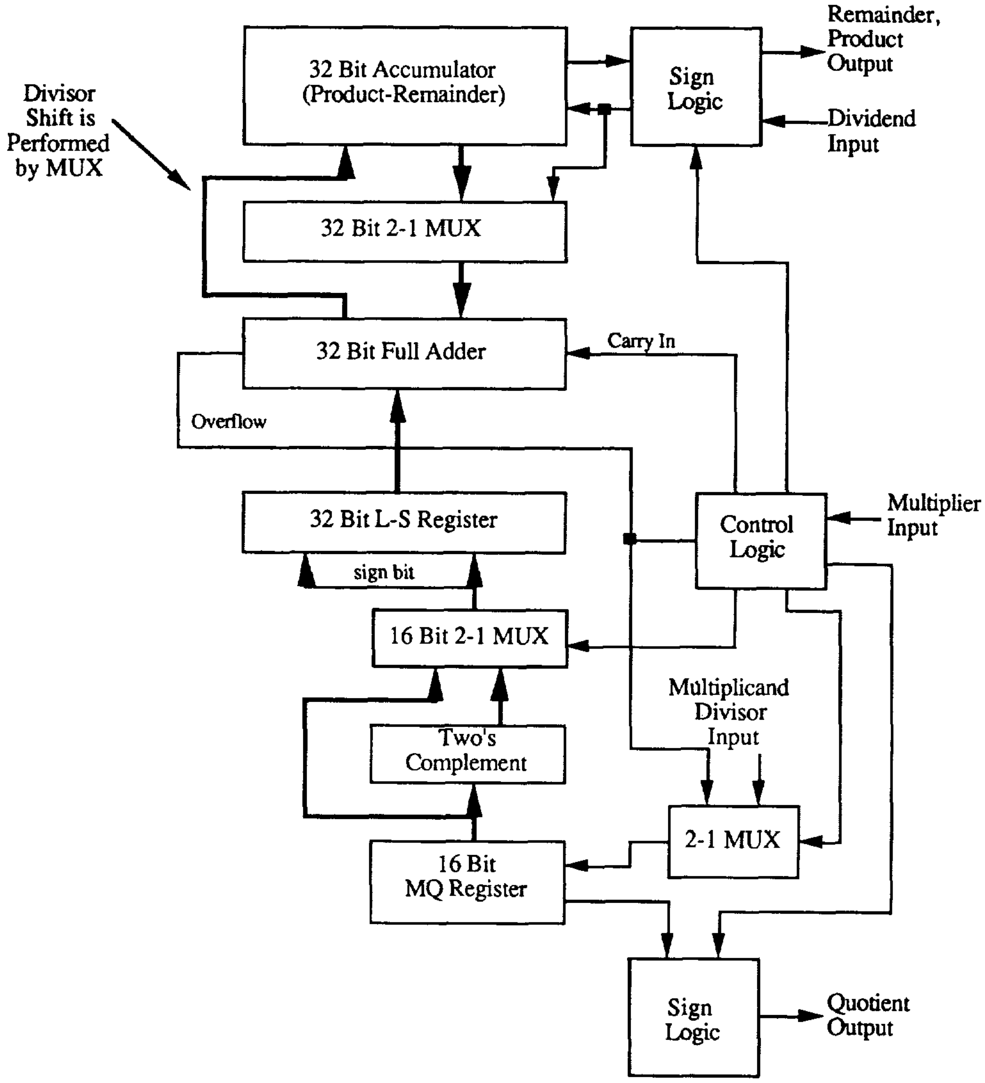
## II. Coprocessor Logic

The COPE coprocessor of Figure 1 consists of three shift registers, a full adder, two's complement circuitry, control logic, and four serial bit manipulation circuits. The shift registers are used to multiply by two, store and retrieve data, and perform serial to parallel data conversions. The bit manipulation logic performs transformations on incoming and outgoing serial data, as required by the multiplication and division algorithms.

The MQ register is used to serially acquire the multiplicand or divisor, or to assemble the quotient. It is also able to dump the quotient serially (right shift). The LS register is used to store either the multiplicand or divisor, and to multiply the left shift the multiplicand. The LS register may be loaded in parallel with either the MQ register or its two's complement. The final register is the accumulator. It serially acquires the dividend, serially dumps the remainder, or latches a new parallel sum. This sum is generated by a parallel full adder.

Two algorithms have been developed for the COPE chip, which perform the desired functions of multiplication and division. The control circuitry to implement these algorithms is split into two sections. A global control section is responsible for generating data independent control signals for each PE on the chip. Each PE also contains local control logic to generate those signals which are data dependent. In this manner, it is possible to implement data dependent algorithms within an SIMD architecture.

CH2649-2/89/0000/0419$01.00 © 1988 IEEE



# Figure 1: The COPE Arithmetic Coprocessor

## III. ALGORITHMS

The COPE chip accomplishes multiplication and division in a serial manner as the operands are shifted into the chip. In both algorithms, one operand is fully loaded into the PE. As each bit of the second operand is presented to the PE, the desired operation is progressively performed. Once the last bit has been used, the result may be shifted out of the COPE. The algorithms for both multiplication and division are described below and are detailed in Appendix I.

### *A. Multiplication Algorithm*

For a multiply, the multiplicand is shifted into the MQ (Multiplicand-Quotient) register. As sign bit of the multiplier is presented to the COPE, the LS register is loaded with either a copy of the multiplicand (positive) or the two's complement of the multiplicand (negative). Also, the accumulator is cleared. Then, as each bit of the multiplier (LSB first) is presented to the COPE, the multiplier's absolute value is generated by the serial signlogic. If the resulting bit is high, the accumulator will be incremented by the value in the LS register. (If the multiplier is negative, the LS register contains the two's complement of the multiplicand, and hence the accumulator will be decremented.) At the end of the cycle, the LS register is multiplied by 2 (left shifted). After the last bit of the multiplier has been processed, the accumulator will contain the desired product, which may then be shifted out of the COPE for further processing.

A multiply-accumulate operation may easily be performed by eliminating two of the steps above. Since the product is required for subsequent operations, the accumulator should not be cleared (except for the first cycle), and the product should not be shifted out of the accumulator (this reduces the cycle time by the product length, 2N).

### *B. Division Algorithm*

The division algorithm is started by shifting the divisor into the MQ register, after which the negative (*i.e.*, -|divisor|) of the divisor is loaded into the LS register, and the accumulator is cleared. Each bit of the dividend is then presented to the COPE (MSB to LSB). If the dividend is negative, its bits will be complemented as they enter the processor. For each dividend bit, the sum of the dividend bit, the LS register, and a left shifted accumulator is generated. If an overflow is generated from this operation, the accumulator is loaded with the sum; otherwise, the accumulator is left shifted, with the dividend bit becoming the accumulator's LSB. Also, the overflow bit is right shifted into the MQ register. Once all of the dividend bits have been processed, the accumulator may need a simple correction. The accumulator is added to the LS register, and the sign of the dividend (this completes the two's complement of the dividend, which was partially done by complementing the incoming bits). If this sum generates an overflow, the accumulator is replaced by the sum. Finally, the quotient may be retrieved from the MQ register, and the remainder retrieved from the accumulator.

The results may still require some simple manipulation, depending upon the sign of the operands. If the dividend is positive, the accumulator will contain the remainder. Otherwise, the accumulator should be incremented by one if the final sum generated an overflow, and its two's complement generated. This will result in the correct remainder. The quotient must be incremented by one if the dividend is negative and an overflow occurred. Then if the operands are of different signs, the quotient's two's complement must be generated. All of these adjustments may easily be accomplished as the results are being shifted out of the COPE, with no time penalty.

### *C. Algorithm Testing*

Both algorithms were tested through extensive simulation. Two programs were independently developed in the C language with slightly different features. The first program simulated the algorithm by using functions which closely emulate the classic two phase registers used in the actual VLSI implementation. This technique helps to guarantee that the data from various registers will be valid at the appropriate times. The second simulation simulates variable length operand feature. From these two programs, the required control signals were verified.

## IV. Performance Comparison

The motivation for the COPE chip was to design a coprocessor for the DAP which performed multiplication, division, and multiply-accumulates in a more timely fashion than an unaided DAP can. After comparing the algorithms used in the COPE to those of the DAP and GAPP, we found:

| Machine | Number of Cycles for Multiplication | Number of Cycles for Division |
|---|---|---|
| DAP | $0.86n^2 + 28n + 105$ | $0.44n^2 + 38n + 227$ |
| GAPP | $6n^2 - 2n + 1$ | $8n^2 - 7n + 3$ |
| COPE | $4n$ | $4n + 1$ |

where n is the number of bits in the operand. For simplicity's sake, it has been assumed that the two operands have an equal number of bits; however, this is not a requirement. It should also be noted that the equation for DAP division was derived from actual cycle counts for the DAP programs.

**Comparative Multiplication Cycles**

| op-size | Number of Cycles | | | COPE Speed Increase Over | |
|---|---|---|---|---|---|
| (bits) | GAPP | DAP | COPE | GAPP | DAP |
| 8 | 369 | 410 | 32 | 11.5 | 12.8 |
| 16 | 1505 | 725 | 64 | 23.5 | 11.3 |
| 32 | 6081 | 1900 | 128 | 47.5 | 14.8 |
| 64 | 24449 | 5400 | 256 | 95.5 | 21.1 |

**Comparative Division Cycles**

| op-size | Number of Cycles | | | COPE Speed Increase Over | |
|---|---|---|---|---|---|
| (bits) | GAPP | DAP | COPE | GAPP | DAP |
| 8 | 459 | 560 | 33 | 13.9 | 17.0 |
| 16 | 1939 | 950 | 65 | 29.8 | 14.6 |
| 32 | 7971 | 1900 | 129 | 61.8 | 14.7 |
| 64 | 32323 | 4350 | 257 | 125.8 | 16.9 |

As mentioned above, a multiply-accumulate operation may be perfromed with no added expense, as the partial sum remains in the COPE to be used in later cycles. For an FIR filter the cylce time is 4NL +2, where L is the filter length, and N is the word length. Relative to the DAP, this is a speed improvement of approximaltely 24 fold for a 16 bit, 32 tap FIR filter.

## V. Conclusions

It has been shown that a coprocessor can be designed which will greatly enhance the functionality of a bit serial processor, such as the DAP. The coprocessor implements multiplication and division algorithms, which operate at least an order of magnitude faster than is possible through software methods. This processor is also able to accomplish multiply-accumulate operations at no extra cost to the user. All of this can be accomplished with an increase in area by a factor of only two at 1.2 μm.

## VI. Appendix I

### *A. Multiplication Algorithm*

```
clear ACC
extend sign bit of multiplicand into MQ
shift multiplicand into MQ
if multiplier is positive
    load MQ into LS (sign extend)
else
    load two's complement of MQ into LS (sign extend)
end if
for each multiplier bit (LSB to MSB)
    let m = multiplier bit
    if multiplier is negative
            let m = two's complement of m
    end if
    if m is set
            let ACC = ACC + LS
    end if
    left shift LS
end for
right shift ACC (LSB to MSB) to obtain product
```

A multiply-accumulate operation may be performed by repeated application of the above algorithm without clearing the accumulator. The result is obtained after the final multiplication has been performed.

### *B. Division Algorithm*

```
clear ACC
extend sign bit of denominator into MQ
shift denominator into MQ
if denominator is negative
    let LS = MQ
else
    let LS = two's complement of MQ
end if
for each numerator bit (MSB to LSB)
    let n = numerator bit
    if numerator is negative
        let n = one's complement of n
    end if
    let x = 2 * ACC + LS + n
    if overflow occurred
        let ACC = x
    else
        left shift ACC with n becoming LSB
    end if
    MQ = 2 * MQ + overflow
end for
if numerator is negative
    let x = ACC + LS + 1
else
    let x = ACC + LS
end if
if overflow occurred
    let ACC = x
cnd if
case
    both numerator and denominator positive:
        right shift ACC to obtain remainder
        right shift MQ to obtain quotient
    numerator positive and denominator negative:
        right shift ACC to obtain remainder
        right shift MQ and generate its two's complement for quotient
    numerator negative and denominator positive:
        if overflow occurred
            right shift ACC and two's complement for remainder
            right shift MQ, add one, and two's complement for quotient
        else
            right shift ACC, add one, and two's complement for remainder
            right shift MQ and two's complement for quotient
        end if
    both numerator and denominator negative:
        if overflow occurred
            right shift ACC and two's complement for remainder
            right shift MQ and add one to obtain quotient
        else
            right shift ACC, add one, and two's complement for remainder
            right shift MQ to obtain quotient
        end if
end case
```

## REFERENCES

[1] S.F. Reddaway, "DAP - A Distributed Array Processor," *First Annual Symposium on Computer Architecture (IEEE/ACM)*, 1973.

[2] E.L. Cloud, "The Geometric Arithmetic Parallel Processor," *2nd Symposium on the Frontiers of Massively Parallel Computation*, 1988.

[3] R.E. Morley and T.J. Sullivan, "A Massively Parallel Systolic Array Processor *System," International Conference on Systolic Arrays*, 1988.

# CONTROLLING AND PROGRAMMING THE SPHINX MULTI-SIMD PYRAMID MACHINE.

J. Méhat
Université Paris VIII, St Denis, France

A. Mérigot
Institut d'Electronique Fondamentale, Université Paris Sud,
Orsay, France

## ABSTRACT

This paper is concerned with the control environment of the SPHINX Pyramid Machine. First, a low overhead interlayer method, to synchronize communication of independently controlled SIMD processor meshes is presented. We show how it can be used to provide multitasking within a mesh to allow opposite data flow to cross safely. Related programming concepts for Multi-SIMD machines — *control transmission* — is presented.

Keywords: Pyramid Machines, Multi-SIMD, Parallel Languages, Interprocessors Synchronization.

## INTRODUCTION

SPHINX is a cellular pyramidal machine primarily designed for image processing applications [4,5], currently under development as a joint effort of University Paris Sud, and ETCA Defence Research labs. It is organized as a set of stacked layers of decreasing size interconnected according to a dual network: a mesh based inter-neighbor interconnection network within a layer, and a binary tree between adjacent layers. The SPHINX processing element relies on bit serial operations and communications. It is formed of an ALU, with enhanced data transfer capabilities, a 256 bits local memory, and a set of special purpose registers. An hardware prototype pyramid 32x32 (2047 PEs) is going to be available in the next future.

Two important differences between SPHINX and other pyramidal machines [1,6,7] are that each PE has 2 sons instead of 4, and that each layer receives its own instruction stream (figure 1); so SPHINX presents both an SIMD aspect, since all the PEs of a same layer share the same instruction stream and a MIMD aspect between layers. It is a Multi-SIMD machine. If we define the power of a pyramid as the number of processors at it's base, SPHINX, as a binary pyramid uses 50% more PEs than a quaternary one of the same power, but presents several advantages:

— operations between layers can generally be pipelined in an efficient way, since each PE has two sons and a two input ALU;

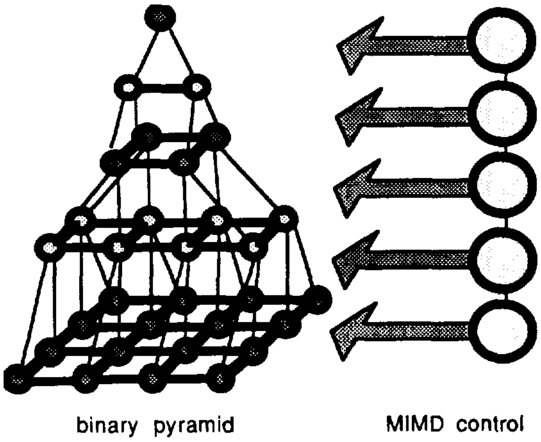


Figure 1 : SPHINX Architecture

— oct-trees as well as quad-trees can be built by grouping PEs of differents layers into virtual PEs;

— thanks to the larger number of layers compared with a quaternary pyramid, we can make a more efficient use of the MIMD aspect.

Arguments against pyramid are often the following [3] : either the pyramid is SIMD, leading to a very poor layer occupation, either, for a Multi-SIMD machine, the interlayer control will slow down data transfer. We present a control scheme with a very limited overhead for SPHINX and related software aspects.

## THE CONTROL STRATEGY

The instruction stream bandwidth problem draw us to spread the control over the pyramid: different layers are independently controlled. With this control scheme, communication between adjacent layers leads to a critical synchronization problem. As data are transferred between layers in a bit serial way, this type of communication implies that one must be able to realize an intercontroller synchronization within a bit serial instruction execution time (typically 100ns).

CH2649-2/89/0000/0423$01.00 © 1988 IEEE

### Two Levels of Synchronization

To obtain the necessary speed of synchronization, we distinguish between two levels of synchronization: a *low level* one insuring integrity of the interlayer transferred data with respect to the state of communication buffers according to a producer/consumer protocol, and a *high level* one allowing coherent manipulation of data within the whole pyramidal structure by means of information passing between adjacent layers.

The high level synchronization is necessary to take into account the constraints of process scheduling, parameter passing and interlayer message passing. To deal with the problem of high level data exchange between layers, it communicates with its two adjacent controllers. It will typically be realized with a standard microprocessor executing compiled code to run application programs generating system control and word level pyramidal instructions, translated into sequence of bit-serial instructions by a simple macro-generator and put into a FIFO for the low level synchronizer.

The low level synchronization has in charge the correct transmission between layers at the bit-serial level. It has to be done within a bit serial instruction execution time, but is very simple as one has only to consider if an instruction is executable with respect to the state of interlayer communication buffers, i.e. the buffers used as source are full at the time of the operand fetch and the ones used as destination will be free when the results will be stored.

The necessity of low level synchronization is not bound to the size of one bit of the communication buffers. If the buffer size were made greater, either through a hardware or software artifice, the problem would be actually identical: a *n* bit buffer permits to overcome the synchronization necessity for the *n-1* first bits, but then the buffer may be full and the problem comes back.

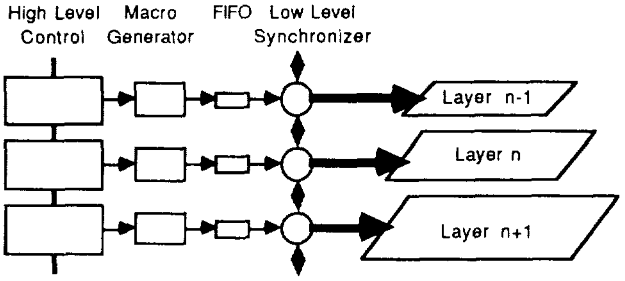


Figure 2: Overview of the Control System.

### The Low Level synchronizer

To illustrate the way program are synchronized, let us consider the following problem : We want to compute the *histogram* of a certain *attribute* stored in the lowest layer of the pyramid, i.e. the number of occurrences of every value of *attribute*, and either output it by the apex of the pyramid, or store it in the upper layer. A straightforward algorithm is the following:

```
for i varying from 0 to the last attributes value
begin
   select the PEs where attribute = i
   count the number of selected PEs
end
```

The *select* operation is a point wise procedure applied to PEs in the lowest layer. The *count* operation uses the vertical communications to accumulate, in a logarithmic time, the number of selected PEs. The actual method is the following: at the lowest layer, PEs send the *selected* bit to their father. At the others layers, PEs add sequentially the bits they receive from their sons, sending the result to their father, followed by the MSB of the result stored in their carry register. The algorithm is the following, assuming the pyramid has *height* layers:

```
at the base
begin
   send the selected bit to the father
end
at layer l such as top ≤ l < base
begin
   repeat height - l times
   begin
      send to father the sum of the bits
         transmitted by left and right sons
   end
   send carry to the father
end
```

Figure 3a presents the *instruction matrix*, i.e. the instructions executed on every layer vs. time, for one step of the histogram, computing the sum for one attribute value on a pyramid of height 5.

Thanks to the MIMD interlayer control mode, it is possible to start another step of this operation as soon as the previous step has completed in the lowest layer. This lead to the idealized version of the histogram computation presented in fig 3b. One can see that MIMD allows a large gain in computing time, as the apparent execution time will only be limited by the data output time, instead of the complete count operation time.

The high level control will execute the previously described algorithm, and send to the low level controllers the instructions streams of fig 3c. The instructions are correct in term of their relative occurrence within a layer, but the actual execution time is not fixed.

The low level controller will synchronize these streams, and send the instructions as soon as it is possible in terms of the producer consumer protocol. This leads to the instruction matrix presented in figure 3d. This matrix is less regular than the idealized

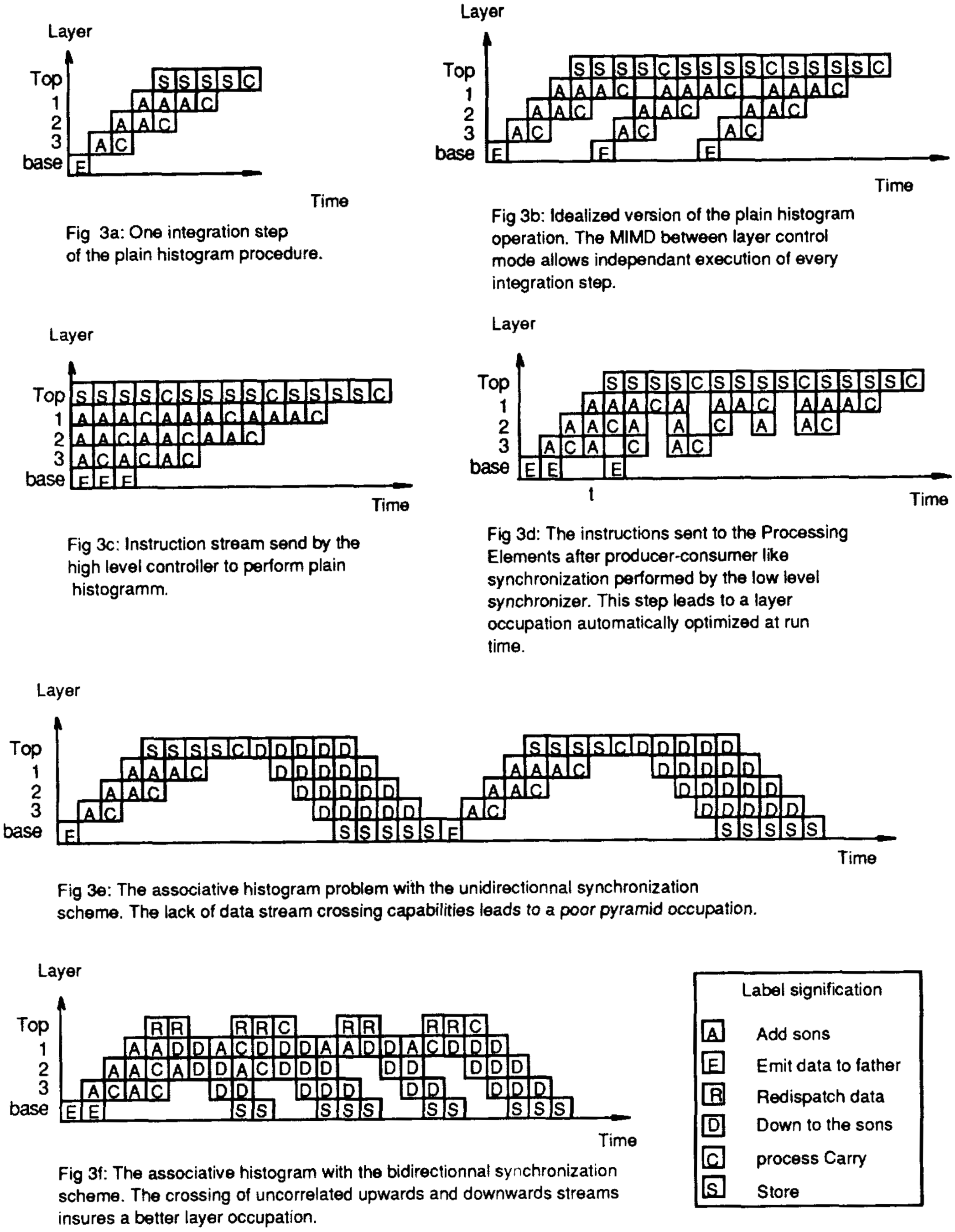


Fig 3a: One integration step of the plain histogram procedure.

Fig 3b: Idealized version of the plain histogram operation. The MIMD between layer control mode allows independant execution of every integration step.

Fig 3c: Instruction stream send by the high level controller to perform plain histogramm.

Fig 3d: The instructions sent to the Processing Elements after producer-consumer like synchronization performed by the low level synchronizer. This step leads to a layer occupation automatically optimized at run time.

Fig 3e: The associative histogram problem with the unidirectionnal synchronization scheme. The lack of data stream crossing capabilities leads to a poor pyramid occupation.

Fig 3f: The associative histogram with the bidirectionnal synchronization scheme. The crossing of uncorrelated upwards and downwards streams insures a better layer occupation.

*Figure 3:* Pyramid occupation vs. time for different histogram operations and different synchronization mecanisms.

version of figure 3b, but more efficient in term of pyramid occupation. More, the programmer just has to take care of the logical correctness of his program, and the synchronizer will automatically optimize its execution at run time.

On figure 3d at date labelled t, we can see a *vertical data shift*. Layer 3 performs an add operation and sends the result to layer 2, whilst this one performs the same operation on previously generated data and so on until the top. This means that the instruction on layer 2 can take place, not because the present state of the father's receive buffer is correct (empty), but because the instruction in the upper layer will empty the buffer, if executed.

This kind of operation where all the concerned communication buffers are full before operand fetch but freed before the result should be stored, requires non local knowledge to be performed. [2] presents a cellular automaton based mechanism able to solve this kind of problems at run time, as well as a complete description of this synchronization scheme.

### The crossing of data movements.

The previous model works very well when all the data movement are of the same direction, but it presents some weakness in other cases. Let us consider the following *associative histogram* problem: each PE at the base of the pyramid contains an attribute value, and we want to associate it, in the PE memory, with the number of occurrences of its value in the base. The vertical connections of the PEs are used both to compute the number of occurrences in the top of the pyramid in a time logarithmic to the number of PEs of the base, and to project the so computed sum from the top PE to the PEs of the base. The computing necessary for each value involves two distincts data movements: an ascending one to compute the sum and a descending one to project it.

There is two way to realize this operation on a pyramid. The first one avoids data crossing by means of the following algorithm:

```
for i varying from the first to the last value
begin
   count the number of PEs where attribute = i
   send the result down
   at the base begin
      where attribute = i begin
         associate the number with the attribute
      end
   end
end
```

As it is shown in figure 3e, this leads to a very inefficient use of the pyramid, most of the layers being idle whilst the data is sent downwards.

Another method consists to allow the crossing of the upwards and downwards data stream. Indeed, these streams are logically uncorrelated, and one can perform the *n*th integration, while the *n-1*th result is sent down. One possible result is shown in figure 3f.

The drawback of this solution, when using the previous synchronization method, is the possible turning up of deadlocks when two data transfers are coexisting. The figure 4 shows the top layers and their communication buffers computing the first bits of the sum in the associative histogram problem. When the first bit of a sum is computed at the top of the pyramid, the 2d bit can't be computed at the top before this 1st bit has been consumed by the level 1. At the same time, the instruction to consume this bit can't be executed by level 1 before the one which compute the 3rd bit has been executed. The two layers are in deadlock.

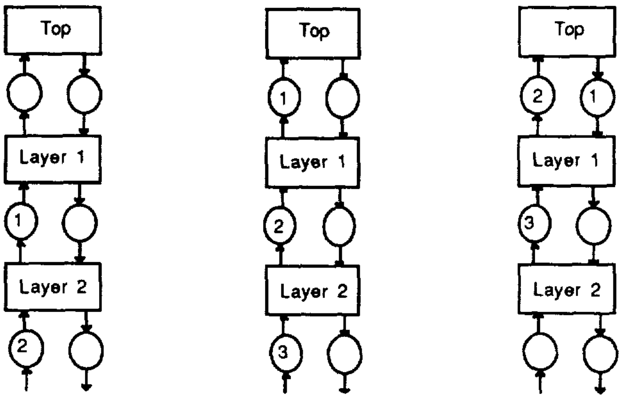


Figure 4: Deadlock emergence in data reverberation.

Of course, one could imagine to synchronize statically the layers at compile time, and to schedule the instructions in a way avoiding deadlocks. This solution leads to several drawbacks. First, it implies that the state of the layers in term of occupation must be well known, and accordingly, all the previously scheduled routines must have completed, disabling any use of pipeline. Second, this leads to an unstructured instruction stream, that imposes a very large data transfer rate, that causing bandwidth problems. Third, this static synchronization is not always possible if we want to be able to use data driven algorithms. For instance, if the global operation termination is controlled by a run time criterion as a convergence test on a layer, it is clearly impossible to perform any kind of static scheduling. We now present a synchronization method able to solve this kind of situation.

### Two FIFO for the synchronizer

A proper automatic deadlock-free run time synchronization of data crossing requires that the upwards and downwards instruction streams are generated by two *independent (pseudo) parallel processes*. The synchronization mechanism should be able to choose the best candidate according to the communication buffers state. The algorithm should be like the following:

```
in parallel begin
    for $i$ varying from $0$ to $n$ begin
      count the number of selected PEs
          where attribute == n
    end
    for $i$ varying from $0$ to $n$ begin
      at all levels except base begin
          send downwards the result of the
            count operation
      end
      at base begin
          where attribute = $i$ begin
            receive the value emitted by
                the upper level
          end
      end
    end
end
```

To perform this, we propose to have two FIFOs in which the synchronizer fetches instructions. One contains the instructions taking part to the ascending data movement, and the other the instructions for the descending one (figure 5). The synchronizer selects one of the two FIFOs. As long as the execution of the instruction on the top of the FIFO is compatible with the state of the interlayer communication buffers, it is sent to the PEs of the layer. When the instruction must be delayed and the other FIFO is not empty, the synchronizer switches the context in the PEs of the layer and uses the instructions of the other FIFO.

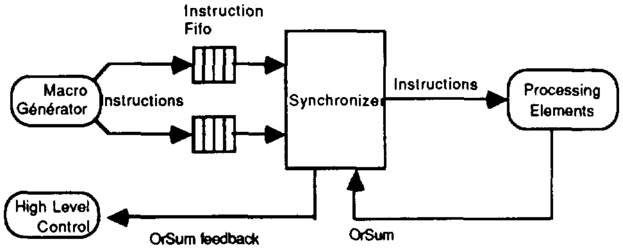


Figure 5: The dual FIFO synchronization.

The context switch is expensive, since on SPHINX five registers have to be saved in the PE memory, but it occurs only when distinct data movements are crossing. When there is only one data movement in a layer, there is no context switch and then no overhead.

We will call *process* the sequence of instruction that takes part to a data movement in a layer. We label the processes as *ascending* or *descending*, according to the data movement to which they are taking part. Only one ascending and one descending process can be simultaneously running in the same layer. The crossing of data movement is accomplished through this coexistence of processes inside a layer.

The use of two FIFOs allows a deadlock free coexistence of the two processes in the same layer, provided they only carry data either upwards or downwards: let's imagine that in the layer $i$ we have a process $P_i$which is blocked in a deadlock. We will suppose that $P_i$ is an ascending process but the demonstration is easily extended to the case where $P_i$ is a descending one. $P_i$ may be blocked either because a process $P_{i-1}$ in the layer $i-1$ doesn't produces the data $P_i$ needs, or because a process $P_{i+1}$ doesn't consumes the data $P_i$ produces. As $P_{i-1}$ and $P_{i+1}$ are ascending processes, they produce their instructions in the FIFO of the ascending processes. As they are the only ascending processes in their layers, their instructions are on the top of their FIFO, and they must be blocked because of the ascending processes of layer $i-2$ or $i+2$ to which the same reasoning may be applied. There can't be a deadlock as long as there is no interaction between ascending and descending processes.

### Vertical communication as resources

Some processes don't fit in the ascending/descending scheme we just described. For example, the process at the top of the pyramid in the associative histogram program, both consume data from the layer 1 like an ascending process, and produce data for the layer 1 like a descending one in the same instruction. We label such a process as *bidirectional*, and activate it according to the following protocol: in each layer, each direction of communication is considered as a resource; an ascending process needs the ascending resource, instantiated by the communication buffers from the lower layer and to the upper layer, a descending process needs the descending resource and a bidirectional process needs both resources. A process is run in a layer only if the resources it needs are available. That way in a layer we can have either at most one ascending and one descending process, or a bidirectional one.

Bidirectional processes are necessary to have crossing data streams performing useful work, as they permit exchange of data between them, but their existence allows occurrences of deadlocks, as is usual when two processes are communicating together via blocking *input and output primitives*. Proper programming allows the avoidance of deadlocks, which are easy to detect as they are always caused by bidirectional processes.

## THE EXPRESSION OF COOPERATION

Independently of the control strategy, one has to provide an effective way to specify the cooperation of different layers, which directs as well how high level synchronization is realized, as how the programmer specify process creation. The control model described above relies on anonymous ports, as any other communication scheme would have implied to carry identifying information about communicating processes with *any* bit of data. Despite this constraint, one has to let the communication expression specify as explicitly as possible the identity of the processes involved.

To express the cooperation of the layers, we propose to use the same scheme that is used for data movement: pyramid algorithms are conceived in term of data movements between layers, and then a task is defined which migrates, accompanying the moving data. We choose not to execute that migration via process migration between layer controllers, because process migration implies that the process environment migrates with the process, and also because data exchange between PEs from differents layers involves two processes: one sending the data and a second one receiving them.

The task migration is accomplished on a local to a layer base, in a way we call *control transmission*, after data transmission. The creation of the task involves three distinct steps: at first, a process is created in the layer initiating the data movement; that process creates in the neighbor level another process and then sends data to the newly created process.

For example, a step of the sum for the histogram will be programmed with:

```
process sum(n)
begin
   if current layer = base then begin
      create process sum(1) in the upper level
      send the selected bit to the upper level
      end
   else if current layer = top then begin
      sum the n bits received from lower level
      end
   else begin
      create process sum(n+1) in the upper
        level
      send to the upper level the n+1 bit sum
        of the n bit received from lower level
   end
end
at base create process sum(0)
```

The associative histogram operation can be programmed in the way. One only needs to activate at the upper layer a descending process to carry data downwards.

Control transmission presents several advantages: involving only transmission between adjacent layers, it can be realized by in a point to point communication between adjacent controllers.

The environment of each process is clearly stated in the program, as each process communicates either with the process which created it, or with a process it has created, or both. Accordingly, provided the algorithm is correct in terms of data transfer i.e. the same amount of data is produced and consumed by communicating processes, the consumer will always be the right one, even though transfers are made through anonymous ports.

Deadlocks coming from process creation obeys the same mechanisms as data communication. One bidirectional processes can, directly or indirectly create deadlocks by process creation requests.

The use of layers being controlled by process execution, it can be dynamically adapted to the state of data in the PEs, allowing a better exploitation of the occupation of the layers.

## CONCLUSION

We have presented a synchronization method to solve the problem of the Multi-SIMD control of the SPHINX Pyramid Computer. Based on a recently designed chip, we are building a 32x32 machine, with a controller relying on these principles. We have as well, designed a high level language—*pyr-e*—based on *C* for which a compiler is on the way.

## AKNOWLEDGEMENTS

This work was supported by ETCA/DRET. Its results from many helpful discussions with Patrick Greussay, Philippe Clermont, Francis Devos and Ni Yang.

## REFERENCES

[1] Cantoni V., et al., "The PAPIA image analysis system," *2nd Int. Techn. Symp. on Optical and Electro-Optical Applied Science and Engineering*, Cannes, Nov. 1985.

[2] Clermont P. and Mérigot A., "Real time Synchronization in a Multi-SIMD Massively Parallel Computer," *Proc. Workshop on Computer Architecture for Pattern Analysis and Machine Intelligence*, Seattle, Oct 1987.

[3] Duff M.J.B. "Pyramids — Expected performance," in *Pyramidal Systems for Image Processing*, V. Cantoni and S. Levialdi ed., Springer Verlag, pp 59-74, 1986.

[4] Mérigot A., Zavidovique B. and Devos F., "SPHINX, a Pyramidal Approach to Parallel Image Processing," in *Proc. Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, IEEE Computer society Press, pp 107-111, 1985.

[5] Mérigot A., Clermont P., Méhat J., Devos F., Zavidovique B., "A Pyramidal System for Image Processing," in *Pyramidal Systems for Image Processing*, V. Cantoni and S. Levialdi ed., Springer Verlag, pp 109-124, 1986.

[6] Schaefer D.H., Wilcox G.C. and Harris V.J. , "A Pyramid of MPP Elements—Experience and Plans," *Proc. 18th Int. Conf. on System Sciences, Honolulu*, 1985.

[7] Tanimoto S.L., "A Pyramidal Approach to Image Processing," *Proc. 10th Annual International Symposium on Computer Architecture,*, Stokholm, pp 253-256, 1983.

# *The Ynet*: An Interconnect Structure for a Highly Concurrent Data Base Computer System

Dr. Philip M. Neches

*Teradata Corporation*

## *Abstract*

The Teradata DBC/1012 Data Base Computer System applies a multiple data stream, multiple instruction stream (MIMD) concurrent processing architecture to implement the relational model. The interprocessor interconnect structure, called the "Ynet", is the basis of the architecture, which can connect up to 1,024 high performance microprocessors to form a very high performance system. This talk describes the design considerations for the Ynet as the interconnect structure for a highly concurrent database system which also requires high availability and real-time operation.

## *Background*

**Design "Givens".** Teradata Corporation was organized in 1979 to develop and market a relational data base computer system. Several key elements of the product architecture were known from the outset of the design process:

(1). The product would implement the relational model of database management. By 1979, the pioneering theoretical work of E. F. Codd and others had resulted in software prototype relational systems, which had established significant productivity benefits for both conventional application development and unanticipated ("ad hoc") requests.

(2). The product would employ a MIMD parallel processing architecture. During the 1970's, several workers in academia produced an extensive literature on parallel architectures to implement some of the operators of the relational model. Hsiao and DeWitt particularly championed MIMD (Multiple Instruction streams, Multiple Data streams) approaches because of the breadth of functions required of a relational system.

(3). The product would employ multiple microprocessors. The advent of the 16-bit generation of microprocessors put the power previously associated with minicomputers onto one chip. Further, both the absolute performance and price/performance of microcomputers would increase at a much faster rate than either mainframe or minicomputer technology. The microprocessor was thus clearly established as the engine of choice for cost-effective designs.

The combination of a MIMD architecture with microprocessor economics permits products with a number of advantages. A design would be inherently very modular, pernitting both small and very large systems to be constructed out of the same "building blocks". By using the most cost effective technology for the building block, the resulting systems should display very competitive price-performance across the range of configurations. Further, the rate of price-performance improvement should be very rapid.

Combining a MIMD parallel architecture with microprocessors to implement the relational model also has several key consequences for the design. The Ynet interconnect design responded to these challenges:

(1). Systems of up to hundreds of processors would be needed to address some of the most demanding applications. The interconnect structure thus has to be capable of extending systems to this scale. We chose 1,024 processors as the design target for this reason.

(2). Work would have to be divided among these processors. This resulted in the functional division between host communication and language processing in the Interface Processor (IFP) and database semantics operations in the Access Module Processor (AMP). (*See Figure 1*). This also led to the design decision to spread the storage of the rows of each table across *all* of the Access Module Processors. Each AMP stores some of the rows of each table: work is managed so that the task of operating on the rows is always done by the processor on which the rows are stored.

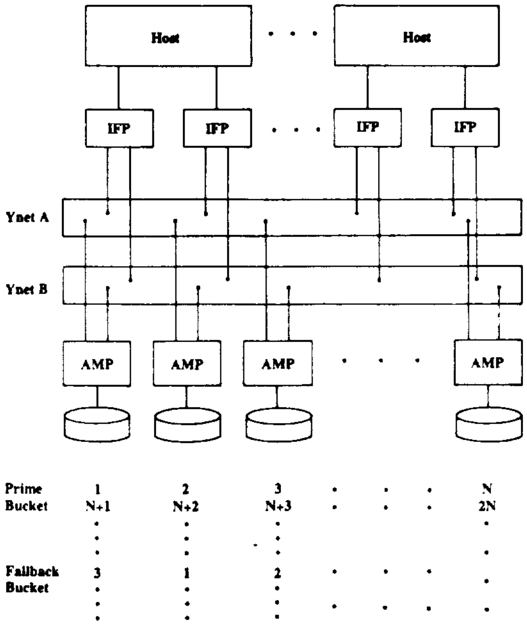


**Figure 1** - *DBC/1012 Overall Architecture*

CH2649-2/89/0000/0429$01.00 © 1988 IEEE

(3). Every parallel processing system must face the issue of how to divide the problem into smaller sub-problems, each of which can be assigned to a processor. Because of the associated problem of managing the distribution of data on secondary storage, this problem proved to be particularly straightforward for a relational database management system.

**Consequences of the Requirements.** The distribution of work following the distribution of data means that the system must have a broadcast mode of operation. Many operations require every AMP to do essentially the same operation to its subset of the rows. Further, the broadcast operation must be reliable: that is, the message invoking the operation must be guaranteed to be received by all of the intended recepients.

To see why this must be the case, consider a system with N Access Module Processors. If N-1 AMPs receive a message which will cause some change to the database, and 1 does not, that 1 AMP's subset of the database will not be consistent with the rest. This is the worst offense imaginable for a database management system: to corrupt the integrity of the information under management. We call this the *reliable multicast* problem.

Next, envision a system with perhaps hundreds of AMPs, each working on some portion of a complex request. There must be an efficient was to monitor the status of requests in process within the system. Such events as completion of a step which must be synchronized in all processors must be ascertained with a minimum of overhead.

Finally, in this kind of MIMD architecture, each AMP produces only a subset of the response from its subset of the database. The system needs a way to merge the partial results from each processor into a single result stream.

This requirement suggested the basic form of the Ynet as a sorting network. Users frequently request reports to be presented in some sorted order, typically different from whatever order was used to store the data. Thus a relational database management system will spend some significant amount of its resources sorting intermediate and final result sets.

The system faces several other important requirements. The system will have many simultaneous users, and thus must be multi-threaded to permit many requests to be in various stages of processing at the same time. In a system with hundreds of processors, and which may have several hundred disk drives, this means that the system must be designed for fault tolerance from the outset. Finally, although the system is composed of many processing elements internally, it must appear to the users and operations personnel as a single system ("single system image").

## *The Ynet*

The Ynet design evolved in response to the requirements outlined above. Historically, these requirements were understood before the design began.

A DBC/1012 system actually includes two completely independent Ynet structures for fault tolerance. (*See Figure 1*). When both Ynets are operational, the processors divide message traffic between the two Ynets. When either Ynet is down, traffic goes over the remaining Ynet, including messages passed by a special diagnostic program which tries to isolate the fault in the failed Ynet.

Each Ynet is a tree-structured network of circuit switching nodes. Each node connects to two elements below and one above: an element can be either another node or a processor interface (YIF board). The following sections describe the node itself, the processor interface, and the protocols which operate in the network.

## *Node*

**Ports.** Each node has three ports: A, B, and C. (*See Figure 2*). The A and B ports connect to either nodes or interfaces in the "down tree" direction. The C port connects to another node in the "up tree" direction.

Each port has a fully duplexed set of signals for data, control, clock, and parity. The data path is 1 byte (8 bits) wide. In addition, there is a COLLISION signal which occurs only in the downtree path.

**Data Paths.** The node logic is divided into the sorting logic in the uptree data path and the broadcast logic in the downtree data path. The sorting logic includes a comparitor, multiplexer, state flip-flops (DECIDED and AorB), parity checking, and control logic. The downtree logic includes only parity checking and latches.

**Flow.** Let's follow a packet through the network. Initially, all of the nodes' DECIDED flip-flops are set to 0, indicating that the network is ready to accept a new packet. Each processor interface sends the next packet that it wants to send into the Ynet. On each clock interval, a new byte from each packet goes from the interface to the first level node. The Ynet is thus byte pipelined.

Each node looks at the two packets coming in on the A and B ports, one byte on each clock interval. If the two bytes are identical, the node logic propagates the byte uptree to the C

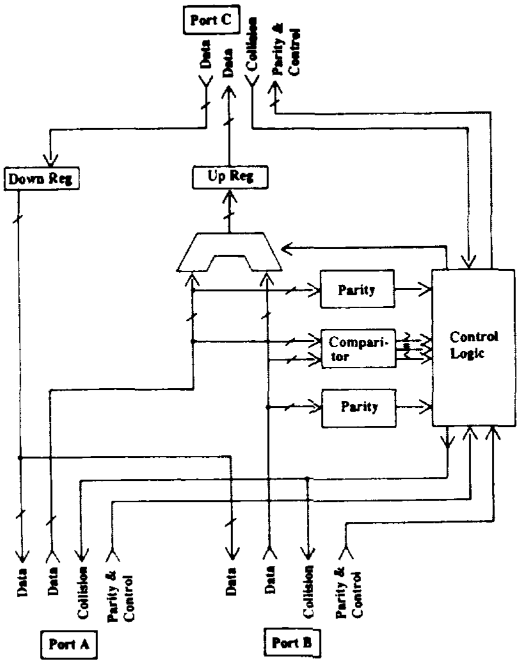


**Figure 2** - *Node Logic, Excluding Clock Circuits*

port, and does not alter the settings of the state flip-flops. However, the the bytes mis-compare, then the node propagates only the lower valued byte to the C port. The node also sets the DECIDED flip-flop to 1 and sets the AorB flip-flop to incidate which port provided the lower-valued byte. The node is now armed to accept further bytes only from the port which sent the lower-valued byte, and thus which has the lower-sorting packet. Finally, the node asserts the COLLISION signal to the losing port only.

**Collisions.** When a processor interface sees a COLLISION signal, it knows that its packet was not the lowest sorting packet: it "lost" in contention with other packets. The collision mechanism arms the processor interface to retry its packet following the acklowledgement sequence, which follows every "primary" packet contention cycle on the network.

Collisions, that is, mis-compares, can occur at any level in the tree of nodes. Thus, is the node sees the COLLISION signal asserted from a higher level node via the C port, it propagates the COLLISION signal downtree to the A and B ports.

In the uptree direction, the nodes implement in hardware an algorithm called "tournament sort". The name comes from the analogy to a sports tournament where the packets are players and the nodes are matches: the winners of the quarter-final matches advance to the semi-finals; the winners of the semi-final matches advance to the finals, and the winner of the final match is the winner of the tournament.

**Broadcast.** At the top or apex of the network, the winning (lowest sorting) packet is turned around and started in the downtree direction. Each node simply propagates the byte which arrives on the downtree Port C data path to the downtree portions of Ports A and B: it thus broadcasts the downtree packet to both ports.

The result is that the winning packet is received by every processor interface at exactly the same time. Subsequent sections will describe the actions taken by the processor interface.

**Packaging.** The node logic consists of about 25 SSI and MSI parts. The design originally used standard 'LS and 'S series TTL; these parts have been largely replaced by 'ALS and CMOS equivalents to reduce power consumption. An ASIC version is planned for a follow-on implementation of the system.

The node logic is packaged on two kinds of boards: the NODE and NODEX. The NODE board (*See Figure 3*) houses seven nodes to connect to 8 processor interfaces by backplane wires using TTL signaling levels.

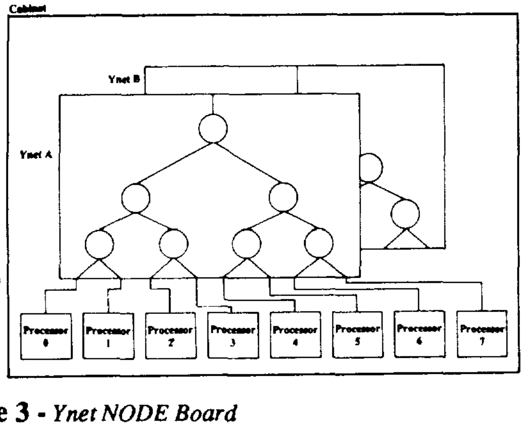


**Figure 3** - *Ynet NODE Board*

The NODEX board (*See Figure 4*) houses 3 nodes, which connect to as many as 4 NODE or NODEX boards via a 25-coax-pair ribbon cable using differential drive signaling. The cable can be up to 10 meters.

A system with only NODE boards can have up to 8 processors. A system with one level of NODEX boards can have up to 4*8 = 32 processors. A system with two levels of NODEX boards can have up to 4*4*8 = 128 processors. And so on.

**Clock.** The Ynet employs a unique scheme for clock distribution. Because the Ynet is a byte-synchronous network, the importance of clock distribution is readily apparent. Each NODE and NODEX board regenerates the clock locally, with a phase-locked loop to keep it synchronized with the board above it. The NODEX board has 4 phase locked loop circuits, one for each of the 4 downtree ports. Each phase locked loop circuit compares the locally generated reference clock with the clock returned from the lower level board. The difference between the local and returned clock signals is the error signal to the phase detector circuit. The phase locked loop scheme results in clocks held synchronous to within 1 nanosecond throughout a system which could fill a large computer room.

## *Processor Interface*

Most of the functionality of the Ynet results from the logic implemented in the processor interface. The interfaces provide a packet-switched interface to software, where the node network is more accurately characterized as a circuit switch. Each DBC/1012 processor has two Ynet interface cards, one for each of the two Ynet structures in the system.

The processor interface is divided into three sections, each of which is implemented by a finite state machine. (*See Figure 5*). These are the Ynet Input, Ynet Output, and CPU Interface sections. A special high speed memory (HSRAM) froms the core of the design of the processor interface. The HSRAM has three ports, one for each of the three finite state machines. The HSRAM and the three FSMs dervie thir clock from the Ynet network.

**HSRAM.** The CPU Interface FSM makes the HSRAM appear to the CPU as part of its memory address space. The programming interface to the Ynet appears through placing messages to be sent in the HSRAM and examining messages

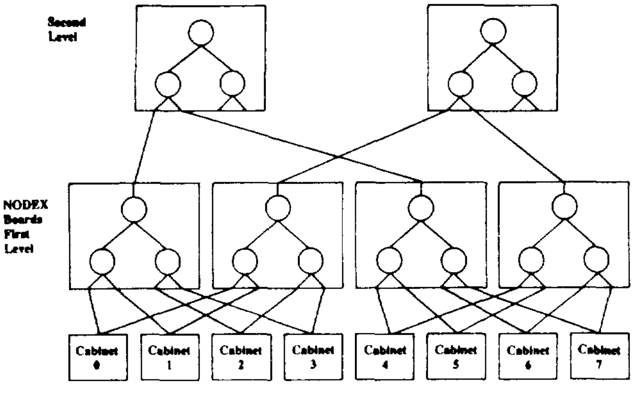


**Figure 4** - *Ynet NODEX Board*

received in the HSRAM using ordinary processor instructions. Messages received from the Ynet appear in the Input Circular Buffer area of HSRAM. The processor places messages to be sent in the Output Buffer area of HSRAM, and sets words in the Transaction Vector area of HSRAM to point to messages to be sent. The Ynet Output FSM places pointers to messages which have been successfully sent in the Output Pointer Circular Buffer, so that software running in the processor can take appropriate action, such as freeing the area occupied by the message itself in the Output Buffer area.

The HSRAM also includes various control information that is used by the processor interface hardware. By placing this control information in HSRAM, software can initialize the Ynet in a very flexible yet powerful manner.

**Transaction Vectors.** A Transaction Vector represents the state of a user transaction or job in each processor in the system. The state of a transaction within the system is derived from the Transaction Vectors with the same designation ("Transaction Vector ID" or TVID) in every processor. The current implementation of DBC/1012 software allocates over 1,000 Transaction Vectors, which permits over 1,000 user requests to be pending in the DBC/1012 simultaneously.

**Hash Maps.** Data rows are assigned to processors in the DBC/1012 by a hashing algorithm. The field(s) composing the primary index of the row are put through a modulo prime number hashing algotithm, which results in a hash "bucket" number. The Hash Map section gives the mapping of buckets to Access Module Processors.

There are two Hash Maps in the system: Prime and Fallback. Each row is stored on the AMP to which its hash bucket is assigned in the Prime Hash Map. In addition, if the user has requested it, a second copy of the row is kept on the AMP to which the corresponding bucket is assigned in the Fallback Hash Map.

There are approximately 3,000 hash bucket numbers. Each AMP is assigned several bucket numbers for both Prime and Fallback Hash Maps. This makes it possible to evenly divide the storage of data, and hence the processing load, among the processors of the system. When additional AMPs are configured into a system, the new AMPs are populated by moving some buckets from the existing AMPs to the new AMPs, and then moving the rows corresponding to the moved buckets. Thus only the minimum amount of data necessary is moved during reconfiguration.

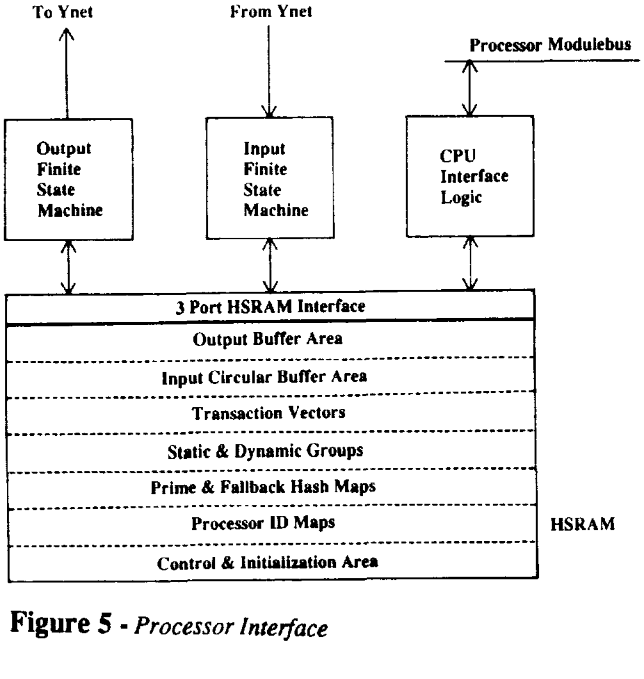


**Figure 5** - *Processor Interface*

Because the Prime and Fallback Hash Maps are independent, the system has a great deal of flexibility in choosing which processors should provide back-up for each other. DBC/1012 software implements a concept called "Fallback Clusters". In this concept, a group of between 2 and 16 AMPs form a cluster: rows that are in the Primary Hash Map of AMPs in the cluster will also be in the Fallback Hash Map within the same cluster. Obviously, the allocation of buckets must be such that the Prime and Fallback copies are on different AMPs. In the cluster concept, the Fallback copies for buckets that are Prime on one AMP in the cluster are equally allocated among the remaining AMPs in the cluster.

Compared to conventional mirroring schemes for reliable operation, the Fallback Cluster concept reduces the impact of a processor failure on the remaining processors in the configuration. Under mirroring, if a processor fails, the mirror processor must assume 100% of the workload of the failed processor. With the cluster concept, for an N processor cluster, each processor assumes 1/(N-1) of the workload. This results in considerably less cost for standby processing capacity required to sustain a workload even with processors down.

**Groups.** The Ynet HSRAM contains a set of groups. A group is represented by three bits: Valid, Member, and Semaphore. If the Valid bit is set in any processor, the group is allocated and is in use. A processor sets its Member bit to indicate that it belongs to the group and should receive messages addressed to the group. The Semaphore bit is used to ensure mutual exclusion during operations which allocate, deallocate, or test the status of the group.

Groups can be allocated statically (when the system starts up) or dynamically. DBC/1012 software uses the Ynet group capability to form groups of processors which are concerned with a given user transaction, so that messages can be sent only to those processors engaged in doing work on behalf of that transaction. Present software allocates over 1,500 groups per system.

Dynamic groups are important for workloads which consist of a large number of relatively simple transations. Each transaction cannot use the full degree of parallelism provided by the DBC/1012: in fact it typically affects only a few processors. However, the workload collectively uses the entire configuration. The dynamic group concept permits the DBC/1012 to be dynamically partitioned into subsets appropriate to each transaction. Note that many overlapping subsets can be in existence simultaneously.

We believe that the dynamic groups concept will be essential to programming and operation of MIMD computers with thousands or tens of thousands of processors which are proposed for the 1990's.

## *Example Message Flows*

The following examples discuss how various features of the Ynet operate for sending messages through the system which evoke new work in the Access Module Processors. These messages are called "Step Messages". They are prepared by the

SQL Parser, which executes in the IFPs. Their transmission to the AMPs is requested by a second program which runs in the IFPs, called the Dispatcher. Each step message requests one AMP, a group of AMPs, or all AMPs to do some operation against the database. The parser generates one or several step messages to effect each SQL statement received from the user.

**Arbitration.** The first example (*See Figure 6a*) shows two IFPs each attempting to send a step at the same time. The first node on the left decides in favor of the step message from IFP1 in the example because it sorted lower. Note that this means that the Ynet not only handles contention, but in doing so automatically enforces system wide message priorities. DBC/1012 software uses this feature to ensure that certain critical control messages always have priority over ordinary messages, for example.

The winning message reaches the top of the network, and is turned around and broadcast down the network. It arrives at the processor interface of each processor at exactly the same time. (*See Figure 6b*).

**Acknowledgement.** Every processor interface generates some kind of acknowledgement to every received primary packet. Acknowledgements are minimum length (2 byte) packets which contain only a command code and the ID of the responding processor. The command code used in acknowledgement packets is a function of whether the processor is intended to receive the packet and whether it can in fact receive the packet.

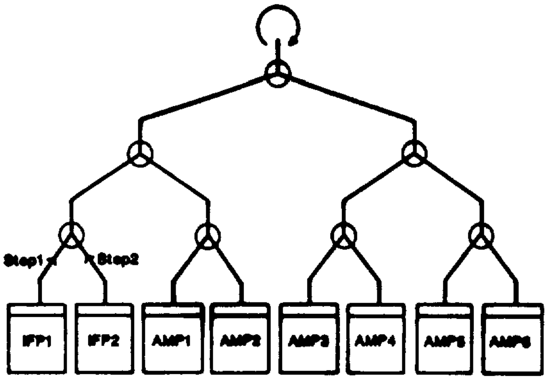


**Figure 6a** - *Both IFP's try to send at the same time*

The acknowledgements are sorted (merged) up the network just like the primary packets that evoked them. Only the lowest-sorting acknowledgement packet makes it to the top of the network: it is in turn broadcasted down the network and is seen simultaneously by all of the processor interfaces.

After dealing with the surviving acknowledgement packet, each processor interface then begins the cycle over again by trying to send its next packet. This will be the same packet as in the previous attempt for all processors except IFP1, which got its message through.

If a processor had no actual message to send, the processor interface sends a special "idle" sequence which is guaranteed to sort higher than any real message.

A primary message can be addressed to a processor id, a hash bucket, a group, or a transaction vector. Step messages are always addressed to a hash bucket, a dynamic group, or to the special static group of all AMPs. We will consider each in turn.

**One AMP (Point to Point).** Step messages are sent to a hash bucket when the Parser recognizes that all of the rows in the request must have the same Prime Key. An example of this kind of request in a banking application would be "find the balance of account 1234567". In this application, account number is the primary key, and account 1234567 hashes to AMP4. (*See Figure 6c*).

Every processor interface generates an acknowledgement to every primary packet. In this case, AMP4 generates an "ACK" or positive acknowledgement: AMP4 is the only processor which has the appropriate hash bucket. All other processor interfaces generate a Not Applicable Processor (NAP) acknowledgement. NAP means that the processor interface determines that the packet was *not* addressed to that processor. The command codes are assigned such that ACK sorts lower than NAP. Since ACK is the lowest sorting acknowledgement packet, it reaches the apex of the network and is broadcast down the network.

When IFP1 sees the ACK, it knows that the appropriate processor could and did receive the intended message. This is reflected by an update to the Output Pointer Circular Buffer in HSRAM. Also, the processor interface generates an interrupt to the processor to tell it to examine the Output Pointer Circular Buffer.

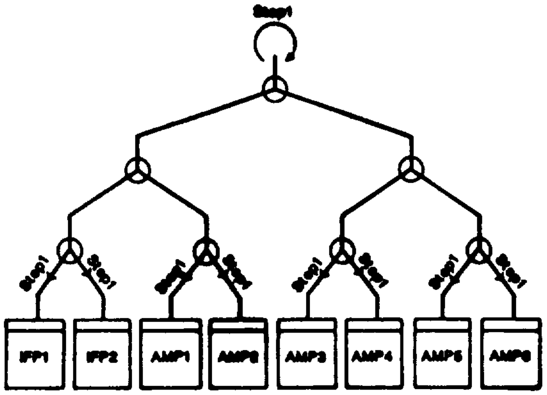


**Figure 6b** - *The winning packet is received by each* processor interface at the same time

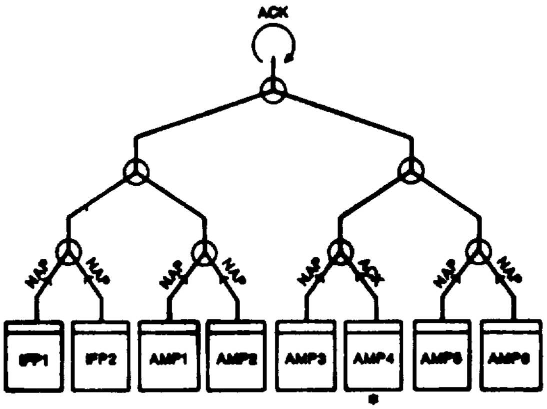


**Figure 6c** - *Acknowledgements for point to point messages*

In AMP4, the ACK confirms receipt of the message, now stored in the Input Circular Buffer. The AMP4 processor interface generates an interrupt to the processor to cause software to process the new message.

All other processor interfaces see both the primary packet and the acknowledgement packet, but take no action. They simply wait for the acknowledgement sequence to complete. At that time, all of the processor interfaces, whether they were involved in the packet we just followed or not, attempt to send thier next primary packet.

For IFP1, this will be the next packet it had ready to send. For all other processors, it will be the same packet they previously tried to send without success, or the idle pattern, or a new packet which software placed into the Output Area and chained to an active Transaction Vector during the processing of the current packet.

**A Few AMPs (Multicast).** Next, conisider a message addressed to a dynamic group. Messages to commit or roll back work for a simple transaction are typically addressed to the dynamic group allocated just for that transaction.

The message is sent from IFP1 in exactly the same way as in the previous example, and is received by all processor interfaces in exactly the same manner as before. (*See Figure 6d*). In this case, only AMP2 and AMP5 are members of the group. (Previous step messages in this transaction would have caused these two AMPs to join the group). That is, only AMP2 and AMP5 have the Member bit set for this particular group.

Thus, both AMP2 and AMP5 generate ACKs; all other processors send NAPs. In fact the ACK sent by AMP2 sorts lowest, because it has a lower processor ID than the ACK send by AMP5. The ACK from AMP2 is thus broadcast to all processor interfaces.

Again, IFP1 deals with the ACK by generating an interrupt to software to signal successful transmission of the message. Both AMP2 and AMP5 generate interrupts to begin processing of the new message in the Input Circular Buffer. All other processors effectively ignore both the primary packet and the acknowledgement and wait until they can send their next primary packet.

**All AMPs (Broadcast).** A message intended for all AMPs is addressed to the special statically defined group of All AMPs by the sending IFP. (*See Figure 6e*). In this case, the processor interface of **every** AMP generates an ACK; only the processor interfaces of IFPs generates NAP acknowledgements.

As in the previous examples, IFP1's processor interface generates an interrupt to software that the message successfully got through to all receipients. Each AMP's processor interface generates an interrupt to cause the new message to be processed.

**Two Phase Commit.** Finally, consider an example in which one of the intended recepient AMPs *cannot* handle the message intended for it. (*See Figure 6f*). This condition typically results from congestion control algorithms in the DBC/1012. In this example, AMP3 is temporarily overloaded, and cannot receive more messages. The processor interface of AMP3 generates a third kind of acknowledgement message: negative acknowledgement (NAK). The command code for NAK is arranged to sort lower than ACK, which in turn sorts lower than NAP.

Since the NAK generated by AMP3 is the lowest sorting acknowledgement packet, it gets to the apex and is broadcast to all processor interfaces.

The processor interface of IFP1 generates an interrupt to software, as before. However, instead of storing the ACK response with the message, it stores the NAK response. This tells software that the message did not get through, and further identifies the lowest-numbered processor which could not receive the message. This information forms the basis for recovery action initiated by software.

In IFP2, the whole proceeding is of no interest, and IFP2 simply waits until it can send again.

In AMP1, the processor interface generated an ACK, and is thus expecting of have its ACK confirmed. The NAK tells the AMP1 processor interface that some other receipeint could not process the message. Thus AMP1 in effect discards the message and generates no interrupt to software. All of the other AMPs' processor interfaces take the same action.

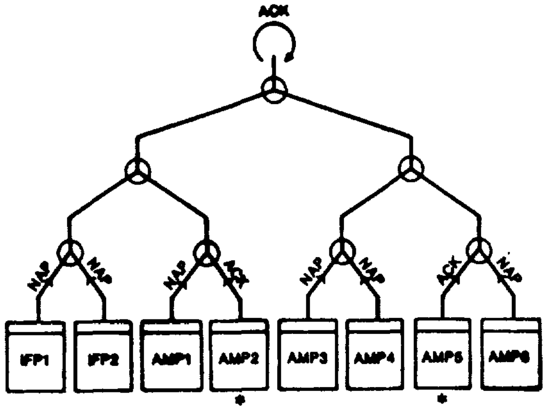


**Figure 6d** - *Acknowledgements for multicast messages to a dynamic group*

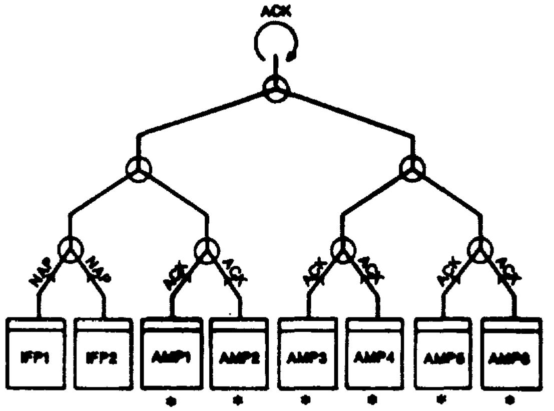


**Figure 6e** - *Acknowledgements for broadcast messages to a static group*

## Conclusions

As can be seen from the foregoing examples, the Ynet implements a two-phase commit protocol on every message, as a simple byproduct of transferring the message and acknowledging it. Two-phase commit is an expensive protocol if implemented in software, and is the only algorithm known for implementing reliable communications in a distributed system.

The combination of two-phase commit and group addressing in the Ynet means that the system can treat a group of processors as a single entity whenever desired. The group, like a single processor, receives messages as an atomic, uninterruptable operation which either entirely succeeds or entirely fails. Further, the processors in the group receive messages at the same time and in the same order.

Because of this property, programs which implement such synchronization intensive functions of a database management system as transaction commit and roll back, and system start-up and recovery -- typically the most difficult algorithms of a database manager and tough enough in a uniprocessor environment -- have straightforward extensions to a concurrent processing environment.

Thus, we have demonstrated how the Ynet design responds to the challenges of a highly concurrent environment, but also to the challenges of real-time, high availability, multi-user applications. In this regard, the Ynet addresses problems usually not considered in the design of exsiting concurrent processing architectures.

While the Ynet design evolved from the requirements of a commercial relational database management system, the structure is suitable for other classes of problems characterized by high degrees of concurrency. These could include numerically intensive problems, such as matrix manipulation and simulation of physical systems, or more general problems such as those posed by artificial intelligence research and implementation of object-oriented environments for large problems where a great deal of computational power must be harnessed.

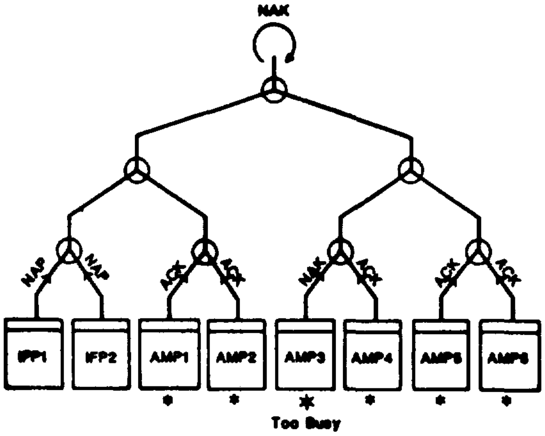


**Figure 6f** - *If any intended recepient cannot handle the message, none receive it (two-phase commit)*

## References

Codd, E. F., "A Relational Model for Large Shared Data Banks", *ACM, 13*, 6, pp. 377-387, June 1970.

DeWitt, D. J., "Data Base Machines", *Proceedings of the Tenth International Symposium on Computer Architecture*, IEEE, Stockholm, June 1983.

Hsiao, D. K., K. Kanan, and D. S. Kerr, "Structure Memory Designs for a Database Computer", *Proc ACM 1977*, pp. 343-350, December 1977.

Neches, P. M., "Hardware Support for Advanced Data Management Systems", *Computer, 17*, 11, pp. 29-41, November 1984.

Neches, P. M., "The Anatomy of a Data Base Computer System - Revisited", *Compcon '86 Digest of Papers*, IEEE, March 1986.

Neches, P. M., United States Patents 4,412,285 (October 1983), 4,445,171 (April 1984), and 4,543,630 (September 1985); others pending.

Neches, P. M. and J. E. Shemer, "The Genesis of a Data Base Computer", *Computer, 17*, 11 pp. 42-56, November 1984.

Smith, D. C. P. and J. M. Smith, "Relational Data Base Machines", *Computer, 12*, 3 pp. 28-39, March 1979.

# A Reconfigurable Optical Interconnection Network for Highly Parallel Architecture

X. Thibault , D. Comte and P. Siron (*)
O.N.E.R.A. / C.E.R.T.
Computer Science Department
2, avenue Edouard Belin, BP 4025
31055 - Toulouse cedex, - France
uucp: xavier@tls-cs.cert.fr

## ABSTRACT

The topology of the elementary processor interconnection in a parallel architecture is an important feature in obtaining high performances . Nevertheless, due to the constraints of electronics, the most of the parallel architectures have fixed interconnection topologies between elementary processors. As far as communications are concerned, optics presents interesting characteristics for application in highly parallel communication networks such : no crosstalk, free space transport and high bandwidth. The O.N.E.R.A./C.E.R.T. MILORD project aims at developing an optical interconnection network to build a reconfigurable highly parallel architecture dedicated to high performance applications. The multiprocessor prototype, which has been developed, is composed of INMOS microprocessors connected by their four serial links on an optical crossbar network. As the interconnection topology can change during the execution at pre-determinated points of the program, various configurations have to be managed. In this paper, we first a give a brief survey of optical network devices. Then, the architectural features and the main characteristics of needed software tools are presented. Finally, we give a survey of applications well suited for reconfigurable architectures.

Keywords : optical reconfigurable network, parallel architecture, transputer, occam, optical free space.

## INTRODUCTION

The increasing demands of applications like signal and image processing, artificial intelligence and numerical applications for growing powerful computation, indicate the needs of highly parallel computers (Ref. 1). With the advances of technology and declining cost of computer hardware, we can envisage the design of massively parallel architectures composed of several hundreds of processors. The elementary processor interconnection network becomes an important part of the parallel processing system. Its efficiency greatly contributes to the global system performance (Ref. 2).

The most attractive interconnection topology would permit any processor to directly communicate with any other in order to fit the application requirements. Most of the highly parallel architectures have a fixed interconnection topology between the processors due to the limitations of electronics for implementing highly parallel communications (pin limitations, wire design and wiring problems). This network rigidity confines them to specific applications. The configurable architectures have introduced the network flexibility : MPP (Ref. 3) and CHIP (Ref. 4). Nevertheless, up to day, no one has been capable of configuring any kinds of topology.

By its own nature, optics presents interesting characteristics : the non-existence of crosstalk between two optical links, the immunity to electromagnetic radiations, the possibility of using free space as a support of the data transport and the intrinsic parallelism (photons don't interact), should contribute to improve the communications in a large parallel computer. It can provide an easy way to implement large reconfigurable interconnection networks. In a such context, the network can be configured to match the communication characteristics of the performed algorithm. Various implementations with the different optical media are feasible : optical fibers (Ref. 5), integrated optics (Ref. 6) and free space (Ref. 7). The basic idea of most existing systems is that a crossbar network can be implemented by a matrix-vector product system (Ref. 8).

According to this idea, we have been developing for two years at O.N.E.R.A./C.E.R.T., a multiprocessor prototype called MILORD, based on a such network to interconnect electronic processors. This project aims at developing an optical crossbar network to build a reconfigurable highly parallel architecture dedicated to high performance applications.

In this paper we first present optical network devices and give the performances that can be expected. In the following section, we describe the architectural features of the MILORD machine, software tools at system and user levels. At last, we give a survey of applications well suited for reconfigurable architectures using large networks builded with optical components.

(*) in collaboration with P. Churoux, M. Fracès and Laug at O.N.E.R.A./C.E.R.T.- Dept of Optics

CH2649-2/89/0000/0437$01.00 © 1988 IEEE

## OPTICAL NETWORK DEVICES

Optics is well adapted to communications due to its long distance capabilities, the non-existence of crosstalk and the three dimensional propagation. The high bandwidth permits the use of bit serial links rather than conventional parallel ones. It reduces the number of communication channels, allowing more processors for a given density limit.

For the last several years, optical fibers have been used for long distance communications. From its widespread use in telecommunications, they are being propelled into data communications within and between computers. But optical networks based on optical fibers suffer of the same limits as electronic wires as a as reconfiguration is concerned. 2x2 fiberswitches are available, but with a too low switching time for our application ( > 1 millisecond ).

Other devices have been proposed to implement optical networks. Integrated optics projects aim an electro-optic crosspoint (Ref. 6), which could be faster than electronic switches (picoseconds range). This device called directional coupler, has two inputs and two outputs and a binary control. With their two internal states ; straight and exchange, so that any input can reach any outputs. In order to realize larger networks, integrated optics must integrate more couplers on a chip or connect the couplers between themselves with optical fibers. The number of ways is limited by the chip size and pinout for the first solution and the coupler attenuation and insertion loss for the second. Today, the number of ways is limited to a few tens.

The optical free space transmission seems a more promising technique to meet our requirements: flexibility, broadcast capability, large parallel system and three-dimension links.

Many optical devices have been described for this approach (Refs. 7-9). The reference 10 analyses optical solutions to implen.ent crossbar network. Systems are mainly based on matrix-vector product concept to implement an optical crossbar switching network. The figure 1 shows the most popular $N^2$ parallel matrix-vector product implementation.

The design consists of a column of emitters (input vector), a row of detectors (output vector) and a sandwich crossbar mask (input matrix). Each emitter broadcasts on an entire row of the mask, so that a binary 1 is represented by light of a fixed intensity and a binary 0 is represented by a zero intensity. Similarly, each detector is capable of summing light from an entire column of the crossbar mask. The summation and the spreading are accomplished in optics by means of lenses.

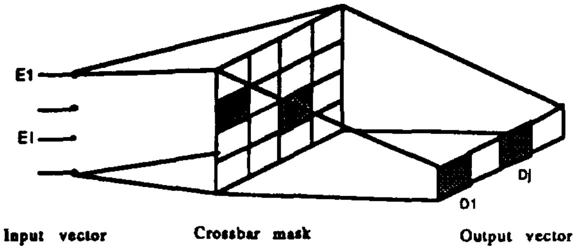


Figure 1. $N^2$ Parallel Matrix-Vector Product Implementation

The mask can be viewed as an array of windows, which are transparent or opaque. When the window of the column i and the row j is transparent (value 1), the $i^{th}$ emitter can send optical signal to the $j^{th}$ detector. In the case of opaque state (value 0) light coming from the $i^{th}$ emitter is blocked : thus, the mask disallows communications between the two elements. The crossbar mask device is set up by the matrix value with optical or electronic control depending of the optical component.

This system can perform matrix-vector product or implement a crossbar network. The crossbar mask acts indeed as a switch at each window to permit or to block incoming light at each position. The switches are commanded to establish a connection, or a global interprocessor topology configuration. For one to one communication, at most one switch in each row and one switch in each column may be opened and all other windows must be closed.

It is interesting to note that this resulting optical network has the broadcast capability (a row of opened switches), added to the basic crossbar function. This capability can reduce the complexity of parallel algorithms

The Spatial Light Modulator (SLM) is one of the devices which can implement the matrix-product function. Its role is to switch inputs to ouptuts (crossbar mask) . It may be either transmissing (as above) or reflective (input and output vector on the same mask side). The table 1 summarizes the main parameters of the most interesting SLMs.

| SLMs | Resolution | Response Time | Control |
|---|---|---|---|
| Liquid Crystal Light Valve (LCLV) | 500x500 | 50 ms | optical parallel |
| Deformable Mirror Device (DMD) | 128x128 | 30 µs | P. L. L. elect.* |
| PLZT | 1000x1000 | 400 µs | optical parallel |
| Non Linear Interferential filters (NLIF ) | 100x100 | 200 µs | optical parallel |
| Magneto-optic | 512x512 | 0,5 µs | P. L. L. elect. * |

* P. L. L. : Parallel Line by Line with electrical control

Table 1 Parameters of some SLMs (Ref. 10)

Most of the given SLMs are not currently available, except the HUGHES Liquid Crystal Light Valve and the Deformable Mirror Device. The others are still studied in laboratories : thus, the given values are approximately. Nevertheless, this list gives a good idea of the performance that we can expected. Three parameters mainly define the characteristics of the resulting network : the resolution, the response time and the control.

The resolution gives the number of pixels provided by the SLMs. In this way it defines, according with the contrast ratio and the efficiency of the SLM, the number of paths that we can expected in a network. The practical limits of SLMs allow the implementation of moderately large networks (at least 100x100).

The response time is the time required by the SLMs to adress a pixel (corresponding to setting up a window). It therefore indicates the time to establish a connection between an input and an output. We note in table 1 that the response time is slow : the speedest modulator (magneto-optic) is still too much slower than switching time in electronic technology. The reconfiguration time needed to establish all the connections of a global interconnection topology, also depends on the SLM control.

There are three modes of access to control the switches: optical parallel, electrical parallel line by line and electrical sequential. The optical control is the most interesting mode because it permits to set up all the pixels at the same time whereas the second mode can simultaneously command only a line of pixels. The sequential control adresses the pixels sequentially. It is possible to evaluate the total reconfiguration time depending on the control mode : $T_r$ (parallel), $N{\bullet}T_r$ (parallel line by line) and $N^2{\bullet}T_r$ (sequential) where $T_r$ is the response time and $N^2$ is the number of pixels. We note that the reconfiguration time in parallel mode doesn't depend of the number of pixels. Then, a such control provides a reconfiguration time, which can reach the performances obtained in large electronic networks

Moreover, the bandwidth of optical crossbar networks is high. It is only limited by the response characteristics of the optical detector array ( < 1 Gigahertzs). A large optical network can be actually implemented with a high data rate. Due to the slow switching time of the current optical technology, it is unrealistic to change any connection at any time. It is better to modify the interconnection topology (all the connections) at pre-defined points of the execution. At these reconfiguration points the right topoloy is required to satisfy the most direct communications between elementary processors. Considering the flexibility of the network, any well known topologies (mesh, tree, hypercube...) or others original topologies can be established. This operation is well adapted to many applications like numerical and image processing.

Let us now give a description of the MILORD machine, which is composed of a such optical network.

## ARCHITECTURE OF MILORD

### Introduction

The MILORD project (Multiprocesseur Interconnecté par Liaisons Optiques Reconfigurable Dynamiquement) aims at developing an optical interconnection switch between elementary processors to face the problem for which electronics suffers fundamental limits. Such a switch allows the parallel architecture to adapt a network configuration to a specific algorithm. Based on this optical switch, we design a massively parallel reconfigurable architecture dedicated to high performance applications. We are building a prototype now to prove the feasibility of such architecture. The design of dedicated tools for reconfiguration management is under way and a survey of applications well adapted to reconfigurable highly parallel architectures has been undertaken.

In this context, the network topology can be altered between different algorithm executions or between different steps of the same algorithm execution. Then, two reconfiguration modes can be taken into account : static and dynamic. In a static mode, a program is written for a fixed topology of the elementary processors. In a dynamic mode, the topology can be dynamically modified at pre-determinated synchronization points of the program, so as to match the different communication schemes required by the algorithm. The MILORD machine, which is implemented in the second mode may be characterized as a reconfigurable architecture. Today several types of parallel architectures have been developped like MPP and CHIP. In contrast with their networks, which can only configure a limited number of topologies, the MILORD machine can realize any required topology.

The architecture of the MILORD machine is mainly composed of an execution unit, an optical crossbar network and a control unit as described in figure 2. This machine is connected to an host computer (PC microcomputer), which supports all the development tools

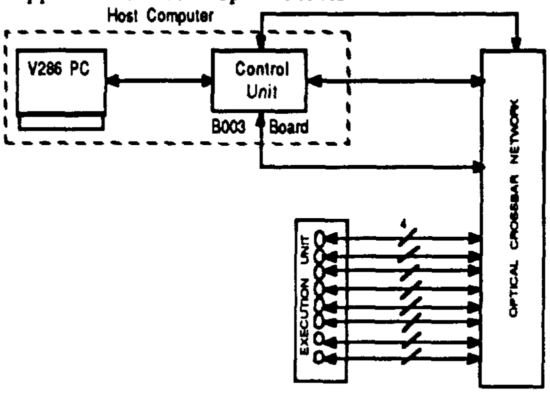


Figure 2. Architecture of the MILORD machine

### The execution unit

The execution unit is built with 8 INMOS T414 Transputers connected to the optical crossbar network. The T414 Transputer is a new generation microprocessor (Ref. 11). On a single chip, there are a fast 32 bit microprocessor, 2 Kilobytes of fast static RAM and four high speed communication links. In addition, the Transputer can access a 256 Kilobytes external memory. The four communication links allow networks of Transputers to be constructed by direct point to point connections. Some Transputer interconnection topologies are given in the figure 3. In the MILORD machine, the four links of each Transputer are connected to the optical network and support a standard operating speed of 10 Megabits/s.

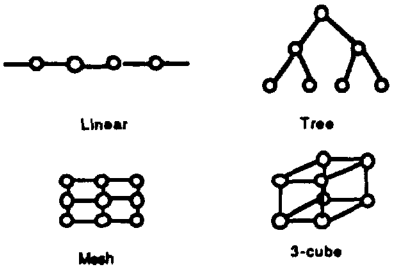


Figure 3. Examples of Transputer interconnection topologies

OCCAM, the standard programming language of the Transputer permits the expression of parallelism between processes (Ref. 12). This OCCAM language has been designed for using with multiprocessor systems. Concurrency processes and message mode communications are deducted from the concept of Hoare's CSP (Communication Sequential Processes) (Ref.13). Communication channels are an explicit part of the language and can be mapped directly on the hardware links connecting different Transputers.

In the MILORD machine, OCCAM programs are decomposed in successive algorithmical sections, each one referencing an optimal network configuration. Between two sections a reconfiguration point synchronises the processors before to reconfigure the toplogy. These reconfiguration points together with the required topology are explicity managed by the programmer using dedicated programming tools.

Within a section, execution takes place in the MIMD mode, corresponding to the asynchronous nature of a set of interconnected Transputers. However, the Transputers involved in the execution of a section can execute the same processes on different data.

### The optical crossbar network

The optical crossbar network is based on matrix-vector concept, as presented above. The optical component that we have chosen is the Liquid Crystal Light Valve (LCLV) from HUGHES Corporation because it is the only ones, which is available commercially and it gives a good compromise between contrast ratio (100) and efficiency (25 %). The figure 4 shows the details of this component.

The network of the MILORD machine provides a 34 by 34 switch allowing any Transputer link to connect to any other link without contention (Ref. 15). Two channels are reserved for host communications and each Transputer has four bidirectionnal channels to the network. The Transputer output link is connected to a laser diode to send data to the network and the Transputer input link is connected to a photodiode to receive data from the network. The switch settings are controlled by a parallel optical signal coming from a Cathod Ray Tube (CRT), which is commanded by a graphics board.

The reconfiguration time of the network, meaning the total time needed to change the settings of all switches, depends especially of the LCLV response time plus the CRT erasing time for taking off the old configuration pattern and the CRT printing time for displaying the new pattern. The total time observed is about 200 milliseconds (Ref. 16). New LCLVs reduce this time significantly.

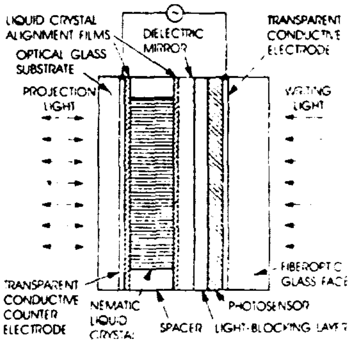

LIQUID CRYSTAL ALIGNMENT FILMS
DIELECTRIC MIRROR
TRANSPARENT CONDUCTIVE ELECTRODE
OPTICAL GLASS SUBSTRATE
PROJECTION LIGHT
WRITING LIGHT
FIBEROPTIC OR GLASS FACE PLATE
TRANSPARENT CONDUCTIVE COUNTER ELECTRODE
NEMATIC LIQUID CRYSTAL
SPACER
LIGHT-BLOCKING LAYER
PHOTOSENSOR

Figure 4. Hughes Liquid Crystal Light Valve (Ref.14)

### The control unit

The control unit of the MILORD machine is implemented on the host computer Beside the classical functions, It assumes three functions :

- Load
- Synchronize
- Configure

The Transputer Development System is provided with a Loader.which loads processes on the Transputers by means of a pipeline topology. But we can modify the load strategy for profiting of the two host channels to the network by loading in parallel on the two channels.

The Synchronizer manages the synchronization points before a configuration command takes place to establish a new interprocessor topology. These rendezvous points are set by the user inside the process of each Transputer. During the execution, each Transputer receives a token from a neighbour and sends it to another neighbour up to the host computer at these points : then, the Transputers are waiting for an OCCAM channel read. When the new topology is set up, the host computer sends a message to the network, which is spreaded to the processors. Then, the Transputers can resume operation on the new topology.

The Configurer commands the setting of the topology pattern demanded by the program. Due to the optical nature of the LCLV command, a matrix pattern is generated on a CRT to establish the required interconnection topology. We use an INMOS graphics board, required by the host computer for drawing the pattern display of the new topology on the CRT. The predefined patterns are compiled in a topology library and stored on the host computer disc. This library contains a set of conventional topology : ring, tree, mesh, hypercube. With the Transputer Development System tools, the user can also define and store others topology patterns

## APPLICATIONS OF MILORD

In the field of numerical applications, a number of algorithms could benefit of a reconfigurable architecture. For example the JACOBI method which solve the linear system : B • x = b by computing the vector x by means of a series of asymptotically values. The algorithm can be written in this form :

```
loop
        Do 10  i = 1 to N
                 $y_i \leftarrow \sum_{j=1}^{N} A_{ij} \cdot x_j^{(p-1)}$
                 $x_i^{(p)} \leftarrow (b_i - y_i) / V_i$
  10             $z_i \leftarrow x_i^{(p)} - x_i^{(p-1)}$
until  // z //  <  epsilon
```

where N is the matrix dimension, A the matrix obtained by suppressing the diagonals of B and V is the vector containing the diagonals of B.

We can split the algorithm in three sections , each one requiring a specific interconnection topology (see figure 5) :

- The matrix-vector product.

- The transposition of the vector y from the first column to the first row and then the computation of x and z.

- The computation of norm.

Considering $N^2$ processors and a NxN matrix, the transfer of the first section's data takes the following times : O(2logN) with the first topology of the figure 5 and O(2(N-1)) with a mesh topology. The benefit of the reconfiguration is more obvious in the second section, because the new interconnection topology minimizes the number of data transfers and doesn't need any routing (only direct connections). The norm between this current result vector and the previous result of the last iteration is computed in the third section. A binary tree topology on the first processor row gives the same result as the first section.

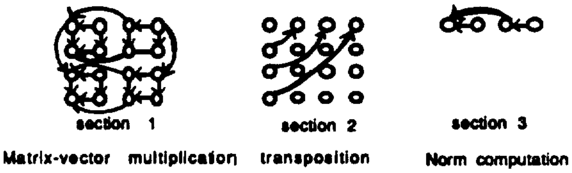


Figure 5. Topologies for the JACOBI method

Then, the flexibility of the network decreases the data transfers of the algorithm in comparaison with the fixed topologies. Moreover, it improves the section execution time as shown by the table 2.

| | Reconfigurable network | Mesh topology |
|---|---|---|
| Section 1 | 2logN • T<br>+ (logN + 1) • C | 2(N-1) • T<br>+ N• C |
| Section 2 | T + (2 • C) | (2N -1) • T + 2 • C |
| Section 3 | logN • (T + C) | (N-1) • (T + C) |
| Total<br>1 Iteration) | (3logN + 1) • T<br>+ (2logN + 3) • C | (5N - 4) • T<br>+ (2N -1) • C |

C : the average computation time.
T : the transfer time.

Table 2 : The execution times of the JACOBI algorithm (Ref. 17)

It is to be noted that the performances of the JACOBI method with a reconfigurable network are better than those obtained with the mesh topology. In this context, the execution time of the algorithm depends of course on the network reconfiguration time. It must be compatible with the transfer and the computation time.

The Gauss algorithm and some differential partial equation problems can also be divided in several sections corresponding to different interconnection topologies. Many other reconfigurable algorithms can be found in Reference 18.

Image processing requires growing power to process increasingly larger images. A reconfigurable network is well suited to certain applications. The low level algorithms (filters, histograms, correlation etc..) rather use mesh topologies with four, eight or more neighbours than pyramidal topologies adapted to high level pattern recognition. The reference 19 shows that flexibility systems are promising for increasing the image processing system.

## CONCLUSION

In this paper we have presented an optical network that allows any link to be connected to any other link without contention. Potentially, optical networks provide a high bandwidth and a large number of ways. Taking into account reconfigurable capabilities when coding parallel algorithm may avoid limitation due to fixed nature of current massively parallel architecture. The interconnection topology is dynamically modified at pre-determinated moments, so as to match the different communication schemes required by the sections of an algorithm. Then, high performance can be achieved over a wide range of applications (image processing and numerical applications) with a reconfigurable architecture like the MILORD machine.

A such optical network takes advantages of the best features of two technologies by using optics for communications and electronics for computation. Massively parallel architectures using large networks can be reasonably envisaged with optical components.

### Acknowledgements

This research is supported by the Direction des Recherches et Etudes des Techniques (DRET).

### References

1. D. Schaefer and J. Fischer, "Beyond the Supercomputer", IEEE Spectrum, March 1982.

2. D. P. Agrawal, V. K. Jonakiran and G. C. Pathak, "Evaluating the performance of multicomputer configurations", IEEE Computer, May 1986

3. K. E. Batcher, "Design of a Massively Parallel Processors" IEEE Transactions on Computers, September 1980.

4. L. Snyder, "Introduction to the Configurable Highly Parallel Computer", IEEE Computer, January 1982.

5. L. Dekker and E.E.E. Frietman, "Optical link and processor clustering in the DELFT parallel processor", Communications of ACM, 1988

6 L. Mac Caughan, "Design and performance limitation of integrated electro-optic cross-points" Proceedings of IEEE,CH2064-4, 1984.

7. A. D. Mac Aulay, "Spatial Light Modulator interconnected computers", IEEE Computer, October 1987.

8. A. A. Sawchuck and B. J. Jenkins, "Optical interconnection networks", Proceedings of Parallel Processing Conference, August 1985.

9. A. A. Sawchuck, B. J. Jenkins, C. S. Raghavendra and A. Varna, "Optical crossbar networks" IEEE Computer, June 1987.

10. P. Churoux, M. Fracès, M. Laug, D. Comte, P. Siron and X. Thibault, "Optical crossbar network analysis", International Symposium on the Technologies for Optoelectronics, November 1987.

11. Inmos Limited, IMS T414 Transputer Data Sheet.

12. D. May, "Occam2 language definition" Inmos Limited, February 1987.

13. C. A. Hoare, "Communicating Sequential Processes" Communications of ACM, August 1978.

14. W. P. Bleha et al, "Application of the Liquid Crystal Light Valve to real time optical data processing", Optical Engineering, July 1978.

15. D. Comte, P. Siron, X. Thibault, P. Churoux, M. Fracès and M. Laug, "Communication par crossbar optique dans une architecture informatique massivement parallèle", Revue de physique appliquée, October 1987.

16. M. Fracès, J. P. Bouzinac, P. Churoux, M. Laug, D. Comte, P. Siron and X. Thibault, "A multiprocessor based on an optical crossbar network: The MILORD project", SPIE Optical Computing Conference, August 1988

17. P. Churoux, M. Fracès, M. Laug, D. Comte, P. Siron and X. Thibault, "Le projet MILORD", Rapport interne n° 1/3272 ONERA/CERT, 1987.

18. P. Berger and M. Dayde, "Architecture à parallélisme massif: une approche algorithmique", Rapport interne n°1/8790 ONERA/CERT, 1987.

19. N. A. Alexandridris, "Architectural adaptations in image processing supersystems", First International Conference on Supercomputers Systems", December 1985.

# THE GAM II PYRAMID

*Zahi Abuhamdeh*
Department of Electronics and Computer Engineering
George Mason University
Fairfax, Virginia

*Abstract* - The GAM II Pyramid is a hierarchical structure with a total of 1365 processing elements. The sequencer designed for controlling this structure is capable of executing pyramid oriented primitives similar to an add and subtract operations, as well as independent procedures similar to convolution. The pyramid contains three control buses embedded in the hardware in anticipation of a future control system that will contain three independent control units.

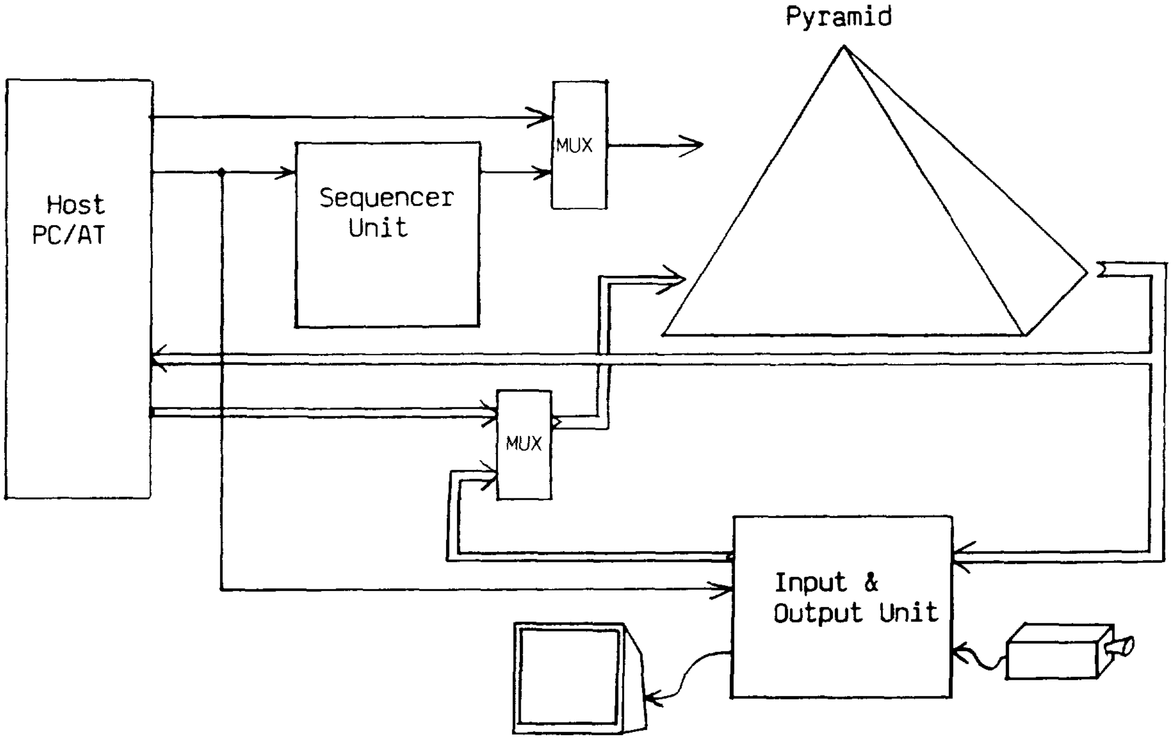
Keywords - Pyramid Data, Scalar Data, GAM I Pyramid, GAM II Pyramid, Corner Turning.

## 1 - Introduction

The GAM II Pyramid is a six level pyramid, which is a one level expansion of the previous GAM I Pyramid, used for image processing applications [1][2][3]. Both systems were developed in George Mason University's Advanced Computer Architecture Laboratory. The GAM II Pyramid, figure 1, is a hierarchical Single Instruction Multiple Data (SIMD) system.

## 2 - The Pyramid Structure

The GAM II Pyramid contains 1365 processing elements, made up of 172 custom microcircuits designed for the Massively Parallel Processor. The pyramid structure is broken down into the following sections: The basic processing element organization, the daughter cards that contain the processing elements, the back plane that contains the daughter cards and the processing element adder network.



The GAM II Pyramid System

Figure 1

### 2.1 - The Processing Element

The GAM II Pyramid's processing elements are identical to the Massively Parallel Processor's [4], with an expanded inter level communication network, figure 2. Each processing element is connected in a quad tree architecture that is six levels deep. A level is an N X N square mesh whose edges on the bottom three levels are connected in a taurus topology and on the top three levels connected to a logical zero, figure 3. A processing element can communicate to four siblings (those to the the North, South, East and West), four children (V, X, Y and Z) and one parent. A Sum-OR circuit on each level signals if any processing element has a value "one".

### 2.2 - The Daughter Card

The six levels are built by using 45 identical daughter boards developed at the Advanced Computer Architecture Laboratory, figure 4. Each GAM II Daughter board has four MPP microchips, and thus has thirty two processing elements configured in an 8 X 4 array slice. There are four Static Random Access Memory (SRAM) microchips configured in an 8K X 8. Eight three-state switch microchips are used to perform level transfers, two per MPP microchip. The three-state switches are attached to every processing element bus, and are enabled whenever a level transfer cycle is performed. Once the switches are enabled, the sending level performs a memory write to the three-state switches and the receiving level performs a memory read from the three-state switches.

### 2.3 - The Back Plane

The back plane of the GAM II Pyramid contains three identical control buses and a communications network that is unique to each card connector. The inter-daughter card communication determines the logical location of that particular set of processing elements. The back plane is an active back plane that performs signal buffering and some logical operations on the control signals.

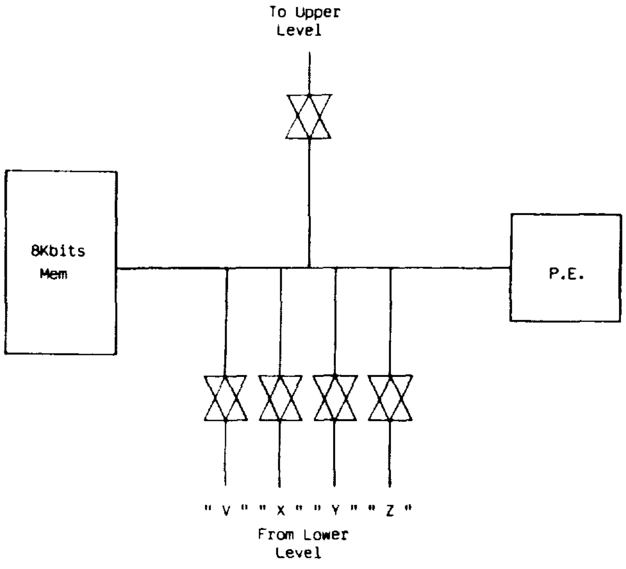


Figure 2

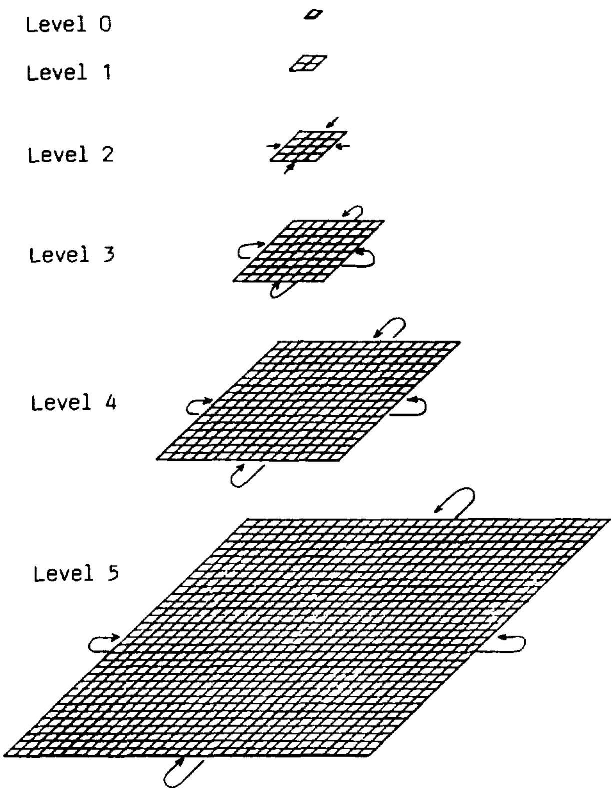
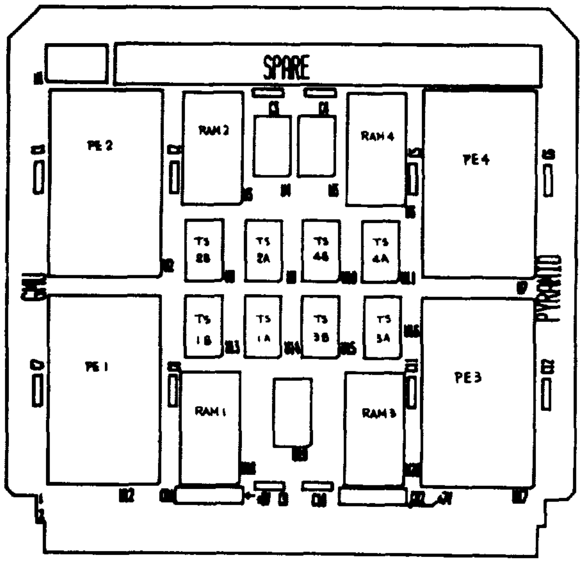


Figure 3



GAM II Pyramid Daughter Card

Figure 4

One of the three control buses, controls the base level, another controls the level above the base and the last controls the top four levels, figure 5. Though currently there is only one controller, the pyramid can handle up to three controllers with each issuing a unique instruction at the particular levels. Each control bus is buffered through a set of latches that could be configured to be transparent for debugging purposes or as part of an instruction pipeline to facilitate higher clock speeds. Discrete logic, on the back plane, has been added that allows for level disabling when not in use and generates a dummy memory read instruction whenever a level transfer operation is in progress.

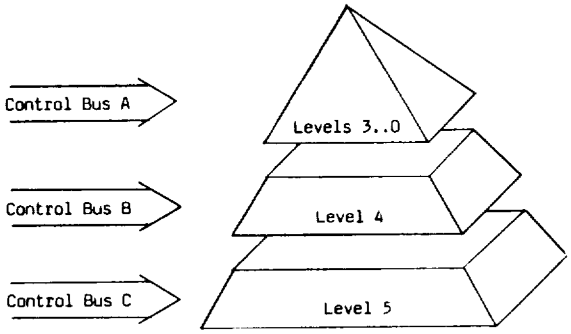
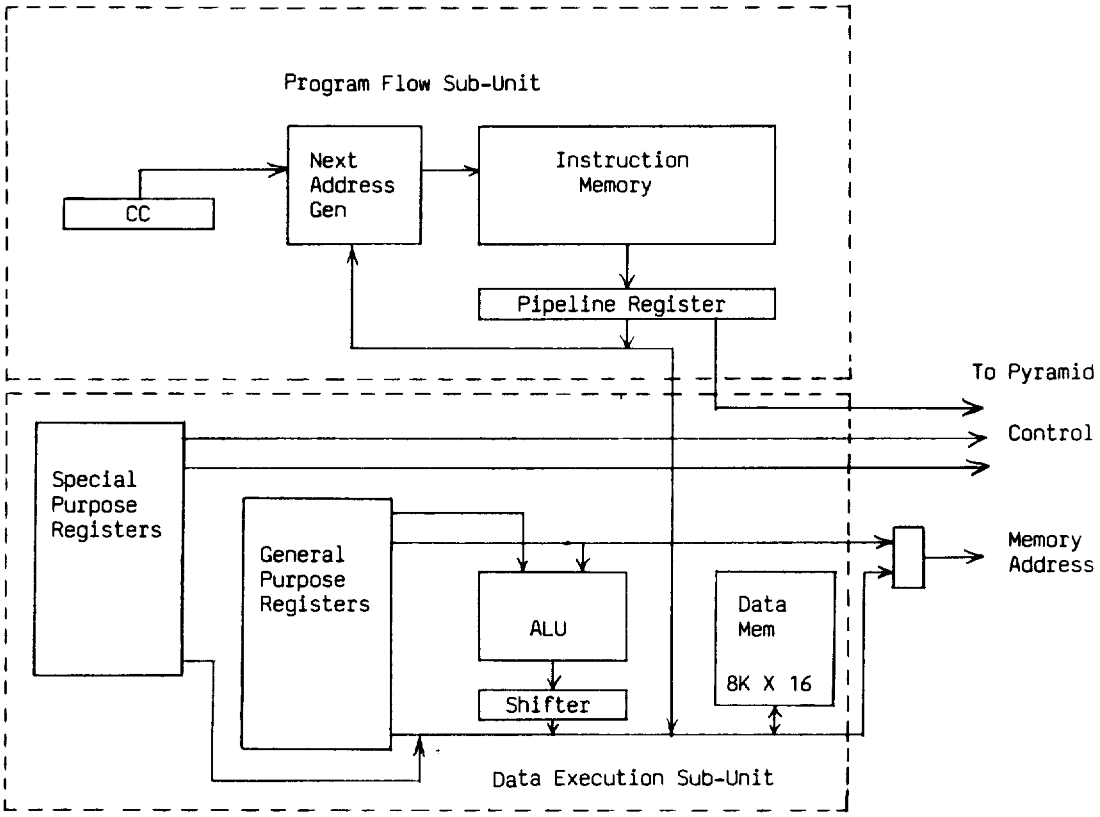


Figure 5

### 2.4 - The Adder Network

The Adder Network is a collection of high density 64K X 8 EPROM and binary adders. It is used to produce a sum of 256 one bit inputs with a value "one". The EPROM is programmed to produce the 5-bit sum of the 16-input address bits that have a value "one". There are sixteen EPROMs that each sum 16 bits for a total of 256 input bits and produce sixteen partial sums. The sixteen partial sums are then added together in pairs by eight binary adders to produce eight partial sums. The process is repeated until all the partial sums are totaled and one number remains. The addition is carried out in parallel and the total delay is 400ns which is well within one cycle of the 500ns targeted cycle time of the GAM II Pyramid clock.



Sequencer Unit

Figure 6

### 3 - The Sequencer

The sequencer is the lowest instruction level interface to the GAM II Pyramid arrays, figure 6. Remote calls are issued to the sequencer by the Host. The calls may request the sequencer to execute a primitive on the pyramid structure similar to an addition or multiplication, or to execute a full procedure that manipulates scalar data as well as pyramid data [5]. The Sequencer Unit is capable of issuing instructions to the pyramid at a rate of 2MHz. The function of the sequencer is to handle program flow control, global scalar calculation and pyramid instruction generation. The sequencer hardware is partitioned into two portions: the Program Flow Sub-Unit and the Data Execution Sub-Unit. The Program Flow Sub-Unit handles the sequencing of program instructions. The Data Execution Sub-Unit handles scalar data computation and storage. Pyramid instructions are generated in coalition with the two Sub-Units.

### 3.1 - The Program Flow Sub-Unit

This portion of the sequencer contains the next address generator, the Micro Memory, the Pipeline Register and the Condition Code register.

### 3.1.1 - The Next Address Generator

The Next Address Generator is based on Advanced Micro Device's 2930, which is a bit-sliced program flow control microchip. The Next Address Generator is composed of four such microchips connected to form a sixteen bit address bus of which only thirteen bits are used. There is an adder module as well as four input sources: an instruction pointer, an auxiliary register, an external data bus and a 17 register deep stack. The microchip can perform its full 32 instructions on the four input sources to produce a straight-through address or an offset branch address.

### 3.1.2 - Micro Memory

The Micro Memory is composed of a configurable Static Random Access Memory. The memory is built using 8K by 8-bit with a 120ns access memory modules and constructed to be addressed as an 8K by 96-bit memory or as a 48K by 16-bit memory. The 96-bit wide memory is used when executing primitives or procedures, during which the memory is in a read only mode. However, during host transfer operations, for example program load time or when the system debugger is operational, the memory is configured as a 16-bit data bus with read and write capability.

### 3.1.3 - Pipeline Register

The Pipeline Register is a 96-bit register that holds a sequencer instruction An instruction has four major fields. The first is the K constant which is used as a branch address or as a constant scalar for the Data Execution Sub-Unit. The second field of the Pipeline Register is used to control the Data Execution Sub-Unit. The third is used to control the Program Flow Sub-Unit. The fourth field is used for pyramid control.

### 3.1.4 Condition Code Register

The Condition Code (CC) register is a set of sixteen flags that are used by the sequencer to perform branching functions. The CC register contains the basic scalar flags generated by the Data Execution Sub-Unit as well as flags that are used to handle communications between the host and the sequencer. The Sum-OR values from all the levels of the pyramid are also latched into the CC register.

### 3.2 - The Data Execution Sub-Unit

The Data Execution Sub-Unit is composed of three basic parts. The General Purpose Registers and their accumulator, the Special Purpose Registers and the Data Memory.

### 3.2.1 The General Purpose Registers

There are sixteen 16-bit registers that are used for general purpose data calculation. These registers are attached to an Arithmetic Logic Unit and a Shifter. They can be used to store results of addition, subtraction and logical operations on scalars. An extremely flexible function of these registers is the generation of pyramid addresses. The registers can contain the address along with offsets that could be loaded from the K constant or the host, an addition or multiplication can take place and a new address will be generated. This same address can be used in a post-increment or a pre-decrement mode to support sequential address traversing for multiple-bit pyramid data.

### 3.2.2 Special Purpose Registers

Sixteen 8-bit registers are used for level and child select generation. These registers are triple ported registers that can output the level mask pattern and the child select values simultaneously. Any of these register values can be enabled to the Sequencer Data Bus.

### 3.2.3 Data Memory

This is a bank of 8K by 16-bits of Static Random Access Memory connected to the sequencer's data bus. The address of the memory is latched in a Data Memory Address Register (DMAR) from the data bus. Data is then read or written to the memory also from the data bus. This memory is used whenever a procedure runs out of register memory. This allows a compiled procedure, from a high level language, to contain large data structures.

### 3.3 Array Instruction Generation

Array Instructions are generated by the different portions of the sequencer. The basic processing element operations are issued from the pipeline register. Array memory source and destination addresses are stored in the Data Memory and the General Purpose Registers where their ALU is used to compute relative offsets, increment and decrement operations. Level masking information as well as an alternative set of child enable signals are maintained in the Special Purpose Registers.

### 4 - The Input and Output Unit

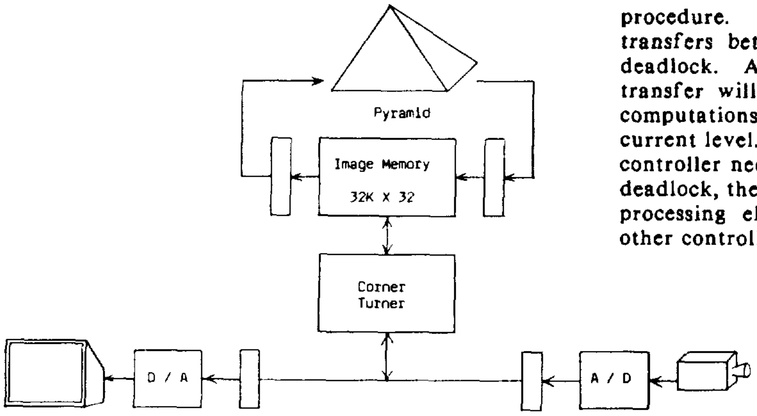
This unit is used to digitize analog input signals from a video camera and to generate analog video signals to display images from the pyramid, figure 7. An image has a 128 by 128 pixel resolution and 6 bits of gray shades. The unit has 128KBytes Image Memory that is capable of storing up to 8 images. An image is stored in sixteen 32 by 32 pixel frames that are shifted in and out of the pyramid by using the S-Registers on the base level of the pyramid (Level 5). The image memory stores data in bit planes and can communicate to the host, camera and the display device through a corner turning interface of shift registers. If a byte is needed from the Image Memory, a block of eight bytes is written to the corner turning block that in turn will be decoded for the appropriate byte.

### 5 - The Host System

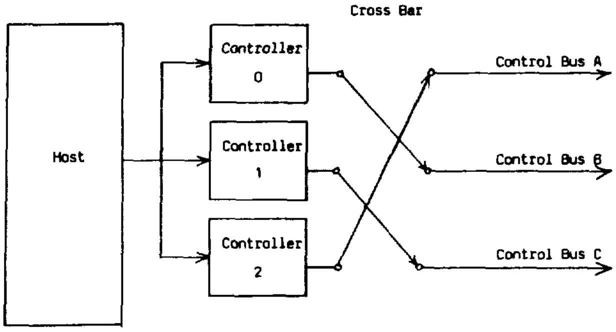
The Host System is an IBM AT compatible system that operates under DOS 3.3. The Host has a 40MByte hard disk. Text is displayed on a Monochrome display terminal and pyramid array graphics is displayed with 256 colors on a Video Gate Array (VGA) display. The Host interfaces to the pyramid by using a set of 32 8-bit registers. The Host can control the Pyramid Arrays, the Sequencer Unit and the Input and Output Unit.

Since an AT clone is used as a front end to the pyramid, PC software is available for program system development. Pyramid system software was developed using C and PASCAL, some of which include: an interactive micro assembly language with a simulated micro sequencer (PYRASM), an interpretive high level language called (Function IV) and a compiler that generates microcode from a high level language similar to C.

The Host was also extremely valuable in debugging the pyramid. Since through software it can single step any micro cycle. A state can be stopped and restarted as though it were running continuously. Special software has also been developed, S_Bug for example, is a debugger for the Sequencer Unit.



Input and Output Unit
Figure 7



Multiple SIMD Controller Configuration

Figure 8

### 6 - Future Control System Expansion

Connecting up to three sequencer type controllers is being investigated for future expansion of the GAM II Pyramid. If a single controller is used, it will be able to clock only one level at a time, the remaining unclocked levels will be idle. In such a situation, for every apex clock there are 1364 processing elements that are idle. A system being investigated contains multiple controllers that can be dynamically attached to any of the three control buses, figure 8. All array operations are memory to memory reference operations. If a controller is allowed to attach to a level and not be interrupted till it is done with that level, no context switching is needed. The data memory locations are pre-allocated at compile time, a controller will attach to a level and read its own data memory area, update the data with the desired operations and restore the updated data back to memory. Another controller can attach itself to that level at that time.

The controllers will be capable of running complete procedures and thus enabling the parallelisation to occur at the high level language level. Most of the program will be executing on the Host system. Parallelism is achieved by the use of Fork and Join operators that control the asynchronous execution of the procedures.

A controller can produce multiple level requests per procedure. This feature is useful to facilitate data transfers between levels and alleviating the problem of deadlock. Any controller that wishes to perform a level transfer will request two levels only when all its other computations are complete and had already released its current level. The next step is to request the two levels the controller needs. Should there be a conflict that results in deadlock, the controller attached to the level with the most processing elements gets its request fulfilled while the other controller waits till the two levels are available.

**Acknowledgment**

This project was supported by a grant from the The Army's Center for the Night Vision and Electro-Optics and a donation of MPP microchips from NASA's Goddard Space Flight Center. The Adder Network described above is the design of Trung Phan. Acknowledgment is also made to the following people for their contribution : Lu Yin, Xiaohong Zhang and Steve Jewel, for the construction and verification of the back plane. John Boyd and Brian Jordan for the construction of the pyramid interface to the PC/AT Computer. Rick Axtell and David Zafitz, for the construction of the Input and Output Unit. Professor David Schaefer, whose direction made this project possible.

**References**

[1] Schaefer, D. H., Ho, P., Boyd, J., Vallejos, C., "The GAM Pyramid". Parallel Computer Vision, L. Uhr, Ed., Academic Press, Boston, 1987, pp. 15-42.

[2] Schaefer, D. H., Ho, p., "Counting on the GAM Pyramid", Pyramidal Systems for Computer Vision. V. Cantoni, S. Levialdi, Ed., Springer-Virlag, Berlin.

[3] Man Bun Chu, "Object Recognition utilizing Combined General Purpose and Special Purpose Hierarchical Structures", Masters Thesis, George Mason University, Fall 1987.

[4] Burkley, J. T., "MPP VLSI Multiple Processor Integrated Circuit Design". The Massively Parallel Processor, J. Potter, Ed., MIT Press, Cambridge, 1985, pp 205-216.

[5] Phillips, Tom, Michael, Bret and Abuhamdeh, Zahi, "Microcode Generation for the Control of a Massively Parallel Computer", Proceedings of the 2nd Symposium on the Frontiers of Massively Parallel Computation.

# SECTION III: ARCHITECTURES

## Part 2: Poster Presentations

# PARALLEL AND PIPELINED VLSI DESIGN FOR THE HISTOGRAMMING OPERATION

M. Abdelguerfi
Dept. of Electrical Engineering
University of Detroit

A.K. Sood
Dept. of Computer Science
George Mason University

S. Khalaf
Dept. of Computer Engineering
Wayne State University

## ABSTRACT

This paper describes the design of a VLSI processing unit for the histogramming operation. The processing unit is composed of several bit-serial processing elements (PE's) connected according to the odd-even network topology. In this approach, histogramming is divided into two stages, the counting process and the filtering process. The filtering process is computationally inexpensive compared to the counting and marking phases. The use of a histogramming unit of fixed size to handle a large number of pixels is considered.

Keywords: Vision Architecture, Parallel Processing, Bit-Serial Architecture, Odd-Even Network, VLSI. MOS.

## INTRODUCTION

This paper presents the design of a MOS [1] technology based special purpose processing unit for the histogramming operation. The architecture is composed of several processing elements connected according to the odd-even network topology [2]. The processing elements operate on pixels bit by bit. As a result, this system is referred to as parallel bit-level pipelined architecture [3,4]. The main advantage of this approach is that the memory requirement of each PE is very small and is independent of the input size. The input operands are processed one bit at a time, hence, the amount of hardware in each PE is reduced. As a result, a large number of PE's can be integrated on a single VLSI chip. The proposed design has several features that are suitable for a VLSI implementation:

1) Simple PE's – The system is composed of one type of simple processing element. Each PE operates on data bit by bit. Thus, the design and verification of the circuit will be easy.
2) Overlap of data I/O and processing – The data processing time is completely overlapped with the inputting and outputting of data to and from each PE.
3) Static interconnection network – For ease of implementation a static interconnection between the different PE's is preferred as compared to a dynamic network. This will allow the system to process several data streams in a pipelined manner.
4) High throughput – This is achieved by the use of several bit-serial PE's operating in parallel and pipelined manner.
5) Low pin count – This property results from the inputting and outputting of operands one bit at a time to and from the PE's.

## THE ALGORITHM

The histogramming operation has been divided into two parts, the count and mark process and the filtering process. The count and mark process is the process of assigning to each grey level Gi, a count field Ci and a mark bit Mi. The filtering process consists of discarding all duplicate grey levels. Figure 1 shows an example of the histogramming operation. Eight pixels { (Ci, Mi, Gi) }, $1 \leq i \leq 8$, are input in parallel to the histogramming unit. It is noted that initially all mark bits and count fields are set to 1. In the first step, count fields and mark bits are computed. In the example sequence there are only four distinct grey level values. As a result, only four pixels will be output with a mark bit set to 1. The second step is the process of removing all duplicate pixels. This is done by discarding all pixels output with a mark bit reset to 0. It is noted that the filtering process is computationally inexpensive compared to the count and mark process duration. In this paper, we will be mainly concerned with the design of a VLSI unit for the counting and marking process. In Figure 2, an 8-input odd-even network is used to count and mark the sequence of the earlier example. The algorithm to be implemented by each PE (Figure 4 gives the block diagram of a PE) is as follows:

```
Procedure COUNT and MARK [(Ci,Mi,Gi),(Cj,Mj,Gj)] Gj)]
    // Two pixels (Ci, Mi, Gi) and (Cj, Mj, Gj) are input
       to a processing element //
Begin
  If Gi > Gj then
     Begin out_1 ← (Ci,Mi,Gi),
                 out_2 ← (Cj,Mj,Gj),end;
  If Gi < Gj then
     begin out_1 ← (Cj,Mj,Gj);
                 out_2 ← (Ci,Mi,Gi),end;
  If Gi = Gj then
     begin
        If Mi ∧ Mj = 1 then[1]
```

[1] Throughout this paper ∧ refers to the logical AND.

CH2649-2/89/0000/0451$01.00 © 1988 IEEE

```
            begin $out_1 \leftarrow (Ci+Cj,Mi,Gi)$;
                              $out_2 \longleftarrow (-,0,Gj)$, end;

        If $Mi \wedge \overline{Mj} = 1$ then
            begin $out_1 \longleftarrow (Ci,Mi,Gi)$;
                              $out_2 \longleftarrow (-,Mj,Gj)$,end;
        else
            begin $out_1 \leftarrow (Cj,Mj,Gj)$
                              $out_2 \leftarrow (Ci,Mi,Gi)$, end;
    End
End //output marked and counted//
```

## PERFORMANCE ANALYSIS

We first investigate the processing time of our approach under the assumption that the number of pixels to be processed is no larger than the number of inputs to the histogramming unit. In the following analysis, the relevant parameters are:

k = number of bits in the grey level representation
p = number of bits in the count field
r = time (in seconds) to manipulate and pass one bit to the neighboring PE
n = number of pixels to be processed.

In our implementation, the processing of the pixels is completely overlapped with the inputting and outputting of the pixels to and from the histogramming unit. Since the longest path in an n-input histogramming unit is $\frac{\log n(\log n+1)}{2}$, processing n pixels will take

$$H_1 = [\frac{\log n(\log n+1)}{2} + (k+p+1)]r$$

In [5] and [6], our approach is compared with the implementation of the histogramming operation on the Massively Parallel Processor (MPP) [7]. It is shown that our approach manifests significant performance improvement over the MPP implementation. For instance, for $p=14$ bits, $k=8$ bits, and $n=16,384$ (128x128 image), our implementation is 16.11 times faster than the MPP approach. It is noted that unlike the MPP implementation our approach allows for the pipelined processing of different streams of grey levels. Suppose that I independent streams of grey levels are to be processed by the histogramming unit. The use of pipelining will reduce the processing time from $IH_1$ to $H_1+(I-1)(k+p+1)r$ (assuming that each stream is composed of n grey levels). For instance, when $I=10$, $n=16,384$, a speedup of 3.82 is achieved with the use of pipelining. The use of pipelining not only reduces the processing time but also allows for a more efficient use of the processing elements.

The histogramming algorithm we have described is internal. That is, the number of pixels to processed is assumed to be no larger than the number of inputs (n) of the histogramming unit. In general, an entire image cannot be processed internally by an odd-even based histogramming unit (because of area and pin count limitations). Figure 3 shows how to obtain a 2n-input histogramming unit from a number of n-input histogramming units. This method is very useful to increase the internal processing capability of a histogramming unit. However, since VLSI chips are of fixed size and the number of pixels is very large, the application of the above method is impaired by financial as well as technological constraints. When the number of pixels is too large to be processed internally by a histogramming unit, an external VLSI algorithm is the most practical solution. It is recalled that an external VLSI algorithm is one that allows a chip (or a set of chips) of fixed size to process an input set of any size [8]. One approach to this problem is based on an iterative use of a histogramming unit of fixed size. The proposed algorithm is based on successively merging histogrammed sets of pixels of increasingly larger size.

The external VLSI algorithm will use a histogramming unit of fixed size (n-inputs) in an iterative manner to process a set of pixels whose size[2] N is larger than n. The external algorithm is divided into two steps. During the first step, the histogramming unit is used to generate histogrammed sets of size n each. The duration of this step is

$$[\frac{\log n(\log n+1)}{2} + \frac{N}{n}(k+p+1)]r$$

In the second step, the histogramming unit is used as a two way merger. The second step requires $\log\frac{N}{n}$ phases. During the i-th phase, $1 \leq i \leq \log\frac{N}{n}$, $(1/2)^{i-1}\frac{N}{n}$ sets of $2^{i-1}n$ histogrammed pixels are converted to $(1/2)^{i}\frac{N}{n}$ sets of $2^{i}n$ pixels each. The merging duration is

$$[\frac{\log n(\log n+1)}{2} + (\sum_{i=1}^{\mathrm{Log}N/n} ((2^{i+1}-1)/2^{i})(k+p+1)]r$$

$$=[\frac{\log n(\log n+1)}{2} + (2\frac{N}{n}\log\frac{N}{n} - \frac{N}{n} + 1)(k+p+1)]r$$

When pipelining is used between the two steps , the overall duration of the external algorithm is

$$H_3 = [\frac{\log n(\log n+1)}{2} + (2\frac{N}{n}\log\frac{N}{n} + 1)(k+p+1)]r$$

## DESIGN AND VLSI IMPLEMENTATION OF THE HISTOGRAMMING UNIT

Figure 4 shows a block diagram of PE. The PE contains five flags (Fi,Fj,F1,F2,F3), a Control Unit (CU), a serial adder, and a Bit Manipulation Unit (BMU). It is recalled that pixels are input, processed, and output one bit at a time. First, the two grey levels $G_m=\{G_m^1\ G_m^2 \ldots G_m^k\}$, $m = i,j$, are input to

[2] Without loss of generality N is assumed to be a multiple of n.

the PE bit by bit starting from the MSB's ($G_i^k$ , $G_j^k$). Next, the mark bits Mi and Mj are input followed by the count fields Cm = { $C_m^p$ $C_m^{p-1}$ ... $C_m^1$ }, m = i,j. It is noted that unlike the grey levels, the count fields are input starting from the LSB's ($C_i^1$ , $C_j^1$ ). In order for the algorithm to work in proper synchrony, every bit must be passed to the right including leading zeros. Note that the grey level comparison, the manipulation of mark bits, and the updating of the count fields are completely overlapped with the inputting and outputting of the pixels to and from the PE's.

Two control signal $s_1$ and $s_2$ are used for control and synchronization. The control signal $s_1$, is a start signal, it is applied to all PE's of column M, $1 \leq M \leq \frac{\log n\ (\log n\ +\ 1)}{2}$, at time instant $t_{M-1}$. At time instant $t_M$, PE's of column M reset their five flags and begin the processing of the two pixels present at their inputs. The control signal $s_2$ is used to indicate the completion of grey levels comparison. This signal is applied to PE's of column M at time instant $t_{k+M}$. At the same time instant, the mark bits are input to the PE of column M. From time instant $t_{k+M+1}$ to $t_{k+M+p+1}$, the count fields are input to the PE's of column M.

The serial adder is used to compute the count sum (S = Ci + Cj). The BMU compute the value of $out_1$ and $out_2$ as a function of the result of the grey level comparison, the mark bit value, and the count sum. The flags Fi and Fj are used to store the mark bits Mi and Mj. The flag F3 is set by the control signal $s_2$. This flag when set indicates that grey level comparison has been completed. Finally, flags $F_1$ and $F_2$ are used to store the result of grey level comparison according to Table 1.

The layout of a PE is given in Figure 5. Each PE is composed of 255 transistors and occupies an area of about 0.2mm$^2$ (assuming a three microns technology ($\lambda$ = 1.5 $\mu$m)).

## SUMMARY

In this paper, an nMOS processing unit for the histogramming operation has been designed. The proposed unit is composed of a number of bit-serial structures connected according to the odd-even network topology. The use of a histogramming chip of fixed size to handle a large number of pixels has been considered.

## REFERENCES

[1] Mead, C., Conway, L., Introduction to VLSI Systems, Reading, Massachsetts: Addison-Wesley, 1980.

[2] Batcher, K.E., "Sorting Network and Their Applications," *AFIPS Proc. Spring Joint Comput. Conf.*, Vol. 32, April, 1968, pp. 307-314.

[3] Batcher, K.E., "Bit-Serial Parallel Processing Systems," *IEEE Trans. Comput.*, C-31, 1982, pp. 377-384.

[4] Hatamian, M., Cash, G.L., "Parallel Bit-Level Pipelined VLSI Designs for High-Speed Signal Processing," *Proc. of the IEEE*, Vol. 75, No. 9, September, 1987, pp. 1192-1202.

[5] Abdelguerfi, M., *et al.*, "Parallel Bit-Level Pipelined VLSI Processing Unit for the Histogramming Operation", *IEEE Computer Conference on Vision and Pattern Recognition*, University of Michigan, June 1988, pp.945-950.

[6] Abdelguerfi, M., *et al.*, "Parallel Bit-Level Pipelined Processing Unit for the Histogramming Operation", *IEEE Trans. on Circuit and Systems* (Submitted for Publication).

[7] Kushner, T., Wu, A.Y., Rosenfeld, A., "Image Processing on MPP:1," *Pattern Recognition*, Vol. 5, No. 3, 1982, pp.836-840.

[8] Bonucelli, M.A., *et al.*, "External Sorting in VLSI", *IEEE Trans. on Comp.*, Vol. C-33, N0. 10, oct. 84, pp. 931-934.

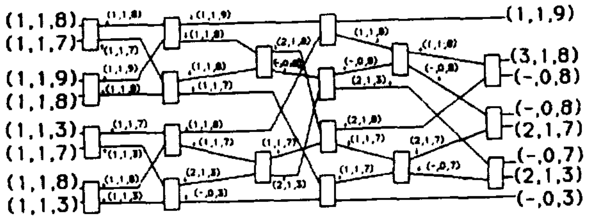


FIGURE 1. An Example

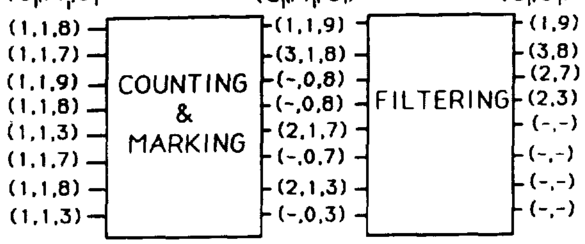


FIGURE 2. Parallel Implementation of the Histogramming Operation

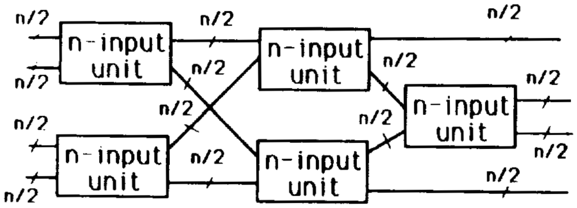


FIGURE 3. Design of a 2n-input Histogramming Unit

| $F_1$ | $F_2$ | CONDITION |
|---|---|---|
| 0 | 0 | $G_i = G_j$ |
| 0 | 1 | — |
| 1 | 0 | $G_i > G_j$ |
| 1 | 1 | $G_i < G_j$ |

TABLE 1. Flip Flop Control States

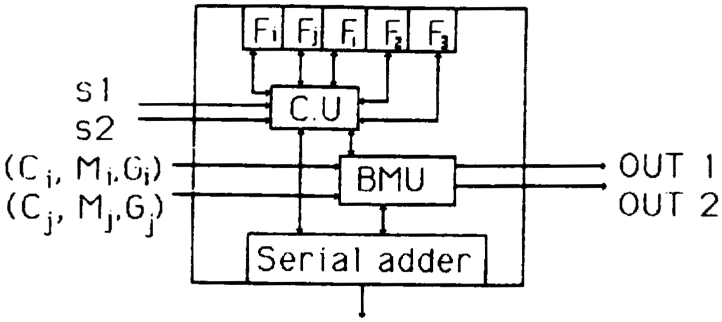


FIGURE 4. Block Diagram of a Processing Element

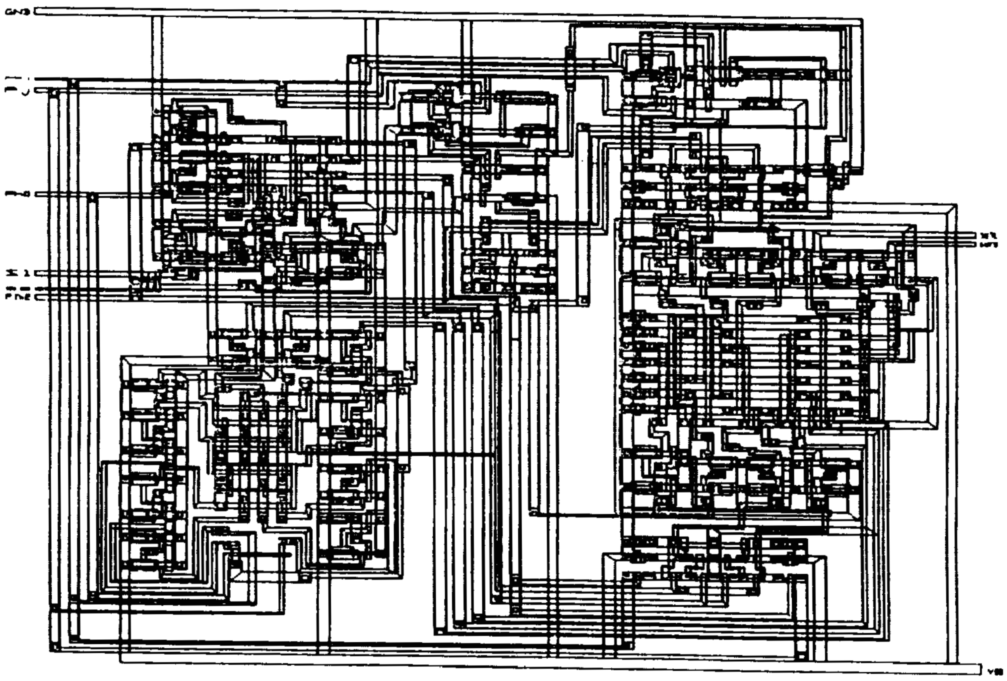


FIGURE 5. LAYOUT OF A PE

# THE FUNCTION OF A CONNECTION NETWORK BETWEEN HOST AND PROCESSING ELEMENTS IN MASSIVELY PARALLEL COMPUTER SYSTEMS

Timothy Bridges

Computer Science Department
Indiana University
Bloomington, IN 47405
tbridg@iuvax.cs.indiana.edu

Massively Parallel Arch. Corp.
P.O. Box 954
Bloomington, IN 47402
(812) 339-3753

## Abstract

Massively parallel computation systems are routinely characterized, identified, and studied based on the connection topology of the processing elements (PEs). This method of classification is flawed since massively parallel computation systems are not stand-alone computers; they are invariably packaged with a standard von Neumann host. This paper examines the function of a connection network between massive numbers of PEs and a single host by comparing the Data Structure Machine (DSM), whose major connection network is a computationally powerful binary tree with the host connected at the root, and the Connection Machine, which provides a very rich and general PE to PE connection network, but whose connection to the host is little more than a buffered wire. The binary tree network used in the Data Structure Machine can be utilized to achieve asymptotic improvements in speed for algorithms that maintain, locate and exploit data parallelism in data structures that can be characterized by a high degree of locality. For example, SUM, MAX, LEFTMOST, and INDEX are all constant time operations on DSM lists, while corresponding Connection Machine algorithms can require linear time to simply find and mark lists.

Keywords: SIMD, Massively Parallel, Connection Topology, Host Interface, Connection Machine, Data Structure Machine

## 1. Introduction

Of the two architectures being compared here, the Connection Machine is widely known, understood, and available as a commercial product. The Data Structure Machine is clearly less known, therefore requiring the following introduction to the parent architecture, APSA, and an overview of the DSM.

The Applicative Programming Systems Architecture, APSA, extends the traditional von Neumann design by including a special purpose functional unit that serves as a Data Structure Memory, DSM [4,6,7]. The DSM is a massively parallel computing system that utilizes thousands, or even millions, of processing elements in an SIMD paradigm to exploit data structure parallelism. A custom VLSI prototype containing 128 Processing Elements, PEs, has been constructed and is under study at Indiana University (16 chips with 8 PEs/chip). The simple tree-based connection topology used in this DSM allows for extending the system by adding more chips of the same design or by increasing the number of processors per chip. A single chip containing upwards of 128 processors is feasible by using the denser and more expensive technologies available today. Also, the total number of PEs in the system does not affect the specification of a individual PE nor does it increase the complexity of the connection network wiring. A project is underway to construct a DSM of an interesting size, 2K - 4K processing elements, and to implement bank switching so that larger machines can be emulated for study.

Because of its simple binary tree connection topology, the APSA DSM will benefit from economies of scale that could lead to implementations containing over a million processors in the near future. However, the use of such a simple connection topology raises serious questions concerning possible applications for such a machine. PE ↔ PE communication is obviously limited by the bottleneck at the root of the tree. What is less obvious, and what I attempt to show in this paper, is that the tree topology that is a bottleneck for PE ↔ PE communication provides an extremely rich Host ↔ PEs connection network. Further, this network can be used to efficiently implement algorithms of a *1 to many* or *many to 1* nature. I will present examples of these algorithms and compare their performance to Connection Machine [1], CM, algorithms. The CM was chosen for this comparison because of its rich PE ↔ PE communication capabilities and its relative lack of a Host ↔ PEs connection network. Figures 1 and 2 depict system level layouts of the Connection Machine and the Data Structure Machine architectures.

## 2. APSA Data Structure Memory Overview

The original APSA architecture was designed for efficient implementation of list processing operations by recognizing that most of these operations involve Host ↔ PE

CH2649-2/89/0000/0455$01.00 © 1988 IEEE

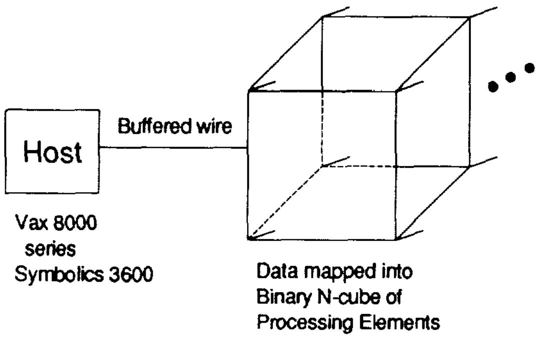


**Figure 1.** Connection Machine high level architecture

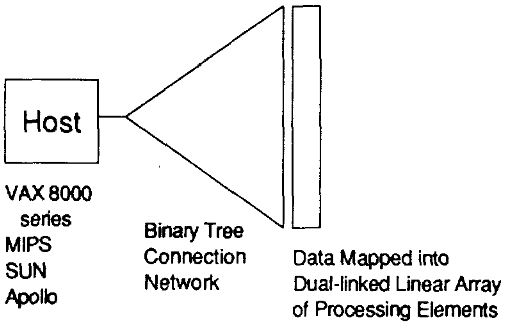


**Figure 2.** DSM high level architecture

communication and computation [4,5]. In this context, the major function of the APSA communication structure is to support efficient operators that 1) find, or mark, a list in the heap, 2) update marked lists, and 3) operate on marked lists as if they were compact linear data structures.

Two simple regular connection networks combine to support these three types of operations. First, a binary tree of processing elements connects a leaf level, formed by PE/ME pairs, to the Host processor connected at the root. The store of the machine is mapped into the leaf level of the tree. Only intermediate status information is stored in the non-leaf nodes of the tree. Using the accepted method of classifying massively parallel systems based on the connection topology of the PEs, the DSM would be classified as having a one dimensional array of processing elements. The binary tree network would be ignored.

A second network connects the leaf level horizontally across the breadth of the tree, allowing bidirectional shifting of data. This second, linear, connection topology implements limited PE ↔ PE communication, that when combined with the tree network can implement a total ordering on the data stored in the leaf level. In this context, a total ordering on the data items in the leaf level means that if a node, $N_i$, occurs before a node, $N_j$, in a traversal of the list, then the node $N_i$ is physically stored to the left of node $N_j$. The linear connections in the leaf level are used to effectively shift a leaf cell from the avail pool, in a single cycle, to a position that maintains the total ordering.

The process of shifting data cells can destroy the validity of explicit, address based, pointers in the heap, which could take significant time to update in a heterogeneous system. APSA solves this problem by using unique labels, instead of addresses, for explicit pointers. A pointer is dereferenced by having all cells perform a match function on the label. This content addressable pointer solution is efficient since updating of pointers is not necessary after a shift operation. A similar system can be implemented on the Connection Machine, but PE ↔ PE communications would then be based on a similar content addressable pointer system which would render most of the CM's routing circuitry useless since explicit pointers would be dereferenced by content-based addressing, while the CM router only works with absolute location-based addressing. Though implementable, such a system wastes most of the bandwidth available, and paid for, on the Connection Machine. In other words, an efficient emulation of the DSM can be implemented on the CM, but fails to exploit the scaling advantage of the tree topology and wastes much of the CM's resources. For the purposes of the comparison in this paper, the CM is assumed not to be running in a DSM emulation mode.

## 3. Timing guidelines for SIMD algorithms

It is necessary to clarify the guidelines used for timing comparisons of algorithms running on von Neumann architectures with those running on SIMD architectures. This clarification is needed because of the dual use of the time complexity of a memory access/update operation on RAM. Any computer scientist asked to analyze the time complexity of the FETCH or STORE instructions for RAM will quickly and easily return the answer $O(log(n))$. However, in analyzing algorithms running on these machines, analysts invariably consider the time complexity of these instructions to be $O(1)$. Historically, this shortcut was taken because all algorithms were run on von Neumann machines; by eliminating this extra *log* from all equations, the comparison between algorithms was clearer. This technique has become so widely accepted that many computer scientists forget that RAM instructions are not constant in time. Several critics of the tree topology have wrongly pointed out that the tree must pay a *log* cost for sweeps up or down the tree. The router in the Connection Machine has probably received similar baseless criticism. In both of these cases the logarithmic delay is analogous to the logarithmic delay for address decoding in RAM. A fair comparison will treat time through the CM router and up or down the APSA tree as constant time operations, asymptotically equivalent to FETCH and STORE times.

## 4. Examples

In the limited space this short paper provides I will present an analysis of two sample algorithms that realize a performance gain by utilizing the Host ↔ PEs network of the APSA DSM. The first example shows an algorithm to sum the elements of an array. The CM algorithm has appeared [2], and works by building a tree, using pointers, within the array to calculate and store intermediate results. This example illustrates the use of the APSA DSM tree as a purely computational network. The second example computes the sum of the contents of a linked list structure and illustrates the value of being able to utilize a total ordering to support implicit pointers. For these examples let $n$ be the number of elements in the array or list, and $e$ be the number of explicit pointers in the CM list. Note that $e$ would be 1 for an APSA DSM list.

**Example 1** – Summing elements of an array

| | **Time Complexity** | |
|---|---|---|
| **Operation** | **CM** | **DSM** |
| 1. Mark array in PEs | 1 | 1 |
| 2. Build pointer tree | $log(n)$ | — |
| 3. SUM by pointers | $log(n)$ | — |
| 4. SUM by network | — | 1 |
| Total Algorithm | $log(n)$ | 1 |

**Example 2** – Summing elements of a list

| | **Time Complexity** | |
|---|---|---|
| **Operation** | **CM** | **DSM** |
| 1. Mark list in PEs | $e$ | 1 |
| 2. Build pointer tree | $e + log(n)$ | — |
| 3. SUM by pointers | $log(n)$ | — |
| 4. SUM by network | — | 1 |
| Total Algorithm | $e + log(n)$ | 1 |

The worst case time for the CM algorithm in Example 2 is $O(n)$ for a severely fragmented list. Also the programmer may choose, for other efficiency reasons, not to implement all APSA DSM pointers implicitly. It is possible to maintain a balance between the number of explicit pointers and sharing between data structures. This can be accomplished under programmer control at runtime. These examples also assume that the CM is not running as a virtual APSA DSM through emulation routines. As mentioned previously, such an emulation could not use the CM's address based routing circuitry for general PE ↔ PE communication.

The decrease in the performance of the CM algorithm for linked lists compared to arrays is the result of the CM's inability to maintain locality within dynamic data structures. An array can be thought of as a list with predefined length and all pointers represented implicitly. The CM can store and efficiently operate on this structure, however, because the CM utilizes absolute location-based addressing, and therefore cannot efficiently shift large blocks of data, it must represent updates to lists using explicit pointers. Dereferencing explicit pointers serializes data parallel operations and accounts for the decrease in performance.

## 5. Other Architectures

The DSM unit of the APSA machine is the only architecture described in the literature that devotes significant power and circuitry to a structure that is not part of the memory map. Other tree-based topologies have been described, yet in each case long-term storage is allowed at all levels. At first glance, this architecture may seem similar to the FFPM [3] since it is a tree based topology that uses a different design for processors for the leaf level. However, there are two major distinctions. First, the granularity of the FFPM is much bigger than that of the DSM; FFPM processors are approximately two orders of magnitude bigger than DSM processing elements. Second, the tree network is not used as an interface to a host. In fact the FFPM is an MIMD machine that could potentially operate without a traditional von Neumann host.

## 6. Conclusions

It is clear that characterizing a massively parallel processor solely on the connection topology between processing elements is not sufficient. In this paper the linear connection topology of the Data Structure Machine's processing elements is able to outperform the binary n-cube topology of the Connection Machine processing elements by utilizing an additional connection network to the host. In this case a binary tree network is able to maintain and exploit locality properties within data structures stored in the array of processing elements at the leaf level.

Just as von Neumann machines may be optimal for *1 to 1* or *few to few* operations, and general communications networks in massively parallel computer systems, such as the Connection Machine, may provide great speedups for *many to many* and *many ( 1 to 1)* operations, this work shows that a limited topology, such as a tree, can yield improvements for classes of *1 to many* and *many to 1* operations. Algorithms that operate on data structures that maintain a locality property are good candidates for inclusion in this class, and can find an asymptotic speedup on the DSM. We have recently extended this class to include nested relational databases, circuit simulation, and low-level image analysis operations. Further, the scalability of the machine can lead to an extremely large PE per chip ratio. This development would allow a moderate number of processors in a very small space or could be used to build a machine with an enormous number of processors.

### Acknowledgements

I would like to thank John T. O'Donnell who developed the architecture and the programming paradigm for the Data Structure Machine, and who shared them with me as my research advisor.

## References

1. W. Daniel Hillis, *The Connection Machine*, Cambridge, Mass.: The MIT Press, 1985.
2. W. Daniel Hillis and Guy L. Steele, Jr., *Data Parallel Algorithms*, Commun. ACM, vol. 29, no. 12, December 1986, pp. 1170–1183.
3. Gyula Magò and David Middleton, "The FFP Machine—a progress report", *Proc. International Workshop on High-Level Language Computer Architecture*, May 1984, pp. 5.13–5.25.
4. John T. O'Donnell, *A Systolic Associative LISP Computer Architecture with Incremental Parallel Storage Management*, Technical Report 81-5, Computer Science Department, University of Iowa, Iowa City, 1981.
5. John T. O'Donnell, "An Architecture That Efficiently Updates Associative Aggregates in Applicative Programming Languages," *1985 IFIP Symposium on Functional Programming Languages and Computer Architecture*, Nancy, France, September 1985, *Lecture Notes in Computer Science 201*, pp. 164–189, New York: Springer-Verlag, 1986.
6. John T. O'Donnell, "Parallel VLSI Architecture Emulation and the Organization of APSA/MPP," *Proceedings of the First Symposium on the Frontiers of Massively Parallel Scientific Computation*, NASA Goddard Space Flight Center, Sept. 1986.
7. John T. O'Donnell, Timothy Bridges, and Sidney W. Kitchel,"A VLSI Implementation of an Architecture for Applicative Programming," *Future Generations Computing Systems*, To Appear.

# DENSE SYMMETRIC NETWORKS FROM LINEAR GROUPS

L. Campbell and M. Fellows
Computer Science Dept., Univ of Idaho
G. Carlsson
Mathematics Dept., Princeton Univ.
V. Faber and J. Moore
Los Alamos National Laboratories
M. Langston
Computer Science Dept., Washington State Univ.
A. Mullhaupt
Mathematics Dept., Univ. of New Mexico
H. Sexton
Lucid, Inc., Menlo Park, California

## ABSTRACT

An algebraic approach to the problem of constructing large networks of bounded degree and diameter is described. Subgroups of *GL[2,n]* are employed to provide a number of record-breaking constructions in the range of potential engineering significance for massive networks. These constructions, all highly symmetric, can be viewed as belonging to a family of constructions based on vector spaces and their automorphism groups that includes hypercubes and cube-connected cycles as special cases.

Keywords: Cayley Graph, Symmetric Network, Hypercube, Parallel Processing, Group.

## INTRODUCTION

The problem of constructing large graphs of a given degree and diameter (called *dense* graphs) has received much attention, and is significant for parallel processing because it models two important constraints in the design of massively parallel processing systems: (1) There are limits on the number of processors to which any processor in the network can be directly connected, and (2) The distance between any two processors in the network should not be too great. Other applications of dense networks include shared-key cryptographic protocols and the design of local area networks. See [3.,6.] for recent surveys.

In this paper we provide evidence that the table of largest known constructions for small values of the two parameters can be improved "almost everywhere" by methods based on finite groups. In many entries the constructions we report are dramatically larger than the best previously known and many of these improvements are in the range of the numbers of processors currently being considered for large parallel processing systems. See Table 1. Our main contribution is the demonstration of the power of an algebraic approach to this problem. For related work see [2.,4.,9.].

Entries improved or matched by our constructions are shown in bold. Most have been obtained by a small amount of searching by simple programs running on small computers (such as an IBM PC). The success of the limited search we have so far conducted seems to indicate that further computational exploration may improve many more entries.

D

| | | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| | 3 | 10 | 20 | 38 | 70 | 128 |
| | 4 | 15 | 40 | 95 | 364 | 731 |
| | 5 | 24 | 70 | 174 | 532 | 2734 |
| Δ | 6 | 32 | 105 | **336** | **1008** | 7817 |
| | 7 | 50 | 122 | **480** | **2016** | 10546 |
| | 8 | 57 | 200 | 807 | **2880** | 39223 |
| | 9 | 74 | 585 | 1248 | **6072** | 74906 |
| | 10 | 91 | 650 | 1755 | **12144** | 132869 |

D

| | | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|
| | 3 | 184 | 320 | 480 | 728 |
| | 4 | 1081 | 2943 | 7439 | 15657 |
| | 5 | **4368** | **8736** | **25308** | **123120** |
| Δ | 6 | **13104** | **50616** | **202464** | **682080** |
| | 7 | **39732** | **101232** | **911088** | **1822176** |
| | 8 | **89280** | **455544** | **1822176** | 3984120 |
| | 9 | 215688 | **682080** | 3019632 | 15686400 |
| | 10 | 486837 | **1822176** | 7714494 | 47059200 |

Table 1

CH2649-2/89/0000/0459$01.00 © 1988 IEEE

## ALGEBRAIC SYMMETRY AS AN ORGANIZING PRINCIPLE FOR PARALLEL PROCESSING.

There are important considerations apart from degree and diameter that must figure in any choice of network topology for parallel computation. Our approach yields *symmetric* constructions, and we believe that in this lies their greater value. Symmetry is one of the most powerful and natural tools to apply to the central problem of massively parallel computation: how to *organize* and *coordinate* computational resources.

The symmetries of the networks we describe are represented by simple algebraic operations (such as 2 by 2 matrix multiplications and modulo arithmetic). The main advantage of algebraic networks is that the developed mathematical resources of algebra are available to structure the problems of testing, data exchange, message routing, scheduling and the mapping of computations onto the network. The appeal of hypercubes, cube-connected-cycles, butterfly networks and others rests in large part on this same availability of easily computed (and comprehended) symmetries. These popular networks and those that we describe all belong to a class of algebraic networks based on vector spaces and their symmetry groups. For recent algebraic approaches to routing algorithms, deadlock avoidance, emulation and scheduling for algebraically described networks of this sort see [1.,2.,7.,8.].

The next section describes our basic approach and some examples of our constructions.

## TECHNIQUE AND EXAMPLE CONSTRUCTIONS

A network is (vertex-) *symmetric* if for any two nodes u,v there is an automorphism of the network mapping u to v. Every Cayley network is symmetric (symmetries are given by group multiplication). If $A$ is a group and $S \subseteq A$ is a generating set that is closed under inverses, i.e., $S = S \cup S^{-1}$, then the (undirected) *Cayley graph* $(A, S)$ is the graph with vertex set $A$ and with an edge between elements $a$ and $b$ of $A$ if and only if $as = b$ for some $s \epsilon S$. It is remarkable (but, indeed, natural) that most networks that have been considered for large parallel processing systems (including hypercubes, grids, cube-connected-cycles and butterfly networks) are Cayley graphs. The degree of a Cayley graph $(A, S)$ is $\Delta = |S|$ and the diameter of $(A, S)$ is $D = max_{a \epsilon A}\{min_t : a = s_1 \cdots s_t, s_i \epsilon S \text{ for } i = 1, \ldots, t\}$.

**Example 1** Degree 5, diameter 7 : 4368 vertices. (Best previous : 2988.)

This is a Cayley graph on the subgroup of *GL[2,13]* consisting of the matrices with determinant in the set {1,-1}. The generators are the following elements together with their inverses.

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} order\ 2 \quad \begin{bmatrix} 11 & 2 \\ 8 & 12 \end{bmatrix} order\ 52 \quad \begin{bmatrix} 11 & 4 \\ 7 & 5 \end{bmatrix} order\ 14$$

**Example 2** Degree 6, diameter 10 : 682,080 vertices. (Best previous : 199,290.)

This is a Cayley graph on the group *GL[2,29]*. The generators are the following elements together with their inverses.

$$\begin{bmatrix} 28 & 10 \\ 8 & 8 \end{bmatrix} order\ 28 \quad \begin{bmatrix} 17 & 13 \\ 16 & 27 \end{bmatrix} order\ 28 \quad \begin{bmatrix} 3 & 4 \\ 27 & 14 \end{bmatrix} order\ 840$$

**Example 3** Degree 10, diameter 5 : 12144 vertices.

(Best previous : 10,000.)

This is a Cayley graph on the group *SL[2,23]*. The generators are the following elements together with their inverses.

$$\begin{bmatrix} 9 & 0 \\ 18 & 18 \end{bmatrix} order\ 11 \quad \begin{bmatrix} 13 & 10 \\ 18 & 21 \end{bmatrix} order\ 11 \quad \begin{bmatrix} 9 & 10 \\ 0 & 17 \end{bmatrix} order\ 22$$

$$\begin{bmatrix} 14 & 7 \\ 19 & 3 \end{bmatrix} order\ 22 \quad \begin{bmatrix} 18 & 13 \\ 17 & 20 \end{bmatrix} order\ 24$$

| Parameters | Order | Group | Generators: order<br>$S = S \cup S^{-1}$ |
|---|---|---|---|
| degree 5<br>diameter 7 | 4368 | index 6 in<br>$GL[2,13]$<br>$(det = r^6)$ | [0,1,1,0]:2<br>[11,2,8,12]:52<br>[11,4,7,5]:14 |
| degree 5<br>diameter 8 | 8736 | index 3 in<br>$GL[2,13]$<br>$(det = r^3)$ | [8,8,10,5]:2<br>[0,3,9,10]:52<br>[1,12,5,3]:4 |
| degree 5<br>diameter 9 | 25308 | index 2 in<br>$SL[2,37]$<br>$(/\{\pm 1\})$ | [0,36,1,0]:2<br>[34,26,34,1]:37<br>[2,16,11,33]:37 |
| degree 5<br>diameter 10 | 123120 | $GL[2,19]$ | [0,1,1,0]:2<br>[16,11,2,0]:45<br>[11,16,0,15]:18 |
| degree 6<br>diameter 4 | 336 | $SL[2,7]$ | [1,6,6,2]:8<br>[0,3,2,5]:14<br>[2,3,1,2]:8 |
| degree 6<br>diameter 5 | 1008 | index 2 in<br>$GL[2,7]$<br>$(det = r^2)$ | [5,0,2,5]:42<br>[0,3,4,2]:24<br>[3,1,5,2]:14 |
| degree 6<br>diameter 7 | 13104 | index 2 in<br>$GL[2,13]$<br>$(det = r^2)$ | [10,12,10,9]:39<br>[12,3,9,8]:84<br>[8,11,5,0]:84 |
| degree 6<br>diameter 8 | 50616 | $SL[2,37]$ | [32,24,35,2]:19<br>[12,24,15,27]:37<br>[23,16,28,34]:36 |
| degree 6<br>diameter 9 | 202464 | index 9 in<br>$GL[2,37]$<br>$(det = r^9)$ | [25,1,31,1]:36<br>[12,35,23,30]:76<br>[12,4,28,16]:152 |

| Parameters | Order | Group | Generators:order<br>$S = S \cup S^{-1}$ |
|---|---|---|---|
| degree 6<br>diameter 10 | 682080 | $GL[2,29]$ | [28,10,8,8]:28<br>[17,13,16,27]:28<br>[3,4,27,14]:840 |
| degree 7<br>diameter 4 | 480 | $GL[2,5]$ | [0,1,1,0]:2<br>[4,3,2,0]:20<br>[3,3,3,4]:24<br>[0,3,1,1]:24 |
| degree 7<br>diameter 5 | 2016 | $GL[2,7]$ | [0,1,1,0]:2<br>[6,1,6,4]:42<br>[5,1,1,4]:48<br>[2,2,5,0]:6 |
| degree 7<br>diameter 7 | 39732 | index 2 in<br>$SL[2,43]$<br>$(/\{\pm 1\})$ | [0,42,1,0]:2<br>[18,16,38,41]:22<br>[8,28,14,33]:43<br>[34,2,37,6]:22 |
| degree 7<br>diameter 8 | 101232 | index 18 in<br>$GL[2,37]$ | [0,1,1,0]:2<br>[21,1,4,2]:9<br>[27,26,4,8]:74<br>[21,34,17,17]:6 |
| degree 7<br>diameter 9 | 911088 | index 2 in<br>$GL[2,37]$<br>$(det = r^2)$ | [0,1,1,0]:2<br>[27,33,19,22]:684<br>[25,16,13,6]:36<br>[23,17,14,26]:18 |
| degree 7<br>diameter 10 | 1822176 | $GL[2,37]$ | [0,1,1,0]:2<br>[35,28,34,12]:456<br>[1,19,14,16]:17<br>[36,1,12,0]:18 |
| degree 8<br>diameter 5 | 2880 | $GL[2,10]$ | [8,3,7,1]:24<br>[9,1,3,4]:24<br>[8,7,7 1]:60<br>[9,4,5,9]:10 |
| degree 8<br>diameter 7 | 89280 | index 10 in<br>$GL[2,31]$<br>$(det = r^{10})$ | [3,1,5,10]:30<br>[6,17,27,5]:30<br>[9,10,22,28]:15<br>[3,29,25,2]:6 |
| degree 8<br>diameter 8 | 455544 | index 4 in<br>$GL[2,37]$<br>$(det = r^4)$ | [28,32,33,33]:171<br>[9,34,25,16]:342<br>[21,9,17,5]:57<br>[0,26,3,1]:171 |
| degree 8<br>diameter 9 | 1822176 | $GL[2,37]$ | [12,13,34,33]:18<br>[35,3,19,35]:684<br>[26,10,36,31]:1368<br>[36,6,20,10]:36 |
| degree 9<br>diameter 8 | 682080 | $GL[2,29]$ | [0,1,1,0]:2<br>[5,22,18,26]:14<br>[17,15,21,4]:840<br>[2,5,10,21]:840<br>[23,12,11,21]:840 |
| degree 10<br>diameter 5 | 12144 | $SL[2,23]$ | [9,0,18,18]:11<br>[13,10,18,21]:11<br>[9,10,0,17]:22<br>[14,7,19,3]:22<br>[18,13,17,20]:24 |

| Parameters | Order | Group | Generators:order<br>$S = S \cup S^{-1}$ |
|---|---|---|---|
| degree 10<br>diameter 8 | 1822176 | index 18 in<br>$GL[2,37]$<br>$(det = r^{18})$ | [35,10,17,32]:684<br>[5,31,35,14]:684<br>[11,3,33,7]:1368<br>[9,12,6,26]:456<br>[21,12,22,5]:57 |
| degree 12<br>diameter 5 | 24360 | $SL[2,29]$ | [1,18,3,26]:58<br>[17,20,14,8]:30<br>[15,6,20,10]:30<br>[23,28,28,19]:28<br>[26,14,11,16]:28<br>[16,7,28,25]:28 |

## References

[1.] S.B. Akers and B. Krishnamurthy, "On Group Graphs and Their Fault-Tolerance," *IEEE Trans. on Computers,* 36(1987) 885–888.

[2.] F. Annexstein, M. Baumslag and A.L. Rosenberg, "Group Action Graphs and Parallel Architectures," COINS Technical Report 87-133, Univ. of Mass., Amherst, 1987.

[3.] J.C. Bermond, C. Delorme, and J.J. Quisquater, "Strategies for Interconnection Networks: Some Methods from Graph Theory", *Journal of Parallel and Distributed Computing* 3, pp. 433–449 (1986).

[4.] J. Bond, C. Delorme and W.F. de La Vega, "Large Cayley Graphs with Small Degree and Diameter," Rapport de Recherche no. 392, Dec. 1987, Orsay.

[5.] G. Carlsson, J. Cruthirds, H. Sexton and C. Wright, "Interconnection Networks Based on a Generalization of Cube-Connected Cycles," *IEEE Trans. Comp.,* C-34(1985), 769–772.

[6.] D.V. Chudnovsky, G.V. Chudnovsky, and M.M. Denneau, "Regular Graphs with Small Diameter as Models for Interconnection Networks," manuscript.

[7.] F.R.K. Chung, "Diameters of Graphs: Old Problems and New Results", Bell Communications Research, Morristown, NJ 07960.

[8.] V. Faber, "Global Communication Algorithms for Hypercubes and Other Cayley Coset Graphs," technical report, Los Alamos National Laboratories, 1988.

[9.] M. Fellows, "A Category of Graphs for Parallel Processing," technical report, University of Idaho, 1988.

# A MASSIVELY PARALLEL PROCESSING SYSTEM BASED ON A HYPER-CROSSBAR NETWORK

C. Chin and W. Lin
General Electric Company / CRD
Corporate Research and Development
Schenectady, NY 12301

## ABSTRACT

Based on the advanced technologies, VLSI and HDI (High Density Interconnection), a parallel processing system consisting of 1024 processors is proposed. A special feature of this system is the reconfigurability of data communication channels between processors, achieved by using a hyper-crossbar interconnection network which facilitates a multiprocessor system to operate as a SIMD, MIMD, MSIMD, ... etc. Each processor possesses two communication channels, separately connected to a local crossbar network and to a global crossbar network (which are sub-networks of the hyper-crossbar network) for local communication and global communication, respectively. Processors connected to the same local network form a processor cluster for the execution of systolic-array-type algorithms. Primarily implemented by LINC chips, the global networks are able to programmably hold or delay operation data to synchronize the data flow for generic applications. With the operation speed of 20 MHz, the system can reach a peak performance of 40 billion operations per second.

Keywords: Computer Architecture, Parallel Processing, Crossbar Network, VLSI, HDI, Reconfigurability.

## INTRODUCTION

Due to the computation demands of the modern applications, e.g. image processing, system simulation, real-time graphics display etc., the computation power required has reached billion operations per second or even higher. Given a general VLSI/CMOS operating clock speed, i.e. 10-40 MHz, it becomes necessary to develop a system with over thousand processors to achieve the required performance. Due to the communication demands from thousands of processors, one of the major bottle-necks existing in any multiprocessor system, interconnection communication networks for such systems have been recently focused and heavily studied (Ref. 1-2).

The most flexible and simplest solution is to have every processor connected to a global crossbar network which provides essentially a non-blocking communication link. Because of the technology limitation, this type of network is not feasible for a large scale system containing over thousand processors. In this paper, an alternative approach, i.e. a hyper-crossbar network based parallel processing system, is proposed. Because of the hyper-crossbar network, the proposed general-purpose multiprocessor system can be configured to operate as a SIMD, MIMD, MSIMD, ... etc. Furthermore, fault-tolerance capability can be enhanced by this approach.

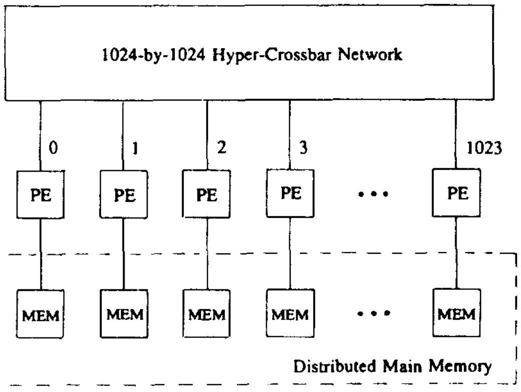


Figure 1. The system architecture

## SYSTEM ARCHITECTURE

The proposed parallel processing system has 1024 processors, a hyper-crossbar network, and a distributed main memory, as shown in Fig. 1. Processors are addressed by 10 bit binary codes $d_9d_8 \cdots d_1d_0$. Each processor consists of a node-level crossbar network, a processor controller, a multiplier, an ALU, a shifter, and a register file, as shown in Fig. 2. In the system, there are 1024 local memories which are individually attached to 1024 processors. The collection of all local memories forms the distributed main memory of the system. Through the node-level crossbar network, operands for the three operators, i.e. multiplier, ALU, and shifter, are provided by four resources: register file (two channels), operation results from previous clock cycle, local memory, and external (through the hyper-crossbar network). The operation results can be either temporarily saved in the register file, transmitted to other processors, or stored in the local memory.

Selectors in a processor provide an option for a half-word operation, while registers hold operands for a certain period. The controller which controls all operations in the processor receives global commands from the external and selectively

CH2649-2/89/0000/0463$01.00 © 1988 IEEE

executes commands pre-stored in the local memory. The local memory is partitioned into two banks to facilitate the memory sharing and updating process. While one bank is being used to serve for the processor control, the other can be updated or read by other processors through the node-level crossbar network at the same time. There are no duplicated copies for information stored in the main memory to avoid memory incoherence problems (Ref. 2).

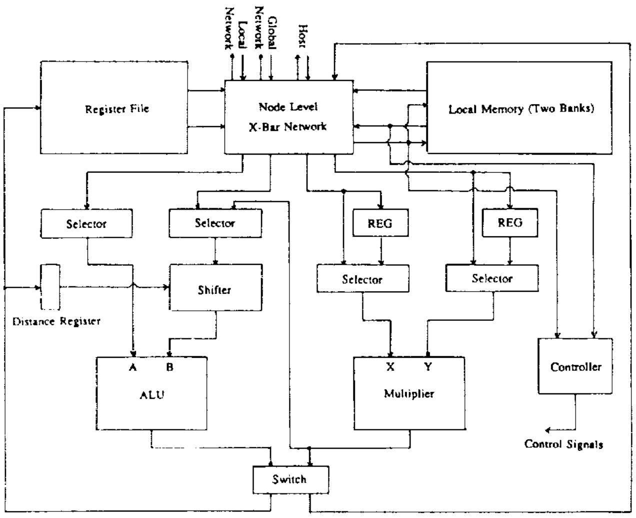


Figure 2. The architecture of a node processor

There is a hyper-crossbar interconnection network connecting all processors together. Instead of using a tightly-coupled connection, the hyper-crossbar network is an accumulation of many individual crossbar networks. The networks are partitioned into two groups, i.e. local network and global network. Processors with the same four leading address code $d_9d_8d_7d_6$ are connected to a 64-by-64 local crossbar network. Those processors which are connected to the same local crossbar network form a processor cluster. Processors with the same six trailing address code $d_5d_4d_3d_2d_1d_0$ are also connected together through a global 16-by-16 network which consists of 4 sets of LINC chips (Ref. 3) as shown in Fig. 3. Each set consists of four/eight LINCs operating in parallel in order to provide 16-bit/32-bit wide communication channels. All the 16 local crossbar networks and 64 global networks are controlled by a host. The host determines the connection patterns, depending on the data flow specified in parallel computation algorithms.

The architecture of LINC chip is shown in Fig. 4. The chip has eight 4-bit data-paths consisting of an 8-by-8 crossbar network, either a FIFO or a programmable delay register for each of its inputs, and a pipeline register file for each of its outputs. The connection pattern between sources and destinations (including broadcasting) is determined by the control pattern register which is updated by the preloaded control pattern memory. Wider data channels can be achieved by combining more LINC chips together, while using the same control signals. With the reconfigurability and programmability, provided by LINC chips, data flow through a global network can be synchronized and redistributed.

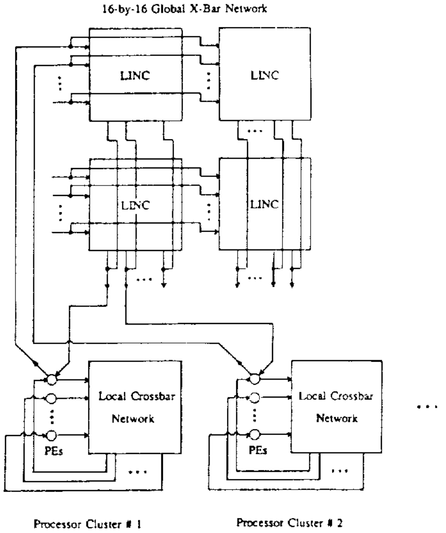


Figure 3. The hyper-crossbar network architecture

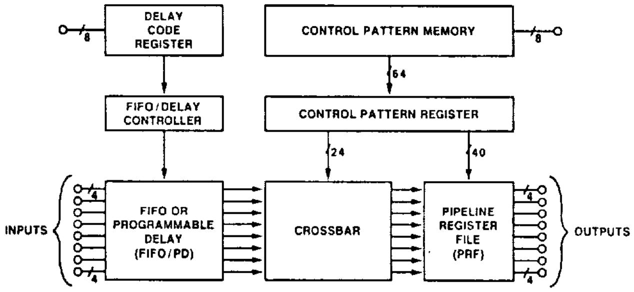


Figure 4. The LINC chip block diagram

## IMPLEMENTATION CONSIDERATION

The area complexity of a node processor has been studied. Using a 1.2 um CMOS technology, a node processor (Local memory is not included.) occupies an area of 340-by-340 mil$^2$ (Ref. 4). In other words, a 4-inch wafer can accommodate more than 80 node processors. Thus, it is sufficient to include one entire processor cluster on a single wafer. The associated 64-by-64 local crossbar network may also be incorporated on the same wafer to interconnect the node processors as shown in Fig. 5. By using the strategy of device redundancy, it is reasonable to assume that the network is fault free.

The area required by a crossbar network is due to switching circuits and data wiring. Although the area complexity of the crossbar switching circuits is known to be O(N$^2$), where N is the number of network terminals, the actual area required for the circuits is almost negligible (only N transistors per channel as shown in Fig. 5). This is particularly true when the switching circuits are compared with 100 thousand transistors in a node processor. It is estimated that the wire routing area required for the proposed crossbar network is 2W·B·N·S, where W, B, N, and S are the pitch width, the channel bits, the network terminals, and the processor perimeter length, respectively. For the case of W = 4 um, B = 16 bits, N = 64, and S = 4 mm, the routing area is about 16 $mm^2$, which is much less than the 64-processor area, i.e. $64 \times 4^2$ $mm^2$. It can be also shown that the multistage interconnection network requires 2W·B·S·N·((logN)−1) routing area on a silicon. It is larger than the area of a crossbar network by a factor of log N.

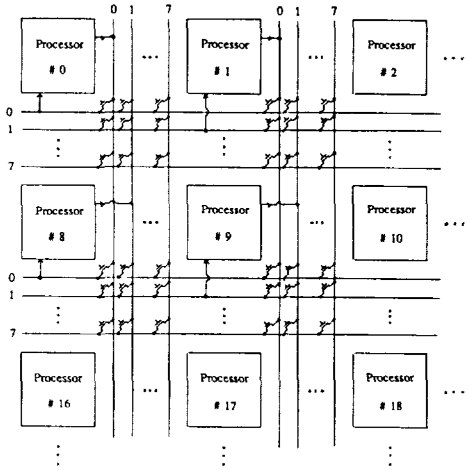


Figure 5. An implementation of a distributed local crossbar network

Other implementation alternative is to use the high density interconnection (HDI) packaging technology (Ref. 5). HDI connects signal I/O pins between bare chips (unpackaged chips) with copper wires by using laser-patterned polymer layer overlays laminated the chips mounted on a silicon substrate. By using VLSI technology in conjunction with HDI, a very reliable processor cluster embedded in a distributed 64-by-64 crossbar network (as shown in Fig. 5) can be easily implemented in a package without any wire routing problem. Similarly a global network could also be realized as a network device. In this way, the system volume is reduced, and the system structure is modulized to facilitate the system expansion.

## SYSTEM APPLICATIONS

In general, the system is designed as a general purpose machine. The data links between processors can be configured into almost any possible pattern for computation-intensive applications.

### Image Processing

By programming the connection pattern in the hyper-crossbar network, the proposed system can be configured as a array-type multi-processor system to process matrix operation or pixel computation for image processing applications. In Fig. 6, an image processing algorithm is partitioned and distributed into different processor clusters (MSIMD). Clusters are pipelined together through global networks to simplify instruction flow and to enhance data throughput. The local network in a cluster can be programmed not only to emulate a mesh-connected network for neighborhood communication, but also to provide direct communication channels for global interactions within a cluster. Intermediate results, generated by each pipeline segment (i.e. processor cluster), can be re-directed, broadcasted, or delayed to maintain a synchronized operand flow by the global networks.

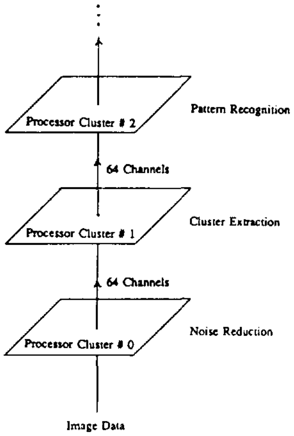


Figure 6. A pipeline configuration for image processing

### Parallelized Looping

Besides the mesh connected pattern, the processors can be configured as a multi-channel pipeline to compute multi-level loop instructions in parallel as shown in Fig. 7. The local networks provide communication links to pipeline the inner loop instructions, while the global networks support

```
For j = 1 to J Do
    For i = 1 to I Do
        I1: f1
        I2: f2 (I1(i), I2(i-1))
        I3: f3 (I2(i,j-1), I2(i))
        I4: f4 (I4(i,j-1), I3(i))
        End
    End
```

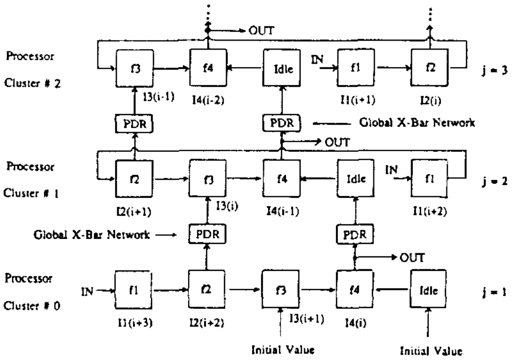


Figure 7. A multi-channel pipeline to execute multi-level loop instructions

data communication between different loops. Because of the programmable delay registers (PDR) and the pipeline register files equipped in LINC, which is the fundamental building block of the global networks, data synchronization and dependency problems can now be easily resolved. Other interconnection patterns or any combination of different patterns may also be established to execute MIMD, MSIMD, or other generic operations.

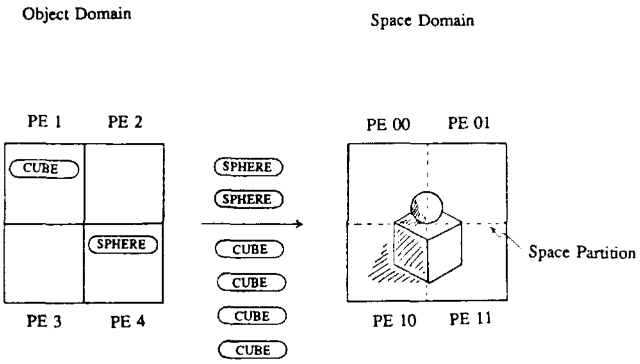


Figure 8. Image object redistribution and 3-D image generation

### 3-D Graphics Display System

An advanced real-time 3-D graphics display system has to manipulate numerous 3-D image objects in a display window. It usually requires a computation capability in the order of 10 billion instructions per second. Using parallel processing, as provided by the proposed system, is the only solution to enhance computation strength for satisfying the requirement. A special space partition algorithm (Ref. 6) to efficiently use multiprocessor elements for real-time graphics display is developed and shown in Fig. 8. The algorithm requires a flexible communication network to support its adjustable space partition and processor assignment, described in Fig. 9. Given the hyper-crossbar network of the proposed system, a flexible environment can be established to partition image space in various topologies which are required by high performance 3-D graphics display applications such as pilot training and mission rehearsal.

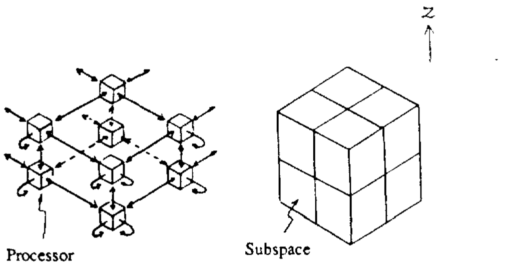


Figure 9. A 3-D cube architecture and the associated image space partition

## PERFORMANCE EVALUATION

The performance of the proposed system is evaluated in two ways, computation capability and communication capability. Operating at the speed of 20 MHz, each processor can perform 40 million operations per second. In total, the system can provide the peak performance of 40 billion operations in every second. It satisfies the speed requirement for most modern applications. The network capacity is determined by the number of channels, the channel width, and the network speed. Given a regular CMOS operating speed, i.e. 20 MHz, a 64-by-64 local crossbar network with 32 bits wide is able to transmit 40 billion bits in a second. In other words, each processor can transmit or receive over 320 million bits per second. Similarly, it can be easily estimated that a global network provides each processor with the same communication capability.

## CONCLUSION REMARKS

Operating at the speed of 20 MHz, the system can reach the peak performance of 40 billion operations per second. Because of the reconfigurability provided by the hyper-crossbar network, the average performance can be easily optimized to approach this peak performance. Systems with more processors can be achieved by either expanding the network size or introducing higher levels of crossbar networks. Furthermore, failed processors can be dynamically by-passed, without interrupting the system service.

## REFERENCES

1. T.Y. Feng, "A survey of interconnection networks," *IEEE Computer*, December 1981, pp. 12-30.

2. K. Hwang and F.A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill Book Co., New York, March 1984.

3. C.Y. Chin, W.T. Lin, J.P. Hwang, S. Chu, G. Forman, R. Dunki-Jacobs, S. Karr, J. Mallick, H.T. Kung, A. Sussman, F.H. Hsu, and T. Nishizawa, "A dynamically reconfigurable interconnection chip," *1987 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, Vol. XXX, Feb. 25-27, 1987, pp. 276-277.

4. C.Y. Chin, G. Buchner, T. Chang, M.J. Hartman, C.Y. Ho, J.R. Jasica, D.E. Orton, and W.D. Smith, "10 MHz IC's for graphics processing designed on a silicon compiler," *1988 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, Vol. XXXI, Feb. 17-19, 1988, pp. 164-165.

5. C.W. Eichelberger, R.J. Wojnarowski, R.O. Carlson, and L.M. Levinson, "High-density interconnects for electronic packaging," *Proceedings of SPIE, Vol. 877, Micro-Optoelectronic Materials*, January 1988, pp. 90-91.

6. C.Y. Chin and W.T. Lin, "A high performance VLSI architecture for computer graphics," *Visual Communications and Image Processing III*, November 1988.

# HYPERCOMPUTERS: DESIGN AND ARCHITECTURE

Ron Coleman and Michael Post
Department of Electrical Engineering and Computer Science
Polytechnic University
Brooklyn, New York 11201
(718) 260 - 3440

Alan Waksman
Plex Systems Research, Incorporated
New York City, NY 10016
(212) 686- 9600

## ABSTRACT

In this paper we introduce the *Hypercomputer* supercomputer, a reconfigurable, massively parallel architecture, a 9,072 processor prototype currently planned. The hypercomputer architecture family is based on arrays of a simple and autonomous unit logic entity, the universal cell. Physically wired in a uniform, eight-degree mesh, the universal cell is a pipelined, 8-bit microarchitecture that provides the logical manifestation of processor elements, switch lattices, memory units, et cetera. In this sense, we say the hypercomputer family is honeycomb reconfigurable. The mesh array is supported by a separate 3D, parallel IO network which provides real-time IO in the planar dimension and distributed configuration and synchronization along the polar dimension. The programming model, which is integral to our design, is based on a new concept we call *computational holism* whereby increasingly abstract and logically seamless clusters of tightly coupled cell chunks or *actors* give rise to parallel and highly specialized centers of computational activity. The software realization of this holistic approach is embodied in *hyperware*, an object-oriented, visually interactive environment for composing algorithmically specialized actors.

## INTRODUCTION

Reconfigurable, parallel architectures offer much greater promise for high performance over a diverse range of algorithms by restructuring their logical resources to accomodate the control or data characteristics of the algorithm. However, in the context of massive parallelism, reconfiguration places additional demands on efficient use of otherwise redundant architectural resources. By simple extrapolation, the rationale for building large machines is obvious: many computationally intensive problems present an opportunity for attack by massively parallelism. Traditional analyses of scalability have focused mainly on *upward scalability*. Unfortunately, upward scalability does not account for non-ideal circumstances (like routing collisions) in which large systems may hampered by their size. Recently, a new scalability analysis called *downward scalability* (Ref. 1, 2) has emerged which provides design motivations for architectures to "scale-down" into smaller, more efficient subnetworks in which the overhead (e.g., communication delay) is proportional to the size of the subnetwork. Thus, it can be shown that no multistage network (e.g., shared memory designs) is downward scalable. On the contrary, mesh and torus architectures (e.g., non-shared memory designs) are indeed downward scalable since data proximity (i.e., locality) can be exploited by nearest-neighbor communications.

Several reconfigurable architectures of this latter class, generically called *spatial arrays* (Ref. 3,4,5), have been suggested in the style of Snyder's seminal CHiP processor architecture (Ref. 6) and the more general *type architecture* (Ref. 7). In both cases, the primary design goal is the establishment of subnetworks which correspond to algorithmically specialized structures. When composed, these structures capture the natural locality and regularity of the underlying problem. A significant advantage of this approach to parallelism is that many VLSI and systolic algorithms can be readily implemented. Physically, the spatial array design is characterized by a two dimensional mesh of processor elements with an embedded switch lattice either integrated in the mesh plane or in a separate, parallel plane. Spatial arrays of this kind have been favored not only for their downward scalability but their upward scalability and bounded degree (i.e., modularity) all of which exploit the physical constraints of VLSI implementation. In this paper, we introduce the *Hypercomputer* supercomputer concept, a 9,072 processor prototype planned for construction at Plex Systems Research, Inc. in New York City. The Hypercomputer family represents an experimental extension of spatial arrays to their logical extreme in both architecture and magnitude. What is new in this approach is that we marry the two concepts of reconfiguration and massive parallelism in an utterly simple, and uniform design. We also suggest how certain practical problems are solved in this framework with respect to other spatial array designs: namely, architecture and programming.

## ARCHITECTURE

### Universal Cell Array Plane

At its foundation, hypercomputers are based on an eight-degree mesh of an autonomous unit logic entity called the *universal cell*. Physically, the universal cell is a microarchitecture with an pipelined, 8-bit data path, 48 8-bit general purpose registers, some control registers, flags and a 128x32-bit control store. Although pipelining is not crucial to our design, we have concluded that its performance (see Table 1) is well worth the small cost: e.g., < 100 transistors of hardware-assisted support is required in the microcontroller. We achieve this low cost by delegating almost all the usual hardware features to software: the microinstruction firing control is static, being computed at compile-time and encoded in the previous microinstruction (Ref. 8). That is, each microinstruction contains a horizontal 3-bit field which has the relative cycle tick on which the next microinstruction is to be loaded from the control store into its corresponding microinstruction register. At compile-time, two software modules called the *linear accelerator* and the *horizontal accelerator* schedule the execution of each microinstruction based on local and global flow analysis. The linear accelerator compacts the microcode temporally by minimizing the pipeline delay and the horizontal accelerator compacts the microcode temporally and spatially by collapsing the operations and reordering their sequence. Horizontal acceleration is based on the concept of trace scheduling as used, for example, in VLIW architectures (Ref. 9).

In the data path, stage 0 provides 1-bit shift (input A) and increment/decrement (input B) as preloading to the ALU, stage 1. The ALU computes three binary functions with programmable feedback to stage 0. The functions are *add*, *bit-wise nor*, and *bit-wise and*. Stages 2 and 3 provide increment/decrement (also with feedback to stage 0) and 1-bit shift, respectively. With the exception of the single control microinstruction, *branch* which operates in one cycle, each microinstruction requires four cycles (1 cycle/stage) and once an instruction begins execution (i.e., "fires"), it runs uninterrupted to completion. See Figure 1. *Very long arithmetic* or VLA processing is a useful application of micropipelining.

CH2649-2/89/0000/0467$01.00 © 1988 IEEE

*Table 1. Pipeline effects.*

| Task | Sequential | Pipelined | Speedup (theorectical max = 4.0) |
|---|---|---|---|
| 32-bit fixed pt add | 16 | 7 | 2.3 |
| 32-bit fixed pt mult | 1732 | 481 | 3.6 |
| 32-bit float pt add | 780 | 202 | 3.8 |
| 32-bit float pt mult | 1684 | 457 | 3.7 |
| 32-bit shift | 16 | 7 | 2.3 |
| VLA 192-bit add | 96 | 28 | 3.4 |
| VLA 384-bit shift | 192 | 52 | 3.7 |
| 8-bit cascaded sum | 40 | 31 | 1.3 |
| 16-bit cascaded sum | 73 | 55 | 1.3 |
| Triangle sort (ref 12) | 587 | 355 | 1.7 |
| Discrete Fourier Transform | 18,090 | 10,565 | 1.7 |

Specifically, *register chaining* is made possible whereby mutiple registers are treated as a single multibyte register so that for example, by chaining all 48 registers, a VLA 384-bit shift can be computed in 52 cycles (with pipelining) as opposed to 192 cycles (without pipelining). We employ this technique extensively for all 32-bit arithmetic, fixed and floating point. Since the universal cell does not have an index register (pointing to its own register file), we also use VLA for indexed, table processing where for example, in the Fourier transform, VLA is used to retrieve entries from the "twiddle factor" table (Ref. 10) and in CORDIC processing, the $\tan^{-1}$ table (Ref. 11).

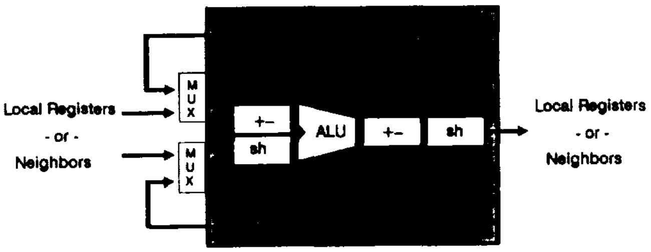


*Figure 1. Universal cell pipelined data path.*

A cell is interfaced to its eight neighbors through extensions of the data path collectively called the *synapse* in which there are four bidirectional, 8-bit channels: alternatively, we can think of the synaptic channels as a very small amount (four bytes) of edge-shared memory. Physically, the sympatic channels are implemented as a two-bit serial shifter whose activation is controlled by a pair of mask registers, one for OR-parallel input and one for AND-parallel output (i.e., local broadcast). By altering the mask registers, a cell can dynamically configure its IO structure. Hence, interprocessor communication is based on the *value passing* paradigm.

Because of the universal cell's generality, the manifest physical components of other spatial arrays designs--namely, processor elements, switch lattices and passive, external memory (or combinations therof)--these items can be configured logically through the universal cell by redefining the cell's operational microscript. In this sense, hypercomputers are *honeycomb* reconfigurable (Ref. 15).

### Input-Output

Whereas the universal cell array plane is the focus of all compute-bound and local IO, another connected but autonomous IO network is the focus of global IO to the array plane as a whole. This global IO network, generically called the *global bus*, is not a bus per se but is really two subnetworks which support the array plane in three dimensions. One perspective of the system architecture is depicted in Figure 2. The *planar controller* drives real-time IO in the planar dimension: that is, along the peripheral edges of the array plane. The *polar controller* drives configuration and synchronization through the polar dimension: e.g., setting the general purpose registers, the IO mask registers and loading the control store. Also, in the event that planar IO at the array edges cannot reach cells embedded in the interior, the polar controller can perform real-time IO by reconfiguring the general registers of these interior cells. Such a scenario could arise if, for example, the embedded algorithmic structure is enfolded in the plane to maximumize cell density.

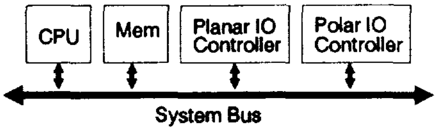


*Figure 2. System Architecture.*

Each IO controller serves a small subnetwork of Multiple Instruction stream, Single Data stream (MISD) IO processors (IOPs) each connected to one another and the controller through a high-speed broadcast bus. It is these IOPs which actually communicate with the array plane. See Figure 3. For example, the polar IOPs physically partition the array plane into rectangular parallel sectors. The IOP can address the cells in the parallel sector discretely, cell by cell or globally through parallel sector-wide broadcasting. The planar IOPs physically partition the array plane at the periphery into *P* linear parallel sectors or λ-parallel sectors. Thus, the host CPU controls the entire machine only indirectly through the substantially fewer networked IOPs. Typically, the host will broadcast a stream of spatial directives (i.e., high level commands) which describe how to configure the logical regions of the array plane. All polar IOPs listen for directives which address their particular parallel sectors and execute pre-compiled, locally stored mount scipts; these mount scripts may in turn invoke other mount scipts or configure the cells under its parallel sector jurisdiction directly. A similar process occurs for the planar IOPs. This IO scheme solves two major practical problems each greatly magnified in the context of massive parallelism. First, the problems of reliably distributing global control signals--power, ground and clock--is avoided. Second, IO can also proceed in parallel, avoiding a bottleneck in host-only operations.

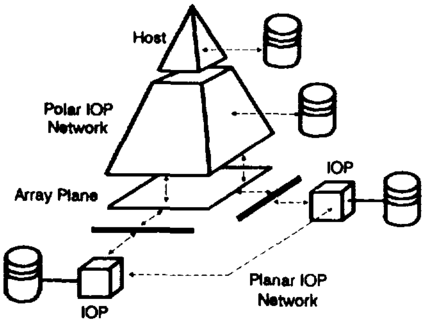


*Figure 3. IO Architecture.*

## PROGRAMMING

### Computational Holism

The limitations of fixed parallel architectures become evident if we consider that many real-world problems are analysed as a set of smaller, *cooperative subproblems* each of which may be realized by different algorithmically specialized organizations. In this model, parallel computation is carried out cooperatively, without global control and using composed, locally interactive agents. Synder (ref. 6) was the first to investigate this technique by way of the CHiP processor architecture. However, cooperative computation remained merely a useful, secondary mechanism to program the CHiP processor: the primary mechanism was Poker (Ref. 13, 14), a process-oriented language developed to explicitly program each of the machine elements, regardless of their numbers. Milutinovic and

Milutinovic (Ref. 15) suggested the possibility of cooperative computation using multicell CPUs, memories and buses but carried their abstraction only to this level. Moreover, we have envisioned scenarios where the collective behaviour of a cell group may express much more fluid characteristics, perhaps being CPU-like for a while, then memory-like and so on. In general, we believe above approaches become impractical in a massively parallel framework. We believe this provides sufficient motivation for a disciplined approach to functional and hierarchical integration which can specify abstract and arbitrarily complex computational behaviors.

We now introduce an approach which allows us to escape the grueling details involved with large conglomerations of cells. In this model which we call *computational holism*, objects or *actors* (Ref. 17) form hierarchies of logically continuous compositions of cell ensembles or *chunks* which cooperate as a single whole giving rise to parallel and highly specialized centers of computational activity. At the lowest level, the primitive actors which make a chunk are strongly interactive, communicating frequently and efficiently, exploiting locality preserved by the cellular mesh. In turn, these specialized, low level chunks participate in cooperative computation with one another and so on in a hierarchy. Thus, we achieve both functional and hierchical integration. Moreover, this approach maps directly to an object-oriented, visual programming environment we call *hyperware*. (In the next subsection, we discuss hyperware and how its maps into the hypercomputer.) From the programmers point of view, hyperware is more than a medium for expressing hypercomputer algorithms: it *is* the hypercomputer, although the physical details of parallel sectors and global communication are transparent. As such, the map from the programmer input to hypercomputer semantics is direct: parallelism is not inferred. The programmer actually makes full use of the machine efficiently. In addition, the programmer can choose the level of abstraction, suppressing details in one case and enhancing them in another.

### Hyperware

The software realization of computational holism is embodied in a visually interactive environment called hyperware. From the user's point of view, hyperware presents two abstract "window" spaces which interact with the user: the *composition space* where actors are composed and the *work space* where actors are staged for functional or spatial manipulation prior to being "chunked" in the composition space: that is, the work space actor becomes part of a larger, tightly coupled assembly of other composition space actors. In one level of detail, the chunked actor relinquishes its simpler, individual identity (e.g., its primitive cell characteristics) to collaborate in the expression of a more complex, logically seamless identity. As a whole, the association consistutes another more abstract actor. In hyperware, all actors are *first-class*, regardless of their constitutions. Although there is one actor per window, each space may have multiple windows so that several actors can be composed simultaneously. Also, each space is local with relative spatial coordinates in the mesh array: global, absolute binding takes place at configuration-load time. Programming begins in one of two ways: either defining a new, primitive actor (which maps directly to a physical cell) in the work space or alternatively, retrieving a previously defined primitive or complex actor from a library of actors. If the actor is new, we supply a script of microinstructions (i.e., a microscript) which defines the local computational and communication behaviors. This level of detail is the "suppressed" and compiled so that we deal only with an object having a set of input and output ports. These IO port details can also be suppressed but at least one port must remain "visible": after chunking the actor, the unsupressed ports are used to connect other actors. This process continues until a new, more complex actor is fully composed. The newly composed actor can be returned to the work space for further manipulation including enhancing or suppressing port details, spatial fixation (i.e., rotating, flipping, flexing, growing, etc.). Alternatively, the actor can be written to a library of actors to be used later.

These ideas are best illustrated with a simple example: in this section we consider building a tree structure of actors. In the next section, "Applications", we show a much more complex, hexagonal actor and its applications. In Figure 4, the root of the subtree is already in the compostion space and a triple chunk is being prepared in the work space. In each space, the lettered arrows represent output ports and the numbered arrows, input ports. The goal is to compose the work space and composition space actors, creating a new actor. The user has only one simple syntax rule to observe: that is, unsuppressed inputs must be connected to complementary unsuppressed outputs or vice versa. Suppressed details obey this rule implicitly. (Actually, the composition driver will inspect all details in the final analysis.) If we chunk the work space actor, connecting outputs (in the work space) to inputs (in the composition space), the composition driver will discover only one possibility, namely *b→1*. (We qualify the transfer as above, to constrain the possibilites the composition driver will have to consider. The same effect could have been achieved by suppressing the composition space actor's input ports.) Figure 5 shows still another alternative in which the work space actor is first rotated by $\pi/4$ radians (counter-clockwise) at the root then chunked.

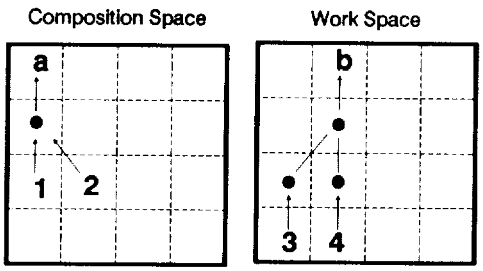


*Figure 4. Tree composition in hyperware.*

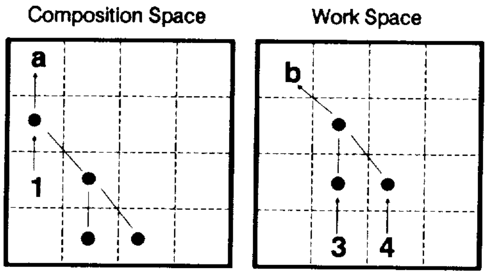


*Figure 5. Rotation and chunking of work space actor.*

## APPLICATIONS

To date, a broad class of many algorithms with different configuration demands have been tested in our design using the hypercomputer simulator. These include novel algorithms for pattern matchining (Ref. 11), sorting (Ref. 12), the multidimensional Fourier transform (Ref. 10), parallel arithmetic and matrix processing (Ref. 16). We anticipate that other applications will also enjoy excellent performance on our architecture including image processing, field interaction problems (e.g., discrete hydrodynamic and Coulombic simulations), connectionist and artificial neural systems. In this section, we present a complex, hexagonal actor as an example for both matrix multiplication and LU-decomposition. We analyse this actor and suggest how it can be architected using hyperware.

### Matrix Multiplication and LU-decomposition

The hexagonal systolic array due to Leiserson (Ref. 18) is the basis of our hexagonal actor. Leiserson shows how this fixed computation and fixed communication structure can be used for multiplying two matricies, **A**, **B** such that **C** = **A** x **B** and for factoring a matrix **A** such that **A** = **L** x **U** where **L** and **U** are lower and upper triangular respectively. Like the systolic versions, the primitive computational agents in our system is the *inner-product-step actor*, flavors of which are shown in Figure 6. Since a

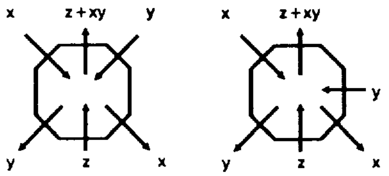


*Figure 6. Inner-product-step actors.*

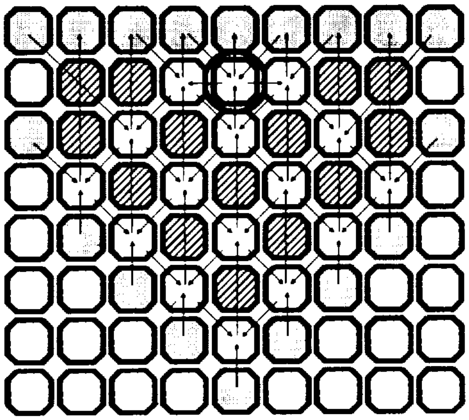


*Figure 7. Hexagonal actor for matrix processing.*

hexagonal array does not map directly into an octagonal array, we use two other primitive actors: one to forward intermediate results along the interior of the array and one to forward external inputs.and final result outputs from and to the world outside the actor. Except for the latter tasks, the two actors are the same. These three primitive actors are shown composed (for matrix multiplication) in Figure 7 as the interior, unfilled cells, the striped cells and the shaded cells embedded in a 72 cell, *9x8* subarray. All other cells in this array are "dead", acting as data sinks. This complex actor can also be used for LU-decomposition as per Leiserson (Ref. 18).

Since the actor is vertically symmetrical, the actor can be built in hyperware quite easily: compose the center column and the left hand side as two separate actors. Compose the right hand side as a horizontal flip of the left hand side. Then, compose the center, left and right hand sides as one actor.

## CONCLUSIONS

We have presented the central ideas of the design and architecture of the Hypercomputer supercomputer concept. In this approach, we believe we have solved not only several problems in its own technical design but by paying careful attention to simple, feature principles, we have addressed a number of critical problems in parallel processing as well. For example, many researchers question the relative merits of fine-grain parallelism as opposed to course-grain parallelism for attacking real-world problems. Indeed, each approach has its strengths and weaknesses. Our high level solution--computational holism--lies somewhere between of the two extremes and allows us to capture the power of course granularity without losing the flexibility of fine granularity. The performance of the computationally intensive multidimensional Fourier transform on the hypercomputer is proof of this claim: see Reference 10. The level of parallelism is another major issue of concern: some researchers maintain that parallelism at the instruction level is simply too low level or too inefficient; algorithm parallelism at a higher level is preferred instead. Again, our solution attracts the benefits of both, one through the simple, unconstrained nanopipeline and the other, through reconfiguration. There are other problems in parallel processing in general and massively parallel processing in particular (e.g., programming) which we also address but space does not permit us to debate the issues fully here. In summary, however, we have seen that our integrated approach offers much greater promise for high performance over a diverse range of issues, computational, structural and architectural.

## ACKNOWLEDGMENTS

This work has been funded in parted by Applied Concepts, Inc.: we appreciate its generosity in supporting our work. We also thank the staff members of Applied Concepts, Inc., Sally Huns, and Maria Gonzalez-Coleman for their spirit of encouragement and assistance, and Haldun Hadimioglu and the Polytechnic Parallel Processing Group for providing insightful technical criticisms and invaluable feedback throughout the evolution of many of the ideas presented here.

## REFERENCES

1. Ma, Y.E., et al, "Reconfigurable Special-Purpose Computers", *Second International Conference on Supercomputing*, May 1987
2. Ma, Y.E., Shea, D.G., "Downward Scalability of Parallel Architectures", *Third International Conference on Supercomputing*, May 1988
3. Hancu, M.V., Smith, K.C., "DYPP: A VLSI Dynamic-Graph Ensemble Machine", *1988 International Conference on Supercomputing*
4. Koren, I., "A Reconfigurable Fault-Tolerant VLSI Multiprocessor Array", *8th Annual Symposium on Computer Architecture*, May 1981
5. Gollakota, N., Gray, G., "Reconfigurable Cellular Architecture", *1984 International Conference on Parallel Processing*
6. Snyder, L., "Introduction to the Configurable Highly Parallel Computer", *IEEE Computer*, 1982
7. Snyder, L., "Type Architectures, Shared Memory and the Corollary of Modest Potential", *Annual Review of Computer Science*, 1986
8. Lusk, M.E., "A (Linear) Scheduler for the Hypercomputer Simulator", *Department of Electrical Engineering and Computer Science Project Report*, Polytechnic University, May 1988
9. Colwell, R., "A VLIW Architecture For a Trace Scheduling Compiler", *IEEE Transactions on Computers*, August 1988
10. Coleman, R., Post, M., Waksman, A., "A Uniform and Reconfigurable Framework for the Multidimensional Fourier Transform", (companion paper), October 1988
11. Coleman, R., Post, M., "Pattern Recognition and the Multidimensional Fourier Transform on the Hypercomputer", *Polytechnic Parallel Processing Group*, Polytechnic University, April 1988
12. Coleman, R., Post, M., "Primary Computational Agents for Sorting on the Hypercomputer", *Polytechnic Parallel Processing Group*, Polytechnic University, November 1987
13. Snyder, L., "Parallel Programming and the Poker Programming Environment", *IEEE Computer*, July 1984
14. Notkin, D., et al, "Experiences with Poker", *Parallel Programming: Experience with Applications, Languages, and Systems*, ACM/SIGPLAN Notices PPEALS, 1988
15. Milutinovic, D., et al, "The Honeycomb Architecture", *IEEE Computer*, April 1987
16. Lee, C., "Primary Computational Agents for Parallel Integer Arithmetic On the Hypercomputer", *Department of Electrical Engineering and Computer Science Project Report*, Polytechnic University, May 1988
17. Agha, G., *Actors, A Model of Concurrent Computation in Distributed Systems*, MIT Press, 1986
18. Leiserson, C., *Area-Efficient VLSI Computation*, Ph.D. Thesis, MIT Press, 1982

# A REDUCED DIAMETER INTERCONNECTION NETWORK*

K. Efe
Center for Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, LA 70504

P. Blackwell T. Shiau W. Slough
Computer Science Department
University of Missouri–Columbia
Columbia, MO 65201

## ABSTRACT

Many interconnection networks have been suggested as the basis for parallel computing architectures. In this paper, we propose a network, based upon the hypercube, which we call the *multiply-twisted cube.* This network preserves many of the desirable properties of the hypercube, but has a diameter which is only $\lceil (n+1)/2 \rceil$ for an $n$–dimensional multiply-twisted cube, a reduction of nearly fifty percent compared to the ordinary hypercube. We discuss some of the basic topological properties of multiply-twisted cubes and present a routing algorithm which produces optimal paths.

Keywords: Interconnection networks, hypercube, parallel processing.

## INTRODUCTION

An important component of a parallel computer is the interconnection network. The designer is confronted with an overwhelming number of choices, including cross-bar, shuffle-exchange, butterfly, mesh, hypercube, hypernet and many others. Among these choices, the hypercube has enjoyed popularity due to many of its attractive properties, including regularity, symmetry, small diameter, strong connectivity and relatively small link complexity. Other properties of the hypercube can be found in [3, 5, 7, 9, 12].

The hypercube topology has received much attention in the literature due to its suitability for general purpose parallel processing [2, 8, 10, 11, 13]. As a result of this focused attention, variations of this topology have been discovered which improve upon the hypercube. For example, Esfahanian *et al.* [4] introduced a class of networks denoted $TQ_n$, for $n \geq 3$, obtained by exchanging any two independent edges in a shortest cycle of the $n$–dimensional hypercube $Q_n$. Introducing such a "twist" reduces the diameter from $n$ to $n-1$, preserving many desirable properties of the hypercube.

Recently, Shiau *et al.* [14] introduced a method of systematically exchanging multiple pairs of edges in an $n$-dimensional hypercube, achieving a graph with diameter $\lceil 2n/3 \rceil$ which has many of the properties of the hypercube. In this paper, we improve these results by introducing an $n$–dimensional "multiply-twisted" cube $MQ_n$ with diameter $\lceil (n+1)/2 \rceil$. We discuss some of the basic properties of this topology and show that it too has many desirable properties of the hypercube, including regularity, small diameter, and large vertex connectivity. We also develop a routing algorithm which guarantees a shortest path between any pair of vertices in $MQ_n$.

## PRELIMINARIES

In this paper, we use undirected graphs to model interconnection networks. We refer the reader to [6] for fundamental graph terminology.

Let $G = (V, E)$ be a finite, undirected graph. The *distance* between vertices $u$ and $v$, denoted $d(u, v)$, is the length of a shortest path from $u$ to $v$. The *diameter* of $G$, denoted $D(G)$, is defined to be $\max\{d(u,v) : u, v \in V\}$. The graphs we consider here, strictly speaking, are labeled graphs. We will, however, often make no distinction between a vertex and its label.

The *vertex connectivity* of a graph G, denoted $\kappa(G)$, is the minimum number of vertices whose removal results in a disconnected or trivial graph. (A *trivial* graph is one with a single vertex and no edges.)

The labels we use are binary strings and, by convention, an $n$-bit string is indexed with the values 0 through $n-1$, letting 0 index the least-significant bit. The notation $b^n$ denotes the string with $n$ repetitions of the bit $b$. $G^b$ denotes the labeled graph obtained by prefixing every vertex label in the graph $G$ with $b$.

Two binary strings $x = x_1x_0$ and $y = y_1y_0$ are *pair-related*, denoted $x \sim y$, if and only if $(x, y) \in \{(00,00), (10,10), (01,11), (11,01)\}$; if $x$ and $y$ are not pair-related, we write $x \not\sim y$.

**Definition 1** *The $n$–dimensional multiply-twisted cube, denoted $MQ_n$, is the labeled graph defined inductively as follows. $MQ_1$ is $K_2$, the complete graph on two vertices with labels 0 and 1. For $n > 1$, $MQ_n$ contains $MQ^0_{n-1}$ and $MQ^1_{n-1}$ joined according to the following rule: the vertex $u = 0u_{n-2}\cdots u_0$ from $MQ^0_{n-1}$ and the vertex $v = 1v_{n-2}\cdots v_0$ from $MQ^1_{n-1}$ are adjacent in $MQ_n$ if and only if*

1. *$u_{n-2} = v_{n-2}$ if $n$ is even, and*
2. *for $0 \leq i < \lfloor (n-1)/2 \rfloor$, $u_{2i+1}u_{2i} \sim v_{2i+1}v_{2i}$.*

Figure 1 shows $Q_n$, $TQ_n$, and $MQ_n$ for $n = 3$ and $n = 4$. For $n < 3$, all three networks are isomorphic.

It follows from Definition 1 that every vertex in $MQ_n$ with a leading 0 bit has exactly one neighbor with a leading 1 bit and vice versa. From this fact and the recursive structure of $MQ_n$, the reader may verify that $MQ_n$ is a connected, regular graph of degree $n$ with $2^n$ vertices.

## BASIC PROPERTIES

In the ordinary hypercube, there is a simple rule which states when an edge is present, *viz.* an edge is incident to vertices $u$ and $v$ if and only if $u$ differs from $v$ in exactly one bit. We can provide a similar characterization for edges in a multiply-twisted cube, albeit the rule is somewhat more complex. The following lemma provides the details. (See [15] for all proofs omitted from this paper.)

**Lemma 1** *For all $n \geq 1$, $(u_{n-1}\cdots u_0, v_{n-1}\cdots v_0)$ is an edge of $MQ_n$ if and only if there exists an $\ell$ with*

1. *$u_{n-1}\cdots u_\ell = v_{n-1}\cdots v_\ell$,*
2. *$u_{\ell-1} \neq v_{\ell-1}$,*
3. *$u_{\ell-2} = v_{\ell-2}$ if $\ell$ is even, and*
4. *for $0 \leq i < \lfloor (\ell-1)/2 \rfloor$, $u_{2i+1}u_{2i} \sim v_{2i+1}v_{2i}$.*

When conditions 1 and 2 of Lemma 1 hold, we say that $u$ and $v$ have a *leftmost differing bit at position $\ell - 1$.* When two adjacent vertices $u$ and $v$ have a leftmost differing bit at position $d$, we say that $v$ is the *$d$-neighbor of $u$* and that the edge $(u, v)$ is an *edge of dimension $d$.* We refer to conditions 3 and 4 as the *pairing condition.*

*This research was funded by a grant from the Research Council of the Graduate School, University of Missouri–Columbia.

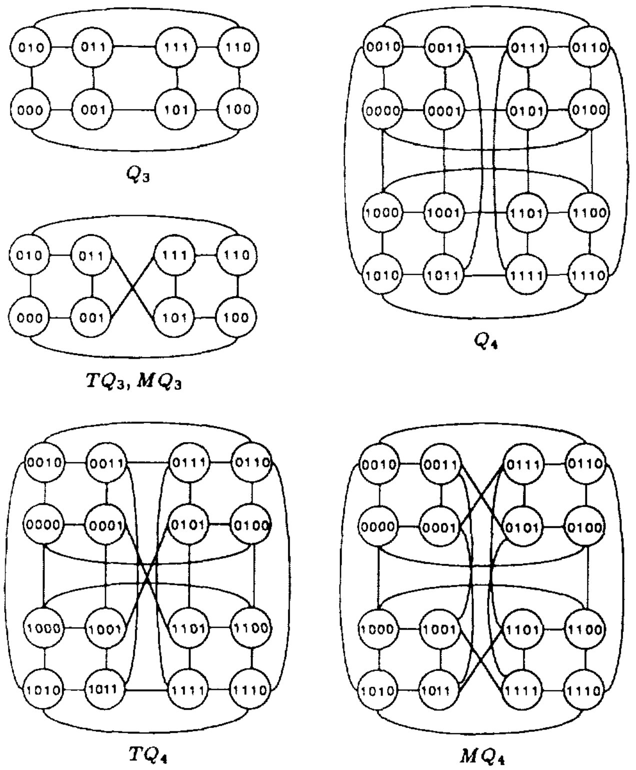


Figure 1: $Q_n$, $TQ_n$, and $MQ_n$ for $n = 3, 4$.

In order to facilitate a discussion of the topological properties of $MQ_n$, it is convenient to introduce a mechanism for identifying induced subgraphs. For this purpose, let $\Gamma_{\alpha,\beta}(G)$ denote the subgraph of the labeled graph $G$ induced by the set of all vertices with prefix $\alpha$ or $\beta$. We also use $\Gamma_\alpha(G)$ as an abbreviation for $\Gamma_{\alpha,\alpha}(G)$.

An $n$-dimensional multiply-twisted cube contains a variety of subgraphs which are isomorphic copies of multiply-twisted cubes of lower dimension. Lemmas 2 through 4 make this idea more precise.

**Lemma 2** *For all $n \geq 2$, $\Gamma_0(MQ_n) \simeq MQ_{n-1}$ and $\Gamma_1(MQ_n) \simeq MQ_{n-1}$. Moreover, the isomorphisms are given by the function which removes the leading bit from every vertex label in a labeled graph.*

**Lemma 3** *For all $k \geq 1$, $\Gamma_{00,10}(MQ_{2k}) \simeq MQ_{2k-1}$ and $\Gamma_{01,11}(MQ_{2k}) \simeq MQ_{2k-1}$. Moreover, the isomorphisms are given by the function which removes the bit at position $(2k-2)$ from each vertex label.*

**Lemma 4** *For all $k \geq 1$, $\Gamma_{\alpha,\beta}(MQ_{2k+1}) \simeq MQ_{2k-1}$ for all $\alpha, \beta$ in $\{(001, 111), (011, 101), (000, 100), (010, 110)\}$. Moreover, these isomorphisms are given by the function which removes bits $(2k-1)$ and $(2k-2)$ from each vertex label.*

The following lemma is useful for arguing about the diameter of multiply-twisted cubes. Roughly speaking, the lemma states that two vertices $u$ and $v$ of a multiply-twisted cube are either contained in a smaller dimension multiply-twisted cube or that $u$ has a neighbor, $u'$, with the property that $u'$ and $v$ are contained in a smaller dimension multiply-twisted cube.

**Lemma 5** *For all $n \geq 2$, if $u$ and $v$ are vertices of $MQ_n$ then either*

1. *$u$ and $v$ belong to a subgraph of $MQ_n$ which is isomorphic to $G$, where $G = MQ_{n-2}$ if $n$ is even, otherwise $G = MQ_{n-3}$, or*
2. *$u$ and $v$ belong to a subgraph of $MQ_n$ which is isomorphic to $G$, where $G = MQ_{n-1}$ if $n$ is even, otherwise $G = MQ_{n-2}$, or*
3. *$u$ has a neighbor, $u'$, where $u'$ and $v$ are in a subgraph of $MQ_n$ which is isomorphic to $G$, where $G = MQ_{n-1}$ if $n$ is even, otherwise $G = MQ_{n-2}$.*

We are now in a position to state and prove that multiply-twisted cubes have relatively low diameter.

**Theorem 1** *For $k \geq 1$, $D(MQ_{2k}) = D(MQ_{2k+1}) = k + 1$.*

**Proof.** To prove the claimed equalities, we show that $k + 1$ serves as both an upper bound and a lower bound for the diameter of each of $MQ_{2k}$ and $MQ_{2k+1}$.

To prove $k+1$ is an upper bound, we use induction on $k$. For $k = 1$, the result is immediate. For the induction, assume $D(MQ_{2i}) \leq i + 1$ and $D(MQ_{2i+1}) \leq i + 1$ for $1 \leq i < k$. Let $u$ and $v$ be vertices of $MQ_{2k}$ (or $MQ_{2k+1}$). Lemma 5 applies, yielding three cases. In cases (1) or (2), $u$ and $v$ belong to a subgraph isomorphic to either $MQ_{2k-2}$ or $MQ_{2k-1}$. In either case, the induction hypothesis applies, yielding $d(u, v) \leq k$. In case (3), a neighbor of $u$, $u'$, belongs to a subgraph isomorphic to $MQ_{2k-1}$; hence $d(u', v) \leq k$ by the induction hypothesis. Since $u$ and $u'$ are adjacent, $d(u, v) \leq 1 + d(u', v) \leq k + 1$.

To prove $k + 1$ is a lower bound, we first note that the characterisation of edges of a multiply-twisted cube given in Lemma 1 provides the following observation.

**Observation 1** *For any edge $(u, v)$ in $MQ_n$, the number of even-indexed bits of $u$ and $v$ which differ is at most one. Furthermore, if there is some $i$ with $u_{2i} \neq v_{2i}$ then $u_{n-1} \cdots u_{2i+1} = v_{n-1} \cdots v_{2i+1}$.*

Let $u = 0^{2k}$ and $v = 1^{2k}$ be vertices of $MQ_{2k}$. We show that any path $\pi$ from $u$ to $v$ has length at least $k + 1$. Each edge $(w, y)$ of $\pi$ causes the vertex label $w$ to be transformed to $y$ in a manner consistent with Lemma 1. The cumulative effect of these changes is to cause each bit of $u$ to be changed. There are $k$ bits with even-numbered indices which must be changed. From Observation 1, any edge in $\pi$ can cause at most one of these even-numbered bits to change; hence, $\pi$ has at least $k$ edges. In addition, none of these $k$ edges cause the most significant bit of $u$ to change, so there is at least one additional edge in $\pi$. Hence, $\pi$ has at least $k + 1$ edges.

For $MQ_{2k+1}$, a similar argument prevails. Let $u = 0^{2k+1}$ and $v = 1^{2k+1}$ be vertices of $MQ_{2k+1}$ and let $\pi$ be any path from $u$ to $v$. There are $k+1$ bits of $u$ with even-numbered indices which must change. Any edge of $\pi$ can change at most one of them; hence, $\pi$ has at least $k + 1$ edges.

Vertex connectivity has been used as a measure of the "robustness" of a network[1], where, informally, a network is considered to be *robust* if its performance does not degradate much in the presence of processor faults. The hypercube and the multiply-twisted cube have the same vertex connectivity, as the following lemma shows.

**Lemma 6** *For all $n \geq 1$, $\kappa(MQ_n) = n$.*

**Proof.** The proof is by induction on $n$. The removal of any vertex from $MQ_1$ yields the trivial graph; hence $\kappa(MQ_1) = 1$. For the induction, assume that $\kappa(MQ_{n-1}) = n - 1$. Removing the $n$ neighbors of any vertex of $MQ_n$ produces a disconnected graph; therefore, $\kappa(MQ_n) \leq n$. We now show that removing fewer than $n$ vertices does not disconnect $MQ_n$.

Let $X$ be a set of $n - 1$ vertices to be removed from $MQ_n$. Either all vertices of $X$ are from $\Gamma_0(MQ_n)$, all are from $\Gamma_1(MQ_n)$, or $X$ has vertices from each of $\Gamma_0(MQ_n)$ and $\Gamma_1(MQ_n)$. We consider each case in turn.

Suppose all vertices of $X$ are from $\Gamma_0(MQ_n)$. Further, suppose the removal of the vertices of $X$ disconnects $\Gamma_0(MQ_n)$. There are, therefore, vertices $u = 0u_{n-2} \cdots u_0$ and $v = 0v_{n-2} \cdots v_0$ which are not joined by a path which lies exclusively in $\Gamma_0(MQ_n)$; however, they are joined by a path which can be seen as follows. Each of $u$ and $v$ have $(n-1)$-neighbors, $u'$ and $v'$. Since no vertex of $X$ is from $\Gamma_1(MQ_n)$, there is a path from $u'$ to $v'$ which lies exclusively in $\Gamma_1(MQ_n)$. This path, together with the edges $(u, u')$ and $(v, v')$ demonstrate the existence of a path from $u$ to $v$. Hence, the removal of the vertices of $X$ does not disconnect $MQ_n$. A similar argument holds when all vertices of $X$ are from $\Gamma_1(MQ_n)$.

Now suppose $X$ has vertices from both $\Gamma_0(MQ_n)$ and $\Gamma_1(MQ_n)$. In this case, each of $\Gamma_0(MQ_n)$ and $\Gamma_1(MQ_n)$ remain connected, by the induction hypothesis. There are at least $2^{n-1} - (n-1) > 0$ surviving edges with dimension $n-1$ after the removal of the vertices of $X$; hence, the resulting graph remains connected.

From a well-known result of Whitney [6, page 48] along with the fact that $MQ_n$ is $n$-regular, we obtain the following.

**Corollary 1** *Every pair of vertices in $MQ_n$ is joined by exactly $n$ vertex-disjoint paths.*

One reason which accounts for the popularity of the hypercube is that many other networks can be embedded, or nearly embedded, within it. It is, therefore, reasonable to ask what other networks can be embedded in the multiply-twisted cube. We briefly address this issue by noting that a ring with $2^n$ vertices can be embedded in $MQ_n$ for all $n \geq 2$. We restate this result in graph-theoretic terms as follows.

**Lemma 7** *For all $n \geq 2$, $MQ_n$ has a Hamiltonian cycle.*

**Proof.** We prove a statement which is somewhat stronger than the lemma; *viz.* for all $n \geq 2$, $MQ_n$ has a Hamiltonian cycle with the edge $(0^n, 10^{n-1})$. We proceed by induction on $n$. Since $MQ_2$ is a cycle with 4 vertices, the basis holds. For the induction, assume the stated assertion holds for $MQ_{n-1}$. In $MQ_n$, consider the two subgraphs $\Gamma_0(MQ_n)$ and $\Gamma_1(MQ_n)$. By the induction hypothesis and Lemma 2, $\Gamma_0(MQ_n)$ has a Hamiltonian cycle $\sigma_0$ with edge $(u,v) = (00^{n-1}, 010^{n-2})$. Also, due to the isomorphism given by Lemma 2, a Hamiltonian cycle $\sigma_1$ in $\Gamma_1(MQ_n)$ may be produced by complementing the first bit of each label in $\sigma_0$. Note that $(u',v') = (10^{n-1}, 110^{n-2})$ is an edge of $\sigma_1$ and that, by Lemma 1, $(u,u')$ and $(v,v')$ are both edges of $MQ_n$. By replacing $(u,v)$ and $(u',v')$ with $(u,u')$ and $(v,v')$ in $\sigma_0$ and $\sigma_1$, a Hamiltonian cycle in $MQ_n$ is produced; moreover, this cycle contains the edge $(u,u')$ thereby meeting the requirements of the assertion. Hence, the lemma follows.

# SHORTEST PATHS AND ROUTING

A parallel architecture based upon the multiply-twisted cube should support a mechanism which allows any two processing elements to exchange data. This may be achieved by finding a shortest path from the source vertex to the destination vertex, taking advantage of the hierarchical nature of multiply-twisted cubes. In this section, we discuss an algorithm which produces a shortest path between any two vertices in a multiply-twisted cube.

As a matter of convenience, we present the algorithm as a sequential algorithm. After this, we discuss some minor modifications which make the algorithm distributed.

In order to find a route between two vertices of a multiply-twisted cube, our algorithm makes extensive use of the topology of $MQ_3$. Suppose $u = u_2u_1u_0$ and $v = v_2v_1v_0$ are nonadjacent vertices of $MQ_3$ with $u_2 \neq v_2$. Since the diameter of $MQ_3$ is two, there is a vertex $w$ which is a common neighbor of $u$ and $v$. For example, the 2-neighbor of 000, 100, is a common neighbor of 000 and 110. This vertex is not unique; the 1-neighbor of 000, 010, is also a common neighbor of 000 and 110. We can summarize this information by noting the dimensions involved—a common neighbor of 000 and 110 may be obtained by choosing a $d$-neighbor of 000, where $d$ is an element of $\{1,2\}$. Due to the symmetry of $MQ_3$, we can complement the leading bits of each of $u$ and $v$ to obtain a similar statement—a common neighbor of 100 and 010 may be obtained by choosing a $d$-neighbor of 100, where $d$ is an element of $\{1,2\}$.

The table shown in Figure 2 summarizes this common neighbor information for $MQ_3$. If $d$ is an element of MOVE$(\alpha,\beta)$, then the $d$-neighbor of $b\alpha$ is also a neighbor of $\bar{b}\beta$.

Before presenting the details of the routing algorithm, we provide an example of how a path between two vertices may be produced.

| $\alpha \setminus \beta$ | 00 | 01 | 10 | 11 |
|---|---|---|---|---|
| 00 | - | {2} | {1,2} | {0} |
| 01 | {0} | {1,2} | {2} | - |
| 10 | {1,2} | {0} | - | {2} |
| 11 | {2} | - | {0} | {1,2} |

Figure 2: MOVE table used in routing.

**Example.** Suppose a path from

$$u = 10\ 10\ 01\ 00\ 01\ 10$$

to

$$v = 10\ 11\ 11\ 10\ 11\ 01$$

is desired. (*Note*: The spacing between bits is present to aid readability.) As a first step, locate the leftmost differing bit position of $u$ and $v$, which occurs at bit 8. Imagine the bits to the right of this bit grouped into pairs. Starting at the leftmost differing bit position, scan $u$ and $v$ from left to right, comparing a pair of bits from $u$ with the corresponding pair of $v$, stopping at the first pair which is not pair-related. In this example, this occurs at the second pair to the right of bit 8. We focus our attention on the three bits from $u$ and $v$ thus identified:

$$u = 10\ 1\boxed{0}\ 01\ \boxed{00}\ 01\ 10$$
$$v = 10\ 1\boxed{1}\ 11\ \boxed{10}\ 11\ 01$$

Viewed as vertices from $MQ_3$, the bits under consideration represent nonadjacent vertices with differing leading bits. The MOVE table provides a way to locate a common neighbor, but rather than use the values provided by this table in an absolute manner, we use them as a relative dimension—2 dictates the selection of the highest dimension from among the three dimensions under consideration, 1 selects the next lowest dimension and 0 selects the lowest dimension. Since MOVE(00, 10) = {1, 2} there are two choices; suppose we use 1. Consequently, we select the 5-neighbor of $u$, yielding the first vertex of the path:

$$w_1 = 10\ 10\ 01\ 10\ 11\ 10$$

By moving to $w_1$, bits 6 through 11 remain unchanged and, in addition, the pair of bits to the right of bit 6 from $w_1$ and $v$ are now pair-related. We repeat the scanning described above for $w_1$ and $v$, starting with the pair to the right of bit 4, which yields

$$w_1 = 10\ 1\boxed{0}\ 01\ 10\ \boxed{11}\ 10$$
$$v = 10\ 1\boxed{1}\ 11\ 10\ \boxed{11}\ 01$$

As before, MOVE(11, 11) provides two possibilities—1 or 2. Suppose we again choose 1. Selecting the 3-neighbor of $w_1$ yields the second vertex of the path:

$$w_2 = 10\ 10\ 01\ 10\ 01\ 10$$

Continuing, we obtain

$$w_2 = 10\ 1\boxed{0}\ 01\ 10\ 01\ \boxed{10}$$
$$v = 10\ 1\boxed{1}\ 11\ 10\ 11\ \boxed{01}$$

from which we select the 0-neighbor of $w_2$, yielding

$$w_3 = 10\ 10\ 01\ 10\ 01\ 11$$

At this point there is no pair to the right of bit 8 which does not satisfy the pairing condition; however, the 8-neighbor of $w_3$ is $v$, which completes the path.

In the preceding example, the leftmost differing bit index, once found, never changes. In general, however, this does not always happen. If the highest of the three dimensions under consideration is used, the leftmost differing bit index in the next step "migrates" to the right—in fact, it will be one of the two remaining dimensions. After this migration occurs, all the bits to its left agree with the corresponding bits of the destination vertex.

Another situation not illustrated by this example occurs when the leftmost differing bit leaves an odd number of bits to its right. When this occurs, imagine the bits paired from the right, leaving a single bit to the right of the leftmost differing bit.

Figure 3 provides the details of the routing algorithm. The expression "choice($S$)" evaluates to an arbitrary element from the nonempty set $S$.

```
procedure EmitDimensions($u, v$)

    $w := u$
    $\ell :=$ LeftmostDifferingBitIndex($w, v$)
    if ($\ell$ is odd) and ($w_{\ell-1} \neq v_{\ell-1}$) then
        $d :=$ choice($\{\ell, \ell - 1\}$)
        emit($d$); $w := d$-neighbor($w$)
        if ($d = l$) then $\ell := \ell - 1$ fi
    fi
    $k := \lfloor \ell/2 \rfloor$
    while ($k \geq 1$) do
        $\alpha := w_{2k-1}w_{2k-2}$
        $\beta := v_{2k-1}v_{2k-2}$
        if ($\alpha \not\sim \beta$) then
            $d :=$ choice(MOVE($\alpha, \beta$))
            case
                $d = 0$:
                    emit($2k - 2$); $w := (2k - 2)$-neighbor($w$)
                $d = 1$:
                    emit($2k - 1$); $w := (2k - 1)$-neighbor($w$)
                $d = 2$:
                    emit($\ell$); $w := \ell$-neighbor($w$)
                    if $w_{2k-1} \neq v_{2k-1}$ then
                        $\ell := 2k - 1$
                    else
                        $\ell := 2k - 2$
                    fi
        fi
        $k := k - 1$
    od
    emit($\ell$)
```

Figure 3: Routing in a Multiply-Twisted Cube.

**Theorem 2** *For any two distinct vertices $u$ and $v$ in $MQ_n$, EmitDimensions($u, v$) produces a sequence of dimensions which specifies a shortest path from $u$ to $v$.*

To perform routing of messages in a distributed fashion, each processor must be capable of performing "local" routing, whereby a processor contributes to the routing of a message by forwarding it to a processor one step closer to its destination. This decision can be made by inspecting an appropriate pair of bits in the source and destination labels, as in the sequential algorithm. In essence, each processor performs the loop of the routing algorithm as previously described. However, rather than simply emit some dimension $d$, the message is forwarded to the processor situated at the $d$-neighbor of the given processor; this fulfills the obligation of the processor in question. As a practical consideration, in order to avoid rescanning bits, the values of $\ell$ and $k$ used in the sequential algorithm can be passed along with the source and destination.

It is also worth noting that the routing algorithm presented has a certain bias in the way the dimensions are produced. There are, however, many other ways in which the routing can be performed. In particular, notice that a pair of strings $\alpha$ and $\beta$ with $\alpha \not\sim \beta$ is sought by the loop of EmitDimensions in a left-to-right scan. This scanning order is more specific than necessary—it is sufficient to find *any* such pair to the right of the leftmost differing bit, so long as there are an even number of bits to their right. Of course, if no such pair exists, routing along the dimension of the leftmost differing bit completes the routing.

# CONCLUSIONS

We have shown how to construct a multiply-twisted cube which has many of the properties of the hypercube, but has diameter only about half as large. This network is self-routing, in the sense that there is a simple distributed routing algorithm which guarantees optimal paths between any pair of vertices. This fact, together with other properties such as regularity, symmetry, high connectivity, and a simple recursive structure, suggests that the multiply-twisted cube may be an attractive alternative to the ordinary hypercube for massively parallel architectures.

# REFERENCES

[1] Becker, B. and H. Simon (1988), "How robust is the $n$-cube?" *Information and Computation,* 77, 162–178.

[2] Chan, T. and Y. Saad (1986), "Multigrid algorithms on the hypercube multiprocessor," *IEEE Trans. Comput.*, C-35, No. 11, 969–979.

[3] Erdös, P. and J. Spencer (1979), "Evolution of the $n$-cube," *J. of Comp. and Math with Appl.*, 5, 33–39.

[4] Esfahanian, A., L. Ni, and B. Sagan (1988), "On enhancing hypercube multiprocessors," *Proc. 1988 International Parallel Processing*, 86–89.

[5] Folds, S. (1977), "A characterization of hypercubes," *J. of Discrete Math.*," 155–159.

[6] Harary, F. (1972), *Graph Theory*, Addison-Wesley.

[7] Hart, S. (1976), "A note on the edges of the $n$-cube," *J. of Discrete Math.*," 157-163.

[8] Hillis, W. (1985), *The Connection Machine*, MIT Press.

[9] Mulder, M. (1980), "$N$-cube and median graphs," *J. of Graph Theory*, 4, 107–110.

[10] Pease, M. (1977), "The indirect binary $n$-cube microprocessor array," *IEEE Trans. Comput.*, C-26, No. 5, 458–473

[11] Ranka, S., Y. Won and S. Sahni (1988), "Programming a hypercube multicomputer," *IEEE Software*, Sept., 69–77.

[12] Saad, Y. and M. Schultz (1988), "Topological properties of hypercubes," *IEEE Trans. Comput.*, C-37, No. 7, 867-872.

[13] Seitz, C. (1985), "The Cosmic Cube," *Comm. ACM*, 28, 22-33.

[14] Shiau, T., P. Blackwell, and K. Efe. (1988), "Multiply-twisted $n$-cubes for parallel computing," *Proc. of 20th Symp. on the Interface: Computing Science and Statistics*, to appear.

[15] Slough, W., K. Efe, T. Shiau, and P. Blackwell (1988), "A Reduced Diameter Interconnection Network,", *Technical Report*, Computer Science Department, University of Missouri-Columbia.

# THE PSMH : A PYRAMID OF FRACTIONAL DIMENSION

Jean Hecquard    Raj Acharya
Electrical Engineering & Computer Science Department
SUNY at Buffalo, Bell Hall Amherst NY 14260

## Abstract

This paper presents a new Interconnection Network, the Partitionable Spanning Multibus Hypercube ( PSMH ). Our aim is to provide an architecture for applications with spatially distributed data. The data are embedded on the PSMH by hierarchical data structures, the $n$-dimensional binary trees. To reduce the scope of this article, we concentrate on a special case of the PSMH, designed for the manipulation and display of 3D objects. Based on the PSMH features, as well as on the properties of the data structure, we first derive a routage algorithm, then show how we can pack the data, to reduce the PSMH size.

**Keywords:** image processing, pyramid, 3D.

## 0. INTRODUCTION

From an architectural point of view, image processing tasks can be divided into 3 levels, a lower one, a higher one and an intermediate one. In the low level, we perform operations such as thresholding or convolution, on an often very large amount of data, organized in a very regular and structured manner. This class of problems is clearly best processed by SIMD architectures. On the other hand, high level processing, i.e pattern recognition, involves a much smaller set of data, for which no unique data dependency graph can be easily found. As a consequence, this processing stage frequently uses a very dense Interconnection Network ( IN ), such as the binary Hypercube or even crossbar, with a set of more powerful processors operating in MIMD mode. Between these 2 processing stages, the transition is much less understood : no clear scheme has emerged until now. Here again we can distinguish 2 main directions. The first one emphasizes the SIMD approach. An example of this is given by the CAPP ( Content Addressable Parallel Processor [7]). This system is a 3 level pyramid in which the bottom is a mesh dedicated to low level processing, while an intermediate level is constructed with a reduced mesh of more powerful processors, with the top being a small set of symbolic processors. Clearly this scheme has the advantage of simplicity and speed. However, its lack of flexibility may waste a lot of processing elements ( PEs ), which becomes redhibitory in higher dimension. Another possible direction is to consider a flexible network, such as the binary Hypercube ( see for instance [3] ). The problem here is reversed. Instead of wasting PEs, this solution may waste links, as the cube's dimension is higher than the problem's dimension. The aim of the PSMH is to find a trade-off between these 2 extreme cases. The plan of this article is as follow. After a brief definition of the PSMH ( §1 ), we present a basic routage algorithm in §2. We then show in §3, how we can compact the data on the PSMH, to reduce its size. As a consequence of this compaction, we finally introduce a neighbor finding technique ( §4 ).

## 1. DEFINITIONS

We define a **PSMH**, as a $D$-dimensional lattice of width $w$ in each dimension, with $N = W^D$ nodes. Each node is connected to $B$ buses over each dimension, but may select only 2 out of the $D \times B$ available ones. One is dedicated to reception, while the other is in charge of the emission. Both emission and reception can take place simultaneously. Each bus is a bidirectional wire, with no controller. To the usual global control scheme is substituted a local scheme, where each PE is in charge of its segment of the bus. A node can either receive from the left or the right part of a bus, and partition it or not. This definition leads to a **network diameter** of $D$, meaning that any 2 PEs can exchange messages in $\Omega(D)$ time. This is useful for operations where we need a rapid propagation of an information to a PE. However, when all the data may be potentially moving, this bound doesn't hold anymore. We get a new bound by using a **wire-cutting argument** [6]. If we slice the PSMH in the middle, we have $\frac{W^D}{2}$ PEs, i.e potential senders, on each side, while only $W^{D-1} \times B$ available buses. This gives us a new bound of $O(\frac{W}{B})$.

As a by-product of the bus partitionability, the PSMH heavily rely on divide-and-conquer methods. This naturally leads us to the choice of hierarchical data structures to embed the data. Among them, the $n$-dimensional binary trees ( or $2^n$-ary trees ) have been developed with the aim of obtaining a systematic way to represent $n$-dimensional objects. To reduce the scope of this article, we concentrate on the 3 dimensional case with the linear octree [1] ( cf figure 1 for the Linear Octree of a planar object ). In the rest of this paper we will consider a PSMH of dimension $D = 2$. This choice is of interest because it opens the way for an easy VLSI as well as WSI implementation. To embed our linear octree onto the mesh, we use the shuffle row major order, i.e the linear quadtree order ( cf figure 2 ).

Research supported by grants from Thomson ( France ), Bell Aerospace, AHA ( NYS ) and NIH BRSG.

CH2649-2/89/0000/0475$01.00 © 1988 IEEE

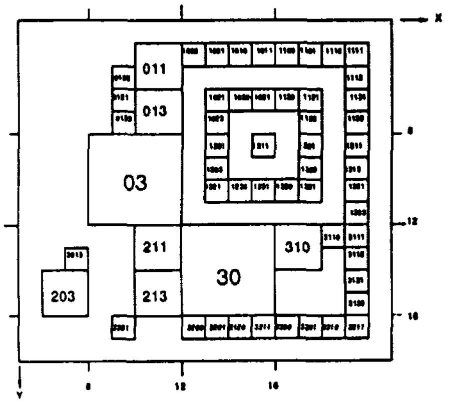


Fig. 1 Example of Linear Octree

For the sake of simplicity of presentation, we first assume that each voxel get a PE. This will be changed to one obel per PE in §3 for the list compaction.

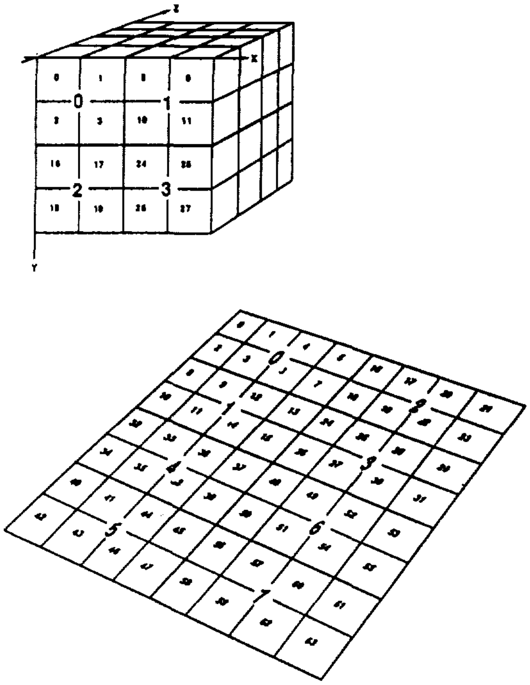


Fig. 2 Embedding on the Mesh

## 2. ROUTAGE ALGORITHM

Our aim is now to get the basic operations of a regular pyramid on the PSMH. Clearly, any semigroup operation can be performed on the PSMH with the same cost as on a pyramid, by using the recursive doubling [4]. Another basic requirement is to be able to shift the data by 1 in any direction. In other words, we want to send a message from one PE to its neighbor at distance 1 in direction $x$, $y$, or $z$. Although the wire-cutting argument of §1 tells us that a sorting/routage algorithm should take $O(\frac{w}{B})$, the cost of this operation will be much less because the data are sorted. The principle of the algorithm is to partition the data in disjoint sets, and to perform the routage within each of these sets in parallel. Following the definition of the linear octree, such sets can be easily created by 'unshuffling' the obels. The result is what we call the **octal planes**. An octal plane of level $i$ for the axis $axis$, is the set $\Pi(i, axis) = \{O \in \Lambda \ / \ (A_1 \wedge A_2)\}$, with $A_1 : (axis_i = 1, n_8 \geq i \geq 1)$, $A_2 : (axis_k = 0, i > k \geq 1)$, where $\Lambda$ is a linear octree of resolution $n_8$ and $axis_k$ is the $k^{th}$ bit of the $axis$-coordinate of an obel $O$. $\Pi(i, axis)$ is the set of planes which are perpendicular to $axis$ and whose distance to the origin on this axis is equal to $(2k+1)2^{i-1}$. For instance the $\Pi(1, axis)$ is the set of planes at distance $2k+1$ of the origin over their respective axis. They correspond to the middle planes of all the octants of volume $8^1$ ( cf figure 3 ).

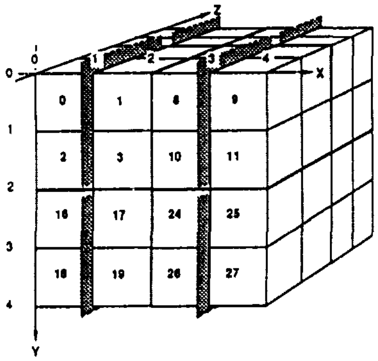


Fig. 3 Octal Plane $\Pi(1, x)$

The algorithm proceeds one octal plane at a time. For each octal plane of level $i$, all the octant of volume $8^i$ can perform their internal communications, i.e through the middle plane, simultaneously, as each octant corresponds to an independent set of PEs. Now that we have partitioned the data, what we have left is to route the data within each octant. We operate in bottom-up/top-down fashion. The reason for this choice is twofold. Firstly this divide-and-conquer approach will allow a partition of the buses. Secondly, by progressively moving the data we get more and more buses to work. The algorithm is as follows : first, we recursively sort/balance the load over the octant. Starting with the suboctants of volume $8^1$, we sort the data in raster order, then balance them so that each row gets the same number of data. The reason for sorting is that it facilitates balancing. We then repeat this with the suboctants of volume $8^2$, and so on, until the whole octant is balanced. Once at the top, we go down unsorting/unbalancing, but according to the destination index. The details of the routage will not be given here. The interested reader may find the whole algorithm in [2]. Because we have at most $4^i$ active PEs within each octant of volume $8^i$, the cost is minimal if we have $B = N^{\frac{1}{3}}$ buses per row/column. The algorithm is then of order $O(\log^3(N))$.

## 3. A PYRAMID OF FRACTIONAL DIMENSION

Following the routage, we want to compact the list to reduce the PSMH size. In fact, an interesting property of the octree is that the number of obels is proportional to the object's surface [5]. This means that we should be able to expect a significant decrease in the PSMH size if we compact the list. The algorithm is mainly dictated by the constraints from the previous routage algorithm. They are twofold. Rule 1: no octant should interfere with another one, so that we can partition the buses. Rule 2: the load has to be well balanced over the mesh. This last requirement, although not vital , is important as it will minimize the bottlenecks, therefore maximizing the system's performances.

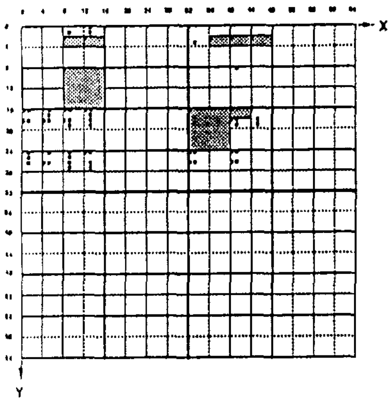


Fig. 4 figure 1 octree's embedding

To take advantage of the compaction, we will now use only one PE per obel. Clearly this is not a problem when a bigger obel needs to communicate with a group of smaller neighbors. The other way is less easy, as 1 PE may have to handle several incoming messages. The solution is to make the neighbors cooperate to present only one message. This cooperation is easy because these neighbors form an interval. We define as interval a set of PEs which are consecutive for a given order traversal. In the present case, we consider the raster order which we get from the bottom-up phase of, for instance, the routage algorithm. In the neighbor's case, the interval consists in fact of more than the neighbors, but the other PEs are inactive and can be bypassed by the buses. The problem is then reduced to either a broadcast or a semigroup operation within the interval.

The compaction algorithm is composed of 2 parts. We first compute the new position of each obel and then move them to destination using a modified routage algorithm. The computation of the new positions is done in bottom-up/top-down fashion. Starting from the bottom of the octree, i.e with the octants of volume $8^1$, we try to merge their suboctants. Two suboctants can be merged if (a) the receiver has enough PEs left, (b) its communication quota is not exceeded. This quota is the number of PEs that may communicate simultaneously per octant. As we proceed one direction at a time, it corresponds to the surface of an octant's side, i.e $4^i$. Because we do not actually move the data until the second part, merging only means that we add to the communication requirements. Once level $i$ is done, we go up to level $i + 1$, and so on. It should be noticed that, to enforce Rule 1, each suboctant has to be considered as an unbreakable entity: we can't merge part of it. The cost of part 1 is of order $O(\log(N))$.

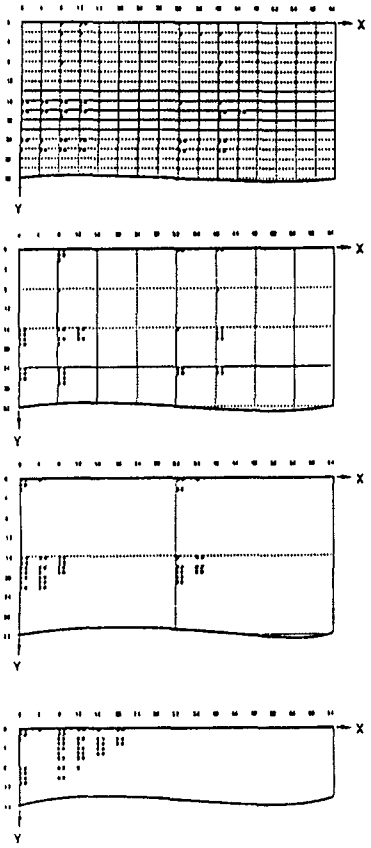


Fig. 5 figure 1 octree's packing

We give an example of packing in figures 4 and 5. The figure 4 represents the embedding of figure's 1 octree on a $64 \times 64$ mesh. The dashed areas correspond to the PEs which would have been used with the original assumption of 1 PE/voxel. The 4 meshes of figure 5 are, from top to bottom, the level 1, 2, 3 and 4 of the packing. For the sake of presentation we show the updated positions at each level although the real algorithm waits until the end to move. The dotted lines delimit the octants of volume, respectively, $8^1$, $8^2$, $8^3$ and $8^4$, i.e also the bus segments. Once each obel knows its final destination, the obels are routed to their final destination with a slightly modified version of the previous algorithm. The basic difference is that we do not work with one octal plane at a time, but consider the whole mesh. Then, while we still sort/balance the data in the first phase, we only unsort in the second phase. Because we may have up to $N$ data to move, we get a cost of order $O(N^{\frac{1}{3}}\log^2(N))$ with $B = N^{\frac{1}{6}}$ buses per row.

## 4. NEIGHBOR FINDING

After the compaction, the obels are still in shuffle row major order. However, we can't get their position by a simple computation. Hence we have to perform a neighbor finding operation before we can route any data. We now present an algorithm that performs this search for all the obels in parallel. For the same reasons as previously, we work one octal plane at a time, so that each octant can proceed independently. The principle of the search is to reverse the usual binary search approach. While the binary search goes through a list, looking for one value at a time, our search makes each PE broadcast its value on a bus in turn, while the others are listening. For the sake of simplicity we assume that we have half as many available buses as we have of active PEs. The idea is to start with the median PE, which broadcasts its value on the median bus. Because all the other PEs are listening, each of them now know where to look the next time. Then the median PEs of the 2 halves broadcast their values, and so on. After log steps, each PE will have found what it was looking for.

The neighbor finding algorithm is as follows : we proceed in 4 stages for each octal plane. Let $\Lambda_R$ and $\Lambda_S$ be respectively the receivers and senders list for one octant of volume $8^i$. The first stage consists of sorting/balancing the octant like for a routage. After this, both lists are spread over the octant, with at most $N^{\frac{1}{6}}$ active PEs on each row. The second part performs a preliminary search by asking the first active PE of $\Lambda_R$ on each row to go on the diagonal and broadcast its value on the column in turn, following the binary search pattern defined above. The result is that all the PEs of $\Lambda_S$ get to know on which row their neighbor should be in $O(\log(N))$ time. The third stage is to send this group of obels on the row oi each $\Lambda_S$'s PE, then to perform a search among the group. This is always possible because we may have at most $O(N_o^{\frac{1}{4}})$ groups of $N_o^{\frac{1}{4}}$ values to be sent on each row of width $w_o = N_o^{\frac{1}{2}}$ (where $N_o$ is the surface of the octant). Finally the fourth part unsorts/unbalances in order to send the result back to the original PEs. The global cost of this algorithm is the same as for the routage, i.e of order $O(\log^3(N))$, with the assumption of $B = N^{\frac{1}{6}}$ buses.

## 5. CONCLUSION

We have presented a new Interconnection Network, the Partitionable Spanning Multibus Hypercube. Its aim is to offer a cost efficient solution to problems with spatially distributed data. To reduce the scope of this article we concentrated on the 3-dimensional case. We introduced a divide-and-conquer routage algorithm which takes advantage of the existing data ordering to reduce the algorithm cost. Because a static allocation map wastes the PEs, we derived a compaction algorithm which respects the constraint of the routage algorithm. As a consequence we derived a neighbor finding technique, based on broadcasting techniques. Most of the operations that need to be performed on the PSMH take either a constant, logarithmic or polylogarithmic time, if we have $B = N^{\frac{1}{6}}$ buses per row/column.

## REFERENCES

[1] **Gargantini I. 1982**
Linear octrees for fast processing of 3 dimensional objects.
CGIP 20,4 (Dec) 365-374

[2] **Hecquard J. and Acharya R. 1988**
The Partitionable Spanning Multibus Hypercube
Technical Report, SUNY at Buffalo, ECE Dept.

[3] **Hillis D. 1985**
The Connection Machine
MIT Press

[4] **Kogge P.M. and Stone H.S. 1973**
A Parallel Algorithm for efficient solution of a general class of recurrence equations
IEEE Trans. Computers, vol C-22(8), pp 786-793.

[5] **Meagher D. 1982**
Geometric modeling using octree encoding.
CGIP 19,2 (June) 129-147.

[6] **Miller R. and Stout Q.F. 1988**
Parallel Algorithms for regular architectures
MIT Press: Series on Scientific Computation.

[7] **Weems C.C., Levitan S.P., Hanson A.R., Riseman E.M. 1987**
The image understanding architecture
University of Massachusetts at Amherst, CS Dept.
COINS TR 87-86.

# ROUTING LINEAR PERMUTATIONS THROUGH THE OMEGA NETWORK IN TWO PASSES

John Keohane and Richard E. Stearns

Computer Science Department
State University of New York at Albany
Albany, New York 12222

## ABSTRACT

The problem of routing permutations through an $\Omega$ network connecting a set of processors is studied in the framework of linear algebra. The class of linear permutations is defined, and it is shown that any linear permutation can be routed through the $\Omega$ network in two passes. Furthermore, the address of the intermediary processor for the routing can be found in $O(n^4)$ time, where n is the size of the address of a processor. The class of linear permutations contains the class of Bit Permute Complement permutations, and the address of the intermediary processor for routing Bit Permute Complement permutations can be found in O(n) time.

## 1. INTRODUCTION

Lawrie [3] has proposed the $\Omega$ network, which can be used as a vehicle for establishing communication among a set of $N = 2^n$ processors, where each processor is identified with a unique n-bit binary address. The $\Omega$ network enables a source processor to establish communication with any other processor, which is called the destination processor, and we consider the case in which each processor acts as both a source processor and a destination processor. The communication pattern may be viewed as a permutation of the addresses of the processors, where the address of each source processor maps to the address of its destination processor.

Parker [6] showed that any permutation can be realized in three passes through the $\Omega$ network. The Benes network [3,4] consists of a reverse $\Omega$ network followed by an $\Omega$ network. Parker [6] also showed that any permutation can be realized in only one pass through the Benes network; however, the Benes network has twice the hardware as the $\Omega$ network. We show that by finding the addresses of intermediary processors, the class of linear permutations can be realized in only two passes of the $\Omega$ network. An important subclass of the linear permutations is the Bit Permute Complement (BPC) permutations studied by Nassimi and Sahni [5] and by Yew and Lawrie [7]. The class of BPC permutations includes three of Lenfant's five families of frequently used permutations [4]. An algorithm for routing BPC permutations through the $\Omega$ network also appears in [7]; however, ours is distinct in both its technique and results. Moreover, if n is the number of bits in an address, our algorithm finds n formulas for the bits of the address of the intermediary processor in time linear in n. Our algorithm for the more general class of linear permutations is $O(n^4)$, and no such algorithm has appeared in the literature. Furthermore, our approach lies in linear algebra, in contrast to that used in [6] and [7].

## 2. THE $\Omega$ NETWORK

The $\Omega$ network for $N = 2^n$ processors consists of n stages of N/2 switches. Each switch has two inputs and two outputs, so there are N wires entering and leaving each stage. Each collection of N wires may be labeled with n vector of *n* bits so that the following conventions hold: for source processor $s_1s_2...s_n$ to communicate with destination processor $d_1d_2...d_n$, the message must be put on wire $s_{i+1}s_{i+2}...s_nd_1d_2...d_i$ after the i-th stage of switches. For source processor $s_1s_2...s_n$ and destination processor $d_1d_2...d_n$, we define the ***i-th window*** as the n bits $s_{i+1}s_{i+2}...s_nd_1d_2...d_i$. Clearly, there is a conflict between two source-destination pairs for the same wire if and only if for some i, $1 \leq i \leq n$, their i-th windows are identical. Thus, a permutation is conflict-free if and only if for each i, $1 \leq i \leq n$, the i-th window of each source-destination pair is unique.

**Lemma 1.** If a permutation P' is obtained from a permutation P by complementing a subset of the bits of the addresses of all destination processors, then P is conflict-free if and only if P' is conflict-free.

**Proof.** For $1 \leq i \leq n$, the i-th windows of two source-destination pairs of P are identical if and only if the i-th windows of the two source-destination pairs of P' are identical for the same two source processors.

## 3. LINEAR PERMUTATIONS

In this section we define the classes of linear permutations and Bit Permute Complement permutations and place the problem of routing them through the $\Omega$ network in the framework of linear algebra. In what follows, + denotes exclusive-or. Note that + is associative.

CH2649-2/89/0000/0479$01.00 © 1988 IEEE

**Definition.** Let $s_1s_2...s_n$ denote the address of a source processor, and let $d_1d_2...d_n$ denote the address of its destination processor. A *linear permutation* is a permutation in which for $1 \leq i \leq n$, each $d_i = s^*_{j_{i,1}}+s^*_{j_{i,2}}+...+s^*_{j_{i,k_i}}$, where $s^*_{j_{i,t}}$ denotes either $s_{j_{i,t}}$ or its complement $\overline{s}_{j_{i,t}}$.

**Definition.** Let $s_1s_2...s_n$ denote the address of a source processor, and let $d_1d_2...d_n$ denote the address of its destination processor. Let $\pi$ be a permutation of $\{1,2,...,n\}$. A *Bit Permute Complement (BPC) permutation* is a permutation in which for $1 \leq i \leq n$, each $d_i = s^*_{\pi(i)}$, where $s^*_{\pi(i)}$ denotes either $s_{\pi(i)}$ or its complement $\overline{s}_{\pi(i)}$.

**Definition.** Let $s_1s_2...s_n$ denote the address of a source processor, and let $d_1d_2...d_n$ denote the address of its destination processor. An *uncomplemented linear permutation* is a permutation in which for $1 \leq i \leq n$, each $d_i = s_{j_{i,1}}+s_{j_{i,2}}+...+s_{j_{i,k_i}}$.

**Definition.** Let $s_1s_2...s_n$ denote the address of a source processor, and let $d_1d_2...d_n$ denote the address of its destination processor. Let $\pi$ be a permutation of $\{1,2,..,n\}$. A *Bit Permute (BP) permutation* is a permutation in which for $1 \leq i \leq n$, each $d_i = s_{\pi(i)}$.

Let P be a linear permutation in which for $1 \leq i \leq n$, $d_i = s^*_{j_{i,1}}+s^*_{j_{i,2}}+...+s^*_{j_{i,k_i}}$. We define its *uncomplemented version* to be the permutation P′ in which for $1 \leq i \leq n$, $d_i = s_{j_{i,1}}+s_{j_{i,2}}+...+s_{j_{i,k_i}}$.

**Theorem 1.** A linear (BPC) permutation is conflict-free if and only if its uncomplemented version is conflict-free.

**Proof.** Let P be a linear permutation in which for $1 \leq i \leq n$, each $d_i = s^*_{j_{i,1}}+s^*_{j_{i,2}}+...+s^*_{j_{i,k_i}}$. Because $\overline{x} = x+1$, $1+1 = 0$, and $x+0 = x$ and because exclusive-or is associative and commutative, each $d_i$ can be rewritten as either $s_{j_{i,1}}+s_{j_{i,2}}+...+s_{j_{i,k_i}}$ or $s_{j_{i,1}}+s_{j_{i,2}}+...+s_{j_{i,k_i}}+1$. Thus, P may be obtained from its uncomplemented version P′ by complementing a subset of the bits of the addresses of all destination processors. By Lemma 1, P is conflict-free if and only if P′ is conflict-free.

Thus, to show that all linear permutations can be routed through the Ω network in some number of passes, we need only show that the class of uncomplemented linear permutations can be routed in that many passes. The same observation holds true for BPC and BP permutations.

It can be seen that the variables $s_1,s_2,...,s_n$ and the constant zero combined with exclusive-or is the vector space of those variables and the constant zero over the field of integers modulo two. Exclusive-or is both associative and commutative. Furthermore, zero functions as the identity, and each vector is its own inverse.

Because the variables $s_1,s_2,...,s_n$ and the constant zero form a vector space under exclusive-or, the class of conflict-free uncomplemented linear permutations can be characterized in terms of linear independence. Let $s_1s_2...s_n$ denote the address of a source processor, and let $d_1d_2...d_n$ denote the address of its destination processor, where each $d_i$ is a linear combination of the $s_j$'s. If $D_n = \{d_1,d_2,...,d_n\}$ is linearly independent, then for each actual destination address, the bits of the address of its unique source processor may be recovered. If, however, $D_n$ is linearly dependent, more than one source processor must have the same destination processor. Thus we have a permutation if and only if $D_n$ is linearly independent. For $1 \leq i \leq n-1$, let $D_i = \{s_{i+1},s_{i+2},...,s_n,d_1,d_2...,d_i\}$. For $1 \leq i \leq n-1$, if $D_i$ is linearly independent, we can determine the bits of the address of a unique source processor using a given wire after the i-th stage. If for some i, $1 \leq i \leq n-1$, $D_i$ is linearly dependent, then the i-th windows of at least two source-destination pairs are identical. Hence an uncomplemented linear permutation is conflict-free if and only if for $1 \leq i \leq n$, $D_i$ is linearly independent.

## 4. A TWO-PASS ROUTING ALGORITHM FOR LINEAR PERMUTATIONS

We present a two-pass routing algorithm for the class of uncomplemented linear permutations, which implicitly provides an algorithm for routing all linear permutations in two passes. Let $s_1s_2...s_n$ denote the address of a source processor, and let $d_1d_2...d_n$ denote the address of its destination processor under some uncomplemented linear permutation. We present an algorithm for finding the address of an intermediary processor $z_1z_2...z_n$ such that the permutations $s_1s_2...s_n \rightarrow z_1z_2...z_n$ and $z_1z_2...z_n \rightarrow d_1d_2...d_n$ are both conflict-free. For $1 \leq i \leq n$, $z_i$ is either $s_i$ or $s_i+s_j$ for some $s_j$. For $1 \leq i \leq n$, let $Z_i = \{s_{i+1},s_{i+2},...,s_n,z_1,z_2,...,z_i\}$, and let $D_i = \{z_{i+1},z_{i+2},...,z_n,d_1,d_2,...d_i\}$. The permutation $s_1s_2...s_n \rightarrow z_1z_2...z_n$ will be conflict-free if and only if, for $1 \leq i \leq n$, $Z_i$ is linearly independent. Furthermore, if $D_i$ is linearly independent, we can determine the unique address of the original source processor $s_1s_2...s_n$ using a given wire after the i-th stage. Thus, if $Z_n$ is linearly independent, we can find the unique address of the intermediary processor $z_1z_2...z_n$ using the wire. If, however, $D_i$ is linearly dependent, there are at least two original source processors that must use the same wire during the second pass through the network. Thus, the two permutations are conflict-free if and only if for $1 \leq i \leq n$, $Z_i$ and $D_i$ are linearly independent.

The process of determining each $z_i$ is accomplished by examining $D_i$. Since $s_1s_2...s_n \rightarrow d_1d_2...d_n$ is a permutation, $D_n$ is linearly independent. At each stage we find $z_i$ such that $D_{i-1}$ is linearly independent if $D_i$ is linearly independent. We finally find $z_1$ such that $Z_n$ is linearly independent if $D_1$ is linearly independent. For $1 \leq k \leq n-i$, let $v_k = z_{i+k}$, and for $1 \leq k \leq i$, let $v_{n-i+k} = d_k$. Thus, $D_i = \{v_j \mid 1 \leq j \leq n\}$. If $\{s_i,v_1,v_2,...,v_{n-1}\}$ is linearly independent, then $z_i = s_i$. If, however, $\{s_i,v_1,v_2,...,v_{n-1}\}$ is linearly dependent, then $z_i = s_i+s_j$, where $s_j$ is one of the terms appearing in $v_n = d_i$. The choice of $s_j$ is governed by Theorem 2 below.

**Lemma 2.** Let $V = \{v_1,v_2,...,v_{n-1}\}$ be linearly independent, and let $W = V \cup \{w\}$ be linearly dependent. There exist $c_i$, $1 \leq i \leq n-1$, such that

$$w = \sum_{i=1}^{n-1} c_i v_i.$$

**Proof.** Since W is linearly dependent, there exist $c_i$, $1 \leq i \leq n$, such that at least one $c_i \neq 0$ and $c_n w + \sum_{i=1}^{n-1} c_i v_i = 0$. Since V is linearly independent, $c_n = 1$. Thus, $w = \sum_{i=1}^{n-1} c_i v_i$.

**Theorem 2.** Let $V = \{v_1, v_2, \ldots, v_n\}$, where $v_n = s_{t_1} + s_{t_2} + \ldots + s_{t_k}$. Let $V' = \{s_i, v_1, v_2, \ldots, v_{n-1}\}$, and for $1 \leq j \leq k$, let $W_j = \{s_i + s_{t_j}, v_1, v_2, \ldots, v_{n-1}\}$. If V is linearly independent and V′ is linearly dependent, then for some j, $1 \leq j \leq k$, $W_j$ is linearly independent.

**Proof.** Assume that for $1 \leq j \leq k$, $W_j$ is linearly dependent. Since any subset of V is linearly independent, we have that by Lemma 2, for $1 \leq j \leq k$, there exist $c_{j,m}$, $1 \leq m \leq n-1$, such that $s_i + s_{t_j} = \sum_{m=1}^{n-1} c_{j,m} v_m$. Consider

$$\sum_{j=1}^{k} (s_i + s_{t_j}) = \sum_{m=1}^{n-1} \left( \sum_{j=1}^{k} c_{j,m} \right) v_m.$$

However,

$$\sum_{j=1}^{k} (s_i + s_{t_j}) = \sum_{j=1}^{k} s_i + \sum_{j=1}^{k} s_{t_j} = \sum_{j=1}^{k} s_i + v_n.$$

If k is even, $\sum_{j=1}^{k} s_i = 0$. Thus, we have $v_n = \sum_{m=1}^{n-1} \left( \sum_{j=1}^{k} c_{j,m} \right) v_m$, and the linear independence of V is contradicted. If k is odd, $\sum_{j=1}^{k} s_i = s_i$, and we have

$$s_i + v_n = \sum_{m=1}^{n-1} \left( \sum_{j=1}^{k} c_{j,m} \right) v_m$$

or

$$v_n = s_i + \sum_{m=1}^{n-1} \left( \sum_{j=1}^{k} c_{j,m} \right) v_m.$$

Because any subset of V is linearly independent and V′ is linearly dependent, by Lemma 2 there exist constants $d_m$, $1 \leq m \leq n-1$, such that $s_i = \sum_{m=1}^{n-1} d_m v_m$. Therefore, $v_n = \sum_{m=1}^{n-1} (d_m + \sum_{j=1}^{k} c_{j,m}) v_m$, and the linear independence of V is again contradicted. Thus, at least one $W_j$, $1 \leq j \leq k$, is linearly independent.

We have just shown that if $D_n$ is linearly independent, then $D_i$, $1 \leq i \leq n-1$, and $Z_n$ are linearly independent. We now show that for $2 \leq i \leq n$, if $Z_i$ is linearly independent, then $Z_{i-1}$ is linearly independent. We first need the following lemma.

**Lemma 3.** Let $V = \{v_1, v_2, \ldots, v_{i-1}, s_i + s_j, v_{i+1}, \ldots, v_n\}$, where $j \neq i$ and for $1 \leq k \leq n$ and $k \neq i$, either $v_k = s_k$ or $v_k = s_k + s_{m_k}$ for some $s_{m_k} \neq s_k$. Let $V' = \{v_1, v_2, \ldots, v_{i-1}, s_i, v_{i+1}, \ldots, v_n\}$. If V is linearly independent, then V′ is linearly independent.

**Proof.** We show that if V′ is linearly dependent, there must exist an infinite, nonrepeating sequence of vectors $v_{x_m}$, $m \geq 1$, where $x_m \neq i$ and $v_{x_m} = s_{x_m} + s_{x_{m+1}}$, with $x_1 = j$. Assume V′ is linearly dependent. Since any subset of V is linearly independent, by Lemma 2 there exist $c_k$, $1 \leq k \leq n$ and $k \neq i$, such that $s_i = \sum_{\substack{k=1 \\ k \neq i}}^{n} c_k v_k$. The remainder of the proof proceeds by induction on m.

**Basis** (m=1). Either $v_j = s_j$ or $v_j = s_j + s_{x_2}$ for some $s_{x_2}$. If $v_j = s_j$, then $s_i + s_j = v_j + \sum_{\substack{k=1 \\ k \neq i}}^{n} c_k v_k$, and the linear independence of V is contradicted. Thus, $v_j = s_j + s_{x_2}$. Furthermore, $x_2 \neq i$; otherwise, the linear independence of V is again contradicted.

**Induction.** Assume there is a nonrepeating sequence $v_{x_k} = s_{x_k} + s_{x_{k+1}}$ for $1 \leq k \leq m-1$, where $x_k \neq i$ for $1 \leq k \leq m$. Note that

$$\begin{aligned}
\sum_{k=1}^{m-1} v_{x_k} &= \sum_{k=1}^{m-1} (s_{x_k} + s_{x_{k+1}}) \\
&= \sum_{k=1}^{m-1} s_{x_k} + \sum_{k=1}^{m-1} s_{x_{k+1}} \\
&= \sum_{k=1}^{m-1} s_{x_k} + \sum_{k=2}^{m} s_{x_k} \\
&= s_{x_1} + s_{x_m} \\
&= s_j + s_{x_m}.
\end{aligned}$$

Either $v_{x_m} = s_{x_m}$ or $v_{x_m} = s_{x_m} + s_{x_{m+1}}$ for some $s_{x_{m+1}}$. If $v_{x_m} = s_{x_m}$, then $s_i + s_j = \sum_{\substack{k=1 \\ k \neq i}}^{n} c_k v_k + \sum_{k=1}^{m} v_{x_k}$, and the linear independence of V is contradicted. Thus, $v_{x_m} = s_{x_m} + s_{x_{m+1}}$. Furthermore, if $x_{m+1} = i$, then $s_i + s_j = \sum_{k=1}^{m} v_{x_k}$, and the linear independence of V is again contradicted.

Assume $v_{x_m} = v_{x_{m-i}}$ for some i, $1 \leq i \leq m-1$. Then $x_m = x_{m-i}$, and $s_{x_m} = s_{x_{m-i}}$. Hence, we have

$$\begin{aligned}
\sum_{k=m-i}^{m-1} v_{x_k} &= \sum_{k=m-i}^{m-1} (s_{x_k} + s_{x_{k+1}}) \\
&= s_{x_{m-i}} + s_{x_m} \\
&= s_{x_m} + s_{x_m} \\
&= 0.
\end{aligned}$$

Again, the linear independence of V is contradicted.

**Theorem 3.** If $Z_i = \{s_{i+1}, s_{i+2}, \ldots, s_n, z_1, z_2, \ldots, z_i\}$ is linearly independent, then $Z_{i-1} = \{s_i, s_{i+1}, \ldots, s_n, z_1, z_2, \ldots, z_{i-1}\}$ is linearly independent.

**Proof.** Either $z_i = s_i$ or $z_i = s_i+s_j$ for some $s_j \neq s_i$. If $z_i = s_i$, there is nothing to prove. If $z_i = s_i+s_j$, then Lemma 3 applies with $v_k = z_k$ for $1 \leq k \leq i-1$ and $v_k = s_k$ for $i+1 \leq k \leq n$.

**Theorem 4.** The vectors $z_i$, $1 \leq i \leq n$, can be found in $O(n^4)$ time.

**Proof.** The linear independence of n vectors can be determined by Gaussian elimination in $O(n^2)$. For each of the n vectors $z_i$, at most $n-1$ tests for linear independence must be made.

## 5. A TWO-PASS ROUTING ALGORITHM FOR BPC PERMUTATIONS

We present a two-pass routing algorithm for BPC permutations. As mentioned before, we need only demonstrate that the class of BP permutations can be routed in two passes. A similar algorithm appears in [7], but ours is distinct and O(n). The algorithm of the previous section also applies to this case, but that algorithm is $O(n^2)$. Let $s_1s_2...s_n$ denote the address of a source processor, and let $d_1d_2...d_n$ denote its destination processor under some BP permutation, where $d_i = s_{\pi(i)}$ for some permutation $\pi$ of $\{1,2,...n\}$.

**Definition.** An integer i *starts a cycle* of the permutation $\pi$ if $\pi^k(i) \geq i$ for all $k \geq 1$.

Again we find the address of an intermediary processor $z_1z_2...z_n$ such that the permutations $s_1s_2...s_n \rightarrow z_1z_2...z_n$ and $z_1z_2...z_n \rightarrow d_1d_2...d_n$ are conflict-free. For $1 \leq i \leq n$, $z_i = s_i$ if i starts a cycle of $\pi$, and $z_i = s_i+s_{\pi(i)}$ if i does not start a cycle of $\pi$. Again, let $D_i = \{z_{i+1},z_{i+2},...,z_n,d_1,d_2,...,d_i\}$, and let $Z_i = \{s_{i+1},s_{i+2},...,s_n,z_1,z_2,...,z_i\}$. Since for $1 \leq i \leq n$, $d_i = s_{\pi(i)}$, $D_n$ is linearly independent. We again show that for $2 \leq i \leq n$, $D_{i-1}$ is linearly independent if $D_i$ is linearly independent and that $Z_n$ is linearly independent if $D_1$ is linearly independent. For $2 \leq i \leq n$, the linear independence of $Z_{i-1}$ follows from the linear independence of $Z_i$ by Theorem 3.

Let $V = \{v_j \mid 1 \leq j \leq n\}$, where $v_j = d_j = s_{\pi(j)}$ for $1 \leq j \leq i-1$, $v_i = s_i$, and for $i+1 \leq j \leq n$, $v_j = s_j$ if j starts a cycle of $\pi$ and $v_j = s_j+s_{\pi(j)}$ if j does not start a cycle of $\pi$.

**Lemma 4.** If i does not start a cycle of $\pi$, then V is linearly dependent.

**Proof.** See [2].

**Lemma 5.** If i starts a cycle of $\pi$ and $V - \{s_i\} \cup \{d_i\}$ is linearly independent, then V is linearly independent.

**Proof.** See [2].

**Theorem 5.** For $2 \leq i \leq n$, if $D_i$ is linearly independent, then $D_{i-1}$ is linearly independent, and if $D_1$ is linearly independent, then $Z_n$ is linearly independent.

**Proof.** See [2].

**Theorem 6.** The vectors $z_i$ can be found in O(n) time.

**Proof.** Given $\pi$, the integers that start cycles in $\pi$ can be found in O(n) time. Once the integers that start cycles are found, each of the n $z_i$'s can be found in constant time.

## 6. CONCLUSIONS

We have studied routing the class of linear permutations through the $\Omega$ network within the framework of linear algebra. Each linear permutation can be routed through the $\Omega$ network in two passes, and a formula for each bit of the address of the intermediary processor can be found in $O(n^4)$ time. Furthermore, for the subclass of BPC permutations, a formula for each bit of the address of the intermediary processor can be found in O(n) time. It remains an open problem to show that either any permutation can be routed through the $\Omega$ network in two passes or there is a permutation that requires three passes to be routed through the $\Omega$ network.

## REFERENCES

[1] G. Birkhoff and S. MacLane, *A Survey of Modern Algebra*. New York: MacMillan, 1970.

[2] J. Keohane and R.E. Stearns, "Routing Linear Permutations Through the Omega Network in Two Passes", Technical Report 88-14, Computer Science Department, SUNY Albany, 1988.

[3] D.H. Lawrie, "Access and alignment of data in an array processor", *IEEE Trans. Comput.*, vol. C-24, pp. 1145-1155, Dec. 1975.

[4] J. Lenfant, "Parallel permutations of data: a Benes network control algorithm for frequently used permutations", *IEEE Trans. Comput.*, vol. C-27, pp. 637-647, July 1978.

[5] D. Nassimi and S. Sahni, "A self-routing Benes network and parallel permutation algorithms", *IEEE Trans. Comput.*, vol. C-30, pp. 332-340, May 1981.

[6] D.S. Parker, "Notes on shuffle/exchange-type switching networks", *IEEE Trans. Comput.*, vol. C-29, pp. 213-222, March 1980.

[7] P.C. Yew and D. Lawrie, "An easily controlled network for frequently used permutations", *IEEE Trans. Comput.*, vol. C-30, pp. 296-298, April 1981.

# PERFORMANCE OF THE ASP
# ON THE DARPA ARCHITECTURE BENCHMARK

A Krikelis, R M Lea

Aspex Microsystems Ltd
Brunel University
Uxbridge, United Kingdom, UB8 3PH

## ABSTRACT

The Associative String Processor (ASP) is a homogeneous, reconfigurable and programmable, massively parallel processor which offers step-function advantages in cost-performance and application flexibility, due to its unique architecture and its exploitation of state-of-the-art microelectronics. This paper briefly describes the ASP architecture, its implementation and reports the results of an evaluation of its applicability to image processing tasks. In order to provide a realistic demonstration of the above-mentioned advantages, a set of independently defined such tasks (viz. the DARPA Image Understanding benchmark) was chosen for the evaluation and the results are used to compare the performance of the ASP architecture with the performances of other parallel computer architectures when applied to the same computer vision tasks.

## INTRODUCTION

Comparing different parallel processing architectures is a very difficult task. Most commercial purveyors promote their machines by quoting only the most favourable performances. Moreover, analysis of parallel algorithms and systems shows that there are always overheads, detracting from performance, which are rarely quoted. Indeed, it is commonly accepted that users can expect parallel processors to provide a speed-up of only O(logN), where N is the number of processing elements.

The field of image processing in general, and computer vision in particular, provides a strong incentive for massively parallel processors; due to the large data volume, high data-rate and algorithmic complexity of its computational tasks. Indeed, researchers, involved in the areas of algorithm and system development for real-time image understanding, need high performance which is easy to use (including programming) and cost-effective. Not surprisingly, therefore, the image processing workers were among the first to attempt the establishment of a realistic benchmark for massively parallel processors. Early computer vision benchmarking attempts included the Abingdon Cross problem (1982) and the Tanque Verde benchmark suite (1984).

A more recent attempt to construct a computer vision benchmark emerged from the DARPA Image Understanding community in 1986, when the University of Maryland defined a set of representative low and intermediate-level vision tasks [1]. High-level vision (such as recognition) were not included, because it was felt that proposed algorithms were too ill-defined to properly evaluate parallel architectures.

The benchmark was intended to achieve an initial understanding .of the general strengths and weaknesses, for computer vision applications, of the growing number of parallel computer architectures and to project the need for future development of parallel architectures to support this field.

This DARPA benchmarking activity has been the most successful to date. Moreover, a second DARPA Image Understanding benchmark suite, based on the experience of the first benchmark and defined by the University of Massachusetts in collaboration with the University of Maryland, has been recently announced [2].

This paper reports the results of the evaluation of the Associative String Processor (ASP) [3], a massively parallel processor emerging from research at Brunel University and being developed by Aspex Microsystems Ltd., for the first DARPA benchmark.

## THE BENCHMARK

The first DARPA Image Understanding benchmark, defined in reference [1] and discussed in reference [4], includes the following computer vision tasks.

A. Edge detection within a 512 x 512 pixel image

   A1. 11 x 11 Laplacian
   A2. zero crossing detection
   A3. border following.

B. Connected component labelling within a 512 x 512 pixel image.

C. Hough transform computation within a 512 x 512 pixel image.

D. Geometrical constructions for a set of 1000 planar points.

CH2649-2/89/0000/0483$01.00 © 1988 IEEE

D1. convex hull
D2. Voronoi diagram
D3. minimum spanning tree.

E. Visibility for a set of 1000 opaque triangles in 3-D space.

F. Graph navigation

F1. Finding subgraphs of a given graph (100 vertices, each with 10 edges) that are isomorphic to another given graph (30 vertices, each with 3 edges).

F2. Finding the minimum cost path between two vertices of an edge-weighted graph (1000 vertices, each with 100 edges).

## ASP ARCHITECTURE

As indicated in Figure 1, an ASP system [3] comprises a dynamically reconfigurable parallel processing structure of communicating ASP sub-strings, each supported with an ASP Data Buffer (ADB), an ASP Controller and an ASP Data Communications Network.

Each ASP substring is a parallel processing computational structure, comprising a string of identical APEs (Associative Processing Elements), as shown in Figure 2. Each APE is connected to an Inter-APE Communication Network (which runs in parallel with the APE string). All APEs share common bit-parallel Data, Activity and Control Busses and a single feedback line (Match Reply, MR), which are maintained by an external ASP Controller, which also maintains the LinK Left and LinK Right ports (LKL and LKR) of the Inter-APE Communication Network.

Each APE incorporates an n-bit Data Register and an a-bit Activity Register, an (n+a)-bit parallel Comparator, where the values of n and a are in the ranges 32-128 and 4-8 respectively, depending on the application class for which the ASP is optimised. Moreover, the APE includes a single-bit full-adder, 4 status flags (viz. C to represent arithmetic Carry, M and D to tag Matching and Destination APEs and A to activate selected APEs) and control logic for local processing and communication with other APEs.

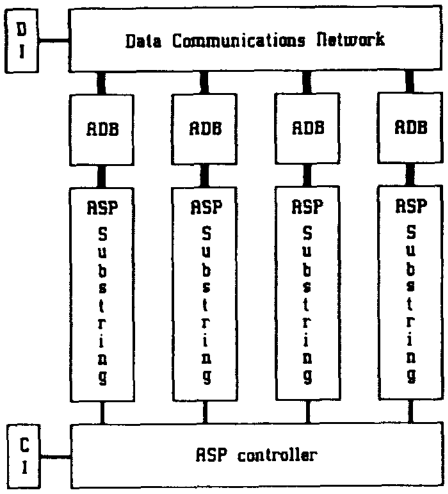


Figure 1. ASP system

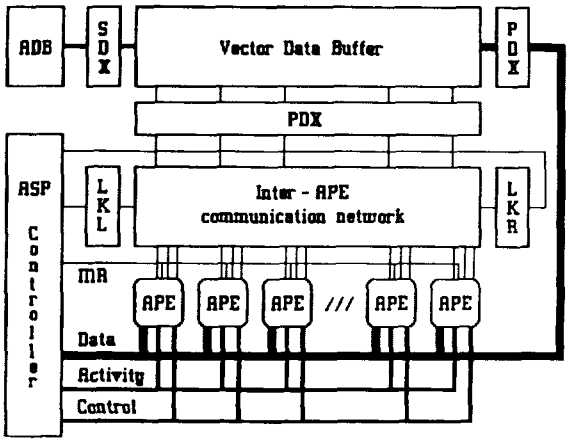


Figure 2. ASP substring

In operation, each ASP substring supports a form of set processing, in which the sub-set of active APEs (i.e. those which match broadcast data and activity values) support scalar-vector and vector-vector operations. The Match Reply (MR) line indicates whether none or some APEs match. Matching APEs are either directly activated or source inter-APE communications to indirectly activate other APEs.

Scalar data are directly broadcast or received by the ASP controller via the bit-parallel Data Bus. Input-output vector data could also be exchanged (viz. output dumped and input loaded in a single step) APE-sequentially via the Data Bus with the bit-parallel Primary Data eXchanger (PDX) shown in Figure 2. However, the Vector Data Buffer supports a much faster APE-parallel exchange facility, in which the bit-serial Primary Data eXchanger (PDX) performs the task at a very high data rate, thereby minimising loss of parallel processing efficiency. Similarly, but at a lower data-rate, the Secondary Data eXchanger (SDX) provides a bit-parallel vector data exchange between the Vector Data Buffer and the external ASP Data Buffer (ADB), which is overlapped with parallel processing and, therefore, does not present a sequential processing overhead.

The Inter-APE Communication Network implements a globally-controlled and dynamically-reconfigurable tightly-coupled APE interconnection strategy, which supports cost-effective emulation of common network topologies with two modes of inter-APE communication:

circuit-switching: asynchronous bi-directional single-bit communication via multiple signal paths, dynamically configured (programmer-transparently) to connect APE sources and corresponding APE destinations of high-speed activation signals, implementing a fully-connected permutation and broadcast network for

APE selection and inter-APE routing functions

packet-switching: synchronous bi-directional multi-bit communication via a high-speed bit-serial shift register, routing M-tag patterns along each APE substring, for data/message transfer.

In order to preserve continuity at the two ends of the Inter-APE Communication Network, the LKL and LKR (shown in Figure 2) allow activation or M-bit signals to be injected and sensed by the external ASP controller and act as the left and right neighbours of the leftmost and rightmost APE in the associative string processor respectively.

## ASP IMPLEMENTATION

The ASP concept is particularly well matched to both the opportunities and constraints of VLSI chip fabrication; owing to its high APE packing density, its highly compact inter-APE communications network and, especially, because its I/O requirement is independent of the string length. The feasibility of a 256-APE VLSI ASP chip was demonstrated in 1986 and Aspex Microsystems are developing 256-APE VLSI ASP chips for ASP substring implementation [3].

Moreover, the ASP is highly amenable to defect/fault-tolerance; owing to its construction from a large number of identical APEs, lack of location-dependent addressing and simple inter-APE interconnection. Consequently, as reducing feature-sizes and increasing chip sizes drive VLSI chip fabrication technology towards the prospect of ULSI chips and WSI devices, the ASP architecture offers consistency and becomes increasingly more cost-effective. Indeed, research at Brunel University has indicated the potential integrating complete ASP systems with 2,048-APE ULSI chips and 8,192-APE WSI ASP devices.

## EVALUATION

In practice, ASP system configurations may be tailored to suit application requirements; the minimum number of APEs being 256 (see above) and the maximum being limited by implementation cost.

Two ASP system configurations were chosen for the DARPA benchmark evaluation [4]:

DARPA ASP: Number of ASP substrings = 512
Number of APEs per substring = 512
Data Register storage = 96 bits
Activity Register storage = 5 bits
Implementation complexity
32 WSI ASP devices
or 128 ULSI ASP chips
or 1,024 VLSI ASP chips plus data communication network, ADB and ASP controller boards
Clock rate = 20 MHz

ASP: whichever ASP system configuration offers optimum performance for the particular benchmark task.

Assuming a clock rate of 20 MHz, the benchmark evaluation results for these configurations are reported in Table 1.

Comparison of the results of Table 1 with those of Tables 2 through 4 indicates the consistency of the performance advantage of the ASP. It is interesting to note the superiority of the two associative architectures. Unfortunately, the DARPA benchmark neglects volume and cost factors, for which highly-compact 1000MOPS/$1000 ULSI/WSI ASPs would excel.

## OTHER ARCHITECTURES

Brief details of other parallel computer architectures which have been evaluated according to the first DARPA Image Understanding benchmark are given below [4].

Medium-grain MIMD multiprocessors (see Table 2 for the reported benchmark results)

- ♦ BB&N BUTTERFLY: 128 shuffle-exchange connected PEs; each 16 MIPS PE comprising a Motorola 68020 32-bit microprocessor, an AMD-2901 bit-slice processors for memory management, a custom-designed VLSI switch circuit supporting 32 Mbits/sec inter-PE communication and up to 4Mbytes of local memory with a 64Mbytes/sec bandwidth. The multiprocessor incorporates 512 I/O channels, each supporting a data rate of 16Mbits/sec operating with a 16MHz clock.

- ♦ Caltech CUBE and MOSAIK: 256 and 16,384 hypercube connected PEs; each 8 MIPS PE comprising Intel 80286/80287 32-bit microprocessors and up to 4.5Mbytes of local memory with a 32Mbytes/sec bandwidth. The multiprocessors are designed to operate with an 8MHz clock.

Medium-grain systolic arrays (see Table 3 for the reported benchmark results)

- ♦ CMU WW-WARP and PC-WARP: 10 linearly connected PEs; each 10 MFLOPS PE comprising a wire-wrapped or printed-circuit board 255 chip implementation of input queues, crossbar, 32-bit processing elements (including a floating point processor), register files, 32 Mbytes of local memory (with an 80Mbytes/sec bandwidth), address generator and microengine. Inter-PE communication PEs can be achieved at 80 Mbytes/sec and the I/O data rate is 40Mbytes/sec. The linear arrays are controlled by an interface unit, comprising 264 chips, which can communicate with the host system through two I/O clusters, based on Motorola 68020 microprocessors. The arrays are designed to operate with an 20 MHz clock.

- ♦ CMU iWARP: 72 linearly connected PEs; each 16 MFLOPS PE being targetted for integration on a single chip being developed in collaboration with Intel Corporation.

Fine-grain SIMD array processors (see Table 4 for the reported benchmark results)

- ♦ Columbia NON-VON: up to 1M tree-connected PEs; each PE being based on a custom-designed VLSI 1-bit (or 8-bit in version 3) array processor chip. In version 3, the tree communication network also incorporates mesh interconnection between its leaves. The array processor is designed to operate with a 10MHz clock.

- ♦ Thinking Machines CONNECTION MACHINE: 65,536 mesh connected PEs; each PE comprising a 1-bit ALU, 8 status flags and 4Kbits of local memory with a 4Mbits/sec bandwidth. 16 PEs are implemented with a custom-designed VLSI chip incorporating a 4 x 4 processor array and one router of a 16Kbytes/sec packet-switching hypercube-connected communications network (overlaying the mesh and implemented separately) and 4 16K-bit static RAM chips. The array can support an I/O data rate of 30Kbits/sec/channel operating with a 10MHz clock.

Fine-grain MIMD/SIMD array processor (see Table 4 for the reported benchmark results)

- ♦ UMass IUA (Image Understanding Architecture): 266,304 three-level mesh connected PEs; comprising a 64 x 64 array of custom-designed VLSI CAAPP (Content Addressable Array Parallel Processor) chips implementing a 512 x 512 array of 1-bit PEs (each incorporating 320 bits of local memory, with a 1Mbits/sec bandwidth) at the lowest level, a 64 x 64 array of Texas TMS 320 16-bit DSP chips at the intermediate level and an 8 x 8 array of Motorola 68020 32-bit microprocessors at the highest level. Overlaying the mesh connected communication network, the Coterie Network allows high speed communication between remote processing elements. Data I/O (at the lowest level) and communication between the three processing levels is achieved through dual-port video RAMs. The entire array processor is designed to operate with a 10MHz clock.

| | DARPA ASP | ASP |
|---|---|---|
| A1 | 0.80 ms | 0.70 ms |
| A2 | 3.9 us | 0.7 us |
| A3 | 4.1 us - 5.5 us | 0.9 us - 5.2 us |
| B | 6.5 us - 2.4 ms/cmp | 6.5 us - 2.4 ms/cmp |
| C | 140 ms | 140 ms |
| D1 | 2.4 ms | 0.2 ms - 0.4 ms |
| D2 | 11 ms | 11 ms |
| D3 | 82 ms | 82 ms |
| E | 0.1 - 128 ms | 0.1 - 128 ms |
| F1 | 0.68 ms/iteration | 0.68 ms/iteration |
| F2 | 15 us - 15 ms | 15 us - 15 ms |

Table 1. Specified ASP system and optimum configuration for each task

| | Butterfly | CUBE | MOSAIK |
|---|---|---|---|
| A1 | 2.9 sec | 100 ms | 2.5 ms |
| A2 | ? | ? | ? |
| A3 | ? | ? | ? |
| B | 7.2 sec | 14 ms | 6 ms |
| C | 7.4 sec | 1.8 sec | 10 ms |
| D1 | ? | ? | ? |
| D2 | ? | ? | ? |
| D3 | ? | ? | ? |
| E | 4.2 sec | ? | ? |
| F1 | ? | ? | ? |
| F2 | ? | 10 ms | 1 ms |

Table 2. Medium-grain MIMD multiprocessors

| | WW Warp | PC Warp | iWarp |
|---|---|---|---|
| A1 | 430 ms/367 ms | 350 ms | 7.8 ms |
| A2 | 170 ms/179 ms | 50 ms | 7.8 ms |
| A3 | n/a | 1.1 sec | 690 ms |
| B | 5.6 sec | 980 ms | 470 ms |
| C | n/a / 2 sec | 340 ms | 60 ms |
| D1 | 9 ms/ 18 ms | 9 ms | 4.3ms |
| D2 | n/a | 290 ms | 140 ms |
| D3 | n/a /160 ms | 160 ms | 43 ms |
| E | 830 ms/400 ms | 400 ms | 40 ms |
| F1 | n/a | 1800 s/sec | 19000 s/sec |
| F2 | 1.4 sec | 69 ms | 25 ms |

Table 3. Medium-grain systolic arrays

| | Non-Von | CM1 | IUA |
|---|---|---|---|
| A1 | 2 ms | 3 ms | 0.2 ms |
| A2 | ? | ? | ? |
| A3 | ? | ? | 0.2 ms |
| B | 1 sec | 400 ms | 8 us |
| C | 400 ms | 700 ms | 27 ms |
| D1 | ? | 200 ms | 15 ms |
| D2 | ? | ? | 50 ms |
| D3 | 40 ms | 2.2 sec | 0.4 ms/124 ms |
| E | 100 ms | 1 sec | 70 ms/290 ms |
| F1 | ? | ? | ? |
| F2 | 40 ms | 50 ms | 1 ms |

Table 4. Fine-grain SIMD array processors

## REFERENCES

[1] A. Rosenfeld, 'A report on the DARPA Image Understanding Architectures workshop', Proc. DARPA Image Understanding workshop, Los Angeles CA, February 1987, pp.298-301.

[2] C. Weems, E. Riseman, A. Hanson & A. Rosenfeld, 'An integrated image understanding benchmark: recognition of a 2½D mobile', Proc. DARPA Image Understanding workshop Vol. 1, Cambridge MA, April 1988, pp. 111-126.

[3] R. M. Lea, 'The ASP: a cost-effective parallel microcomputer', IEEE Micro, October 1988, pp. 10-29.

[4] 'Performance of the ASP on the DARPA architecture benchmark', Aspex Microsystems Tech. Memo., May 1988.

# SIMULATION AND ANALYSIS OF ENHANCED SWITCH ARCHITECTURES FOR INTERCONNECTION NETWORKS IN MASSIVELY PARALLEL SHARED MEMORY MACHINES

Yue-sheng Liu and Susan Dickey
liuys@nyu.edu, dickey@nyu.edu

Ultracomputer Research Laboratory
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York, NY 10012
(212) 998-3339

*Abstract*

Differences in switch architecture can have a significant effect on both latency and throughput in interconnection networks. We assess improvements in performance which can be obtained by adding buffers to a crossbar switch, by changing the configuration of the buffers, and by adding the capability of combining messages to the buffers.

Four basic $k \times k$ crossbar switch types are described: unbuffered; $k$-input buffers, one per output port; one-input buffers, one per input port; and one-input buffers, $k$ buffers per output port. We review previous analytical work and simulation studies of some of these switch types and extend the analysis to the other types. An analytical model for simple "hot spot" traffic is presented and simulation results are shown for different kinds of message combining.

*Keywords: Interconnection Network, Crossbar Switch, Shared Memory Multiprocessor*

## INTRODUCTION

The basic system model under consideration is $N$ independent processing elements (PEs) connected to $M$ shared memory modules (MMs) through a logarithmic interconnection network. Examples of this architecture include the NYU Ultracomputer [4], the BBN Butterfly [20], and the IBM RP3 [19].

In this model, traffic through the network consists of *requests* from PEs to MMs and *responses* from MMs to PEs. Requests and responses are typically sent as *messages* of at most a few hundred bits, divided into *packets* of the same size as the data path width from switch to switch within the network. Packets within the same message are pipelined.

Overall system performance for very large numbers of processors critically depends on the message throughput that can be achieved by the interconnection network. In practice, message throughput is limited not only by the theoretical bandwidth of the network but by its latency, the time from the generation of a request until a response is received. Though processors can be designed to tolerate some latency [6], a processor will eventually be unable to generate new requests until it has received the response to some previous request.

*Delta* networks [15] connect $N = a^n$ PEs to $M = b^n$ MMs via an $n$-stage network composed of $a \times b$ switches. In this paper we study latency and throughput of square delta networks, with $M = N$ and $a = b = k$; in particular we will concentrate on *omega* networks (a sub-class of square delta networks with perfect shuffle connections [11]) composed of $2 \times 2$ switches.

Using the fetch-and-add synchronization primitive, the NYU Ultracomputer project has developed operating system and applications software designed to avoid critical sections and scale to thousands of processors [3][5]. Good network performance in the presence of synchronization "hot spots" is required for the efficient implementation of such code. We have studied ways of combining messages as they traverse the network to avoid a hardware critical section at memory and to prevent the degradation of performance in the entire network [18].

## NETWORK SIMULATOR

Networks composed of $2 \times 2$ switches under various assumptions of switch architecture, PE request generation and memory behavior are modeled in the simulator. Output statistics include the average request latency, average bandwidth, average queue length per stage and total number of combines that occur.

Switches with 2 two-input buffers, 4 one-input buffers and 2 one-input buffers can all be simulated. Buffer size at switches can be varied. Different clear-to-send protocols can be tested, and different combining alternatives corresponding to different hardware implementations can be specified.

PE request generation can be modeled in two ways: (1) generate a request with some fixed probability whenever permitted by the first stage of the network or by the capacity of a finite PE request queue; (2) generate a request with some fixed probability whenever a PE "wants to," which requires the simulation of an infinite request queue at the PE. In the first case, the *offered load* is the probability of generating a request on any cycle when the PE is not blocked. The *effective throughput* will be less than the offered load. In the second case, messages accumulate latency in the request queue at the PE, but as long as that queue reaches a steady state, the effective throughput will be equal to the offered load.

Our results show little difference between the two models of PE request generation, except when the network is very congested. With uniform traffic and a heavy offered load, both latency and effective throughput are greater with an infinite request queue. When hot spot requests induce congestion, a system modeled with a finite PE request queue, operating at maximum offered load, will show increased latency without a corresponding increase in effective throughput once the buffers are larger than a certain size [2]. With an infinite PE request queue, the same throughput can be achieved at a much lower offered load; in this case, latency decreases as switch buffer size increases. A system imposed limit on the PE's maximum request rate might be desirable in a real system, to avoid loading the network above the capacity at which it functions well.

PE request generation can also simulate a processor which is allowed only a small number of requests outstanding before it must quit generating requests.

Memory behavior is modeled with both a cycle and an access time. The cycle time is used to determine whether or not the memory will accept a message; the access time is used in computing total latency from processor to memory. Cycle and access times may vary for loads, stores and other operations. The intervention of

This work was supported in part by the Applied Mathematical Sciences Program of the U.S. Department of Energy under contract DE-AC02-76ER03077, in part by the National Science Foundation under grant DCR-8413359, and in part by I.B.M. under joint study agreement N00039-84-R-0605(Q).

CH2649-2/89/0000/0487$01.00 © 1988 IEEE

the memory makes delays on the forward and return paths asymmetric, and must be included for accurate system modeling.

## SWITCH ARCHITECTURES

Consider a $k \times k$ crossbar switching component in a delta network. Its basic function is to forward messages from any of its $k$ inputs to any of its $k$ outputs. It may include buffers to hold messages in case of conflicts for the output ports or blocking from later stages. These buffers may be associated with either input or output ports.

In this section, performance comparisons of the different switch configurations are based on a traffic model in which addresses are uniformly distributed among the MMs, and the interarrival time of requests at the first stage of the network is geometrically distributed. The network is assumed to be an $N \times N$ square delta network, composed of $k \times k$ switches, with $\log_k(N) = n$ stages.

In the simplest unbuffered switch design (see Figure 1a), a protocol must be used to kill messages in case of conflict. Lost messages must be retransmitted. The probability of an output at a switch in the $i^{th}$ stage is $p_i = 1-(1-p_{i-1}/k)^k$, where $p_0 = p$ is the offered load on an input port to the network [15]. This can be approximated by $p_n = 2k/((k-1)n + 2k/p)$ (see [7]). Thus for a square delta network with $N$ PEs and $N$ MMs, the throughput at each output port of the network is $O(1/logN)$, holding $k$ and $p$ constant. The overall bandwidth of the network is then $O(N/logN)$.

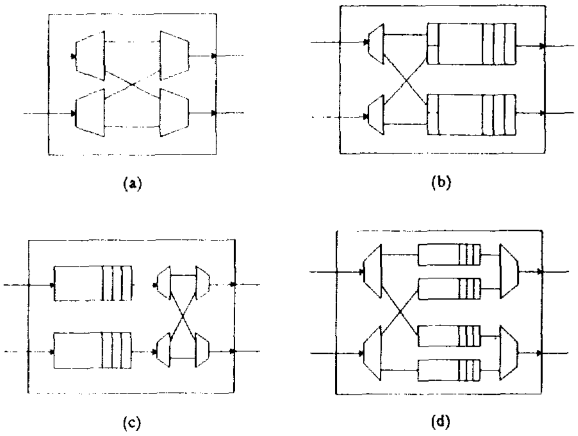


Figure 1. (a) Unbuffered Switch. (b) $k$-Input Buffers, One per Output Port. (c) One-Input Buffers, One per Output Port. (d) $k$-Input Buffers, $k$ per Output Port.

The latency of a message, measured from the time a processor makes a request until it is satisfied, is difficult to estimate because of retransmission. Suppose the processor can actually generate requests at a rate $b$, independent of any responses it receives. Over time, if $b$ is less than the maximum bandwidth of the network, retransmissions will accumulate until the offered load on the network $p$ minus the rate $r$ of rejected messages gives the desired $b$. At this point, the offered load should stabilize at $p = b + r$. For a given $b = p_n$ the output probability equation for stage $n$ can be solved for $p$, $b/p$ will give the probability of a message being accepted, and $p/b$ the average number of trials until it is accepted. If we assume that the transit time in switch cycles of a message accepted by the network is $n + m - 1$, where $n$ is the number of stages and $m$ is the number of packets per message, and that a rejected message is retransmitted after twice this amount of time (when no response or a negative acknowledgement is received), then the expected value of the round trip latency is $2 \times (p/b) \times (n + m - 1)$.

A hardware buffer capable of accepting $k$ inputs in one cycle can be used to construct a switch with one buffer per output port (see Figure 1b). This is the type of switch that has been most thoroughly analyzed in the literature, especially in [8]. According to that analysis (see also [17]), if the queues at each switch may grow without bound, ("infinite buffers") then the average switch delay at the first stage is $1 + m^2p(1-1/km) / 2(1-mp)$ where $m$ is the number of packets in a message, and the average switch delay at subsequent stages is approximated by the expression $1 + (1 + 4mp / 5k)(m^2p(1-1/k) / 2(1-mp))$. The initial 1 corresponds to the time required for a message to be transmitted through a switch without being queued (the switch service time). The average network traversal time (in one direction) is the sum of the individual stage delays plus the setup time for the pipe, i.e. $(m-1)$.

Note that the network has a capacity of $1/m$ messages per switch cycle per PE. That is, each PE cannot enter messages at a rate higher than one per $m$ cycles, and, conversely, the network can accommodate any traffic below this threshold. Thus, the global bandwidth of the network is theoretically proportional to the number of PEs connected to it.

Using the above formulas, Table 1 compares the analytical performance predictions for a 1024 PE network containing unbuffered switches with a network containing switches with $k$-input buffers. The advantage of buffered switches increases for heavier loads.

| Request | Unbuffered | | Buffered | |
|---|---|---|---|---|
| rate | 2 × 2 | 4 × 4 | 2 × 2 | 4 × 4 |
| .1 | 26.7 | 12.3 | 21.1 | 10.5 |
| .2 | 40.0 | 12.3 | 22.5 | 11.2 |
| .3 | - | 22.8 | 24.5 | 12.1 |
| .4 | - | - | 27.2 | 13.4 |
| .8 | - | - | 69.5 | 31.6 |

Table 1. Round-trip Latency. 1024 PEs, analytical predictions, buffered and unbuffered switches, different crossbar sizes.

The configuration using $k$-input buffers is somewhat difficult to realize in hardware. For a simpler hardware implementation, one-input buffers, one per input port, may be used. Outputs of the buffers are multiplexed, and a buffer may be blocked on output by another buffer (see Figure 1c). In [9] this arrangement was called "buffers between the switches." Recent work [16] has shown that the effective throughput of this type of switch cannot exceed .75 messages per cycle per port, but that for rates under .75, the queues are stable and output rate equals input rate. Recurrence relations for the queue length probabilities have also been developed, and have been used to compute the data in Table 2.

| Switch | Request rate | | | |
|---|---|---|---|---|
| architecture | .2 | .4 | .6 | .8 |
| (b) and (d) | .012 | .067 | .225 | .788 |
| (c) | .014 | .094 | 1.044 | |

Table 2. Expected Queue Length. Single stage, "infinite buffers," labels as in Figure 1.

Using only one-input buffers, the performance of the $k$-input buffers may be approximated using $k$ buffers per output port (see Figure 1d). A packet leaves an output port whenever any of the $k$ associated buffers has data; if more than one has data, arbitration must occur. In [9], this configuration was called "buffers within the switches" and showed better performance than the "buffers between the switches," especially for high load.

By analyzing the $k$ buffers associated with an output port as a single queueing system with a service rate of 1 per cycle, the expected waiting time in the first stage can be seen to be the same as that of the $k$-input buffer. However, the service discipline within the queueing system is no longer first come, first serve, so the variance of the waiting time may differ. This in turn may affect the result at later stages.

Table 2 compares analytical results for the three types of buffered switches, showing the expected queue length at the first stage, assuming "infinite buffers."

Simulations for this section were carried out with one packet messages, a large buffer size per output port (40 packets), and no delay at memory. We use a finite PE request queue (in the simulations below, of size 0) as a more realistic model, since actual processors cannot to generate requests indefinitely when blocked. All results given here are for a 1024 PE omega network composed of 2 × 2 switches.

Figure 2 shows the latency of different buffered switch types at different effective throughputs. As in the analytical results for a single stage, switches (b) and (d) show equivalent delay, while (c) is not as good, especially at higher loads. Even in a system where the average load is expected to be light, the switch types with better performance at heavier loads may make the network more robust to bursty traffic.

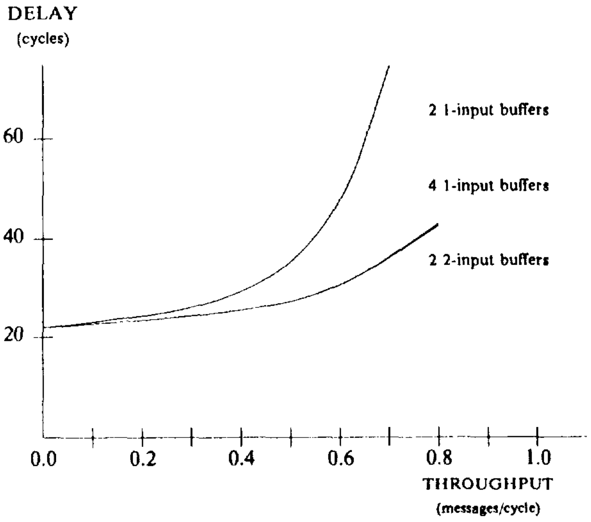


Figure 2. Comparison of Three Buffered Switch Architectures. Simulation data, 1024 PEs, single-packet messages, buffer size 40 packets.

## "HOT SPOT" TRAFFIC AND COMBINING

The above results assume that memory reference patterns are perfectly uniform. As shown in [10], "hot spot" traffic can be particularly damaging in a buffered network. We develop a model of hot spot traffic which gives an estimate of the maximum amount of such traffic that can be handled for large numbers of processors, describe how combining memory requests can be used to mitigate the effects of hot spots, and show simulation results for different combining alternatives.

### A model for single "hot spot" traffic

Suppose each PE issues two types of requests: hot spot requests directed to a particular MM and other requests uniformly distributed among all $N$ MMs. Assume the uniform requests have request rate $W$, and the hot spot requests rate $R$. Each PE issues $p = W + R$ requests per cycle. The paths of the hot spot requests produce a traffic tree rooted at the hot MM, and spanning all the PEs as leaves.

In [14] it is shown that each switch node in a delta network with a single hot spot has the same traffic distribution on all its inputs. We denote the input probabilities as $p_{j,i-1}$ where $i$ is the stage of the node, and $j$ is the number of stages the traffic at this input has conflicted with traffic directed to the hot spot.

For an unbuffered network, if the switch node is in the "hot" traffic tree, the probability that a "hot" output has a message is:

$$p_{j,j} = 1 - [(1 - r_{j-1}p_{j-1,j-1} - (1 - r_{j-1})p_{j-1,j-1}/k]^{k.}$$

and the probability a "cool" output has a message is:

$$p_{j-1,j} = 1 - [1 - (1 - r_{j-1})p_{j-1,j-1}/k]^{k.}$$

where,

$$r_j = \frac{Rk^j}{Rk^j + W} \quad j = 0,1,\ldots,n.$$

For nodes not in the "hot" traffic tree,

$$p_{i,j} = 1 - (1 - p_{i,j-1}/k)^{k.}$$

Take a $4^5 \times 4^5$ delta network as an example. At the outputs of the last stage, traffic can be divided into 6 classes: $p_{5,5}$, the "hot" traffic, one output only; $p_{4,5}$, traffic that conflicts with the "hot" traffic for 4 stages, and has 3 outputs; $p_{3,5}$, 3 stages conflict, 12 outputs; $p_{2,5}$, 2 stages conflict, 48 outputs; $p_{1,5}$, 1 stage conflict, 192 outputs; and $p_{0,5}$, traffic that has no conflict with the "hot" traffic, 768 outputs.

The effects of "hot" traffic on the other five classes of traffic are shown in Table 3, with throughput under uniform traffic as a comparison.

| Offered | Traffic class | | | | | | |
|---|---|---|---|---|---|---|---|
| load | $p_{0,5}$ | $p_{1,5}$ | $p_{2,5}$ | $p_{3,5}$ | $p_{4,5}$ | $p_{5,5}$ | Uniform |
| .2 | .139 | .137 | .131 | .110 | .061 | 1.00 | .144 |
| .4 | .216 | .212 | .196 | .148 | .066 | 1.00 | .222 |
| .6 | .263 | .257 | .233 | .165 | .067 | 1.00 | .269 |
| .8 | .294 | .285 | .255 | .173 | .067 | 1.00 | .299 |
| 1.0 | .315 | .305 | .269 | .177 | .067 | 1.00 | .320 |

Table 3. Throughput in $4^5 \times 4^5$ Unbuffered Delta Network. 5% hot spot.

For buffered networks, assuming infinite buffers, the network is non-blocking and every request is able to get into the network. If the network has a stable state for a given offered load,

$$p_{i,i} = W + Rk^i, \quad i = 0,1,\ldots,n$$

and

$$p_{j,i} = W, \qquad i > j,\ i = 1,2,\ldots,n.$$

For the network to be stable, we must have

$$W + NR \le 1.$$

which severely restricts the amount of hotspot traffic which can be handled without combining for large N.

### Combining messages

The NYU Ultracomputer project has proposed combining fetch-and-add operations as well as loads and stores at the switches [1][4]. The fetch-and-add operation, useful as a synchronization primitive and in many parallel algorithms, is an indivisible add to memory; its format is F&A($X$,$e$), where $X$ is an integer variable and $e$ is an integer expression. The operation is defined to return the (old) value of $X$ and to replace $X$ by the sum $X + e$.

When two fetch-and-adds referencing the same shared variable, say F&A($X$, $e$) and F&A($X$, $f$), meet at a switch, the switch forms the sum $e + f$, transmits the combined request F&A($X$,

$e+f$), and stores the value $e$ in its local memory. When the value $Y$ is returned to the switch in response to F&A($X$, $e+f$), the switch returns $Y$ to satisfy one request, F&A($X$, $e$), and $Y+e$ to satisfy the other, F&A($X, f$).

Since combined requests can themselves be combined, any number of concurrent memory references to the same location can be satisfied in the time required for one shared memory access from a single PE.

## Different Combining Strategies

The NYU Ultracomputer's current hardware design uses a systolic implementation of "pairwise combining," in which a message may combine with only one other message at any given stage. Other researchers [12][13] have shown that for very large interconnection networks, pairwise combining may not be sufficient to prevent network degradation due to hot spots. They suggest "$k$-way combining," where $k=3$ means a message may combine with two other messages at a stage. According to simulations in [13], 3-way combining has performance almost as good as unlimited combining.

Our own simulations show acceptable performance for two-way combining, even for systems with 1024 processors. A simple extension of our current systolic queue design will give "two-and-a-half way" combining, in which a message may combine with two other messages as long as they each come from different input ports at that stage. Table 4 compares no combining with pairwise, two-and-a-half way, and 3-way combining.The results for 3-way combining assume larger return path queues, as in [12].

| Offered | Throughput(%) | | | | Latency(cycles) | | | |
|---|---|---|---|---|---|---|---|---|
| load | 0 | 2 | 2-1/2 | 3 | 0 | 2 | 2-1/2 | 3 |
| 70 | 7.2 | 69.9 | 69.9 | 69.9 | 165 | 21.6 | 21.0 | 21.5 |
| 90 | 7.2 | 82.8 | 84.4 | 84.7 | 167 | 30.5 | 30.7 | 30.9 |

Table 4. Latency and Throughput for Different Combining Strategies. Simulation data, 128-PE network, single packet messages, 2-input queues, 10% hot spot rate.

For a 128-PE network, the differences among all three schemes are small. Three-way combining with the same return path buffer size is actually worse than two-and-a-half way combining, with latency of 31.3 cycles and throughput of 77.8% at 90% offered load. We are currently running simulations of larger systems to see at what size system a more complicated combining scheme becomes worthwhile.

## FURTHER WORK

We are continuing work on analytic solutions, in particular for the output distribution, and on finite buffers. We are interested in developing analytic models that include the effect of routing reversal at memory on the round trip performance. Better analytic solutions are needed to judge initial proposals for network design; simulations can be most usefully conducted only after a design has been elaborated.

We are also carrying out simulations under more complicated traffic models, with multiple hot spots and varying hot spot frequencies and locations. Since the usefulness of increased switch functionality is heavily dependent on the patterns of network load, further work must concentrate on characterizing that load. We are gathering traces from parallel programs to be used in conducting trace-driven simulations of the network. Our goal is to integrate the network simulator into a system simulation environment that can simulate increasingly complex models of processor and memory, as well as switch behavior.

## References

[1] S. Dickey, R. Kenner, M. Snir and J. Solworth, "A VLSI Combining Network for the NYU Ultracomputer," *Proceedings of the International Conference on Computer Design*, pp. 110 - 113, October 7, 1985.

[2] S. Dickey and Y. Liu, "Simulation and Analysis of Different Switch Architectures for Interconnection Networks in MIMD Shared Memory Machines", NYU Ultracomputer Note #141, June 1988.

[3] J. Edler, A. Gottlieb, C. Kruskal, K. McAuliffe, L. Rudolph, M. Snir, P. Teller, and J. Wilson, "Issues Related to MIMD Shared-Memory Computers: The NYU Ultracomputer Approach", *Proc. of the 12th Annual Intl. Conf. on Comp. Arch.*, pp. 126-135, 1985.

[4] A. Gottlieb, "An Overview of the NYU Ultracomputer Project", *Experimental Parallel Computing Architectures*, J. Dongarra, ed., North Holland, pp. 25-95, 1987.

[5] A. Gottlieb, B.D. Lubachevsky, and L. Rudolph, "Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors", *ACM TOPLAS* 5, pp. 164-189, Apr. 1983.

[6] R. Kenner, S. Dickey and P. Teller, "The Design of Processing Elements on a Multiprocessor System with a High-bandwidth, High-Latency Interconnection Network," to appear in *Proceedings of the Hawaii International Conference on System Sciences*, January, 1989.

[7] C.P. Kruskal and M. Snir, "The Performance of Multistage Interconnection Networks for Multiprocessors", *IEEE Trans. Comp.* C-32, pp. 1091-1098, 1983.

[8] C. P. Kruskal, M. Snir and A. Weiss, "The Distribution of Waiting Times in Clocked Multistage Interconnection Networks," *Proc. 1986 Intern. Conf. on Parallel Processing*.

[9] M. Kumar and J. R. Jump, "Performance Enhancement in Buffered Delta Networks Using Crossbar Switches and Multiple Links," *Journal of Parallel and Distributed Computing*, Vol. 1, No. 1, August 1984, pp. 81-103.

[10] M. Kumar and G. F. Pfister, "The Onset of Hot Spot Contention," *Proc. 1986 Intern. Conf. on Parallel Processing*, pp 28-34.

[11] D.H. Lawrie, "Access and Alignment of Data in an Array Processor", *IEEE Trans. Comp.*, C-24 , pp. 1145-1155, Dec. 1975.

[12] G. Lee, "Another Combining Scheme to Reduce Hot Spot Contention in Large Scale Shared Memory Parallel Computers," *First International Supercomputing Conference*, Athens, 1987, pp. 68-79.

[13] G. Lee, C. P. Kruskal and K.G. Kuck, "The Effectiveness of Combining in Shared Memory Parallel Computers in the Presence of 'Hot Spots'," *Proc. 1986 Intern. Conf. on Parallel Processings*, August 1986, pp. 35-41

[14] Y. Liu, "Delta Network Performance and 'Hot Spot' Traffic," NYU Ultracomputer Note #132, 1988.

[15] J.H. Patel, "Performance of processor-memory interconnection networks for multiprocessors," *IEEE Trans. Comp.*, C-30, pp. 771-780, Oct. 1981.

[16] O. Percus and S. Dickey, "Some Properties of Clock-regulated Queues in 2 by 2 Switch Architectures," NYU Ultracomputer Note, in preparation.

[17] O. Percus and J. Percus, "Elementary properties of Clock-regulated Queues," NYU Ultracomputer Note #138, May 1988.

[18] G.F. Pfister and V.A. Norton, " 'Hot Spot' Contention and Combining in Multistage Interconnection Networks", *Proc. 1985 Intern. Conf. on Parallel Processing*, pp. 790 - 797.

[19] G. F. Pfister et. al, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *Proc. 1985 Intern. Conf. on Parallel Processing*, pp. 764-771.

[20] R. Rettberg and R. Thomas, "Contention Is No Obstacle to Shared-Memory Multiprocessing", *CACM*, Volume 29, Number 12, December 1986, pp. 1202-1212.

# FAULT TOLERANCE OF ALLOCATION SCHEMES IN MASSIVELY PARALLEL COMPUTERS

Marilynn Livingston[*]
Department of Computer Science
Southern Illinois University
Edwardsville, IL 62026-1653

Quentin F. Stout[†]
Dept. of Elec. Eng. and Comp. Sci.
University of Michigan
Ann Arbor, MI 48109-2122

**Abstract**

This paper examines the problem of locating and allocating large fault-free subsystems in multiuser massively parallel computer systems. Since the allocation schemes used in such large systems cannot allocate all possible subsystems a reduction in fault tolerance is experienced. We analyze the effect of different allocation methods including the buddy and Gray-coded buddy schemes for the allocation of subsystems in the hypercube and in the 2-dimensional mesh and torus. Both worst case and expected case performance is studied. Generalizing the buddy and Gray-coded systems, we introduce a new family of allocation schemes which exhibits a significant improvement in fault tolerance over the existing schemes and which uses relatively few additional resources. For purposes of comparison, we study the behavior of the various schemes on the allocation of subsystems of $2^{18}$ processors in the hypercube, mesh, and torus consisting of $2^{20}$ processors. Our methods involve a combination of analytic techniques and simulation.

**Keywords** fault tolerance, allocation, hypercube computer, mesh,torus, buddy system.

## 1 Introduction

Parallel computers incorporating thousands of processors must be able to tolerate faulty processors and communication links if they are to achieve a usable mean-time-to-failure. In these large systems, processor allocation is needed for both multiuser environments, such as is provided with the NCUBE series of hypercubes, and for single user systems with multiple subtasking capabilities. In such a computing environment the problem of locating and allocating large fault-free subsystems is computation-intensive and, in practice, some allocation scheme which recognizes only a subset of the existing subsystems is used. The allocation scheme often has a dramatic effect on the fault tolerance of the system, thus forcing a trade-off to be made between space and computation time devoted to the allocation scheme versus minimum acceptable level of fault tolerance of the system.

In this paper we examine allocation schemes for large subcubes of a hypercube and large subsquares of a two-dimensional mesh and torus, considering worst case and expected case fault tolerance of the interconnection network, and the reduction in fault tolerance caused by the fact that the allocation scheme used cannot allocate all subsystems. We focus on the allocation of large subsystems because we believe that most massively parallel machines are purchased in order to support large tasks as opposed to hundreds of simultaneous users with small tasks. For comparative purposes we analyze allocating a subcube or subsquare of $2^{18}$ processors in a machine of $2^{20}$ processors.

In a $d$-dimensional hypercube there are $\binom{d}{q}2^{d-q}$ subcubes of dimension $q$, called *q-subcubes*, and each processor belongs to $\binom{d}{q}$ of them. Thus in a hypercube of $2^{20}$ processors a faulty processor makes 190 of the existing 760 18-subcubes faulty. The smallest number of faulty processors which makes all subcubes containing $m$ processors faulty in a hypercube containing $n$ processors is denoted by $Q_w(n,m)$. Analogously, the expected number of processor faults which makes all subcubes of size $m$ faulty is denoted by $Q_e(n,m)$, where we assume that faults are independent and uniformly distributed. We find that $Q_e(n,m)$ is significantly larger than $Q_w(n,m)$ and, in particular, $Q_w(2^{20},2^{18}) = 8$ while $Q_e(2^{20},2^{18}) \approx 24.5$. A discussion of these functions and their analogs for the 2-dimensional mesh and torus is included in Section 2.

Current hypercube allocation schemes do not allocate all $q$-subcubes but instead employ some form of the *buddy system* approach, where the only $q$-subcubes allocated are those consisting of all processors determined by arbitrarily fixing the high-order $d-q$ address bits. There are only $2^{d-q}$ such $q$-subcubes and each processor is in exactly one of them. Using $B_w(n,m)$ and $B_e(n,m)$ to denote the worst case and expected case number of faults needed to make all the buddy system subcubes of dimension $\lfloor \lg m \rfloor$ faulty, one has $B_w(2^{20},2^{18}) = 4$ and $B_e(2^{20},2^{18}) \approx 8.3$. Thus the use of the buddy system allocation scheme results in a considerable reduction in fault tolerance. In Section 3 we discuss the fault tolerance properties of the buddy system along with several variants including a *double buddy system* ($DB$), a *Gray-coded buddy system* ($G$), and a *double Gray-coded buddy system* ($DG$) defined for the hypercube, and the 2-dimensional mesh and torus.

In Section 3.3 we introduce a new family of allocation schemes which generalizes both the buddy and Gray-coded buddy systems and exhibits improved fault tolerance with relatively little increase in search time. The best of the schemes considered in our simulation studies, denoted $DA^2$, allocates only 48 subcubes of dimension 18 in a 20-dimensional hypercube, and yet $DA^2_e(2^{20},2^{18}) \approx 15$. Thus $DA^2$ achieves more than half of the fault tolerance of the hypercube in which all 760 subcubes of dimension 18 are allocable.

For purposes of comparison, we include in Section 4 simulation results for the hypercube with $2^{20}$ processors and the various schemes for allocation of subsystems possessing $2^{18}$ processors.

[*] Partially supported by National Science Foundation grant CCR-8808839

[†] Partially supported by National Science Foundation grant DCR-8507851 and an Incentives for Excellence Award from Digital Equipment Corporation

CH2649-2/89/0000/0491$01.00 © 1988 IEEE

## 2 Subsystems

Throughout, lg means $\log_2$. We use $n$ to denote the total number of processors in the system, and for convenience of notation, suppose $n$ is an even power of two. Let $\mathcal{M}(n)$ denote a two-dimensional square mesh with grid points $\{(x,y) : 1 \le x, y \le \sqrt{n}\}$, where two grid points are connected if and only if their coordinates differ by one in exactly one coordinate position. The corresponding $\sqrt{n} \times \sqrt{n}$ two-dimensional torus will be denoted by $\mathcal{T}(n)$. It has the same grid points as $\mathcal{M}(n)$, and includes the connections of $\mathcal{M}(n)$, but in addition its boundary points $(x, 1)$ and $(x, \sqrt{n})$ are adjacent as are $(1, y)$ and $(\sqrt{n}, y)$ for $1 \le x, y \le \sqrt{n}$. We will denote by $\mathcal{Q}(n)$ the $d$-dimensional hypercube with $n = 2^d$ nodes which are the binary $d$-tuples and where two $d$-tuples are connected if and only if they differ in exactly one position.

The mesh $\mathcal{M}(n)$ has $(\sqrt{n} - \sqrt{m} + 1)$ 2 subsquares of size $\sqrt{m} \times \sqrt{m}$, the torus $\mathcal{T}(n)$ has $n$ such subsquares, while the hypercube $\mathcal{Q}(n)$ has $\binom{d}{q} \cdot 2^{d-q}$ subcubes of dimension $q$, where $d = \lg n$. Thus we see that both $\mathcal{M}(n)$ and $\mathcal{T}(n)$ have $\Theta(n^{1.5})$ square subsystems whereas $\mathcal{Q}(n)$ has $\Theta(3^d) = \Theta(n^{\lg 3})$ subcubes.

Let us now examine the worst case and expected case fault tolerance when all subsystems of a given size are allocable. Denote by $M_w(n,m)$ the smallest number of faulty processors which

The mesh $\mathcal{M}(n)$ has $(\sqrt{n} - \sqrt{m} + 1)$ 2 subsquares of size $\sqrt{m} \times \sqrt{m}$, the torus $\mathcal{T}(n)$ has $n$ such subsquares, while the hypercube $\mathcal{Q}(n)$ has $\binom{d}{q} \cdot 2^{d-q}$ subcubes of dimension $q$, where $d = \lg n$. Thus we see that both $\mathcal{M}(n)$ and $\mathcal{T}(n)$ have $\Theta(n^{1.5})$ square subsystems whereas $\mathcal{Q}(n)$ has $\Theta(3^d) = \Theta(n^{\lg 3})$ subcubes.

Let us now examine the worst case and expected case fault tolerance when all subsystems of a given size are allocable. Denote by $M_w(n,m)$ the smallest number of faulty processors which

make all subsquares of $\mathcal{M}(n)$ with $m$ processors faulty, and let $M_e(n,m)$ denote the *expected* number of faulty processors which must occur before all subsquares with $m$ processors are faulty, assuming that faults are independent and uniformly distributed. The expressions $T_w(n,m)$ and $T_e(n,m)$ denote the corresponding quantities for the torus $\mathcal{T}(n)$ and $Q_w(n,m)$ and $Q_e(n,m)$ denote the corresponding quantities for the hypercube $\mathcal{Q}(n)$.

It is straightforward to establish that the functions $M(n,m)$, $T(n,m)$, and $Q(n,m)$ are monotone non-decreasing functions of $n$ and monotone non-increasing functions of $m$. We state these results without proof in the following.

**Proposition 2.1** *If $n' > n$, $d' > d$, $m' > m$, and $n \ge m'$ then*

( *i* ) $M_w(n',m) \ge M_w(n,m)$, *and* $M_w(n,m') \le M_w(n,m)$,

( *ii* ) $M_e(n',m) > M_e(n,m)$, *and* $M_e(n,m') < M_e(n,m)$,

( *iii* ) $T_w(n',m) \ge T_w(n,m)$, *and* $T_w(n,m') \le T_w(n,m)$,

( *iv* ) $T_e(n',m) > T_e(n,m)$, *and* $T_e(n,m') < T_e(n,m)$,

( *v* ) $Q_w(d',m) \ge Q_w(d,m)$, *and* $Q_w(d,m') \le Q_w(d,m)$,

( *vi* ) $Q_e(d',m) > Q_e(d,m)$, *and* $Q_e(d,m') < Q_e(d,m)$.

□

The values of $M_w(n,m)$ and $T_w(n,m)$ are relatively easy to determine. To illustrate for $m = n/4$, subdivide the $\sqrt{n} \times \sqrt{n}$ mesh into four $\sqrt{n/4} \times \sqrt{n/4}$ submeshes, designate a faulty processor $p_0$ in one of the submeshes and designate its translate as faulty in each of the other submeshes. This results in every square submesh of size $n/4$ being faulty, and the same fault pattern also makes every square submesh of the torus $\mathcal{T}(n)$ faulty. On the other hand, since the 4 subsquares are nonoverlapping, at least 4 faulty processors are required to cause them to be faulty. Using this reasoning, it follows that

$$M_w(n, n/4^i) = T_w(n, n/4^i) = 4^i$$

for any natural number $i$.

In the case of the hypercube the function $Q_w(n,m)$ is much more difficult to compute. However, it is known [1,2] that $Q_w(n,n/4)$ is the minimum positive integer $r$ such that $\binom{r-1}{\lfloor r/2 \rfloor - 1} \ge \lg n$ which yields

$$Q_w(n, n/4) = 8.$$

Thus, while in a square mesh or torus of 1024 × 1024 processors, as few as 4 faulty processors can make every 512 × 512 submesh faulty, in a 20-dimensional hypercube consisting of the same number of processors, at least 8 processors must become faulty before all 18-dimensional subcubes become faulty. As the number of processors increases, this difference becomes more pronounced since it has been shown [1,2] that

$$Q_w(n, n/4) = \lg\lg n + \frac{1}{2}\lg\lg\lg n + O(1).$$

We used simulation to investigate the expected number of faulty processors that make the analogous subsystems faulty. The 1024 × 1024 mesh and torus was approximated by the continuous unit square, and the 512 × 512 submesh by a continuous $\frac{1}{2} \times \frac{1}{2}$ square. In each trial, faults were successively generated randomly and uniformly in the unit square until no fault-free subsquare of size $\frac{1}{2} \times \frac{1}{2}$ remained. The results of 100,000 trials yielded the following:

$$M_e(2^{20}, 2^{18}) \approx 13.54, \text{ and } T_e(2^{20}, 2^{18}) \approx 19.89\ .$$

For the 20-dimensional hypercube we used a simulation with 10,000 trials and found that

$$Q_e(2^{20}, 2^{18}) \approx 24.50.$$

The program developed for this simulation, which will be described in Section 4, can also produce mean-time-to-failure values by incorporating a given probability distribution for the faults.

## 3 Allocation Schemes

In comparison with the 2-dimensional mesh and torus, the hypercube displays a high degree of fault tolerance with respect to large subsystems. However, this advantage is lost when only a small subset of the existing subsystems are allocable. In this section we will describe a few analytic results that indicate how the buddy system, and other allocation schemes, affect the fault tolerance properties of the hypercube, mesh, and torus.

### 3.1 Allocation Schemes for Hypercubes

Most allocation schemes for the hypercube employ the *buddy system* approach, where the only $q$-subcubes allocated in $Q(2^d)$ are those of the form $a_1 a_2 \dots a_{d-q} * \dots *$, that is, the high-order $d-q$ address bits are fixed in each allocable $q$-subcube. There are $2^{d-q}$ of these subcubes and they form a partition of $Q(2^d)$. We will use $BQ_w(n,m)$ and $BQ_e(n,m)$ to denote the worst case and expected case number, respectively, of faults needed to make all the buddy system subcubes of size $m$ faulty.

To obtain a good upper bound for $BQ_e(n,m)$, we consider the following model: balls are tossed at random into $x$ identical boxes until each box contains at least one ball. Suppressing details, we can show that the expected number of balls required is $\sum_{j=0}^{x-1} \frac{x}{x-j}$. The balls correspond to faults, and the boxes represent the disjoint subcubes allocated by the buddy system. Thus $x = n/m$. The number of balls required is a slight overestimate of the number of faults required because a subcube with one or more faults is slightly less likely to acquire another fault. From this model we have the following.

**Theorem 3.1** *For $n \geq m \geq 2$ and both $n$ and $m$ even powers of 2,*

( *i* ) $BQ_w(n,m) = n/m$,

( *ii* ) $BQ_e(n, n/2) \leq 3$ *and* $\lim_{d \to \infty} BQ_e(n, n/2) = 3$,

( *iii* ) $BQ_e(n,m) \leq n/m(\ln 2 \lg(n/m) + O(1))$.

□

Several allocation schemes which are more fault tolerant than the buddy system have been proposed. For example, two "orthogonal" buddy systems, which we call *double buddy systems* (DB), can be used to allocate $m$-subcubes of the form $a_1 a_2 \dots a_{d-m} * \dots *$ and $* \dots * a_{m+1} \dots a_d$ . With twice the number of subcubes allocated and roughly twice the overhead in the allocation algorithm we shall see an increase of approximately 25% in fault tolerance. In general, multiple buddy systems use multiple permutations of the index set $\{1, 2, \dots, d\}$, and for each permutation $\pi$ they allocate $m$-subcubes by fixing index positions $\pi(1) \dots \pi(d-m)$ and varying the remaining $m$ indices.

Another class of allocation schemes for the hypercube involve the use of the Gray code numbering of the nodes of $Q(n)$. Let $g_d$ denote the binary reflected Gray code map from $\{0 \dots 2^d - 1\}$ to $d$-bit strings. A single *Gray-coded buddy system*, denoted by $G$ here, allocates $q$-subcubes that arise as pairs of $(q-1)$-subcubes of the form $\{a_1 \dots a_{d-q+1} * \dots *, b_1 \dots b_{d-q+1} * \dots *\}$, where $g_{d-q+1}^{-1}(a_1 \dots a_{d-q+1})$ and $g_{d-q+1}^{-1}(b_1 \dots b_{d-q+1})$ are consecutive mod $2^{d-q+1}$. An approximate analysis of the behavior of the Gray-coded buddy system can be obtained through the use of a model in which $x$ identical boxes are arranged in a ring and balls are tossed at random into the boxes until no two adjacent boxes are empty. Here $x = 2^{d-q+1}$, which is the number of $q$-subcubes allocable by scheme $G$. While it is straightforward to show that $\lim_{d \to \infty} GQ_e(2^d, 2^{d-k})$ is trapped between $2^k k \ln 2$ and $2^{k+1}(k \ln 2 + O(1))$, we can obtain a more exact description. Let $p(x,i)$ denote the conditional probability that there are two adjacent empty boxes, given that $i$ balls have been tossed at random into the boxes. We have evaluated $p(x,i)$ which, in turn, gives an explicit expression for the limiting behavior of $GQ_e(n,m)$ as stated in the following.

**Theorem 3.2** *For $n \geq m \geq 2$ and both $n$ and $m$ even powers of 2,*

( *i* ) $GQ_w(n, n/4) = n/4$,

( *ii* ) $GQ_e(n, n/2) \leq 3\frac{2}{3}$ *and* $\lim_{n \to \infty} GQ_e(n, n/2) = 3\frac{2}{3}$,

( *iii* ) $GQ_e(n,m) \leq \sum_{j=0}^{x-1} p(x,i) \frac{x}{x-j}$, *where* $x = n/m$ *and* $p(x,i)\binom{x}{i} = \binom{x}{i} - \binom{i}{x-i} - \binom{i-1}{x-i-1}$, *and* $\lim_{n \to \infty}$ , $GQ_e(n, n/x) = \sum_{j=0}^{x-1} p(x,i) \frac{x}{x-j}$.

□

Multiple Gray-coded buddy systems combine Gray codes with the multiple index permutations of multiple buddy systems. We will use $DG$ to denote the double Gray-coded buddy system which allocates $q$-subcubes from pairs of $(q-1)$-subcubes in which the first $d-q+1$ index positions have been fixed together with the pairs of $(q-1)$-subcubes in which the last $d-q+1$ index positions have been fixed. Chen and Shin [4] have suggested $DG$ as an improved allocation scheme for $Q(n)$.

To obtain corresponding analytic results for $DB$ and $DG$, we can use the same model but the analysis involves the consideration of many special cases. We have resorted to simulation to help us understand the performance of these schemes and will report the results in Section 4.

### 3.2 Allocation Schemes for the Mesh and Torus

A buddy system for the allocation of square submeshes containing $m$ nodes in $\mathcal{M}(n)$ is analogous to that described for hypercubes. There are $n/m$ subsquares of dimensions $\sqrt{m} \times \sqrt{m}$ allocated and these are of the form $\{(x,y) : (j-1)\sqrt{m} + 1 \leq x \leq j\sqrt{m}, (k-1)\sqrt{m} + 1 \leq y \leq k\sqrt{m}\}$ for $1 \leq j,k \leq \sqrt{n/m}$. So, for example, if $m = n/4$, there are only 4 allocable square submeshes containing $n/4$ processors. In the worst case we see that only 4 faulty processors are needed to make every buddy system submesh of size $n/4$ faulty. Assuming the faults are randomly and uniformly distributed, the expected number of faulty processors needed to make every buddy system submesh of size $n/4$ faulty can be found by considering the same model as that used for the buddy system for hypercubes. This gives an expected value of approximately $8\frac{1}{3}$. The same arguments hold for the torus, as we state in the following.

**Theorem 3.3** *For $n \geq m \geq 2$, and both $n$ and $m$ even powers of 2,*

( *i* ) $BM_w(n,m) = BT_w(n,m) = n/m$,

( *ii* ) $BM_e(n,m) = BT_e(n,m) \leq (n/m)[\ln(n/m) + O(1)]$.

□

The Gray-coded buddy system for $\mathcal{M}(n)$ and $\mathcal{T}(n)$ is the same as the double buddy system $DB$ here. $\mathcal{M}(n)$ is partitioned into square submeshes consisting of $m/4$ nodes each and $G$ allocates a square submesh of size $m$ when that submesh arises as a $2 \times 2$ array of the smaller $m/4$ submeshes in the partition. Thus $G$ allocates a total of $(2\sqrt{n/m} - 1)^2$ $m$-node square submeshes for $\mathcal{M}(n)$ and a total of $4n/m$ for $\mathcal{T}(n)$.

When $n/m$ is an integral power of 4 the buddy systems are already worst-case optimally fault tolerant for the mesh and torus, and hence also $GM_w(n,m) = GT_w(n,m) = n/m$. To obtain analytic estimates of $GM_e(n,m)$ and $GT_e(n,m)$, the same probability model as used in the case of the hypercube can be utilized here. However, the number of configurations to handle becomes large and we have not carried the computations out but rather have resorted to simulation studies instead. In the study, we approximated $\mathcal{M}(n)$ and $\mathcal{T}(n)$ by the continuous unit square, and the $\sqrt{n/4} \times \sqrt{n/4}$ submesh by a continuous $\frac{1}{2} \times \frac{1}{2}$ square as we described in Section 2. The results of 100,000 trials gave

$$GM_e(n, n/4) \approx 10.0 \,\text{and}\, GT_e(n, n/4) \approx 11.7.$$

### 3.3 A New Family of Allocation Schemes

We will describe here a new family of allocation schemes for the hypercube and note that an analogous family can be described for the 2-dimensional mesh and torus.

Let $k \geq 1$, and consider an allocation scheme $A^k$ that, for a given $d$ and $q$, will allocate $q$-subcubes whose nodes are $d$-tuples in which the last $q - k$ bits are arbitrary and the first $d - q + k$ bits are the nodes of a $k$-subcube in $Q(2^{d-q+k})$. For example, $A^2$ will allocate 18-subcubes in $Q(2^{20})$ of the form $a_1a_2a_3a_4 * \ldots *$ in which the *last* 16 components are $*$'s and where two of the $a_i$ have values 0 and 1 and the other two $a_i$ are equal to $*$. Thus, $A^2$ allocates 24 subcubes of dimension 18. The *double* $A^2$ family, denoted by $DA^2$, allocates a set of $q$-subcubes of $Q(n)$ which consist of the set of $q$-subcubes allocated by $A^2$ together with a corresponding set in which the *first* 16 components are $*$'s and the last four components are chosen in an analogous way to the first four components for $A^2$. In general, $A^k$ allocates $\binom{d-q+k}{k}2^{d-q}$ subcubes of dimension $q$ in $Q(2^d)$ and $DA^2$ allocates twice this number. In the families $A^k$ and $DA^k$, increasing $k$ clearly increases the number of allocable subcubes and hence increases the fault tolerance, at a cost of increased search time. Analytic results for these families, similar to those for the buddy and Gray-coded buddy systems has not been done as yet. However, simulation studies we have done on these schemes show them to yield a significant improvement in the fault tolerance of the hypercube as we will see in the next section.

## 4 Simulation Results

We illustrate in the table below the results of simulation studies of the buddy ($D$), double buddy ($DB$), Gray-coded buddy ($G$), double Gray-coded buddy ($DG$), the new allocation families ($A^1$) and ($A^2$), and their doubles ($DA^1$) and ($DA^2$). We include the results for the scheme ($E$) in which every subcube is allocable, as well as including the worst case results and a listing for the number $S_a$ of subcubes allocated by each scheme. These schemes were used in the allocation of subcubes in $Q(2^{20})$ of size $2^{18}$. The values for $Q_e$ shown in the table were obtained from 1000 trials. In each trial, a list of random faults sufficient to make every 18-subcube faulty was generated. For each allocation scheme $X$, the list was scanned to identify the first fault on the list that resulted in all of the 18-subcubes allocated by $X$ being made faulty. Although we have not done so, we could easily modify our program to compute mean-time-to failure for each of the allocation schemes once a probability distribution for faults was specified. The expected case values in the table are based on the assumption that faults are randomly and uniformly distributed with respect to the nodes of the hypercube $Q(2^{20})$.

**Table: Expected and Worst Case Behavior of Allocation Schemes**

| | $E$ | $B$ | $DB$ | $G$ | $DG$ | $A^1$ | $DA^1$ | $A^2$ | $DA^2$ |
|---|---|---|---|---|---|---|---|---|---|
| $S_a$ | 760 | 4 | 8 | 8 | 16 | 12 | 24 | 24 | 48 |
| $Q_w$ | 8 | 4 | 4 | 4 | 4 | 4 | 4 | 5 | 5 |
| $Q_e$ | 24.6 | 8.1 | 10.1 | 9.8 | 11.9 | 10.7 | 13.0 | 12.8 | 15.4 |

## 5 Conclusion

Our results show that simple allocation schemes such as those based on the buddy system lose much of the fault tolerance of the system, but that much of this loss can be regained by a more sophisticated allocation scheme such as the new family $DA^k$ that we described. Results of our simulations indicate, for example, that by using the allocation scheme $DA^2$ on the hypercube of dimension 20, we can roughly double the fault tolerance of that provided by the buddy system. Moreover, very little overhead is involved in the implementation of $DA^2$. As a final note, we observe that the hypercube is significantly more fault tolerant than the mesh or torus, and that the use of the buddy system for large subcube allocation reduces the fault tolerance to that of the mesh.

## References

[1] B. Becker and H. Simon. How robust is the $n$-cube? *Proc. 27th Annual IEEE Symp. on Foundations of Comp. Sci.* (1986) 283-291.

[2] N. Graham, F. Harary, M. Livingston, and Q. F. Stout. Subcube fault tolerance in hypercubes. (Submitted)

[3] M. Livingston and Q. F. Stout. Fault tolerance of allocation schemes in massively parallel computers. *Proc. Third Conf. on Hypercube Concurrent Computers and Applications, Pasadena, CA* (1988).

[4] M.-S. Shen and K. Shin. Processor allocation in an $n$-cube multiprocessor using gray codes. *IEEE Trans. Computers* C-36 (1987), 1396-1407.

# COMPUTER ARCHITECTURE FOR INTELLIGENT, REAL-TIME, NUMERIC AND SYMBOLIC PROCESSING

R.K. Mahadevan and C.C. Carroll
Computer Architecture Research Laboratory
Department of Electrical Engineering
University of Alabama
Tuscaloosa, Alabama

## Abstract

*Numeric processing governed intelligently, permits achieving solutions at a much faster rate than by normal procedures. An architecture that has knowledge of the processes it controls and the power to alter the course of the process using expert resources, enables quicker solutions of numeric and symbolic problems. Algorithms written in a mix of numeric and symbolic computation would perform better as they parallel the expert mind more closely than exclusively numeric or exclusively symbolic algorithms. The architecture presented, while MIMD massively parallel, is an optimum parallel environment for real-time applications (OPERA), and follows the idea of intelligent processing to create a coupled system to support real-time simulation and control tasks.*

## 1.0 Introduction

Continuous simulation on computers, has been one of the modes to study the mechanics of various physical systems. The first step is to represent these systems as closely as possible in mathematical forms, to create models. The study of the behavior of the system is then conducted by subjecting the described model to various external influences and observing its response. To understand certain models, a qualitative design, which uses little or no mathematical description and which has the least of assumptions in describing the model in its component parts, is better than a quantitative model. In most models, the understanding of both the physical and logical relationships provides a more fertile area for study.

The complexity of certain applications, or the flexibilty needed by them for accurate results often balks the speed and performance of existing machines.

Besides having the knowledge of the type, size and scope of the simulation to be processed within the processing environment, the knowledge of the systems resources and their capabilities is essential. There are occasions where an expert is willing to compromise a certain degree of accuracy to obtain a satisfactory result. This could be in the control of input-output actuators or sensors, in the replacement of a block of computation by a fixed value, or in the rearrangement of computation itself. There are also the situations where a certain set of computations may be unnecessary or against the grain of the computation being performed. At times these irregularities are difficult to detect, or completely imperceptible, and may be more easily detected by a run-time supervisor. There are also the expert applications themselves, that have a great numeric orientation; in which experts have a great dependency on extensive mathematical computation to arrive at conclusions. The University of Alabama's proposed OPERA architecture seeks to provide a good balance in hardware support to both of these applications [1-4]. In the initial proposal the target languages are the Advanced Continuous Simulation Language (ACSL, not as a preprocessor to FORTRAN, but on its own) and LISP. Further changes and expansions are being investigated.

## 2.0 Software Considerations

ACSL [5] as currently implemented, is a preprocessor to FORTRAN. It provides most functions necessary in continuous simulation, and special facilities for input-output functions that permit control and communication from and to the external world. The major modules of an ACSL program are the INITIAL, TERMINAL and DYNAMIC blocks. The INITIAL block serves to establish preprocessing conditions, by performing tasks such as setting up initial values of variables. The TERMINAL block performs all output jobs and minor processing that may be necessary at the end of a run. The DYNAMIC block is the main module that performs

*This research was funded in part by Electronics Associates Inc. and The Alabama Research Institute. The opinions expressed in this paper though, are wholly those of the authors.*

the iterative computation on the model; periodically communicating vital statistics to the external environment. ACSL has provisions to perform various integration algorithms, permitting the user to alter the algorithm between different simulation runs.

Many of the ACSL simulation benchmarks have been executed at the University of Alabama using C, LISP and assembly language. The advantage of parallel techniques over their sequential versions was studied [6-13], and was found to be beneficial to the applications. The main strategies employed in creating the parallel versions were, the decomposition of equations to create suitable parallel versions, and the direct application of parallel algorithms to the problem. Extensive simulation showed that neither one of these stategies was superior to the other, but that the combination of the two produced more favorable results. During these simulations, the allocation was performed using a random allocation process to arrive at the best fit. In the final form it is intended that the expert system select or suggest one of many allocation algorithms that will be available to the user[14-16]. At certain points the deviation in accuracy had to be compromised, because of the greater approximations that resulted because of parallelism. In situations such as these expert judgement was required to decide if the sacrifice in accuracy of computation would actually compromise the validity of the result. On occasions where an enormous combination of runs had to be conducted to arrive at a result, an expert system could have restricted the problem space to only the actually necessary combinations to reach the target. The variations which could cause deviations in real-time control tasks may also be eliminated. The parallelism possible in the expert system itself was studied to speed up expert resource use [4,17-20].

## 3.0 Architectural Configuration

The design of the architecture allows for hierarchical levels of parallelism, that enable the exploitation of parallelism in the application at different grain levels. Specifically, the environment is designed to suit two levels of parallelism, the grain of which have been selected giving consideration to factors such as processing element capabilities, software decomposition facilitated, and networking performances. Granularity at the expression level or construct level is termed fine, and granularity at the procedure level or program level is classified coarse. To address the problem at the lower level of granularity the architecture provides processing elements capable of performing independently as nodes. These processing elements themselves are designed as complements of processors, each dedicated to a specific detail of the node's total capability. Sixty four such processing elements are grouped to constitute a cluster. Fifty six of these processing elements are assigned to the task of processing, four are dedicated to communication and the four others remain in a hot stand-by mode to aid in reconfiguration. Sixty four such clusters are interconnected to make up the entire system. The clusters attend to the parallelism classed as coarse-grain. The selection of the number sixty four is based on the interconnection network delays and optimal parallelism considerations. This hierarchical treatment is applied in both the hardware and the software of the architecture. In many parallel architectures the parallel architectures the parallelism is controlled at an operating system level. This form of control is limited to a very low level, allowing control at program level to be emphasized.

### 3.1 Processing Element

The processing element is a module capable of performing any task independently and also being capable of performing a task in unison with any other processing elements involved. The major elements into which the processing element (PE) at the node can be broken down are; a symbolic processor, a numeric processor and data control [4,20]. The Mega-chip architecture of TI is currently chosen to represent the symbolic processor. Another heavily branch oriented virtual processor is also being recommended for the same purpose. The numeric processor which is a stand alone processor is implemented in a custom design. It is constructed to perform high-speed 32-bit floating point operations such as, 32x32 bit multiplication in a cycle to produce a 55-bit result. Data control is a functional unit which encompasses all aspects that relate to data transfer. It includes the cache and cache-controller system, the array and table handling unit, and an external communications unit which operates on its own. The design of this processing element is very memory intensive. The idea which was spurred by the design of the Megachip achieves modularity in computational packages that permit the maintenance of all code and data in PE memory, and thereby avoiding any run-time reference to main-memory. This alone stands to saves enormous amounts of computational time. The random-accessing of complex data-structures is implemented in hardware to speed up memory accesses. Accessing of array subscripted variables for example, relative to accessing a simple data object is very tedious, and burdensome when the number of dimensions of the array increase. The hardware access scheme decreases the access time, and directly improves the facilities for problem definition of high-speed real-time models. Detection and diagnostics are minimal in the hardware level.

A very efficient communication system which offers separate communication facilities for memory access and inter-processor communication, is implemented internal to the processing element. The numeric and symbolic processors may, and usually operate concurrently within the PE. The symbolic processor serves as the expert to monitor the process of the numeric algorithm, and/or accomplishes some symbolic processing of its own. The differences in data type between the two processors are handled by specialized interfaces. Partitioning of algorithms take place at two levels. First, into modules small enough to be contained in individual nodes throughout the process and second, as independent or semi-independent subprograms with limited coupling between them. Communication is conducted on a priority basis if the usage is not very intense. If traffic does become dense, a polling routine is employed. The I-O processor handles the error-detection and correction, encoding, decoding, serializing and deserializing of data. It has a small local memory addressable by other processors.

## 4.0 Networking

There are two levels of interconnection that are considered in this architecture. They are associated with the intra-cluster and the inter-cluster level of communication. The network as was judged from an applications point-of-view necessitated tight coupling between elements, and is constructed using delta-networks [24]. At the cluster level of communication, the network is designed using a 64 × 64 delta-network which has buffers built into the switches to reduce blocking and increase throughput. At this level the communication is more in short and rapid bursts of messages and to maintain a real-time speed of communication, having a high transfer-rate across the network is essential. The delay caused by the inclusion of buffers is minimal but the the throughput of the network increased tremendous, which definitely justifies the inclusion of buffers. At the inter-cluster level of communication, the communication technique is varied a little to accommodate the intra-cluster format. It is also extended to accommodate varying and larger formats in communication packets, restricted only by a maximum size. Buffering is not considered here because of the larger size of communication packets. A flag is used to prevent switch blocking at this level.

## Summary

An architecture to provide for the expert control, of numerical control was described. Simulations run to explore the possibility of the exploitation of parallelism have shown positive results, in both the area of numerical processing and expert symbolic processing. Research to study the positive blend of expert control in this architecture is being conducted.

Example applications towards which such an environment is directed are real-time aerospace, nuclear system modelling, and simulation. Another application class which is being considered is, expert systems which require a heavy load of numerical computing to analyze or synthesize their results.

## References

1. Consortium for Supercomputer Research, Inc., " Consortium for Supercomputer Research,"Applications, PART-A : Scientific and Engineering, vol. 1, September 1987, pp. VI-15.

2. Despain, A.M. and Patt, Y.N., "Aquarius-A High Performance Computing System for Symbolic/Numeric Applications," The Proceedings of COMPCON S'85, 1985, pp. 376-382.

3. Kowalik, Janus S., Coupling Symbolic and Numerical Computation in an Expert System, Elsevier Science Publishers B.V., 1986.

4. Amundsen, Micheal J., et al., "The Compact Lisp Machine, A 'LISP MACHINE IN A SHOEBOX'," Proceedings of the IEEE Aeronautics Conference, vol. 4, 1982, pp. 1309-1314.

5. ACSL User Guide/Reference Manual, Mitchell and Gauthier, Associates, 1981.

6. Miranker, W.L. and Liniger W., "Parallel Methods for the Numerical Integration of Ordinary Differential Equations," Math. Comput., vol. 21, 1967, pp. 303-320.

7. Burden, Richard L., Faires, Douglas J., and Reynolds, Albert C., Numerical Analysis, 2nd ed., Prindle, Weber & Schmidt, 1981.

8. Dorn W.S., "Generalizations of Horner's rule for polynomial evaluation," IBM J.Res. and Develop. 6, 1962, pp. 239-245.

9. Miranker, W.L., "A Survey of Parallelism in Numerical Analysis," Siam Review vol 11, No. 4, October 1974, pp. 524-547.

10. Schedler G.S., "Parallel numerical methods for solution of equations," Comm. ACM 10 off, 1967, pp. 286-290

11. Hambler, J.O., " Parallel Continuous System Simulation Using the Transputer," Simulation, Vol., 49, No. 6, December 1987, pp. 249-253.

12. Franklin, F.A.," Parallel Solution of Ordinary Differential Equations," IEEE Transactions on Computers, Vol. C-27, No. 5, May 1988, pp. 413-420.

13. Well Jr., B.E. and Carroll, C.C., " An Intelligent Processing Environment for Real-Time Simulation," Bureau of Engineering Research Report No. 426-17, The University of Alabama, Tuscaloosa, AL, May 1988.

14. Sinclair, J.B.," Efficient Computation of Optimal Assignments for Distributed Tasks," Journal of Parallel and Distributed Computing, Vol. 4, No. 4, August 1987, pp. 342-362.

15. Zhao, W., Ramamritham, W. and Stankovic, J.A., " Preemptive Scheduling Under Time and Resource Constraints," IEEE Transactions on Computers, Vol. C-36, No. 8, August 1987, pp. 949-960.

16. Carroll, C.C. and Ananthram, K.G., " An Intelligent Allocation Algorithm for Parallel Processing," Bureau of Engineering Research Report No. 417-17, The University of Alabama, Tuscaloosa, AL, January 1988.

17. Patrick Henry Winston, Artificial Intelligence, 2nd ed., Addison-Wesley, 1981.

18. Akl, Selim G., Barnard, David T. and Ralph Doran, "Design and Analysis, and Implementation of a Parallel Tree Search Algorithm," IEEE Transactions on Pattern-Analysis and Machine Intelligence, vol. point 4, No. 2, March 1982.

19. Chu Y., Abrams M., " Programming Languages and Direct Execution Computer Architectures," Computer, July 1981, pp. 22-32.

20. Sussman, G.J., Holloway, J., Steel, Jr. G.L., and Bell, A., " Scheme-79 Lisp on a Chip," Computer, July 1981, pages 10-21.

21. Sridhar, et al., " Analysis and Simulation of Parallel Signature Analyzers," paper 22.3, IEEE Test Conference, 1982.

22. Fenton, B.P., Grant, I.M. and Thiessen, D.R., "Multiple-line Signature analysis using Multiple Line Feedback Shift Register," IBM Technical Disclosure Bulletin,vol. 26, No 12, May 1984, pp. 6358.

23. Lala, Parag K., Fault Tolerant and Fault Testable Hardware Design, Prentice/Hall International, 1985.

24. Kumar, M. and Jump, J.R., "Performance Enhancement in Buffered Delta Networks Using Cross-bar Switches and Multiple Links," Journal of Parallel and Distributed Computing 1, 1984, pp. 81-103.

# REGULAR PROCESSOR ARRAYS

Allen D. Malony

Center for Supercomputing Research ..d Development
University of Illinois at Urbana–Champaign.

## ABSTRACT

*Regular* is an often used term to suggest simple and uniform structure of a parallel processor's organisation or a parallel algorithm's operation. However, a strict definition is long overdue. In this paper, we define regularity for processor array structures in two dimensions and enumerate the eleven distinct regular topologies. Space and time emulation schemes among the regular processor arrays are constructed to compare their geometric and performance characteristics. The hexagonal array is shown to have the most efficient emulation capabilities.

*Keywords: regularity, processor arrays, emulation, interconnection networks*

## INTRODUCTION

The most widely debated topic in parallel processing research is how to interconnect multiple processors. The arguments take place across many different cost/performance criteria such as algorithm mapping, scalability, reconfigurability, communication efficiency, graph embedding, fault tolerance, and VLSI implementation.

Mesh connected processor arrays were among the first processor interconnection structures proposed for parallel processing [1] [7]. Their distinguishing feature is the connection of processors only to immediate neighbors where the connection degree is uniform throughout the array. The original motivation for mesh topologies came from their ability to easily represent the natural data flow patterns found in many algorithms [8] [7] [13] [18] [17] [27].

The thought of interconnecting thousands of processors brought on a wave of new processor interconnection structures aimed at providing cost–effective solutions to certain key scalability issues such as mean internode distance, communication traffic density, connections per node, link visit ratios, and fault tolerance [21] [28]. The processor arrays proposed included the torus, X–tree, chordal ring, R–ary N–cube, cube–connected cycles, spanning bus hypercube, and dual bus hypercube, in addition to the standard bus, crossbar, ring, and tree architectures [28]. Although favored for their regular geometry, uniform communication and simple extension, the mesh connected processor arrays were generally less desired because of the fact that internode communication delays increase as the square root of the number of nodes in the system.

Systolic array research approached the problem of designing processor arrays by concentrating on requirements for an effective VLSI implementation of a parallel algorithm [11] [9] [10]. Chip area, time and power required to implement an algorithm in VLSI are dominated by the communication geometry of the algorithm [25]. The effects of the area and time parameters of VLSI can be reduced to a large degree if very simple and regular patterns of interconnections between elements are used [16] [26]. The regularity requirement imposed on interconnection structures, in a broad sense, deals with the layout of the communication geometry in a two–dimensional area [22]. Simple and regular interconnection geometries that are two–dimensional and plane filling lead to cheap implementations and high chip density. Also, parallel algorithms with simple and regular communication and data flows are more appropriate for VLSI implementation and will result in higher performance.

The choice of processor array design to achieve good generalized communication performance conflicted with the simple processor arrays favored for specialized VLSI systems. If only the more sophisticated communication topologies were implemented in VLSI, then their communication efficiencies could be combined with the faster VLSI speeds. However, several recent results suggest that mesh–connected arrays' have comparable, if not better, general communication efficiency and performance when implemented in VLSI as compared to other networks [15] [19]. In addition, there has been much work done on making regular mesh arrays more flexible [3] [23] [4] [2] [20].

In this paper, we consider the question of what are the simple and regular processor array topologies? The primary contribution of this work is the enumeration and analysis of the "regular" two–dimensional processor array topologies using a geometric definition of regularity. Several topologies are shown that have not appeared in the computer science literature previously. Our analysis of the regular processor arrays is based on their ability to emulate the other members of the class. We consider both space emulation (processors of the host array are combined into "logical" nodes of the target array) as well as time emulation (the interconnection geometry of the target array is provided by time–multiplexing the links).

## REGULARITY

Intuitively, the term *regular* implies simplicity and uniformity in space. A more quantitative geometrical definition of regularity can be formulated from the extensive mathematical literature on graphs [5] [6] [12] [24]. Although regularity can be defined for multiple dimensions, our discussion is restricted to graphs that are two–dimensional, i.e. planar. A second requirement is that the graph have a simple description and be uniformly extensible following a basic set of construction rules. By the graph being uniformly extensible, we mean that the properties of the vertices and edges do not change as the number of nodes is increased; e.g., the length of an edge. Another requirement for regular graphs is that the vertices have equal degree. The final requirement is that regular graphs be plane filling. That is, the infinite graph completely covers the two–dimensional plane.

The requirements placed on regular graphs are not without mathematical precedence. Justification comes from the old geometrical problem of determining those convex polygon figures that tessellate the plane [5] [6]. In particular, the problem is to construct tilings of the plane where a single convex polygon of $r$ sides is used. Based on Euler's theorem $v - e + f = 1$ ($v$ vertices, $e$ edges and $f$ faces of a polygonal network of tiles) and basic Diophantine analysis, it is a simple consequence that $3 \leq r \leq 6$ [5].

Although there are eighty–one types of isohedral tilings in the plane [5], there are ONLY eleven topologically distinct types of *Laves*

This work was supported in part by the National Science Foundation under Grants No. US NSF DCR84-10110, the U. S. Department of Energy under Grant No. US DOE-DE-FG02-85ER25001, the U.S. Air Force Office of Scientific Research Grant No. AFOSR-F49620-86-C-0136, and the IBM Donation.

CH2649-2/89/0000/0499$01.00 © 1988 IEEE

*nets* [12] (also called *regular* or *Subnikov nets* [24]) which are the "skeleton" graphs consisting of tile "vertices" (where three or more tiles meet), and tile "edges" where two tiles intersect. Figure 1 shows the eleven Laves nets along with symbols denoting the valences of the vertices as the tessellating *r*–gon is traced; e.g., $3^2$.4.3.4 describes a pentagon tessellation where the pentagon meets 3 other tiles, then 3, 4, 3, and finally, 4 other tiles. The geometry of the distinct tessellation topologies can be described from this simple vertex valency syntax.

Tessellation structures embody the requirements set forth for regular graphs: they are two–dimensional, they have a simple description (tile vertex valency syntax), all tiles used in a tessellation have the same number of edges (*r*–gon), they are uniformly extensible, and they are plane filling. If we associate a tile to a processor array node and the links to tile intersections (tile edges), the resultant interconnection topology will embody the same regular properties.

The regular processor interconnection graphs can be generated by taking the *dual* of the Laves nets, i.e. the faces (tiles) are mapped to vertices, the tile vertices are mapped to faces, and tile edges map to edges between the new vertices [5]. Because the graphical duality mapping is isomorphic, there are exactly eleven distinct regular processor array topologies. These topologies are also known as the familiar *nearest neighbor* topologies because all vertices are of equal degree and each vertex connects to that many of its nearest neighbors.

**Definition:** A graph is regular if it is two dimensional, all vertices have equal degree and the dual of the graph is a tessellation.

**Definition:** A processor array is regular if its interconnection topology is a regular graph.

In the next section, we consider emulations among the regular processor arrays. In particular, we focus on the triangular $(6^3)$, the orthogonal $(4^4)$ and the hexagonal $(3^6)$ topologies. These have been defined to be *strongly regular* because they form a set closed under duality: the triangular graph is the dual of the hexagonal and vice versa, and the orthogonal graph is the dual of itself [14].

## REGULAR PROCESSOR ARRAY EMULATION

Although the number of regular processor arrays is finite, it would be cost inefficient to include each array in a parallel processing system and use an array only when there is an appropriate match between an algorithm's communication geometry and that array's topology. Instead, we would like to design the system with a single processor array that offers good performance across a wide range of algorithms. The versatility of a processor array is measured not only by the range of algorithms for which it is specifically suited but also by the ease to which other algorithms can be mapped to its communication geometry [2], and the ability of the array to reconfigure it communication geometry to that of the algorithms or other array topologies [3] [4] [23]. We evaluate the regular processor arrays based on their ability to emulate other regular arrays.

### Emulation Philosophy

The goal of emulating a *target* regular array by a *host* regular array is to reproduce the communication properties of the target array in the host. The emulation can take place either in space or in time. *Space emulation* structurally maps the host array to the target array by physically grouping host nodes into logical target nodes and activating the appropriate host links such that the communication topology of the target array is realised. If the target array cannot be embedded in the host array with a one to one node mapping, the space emulation will necessarily result in a reduction of the effective size of the emulated target array.

*Time emulation* realises the communication properties of the target array by time multiplexing the host array links. Once a one to one node mapping is made between the target and the host, the maximum number of host "minor" communication time cycles needed to realise the communication connectivity of one "major" target time cycle can be determined. If the target array cannot be embedded in the host array with a one to one link mapping (we already assume a one to one node mapping), the time emulation will necessarily result in an increase in the number of time cycles needed to execute an algorithm on the emulated target array.

### Space Emulation

An optimal space emulation scheme should minimise the average number of host nodes used to emulate a node in the target array.

**Definition:** The *space emulation efficiency* $S_M(N)$ of a space emulation scheme used by regular array $M$ to emulate regular array $N$ is the average number of nodes of $M$ required to emulate one node of $N$. If $N$ contains $n$ nodes, the size of the emulated target array will be $n \ / \ S_M(N)$ nodes. A lower bound on $S_M(N)$ can be determined by calculating the number of host array nodes needed to match the node degree of the target array. An *optimal* space emulation scheme achieves the lower bound of the average number of host nodes required for a node of the emulated target array. That is, no more than the number of host nodes needed to meet the node degree requirements of the emulated target array are used.

The process followed to construct an emulation scheme begins by grouping adjacent nodes together to form logical nodes of the emulated target array. "Active" host links are then selected to realise the target communication geometry. During operation, nodes within a group coordinate their actions to correctly communicate data across the active links. Instead of enumerating all space emulations for all regular host arrays ad nauseam, we instead concentrate on the strongly regular arrays.

**Theorem 1:** The triangular array can optimally emulate the hexagonal array with a space emulation efficiency of four and the orthogonal array with a space emulation efficiency of two.
**Proof:** The emulation schemes are shown in Figure 2.

**Theorem 2:** The orthogonal array can optimally emulate the hexagonal array with a space emulation efficiency of two and the triangular array with a space emulation efficiency of one.
**Proof:** The emulation schemes are in [14].

**Theorem 3:** The hexagonal array can optimally emulate the orthogonal array with a space emulation efficiency of one and the triangular array with a space emulation efficiency of one.
**Proof:** The emulation schemes are shown in Figure 3.

**Theorem 4:** $S_{triangular}(R) \leq S_{orthogonal}(R) \leq S_{hexagonal}(R)$ where $R$ is a regular array.
**Proof:** Any space emulation scheme used by the triangular array to emulate another regular array can also be used by the orthogonal and hexagonal arrays since only one node is required by the orthogonal and hexagonal arrays to emulate a node in the triangular array. Therefore, any emulation scheme used by the orthogonal and hexagonal arrays for emulating another array must be at least as efficient as the optimal scheme that would be used by the triangular array. A similar argument is applied to show $S_{orthogonal}(R) \leq S_{hexagonal}(R)$.

### Space Emulation Schemes

The space emulation schemes for the regular processors arrays using the strongly regular arrays are shown in Figure 2 for the triangular host array and Figures 3 for the hexagonal host array [14]. Node groupings for the triangular host array are shown shaded. The dashed lines in the hexagonal host array indicate inactive links.

### Space Emulation Efficiency

The space emulation efficiencies of the schemes presented for the strongly regular arrays are shown in Tables 1. As expected, the hexagonal array shows the best efficiencies with nine of the schemes using an optimal emulation of one. The inability to achieve optimal schemes for $3^4.6$ and 3.6.3.6 is attributed to the rigid structure of those topologies.

Notice that the triangular array was able to achieve more optimal space emulations than the orthogonal array. In part, this has to do with the orthogonal array's inability to realize triangular interconnection paths present in some of the arrays such as $3^4.6$, $3^2.4.3.4$ and 3.6.3.6.

An interesting observation from the table is that $S_{triangular}(3.4.6.4) = 3$, yet $S_{triangular}(4^4) = 2$ and $S_{orthogonal}(3.4.6.4) = 1 \frac{1}{3}$. One quickly realizes that a better space emulation scheme could be achieve for 3.4.6.4 using the triangular array if the orthogonal array was first emulated and then its emulation scheme applied to realize 3.4.6.4. This would result in an emulation efficiency of $2 \frac{2}{3}$ instead of 3.

Space emulation analysis allows a simple measure of cost, $S_M(N)$, to be used for comparing the versatility of the different regular arrays. Additionally, the pay back for adding additional links to the array can be easily discerned. Finally, algorithms be designed for one particular regular array can executed on another array with bounded performance degradation.

### Time Emulation

Time emulations among the regular processor arrays are more complex to construct because a mapping from nodes of the host array to nodes of the target array must first be devised. We employed some convenient shortcuts that allowed us to develop a collection of time emulations for the strongly regular arrays as target topologies.

**Definition:** The *time emulation efficiency* $T_M(N)$ of a time emulation scheme used by regular array $M$ to emulate regular array $N$ is the number of communication time cycles required in $M$ to realize the data transfer between nodes possible in one cycle in $N$. Assuming the processor array speeds are equal, if $N$ completes an algorithm in $t$ time cycles, the time emulation scheme used by $M$ will finish in $T_M(N) * t$ time cycles.

Notice that if $S_M(N)$=1, $T_M(N)$=1. We can make use of this fact to compute bounds on time emulations based time emulations already known. That is, if $T_M(N)=t_1$ and $T_N(O)=t_2$, then $T_M(O) \leq t_1 * t_2$.

Initial time emulations can be constructed by looking at the space emulations with efficiency one. Since these already give a one to one node mapping, an optimal time emulation of the host array (in the space emulation) by the target array (in the space emulation) can be devised and its efficiency calculated by visually following the shortest path to establish the single link connections of the underlying host array. For instance, we compute $T_{3.4.6.4}(3^6)$ to be three by looking at the space emulation scheme of 3.4.6.4 by $3^6$ and following the shortest path between connected hexagonal nodes using only the 3.4.6.4 links.

Following the above procedure, we were able to construct optimal time emulations of the hexagonal array for most regular arrays. The time emulation efficiencies are shown in Table 2. The parentheses indicate upper bounds determined by applying the above formulas to these optimal hexagonal and orthogonal time emulations. Other entries in the table come from visually mapping one processor array onto another as in the case of the square array onto $3^2.4.3.4$ and $3^3.4^2$.

The interest in time emulations comes from the fact that the emulated target array is not reduced in size. Instead, a more complex routing of data in multiple time steps is required to emulate the target array's communication properties. However, we cannot ignore the time needed to route data in a space emulation scheme. In fact, we see that there are cases where a time emulation will be actually faster than a space emulation; a time emulation of a hexagonal array using a triangular array will take three time cycles whereas the space emulation requires four. In other cases, the opposite is true; consider the triangular array emulating the orthogonal array.

## CONCLUSION

Processor interconnection topologies incorporating communication and spatial regularity will become increasingly important as VLSI dimensions continue to decrease. Although mesh processor arrays have known scalability limitations with respect to communication [28], several recent reports suggest that the communication efficiency of two dimensional meshes is better than other interconnection topologies when compared for VLSI implementation [15] [19].

The regular processor arrays described in this paper are geometrically defined based on nearest neighbor connections and space-filing properties. Interestingy, only eleven processor arrays of regular topology exist in two dimensions. We have enumerated these arrays as well as presented space and time emulation schemes. A natural extension of the work presented here concerns regular three dimension organizations. Research in this area will become more important and necessary as VLSI begins to offer three dimensional interconnects.

## REFERENCES

[1] G.H. Barnes, R.M. Brown, M. Kato, D.J. Kuck, D.L. Slotnik, R.A. Stokes. *The ILLIAC IV Computer.* **IEEE Trans.** Comp., Vol. c-17, 1968, pp. 746-757.

[2] S.H. Bokhari. *On the Mapping Problem.* **IEEE Trans.** Comput., Vol. C-30, March 1981, pp. 207-214.

[3] Naga S. Gollakota and F.Gail Gray. *Reconfigurable Cellular Architecture.* **Proc. 11th Int. Symp. Computer Arch.**, 1984, pp. 377-379.

[4] D. Gordon, I. Koren and G. Silberman. *Embedding Tree Structures in VLSI Hexagonal Arrays.* **IEEE Trans.** Comput., Vol. C-33, Jan. 1984, pp. 104-107.

[5] B. Grunbaum and G.C. Shephard. *The Eighty-one Types of Isohedral Tilings in the Plane.* **Math. Proc. of the Cambridge Phil. Soc.**, Vol. 82, Sept. 1977, pp. 177-196.

[6] B. Grunbaum and G.C. Shephard. *Tilings with Congruent Tiles.* **Bulletin of the Amer. Math. Soc.**, Vol. 3, No. 3, Nov. 1980, pp. 951-973.

[7] W.H. Kauts, K.N. Levitt and A. Waksman. *Cellular Interconnection Arrays.* **IEEE Trans.** Comp., Vol. C-17, No. 5, May 1968, pp. 443-451.

[8] D.J. Kuck. *ILLIAC IV Software and Applications Programming.* **IEEE Trans.** Comp., Vol. C-17, No. 8, Aug. 1968, pp. 758-770.

[9] H.T. Kung. *Let's Design Algorithms for VLSI Systems.* Caltech Conf. on VLSI, Jan. 1979, pp. 65-90.

[10] H.T. Kung. *Why Systolic Architectures?.* Computer, Vol. 15, No. 1, Jan. 1982, pp. 97-107.

[11] H.T. Kung and C.E. Leiserson. *Algorithms for VLSI Processor Arrays.* in **Introduction to VLSI Systems**, by C. Mead and L. Conway, Addison-Wesley, 1980, pp. 271-292.

[12] F. Laves. *Ebenenteilung und Koordinationszahl.* **Z. Kristallogr.**, Vol. 78, 1931, pp. 208-241.

[13] K.N. Levitt and W.H. Kauts. *Cellular Arrays for the Solution of Graph Problems.* **CACM**, Vol.15, No. 9, 1972, pp. 789-801.

[14] Allen D. Malony. *Regular Interconnection Networks.* Master's Thesis, Univ. of California, Los Angeles, August 1982.

[15] Pinaki Masumder. *Evaluation of Three Interconnection Networks for CMOS VLSI Implementation.* **1986 ICPP**, Aug. 1986, pp. 200-207.

[16] C. Mead and L. Conway. *Introduction to VLSI Systems.* Addison-Wesley, Reading, Mass., 1980.

[17] D. Nassimi and S. Sahni. *Bitonic Sort on a Mesh-Connected Parallel Computer.* **IEEE Trans.** Comput., Vol. C-27, Jan. 1979, pp. 2-7.

[18] S.E. Orcutt. *Implementation of Permutation Functions in ILLIAC IV-Type Computers.* IEEE Trans. Comp., Vol. C-25, No. 9, Sept. 1976, pp. 929-936.

[19] A.G. Ranade and S.L. Johnson. *The Communication Efficiency of Meshes, Boolean Cubes and Cube Connected Cycles for Wafer Scale Integration.* Proc. 1987 Inter. Conf. on Parallel Proc., Aug. 1987, pp. 479–482.

[20] D.A. Reed, L.M. Adams and M.L. Patrick. *Stencils and Problem Partitionings: Their Influence on the Performance of Multiple Processor Systems.* to appear in IEEE Trans. Comput..

[21] D.A. Reed and H.D. Schwetman. *Cost-Performance Bounds for Multi-microcomputer Networks.* IEEE Trans. Comput., Vol. C-32, No. 1, Jan. 1983, pp. 83–95.

[22] J.E. Savage. *Planar Circuit Complexity and the Performance of VLSI Algorithms.* Proc. of the CMU Conf. on VLSI Systems and Computations, 1981, pp. 61–67.

[23] L. Snyder. *Introduction to the Configurable, Highly Parallel Computer.* Computer, Vol 15, No. 1, Jan. 1982, pp. 47–56.

[24] A. Subnikov. *K voprosu o stroenii Kristallov.* Bulletin Acad. Imp. Sci., Ser. 6, Vol. 10, 1916, pp. 755–779.

[25] I.E. Sutherland and C.A. Mead. *Microelectronics and Computer Science.* Scientific American, Vol. 237, Sept. 1977, pp. 210–228.

[26] C.D. Thompson. *Area-Time Complexity for VLSI.* Proc. Caltech Conf on VLSI, Jan. 1979, pp. 405–508.

[27] C.D. Thompson and H.T. Kung. *Sorting on a Mesh-Connected Parallel Computer.* CACM, Vol. 20, April 1977, pp. 263–271.

[28] L.D. Wittie. *Communications Structures for Large Networks of Microcomputers.* IEEE Trans. Comput., Vol. C-30, No. 4, April 1981, pp. 264–273.

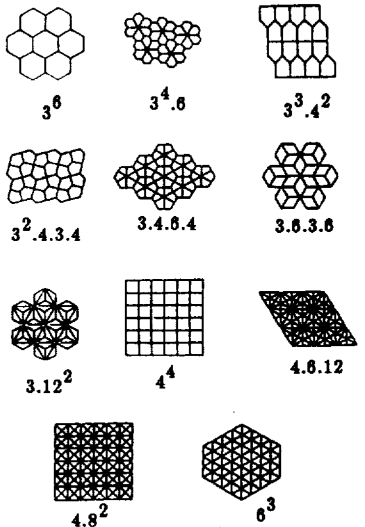


Figure 1

| Target Topology | Optimal Emulation (host/target) | | | Emulation Efficiency (host/target) | | |
|---|---|---|---|---|---|---|
| | Tri. | Ortho. | Hex. | Tri. | Ortho. | Hex. |
| $3^6$ | 4 | 2 | 1 | 4 | 2 | 1 |
| $3^4.6$ | 3 | 2 | 1 | 3 | 2 1/3 | 1 1/6 |
| $3^3.4^2$ | 3 | 2 | 1 | 3 | 2 | 1 |
| $3^2.4.3.4$ | 3 | 2 | 1 | 3 | 2 1/4 | 1 |
| 3.4.6.4 | 2 | 1 | 1 | 3 | 1 1/3 | 1 |
| 3.6.3.6 | 2 | 1 | 1 | 2 | 1 1/3 | 1 1/3 |
| $3.12^2$ | 1 | 1 | 1 | 2 | 2 | 1 |
| $4^4$ | 2 | 1 | 1 | 2 | 1 | 1 |
| 4.6.12 | 1 | 1 | 1 | 2 2/3 | 1 1/3 | 1 |
| $4.8^2$ | 1 | 1 | 1 | 2 | 1 | 1 |
| $6^3$ | 1 | 1 | 1 | 1 | 1 | 1 |

**Table 1.** Space Emulation Efficiency

| Host Topology | Emulation Efficiency in Cycles | | |
|---|---|---|---|
| | Hexagonal | Orthogonal | Triangular |
| $3^6$ | 1 | 1 | 1 |
| $3^3.4^2$ | 2 | 1 | 1 |
| $3^2.4.3.4$ | 2 | 1 | 1 |
| 3.4.6.4 | 3 | (3) | (3) |
| $3.12^2$ | 6 | (6) | (6) |
| $4^4$ | 2 | 1 | 1 |
| 4.6.12 | 5 | (5) | (5) |
| $4.8^2$ | 4 | 3 | (3) |
| $6^3$ | 3 | 3 | (3) |

**Table 2.** Time Emulation Efficiency

Figure 2
TRIANGULAR SPACE EMULATION

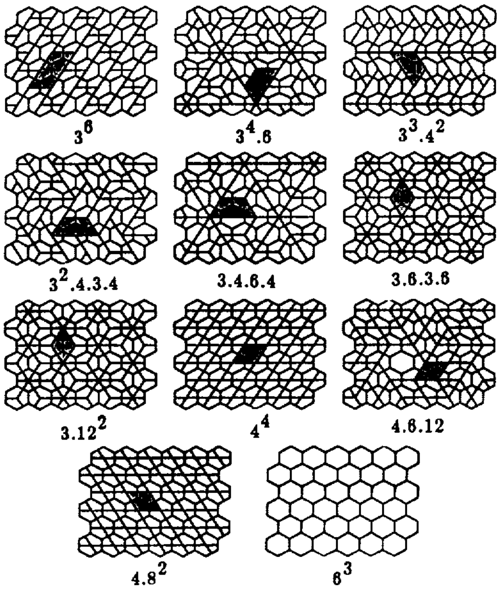
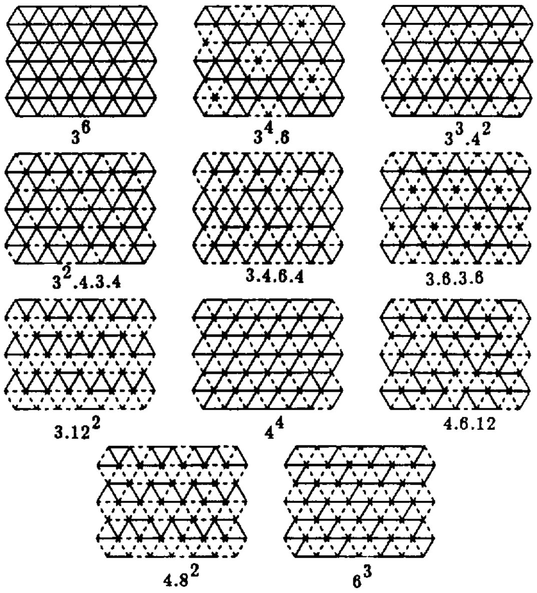


Figure 3
HEXAGONAL SPACE EMULATION



# Interconnection Networks for Fifth-Generation Computers

Bernard L. Menezes

Dept. of Elec. Eng. and Inst. for Advanced Computer Studies
University of Maryland
College Park, MD 20742

**Abstract:** An important subset of Fifth Generation Computers are Expert Systems. Support for the database management subsytem of such a machine must include parallel I/O, a relational engine and an interconnection network. Properties of a multiple tree-based network called KYKLOS are presented and examined in the context of this application. It is shown that this structure effectively supports the parallel executions of database operations such as the join. A brief description of the hardware of a single I/O node is included.

## .1 Introduction

Since the early part of this decade, it has been recognized that the present (fifth) and future generations of computers would have to be adept at processing knowledge about objects and the relationships between them. The logic base representing this knowledge is composed of a set of facts (the extensional database) and a set of rules (intensional database). Data-intensive logic base systems have a large extensional database and are characterized by the fact that they are expected to respond to all possible solutions to a query rather than only one solution as in AI-oriented logic processing systems. The two major components of such a logic base or expert system are the relational database management system (RDBMS) and the inferential subsystem. To facilitate parallelism in search, join, aggregate operations, etc., each relation in the extensional database is assumed to be fragmented across several I/O nodes. The concern of this paper is the provision of architectural support for the parallel execution of queries directed at the RDBMS. In particular, the communication medium or interconnection network (ICN) linking the I/O nodes will be the principal focus.

An ICN linking processors, memory modules and I/O nodes in a system may, in addition, have dedicated switching nodes as shown in Fig. 1(a). These networks, called dynamic or switching topologies, were first proposed for telecommunication where a great deal of research was conducted on the feasibility of using such networks to ensure nonblocking communication paths between every pair of subscribers. A special class of these networks called multistage interconnection networks (MIN's) has been investigated in connection with their applicability to multiprocessing systems. MIN's are parameterized by their spread and fanout, respectively designated s and f. On the other hand point-to-point networks do not have nodes dedicated to switching. An example of this class of networks is the hypercube (Fig. 1(b)).

The design of an ICN may be motivated by considerations that make no a priori assumptions of the application(s) to be run on the system. These considerations are typically topological: minimization of average or worst case interprocessor distances, for example. Indeed many of the general concerns of performance, cost and reliability may be expressed in graph-theoretic terms. Cost concerns translate to low node fanout and small bisection width if the network is to be laid out in VLSI. Finally, provision for multiple paths between node pairs would provide a level of fault tolerance. Algorithms are typically mapped onto this network. By contrast, a top down approach would select a network based on its suitability for one or more specific application(s). The methodology reported herein in similar in spirit to the bottom-up approach - a network is proposed, the graph-theoretic properties of the network alluded to are investigated and its utility in the RDBMS is explored.

The paper is organized as follows. In Section 2, some of the ICN's proposed for the RDBMS are surveyed. The KYKLOS ICN is introduced in Section 3 and its main properties are summarized. Finally, the proposed RDBMS architecture is presented in Section 4.

## .2 Background

There have been a plethora of papers on the utilization and comparison of various ICN's for parallel RDBMS's. Strategies for performing database operations in a hypercube-based multicomputer system with parallel I/O that involve data redistribution operations such as tuple balancing were presented in [BARU87]. In [BABA87], a cost-performance analysis of a switching network as a function of the number of stages was undertaken. In [HSIA87], several networks

CH2649-2/89/0000/0503$01.00 © 1988 IEEE

including the bus, tree and ring were compared. Both tree networks as well as the broadcast bus fared well in the comparison for a host of database operations. In addition, a tree architecture unlike the bus lends itself well to the sort-and-merge operation (in O(n) time) which is of importance in database applications. Tree networks have other virtues: they are simple to build, are incrementally extensible, have small fanout and logarithmic delay between any two nodes in the case of the full m-ary tree.

While a tree is an ideal architecture for a broadcast and for the sort-and-merge operation, there is another comunication paradigm that is frequently encountered in query processing of horizontally fragmented relations (i.e. relations fragmented by rows). This is the N-broadcast which involves transmission of a message, $m_{ij}$ from a node i to node j for every pair i, j. Clearly, half the messages would pass through the root resulting in a bottleneck. The other problem with the tree is poor reliability.

There have been several approaches to the rectification of these problems. Adding links between nodes at the same level in the tree to bypass traffic through the root – this strategy was implemented in the half-ring and full-ring X-Tree [DESP78] and in the Hypertree [GOOD81] (see Figs. 2(a) and 2(b)). A scheme by Hayes [HAYE76] (Fig. 2(c)) provided 1 fault-tolerance (tolerance to any single node fault) by the addition of a spare node at each level of the binary tree and the addition of extra links between every pair of adjacent levels. In addition, it was shown to be optimal with respect to the number of links added.

The KYKLOS Network was an attempt to integrate the concerns of performance and fault tolerance. This structure makes use of two or more trees to increase the communication bandwidth and provide fault tolerance without increasing the fanout of the nodes in the original binary tree. In addition, a skew in the connections between the trees alleviates the root bottleneck problem.

## .3 The KYKLOS Interconnection Network

The general form of KYKLOS is an interconnection network consisting of r sets of m-ary trees joined so that they share a common set of leaf nodes [MENE88]. Fig. 3(a) shows a KYKLOS of 4 ternary trees sharing the same set of leaf nodes.

### .3.1 Interconnection Strategies

**KYKLOS-I**

The simplest version of KYKLOS is the simple double tree (Fig. 3(b)) used for database problems[SONG80]. Here, the bottom tree is a mirror image of the top tree.

**KYKLOS-II: The shuffle-connected Bottom Tree**

The connection between any two adjacent levels of the bottom tree (level -i and -(i+1)) may be visualized by considering level -i nodes, $0 \leq i < n$, split into two groups ı.e. node 0 to node $2^{n-i-1} - 1$ in one group and nodes $2^{n-i-1}$ to $2^{n-i} - 1$ in the second group (Fig. 3(c)). The jth node at level -(i+1) is connected to the jth node in each of the two groups. Descendants of level -(i+1) nodes from left to right are thus ordered as a perfect shuffle. This idea can be generalized to a double tree with arbitrary branching factor where an m-way shuffle may be used to define the bottom tree of KYKLOS-II. Thus

For 1 ≤ j < n;

level -j node <-j, i>, 0 ≤ i < $m^{n-j}$

is connected to level -(j-1) nodes:

<-j+1, i>, <-j+1, i+$m^{n-j}$ >,... <-j+1, $i + (m - 1)^{n-j}$ >

and to level -(j+1) node

<-j-1, i mod $m^{n-j-1}$ >

Root node <-n,0> is connected to nodes:

<-n+1, 0>, <-n+1, 1>,... <-n+1, m-1>.

The interconnection for the top tree is as in KYKLOS-I.

### .3.2 Properties

The key properties of the KYKLOS network are (details of these may be found in [MENE88]):

1) *Network Cost* (as measured by the number of non-leaf nodes) is a linear function of the number of leaf nodes. Also, the degree (fanout) of each non-leaf node is uniformly three or less.

2) The *maximum traffic density* under the uniform message distribution assumption in KYKLOS-II is $O(N^{1.5})$ as compared with the $O(N^2)$ traffic density in the single binary tree or KYKLOS-I.

3) *Average Communication Delay* (leaf-to-leaf) is further reduced over that in the simple binary tree.

4) *Routing* is straightforward. Also there are several routing strategies, each of which map to a different set of traffic or distance characteristics.

5) *Fault Tolerance* is improved by virtue of the multiple tree nature of the structure. Further, conectivity in the case of KYKLOS-II is superior to that of KYKLOS-I.

## .4 The KYKLOS-based RDBMS Architecture

Fig. 4 is a block diagram of the proposed RDBMS. The host processor is assumed to be a supercomputer connected to the roots of the KYKLOS-II network. The leaves, called I/O nodes in this application, consist of a conventional moving head disk, associative disk cache, a sort engine, and a general-purpose microprocessor. An important component of the I/O node is the sort engine design utilizing an array of Content Addressable Data Manager (CADM) chips manufactured by Advanced Micro Devices. 16 CADM chips (a total of 16K bytes i.e. (# of records).(key width + pointer width) = 16KBytes) comprise the sorter, it is expandible to 256K. The choice of these CADM chips was based on its sort speedup of 15-50 over the VAX 11/780 and 1.5 over software sort on the CRAY X/MP-24 in addition to performing search and set operations. The nonleaf nodes couple host and leaf (I/O) nodes and incorporate logic and buffering to support merge operations on data streams.

The database is organized as a set of relations R, S, etc. which are the base relations upon which the leaf level operations of a query tree operate. Each of these relations is partitioned horizontally into fragments $r_i$, $s_i$, etc. residing at I/O node i so that $\cup_{i=0}^{N-1} r_i = R$. Each fragment has associated with it a qualification which describes the common properties of tuples in that fragment. The fragmentation scheme enables fragments of a relation to be accessed in parallel besides parallel processing of a query. Queries from the host are compiled into a set of relational algebraic operations at the root node. The utility of the proposed configuration for the computation-intensive join operation is next exemplified. It is assumed that a join between relations R and S is to be performed. The results could be sent back to the host or may return to the leaves as a temporary relation for the next phase of a query such as a multi-way join.

Suppose that relations R and S are partitioned on attributes a and b respectively, $a \neq b$, and a join on attribute a is to be performed. A hash-based join [SHAS86] would necessitate repartitioning S on attribute a based on hash function h used to partition R i.e. if t ∈ S, t is sent to h(t.a). The communication requirements for this phase can be modeled using the N-broadcast paradigm (i.e. each pair of processors needs to communicate). This is particularly well-suited to the KYKLOS-II network which shows an asymptotic improvement over the binary tree or KYKLOS-I. Once repartitioning takes place, the partial joins may be computed locally. If the tuples need to be sorted on an attribute other than the join attribute as is often the case, the sort engine at each I/O node could be used and the results merged using the two trees. The network in this example has been used for communication and for the merge operation. However, the nonleaf nodes may be used to perform some of the partial joins as in the semi-join algorithm described next.

The semi-join algorithm involves a broadcast of the join attribute values from each node upon which a selection of joinable tuples between each pair of nodes is made. Let $r_{i,j}$ and $s_{j,i}$ denote the fragments of $r_i$ and $s_j$ that participate in the join. The partial join between $r_{i,j}$ and $s_{j,i}$ may then take place at i, j, or a predetermined site in the network. To distribute the joins throughout the network, the sites for the partial joins could be assigned based on the set of mappings, $\nu_i, 0 \leq i < N$ :

$$\nu_i(l) = < -(k+1), l \; mod \; 2^{n-k-1} >$$

where $i \oplus l = 0^k 1u$, i and l are n-bit binary strings, u is a n-k-1 bit binary string with $0^k$ representing a string of k consecutive zeros, and n is the height of each tree in KYKLOS.

Then the partial join between i and j takes place at $\nu_i(j)$. Note that $\nu_i(j)$ represent a non-leaf node address; the first element of the tuple represents level number and the second element represents a node within that level.

An example of this allocation strategy for i=0 is shown in Fig. 5.

Two characteristics of this strategy are:

1) *Load Balancing* The joins are uniformly distributed throughout the nodes of the network. Note that if $r_i \bowtie s_j$ takes place in Tree 2, $s_i \bowtie r_j$ could take place in Tree 1 by an assignment which corresponds to $\nu_i$ with k defined by $i \oplus l = u10^k$.

2) *Traffic Equalization* The traffic of the inputs of the partial join is always rootward in each tree. Thus, there is almost a perfect balance in traffic through every link in the tree.

In conclusion, an augmented binary tree such as KYKLOS appears to be an attractive choice in the design of the RDBMS of an expert system because the operations involved in parallel query processing such as sort-and-merge, broadcast and N-broadcast are well supported with this structure. Finally, logarithmic internode path lengths, low fanout, fault tolerance and ease of interprocessor routing are desirable characteristics that further enhance its utility.

### Bibliography

[BABA87] T.Baba et al, "A Network Algorithm for Relational Database Operations" in Database Machines and Knowledgebase Machines by M. Kitsuregawa and H.Tanaka, Kluwer Publishers, 1988.

[BARU87] C.K.Baru and O.Frieder, "Implementing Relational Database Operations in a Cube-connected Multicomputer System", *Proceedings of the Third Int. Conf. on Data Engineering*, Feb. 1987, pp. 36-43.

[DESP78] A.M.Despain and D.A.Patterson, "X-Tree: A Tree Structured Multi-processor Computer Architecture", *Proceedings of the Fifth International Symposium on Computer Architecture*, April 1978, pp. 144-151. Computer Architecture, Dec. 1973, pp. 21-28.

[GOOD81] J.R.Goodman and C.H.Sequin, "Hypertree: A Multiprocessor Interconnection Topology," *IEEE Transactions on Computers*, C-30, Dec. 1981, pp. 923-933.

[HAYE76] J.P.Hayes, "A Graph Model for Fault-Tolerant Computing Systems," *IEEE Transactions on Computers*, C-25, Sept. 1976, pp. 875-884. the University of Texas at Austin, Austin, Tx., May 1987.

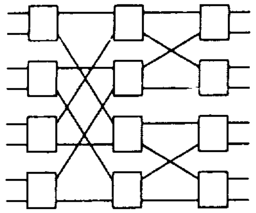
[MENE88] B.L.Menezes, " The KYKLOS Multicomputer Network: Interconnection Strategies, Properties, and Applications" Ph.D. Dissertation, Univ. of Texas at Austin, Austin, Texas 78712.

[REED87] D.A.Reed and D.C.Grunwald, "The Performance of Multicomputer Networks," *Computer*, 20-6, June 1987, pp. 63-73.

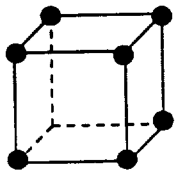
[SHAS86] D.Shasha, "Query Processing in a Symmetric Parallel Environment", *Proc. of the Sixth Advanced Database Symposium*, Aug. 1986.

[SONG80] S.W.Song, "On a High-Performance VLSI Solution to Database Problems", *Ph.D. Thesis*, Department of Computer Science, Carnegie Mellon University, Pittsburgh, Pa., 1980.

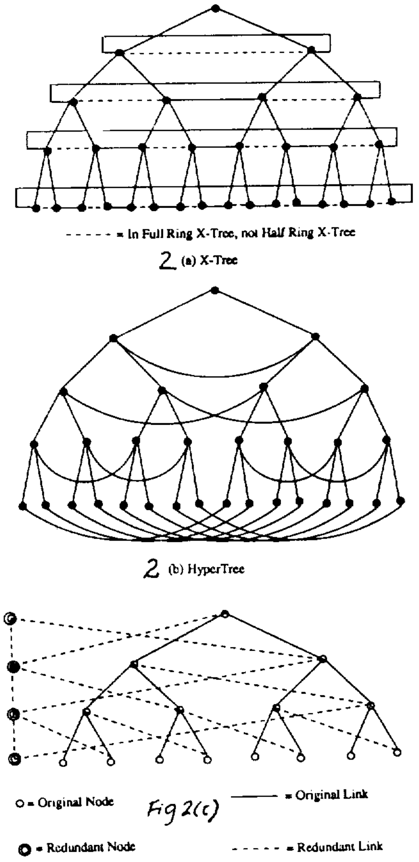
[WU84] C.Wu and T.Feng, *Tutorial: Interconnection Networks for Parallel and Distributed Processing*, IEEE



1a) Dynamic Switching Network



1(b) Static Network


----- = In Full Ring X-Tree, not Half Ring X-Tree
○ = Original Node
——— = Original Link
◎ = Redundant Node
- - - - - = Redundant Link

2 (a) X-Tree

2 (b) HyperTree

Fig 2(c)

Fig 2: Augmented Binary Tree Topologies

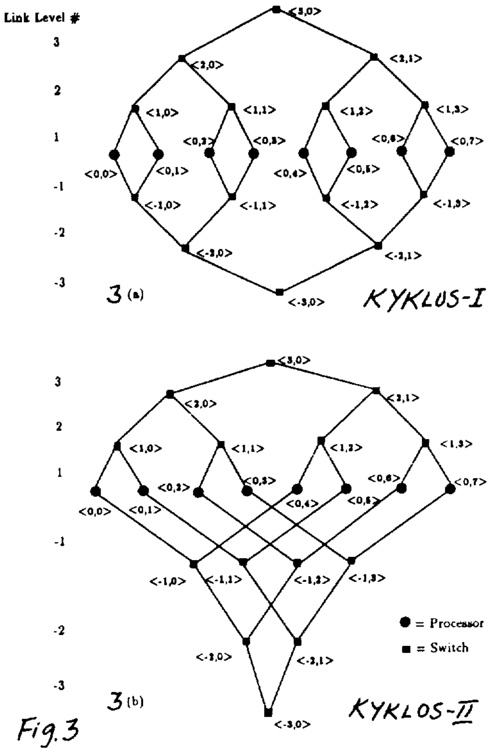


Fig.3

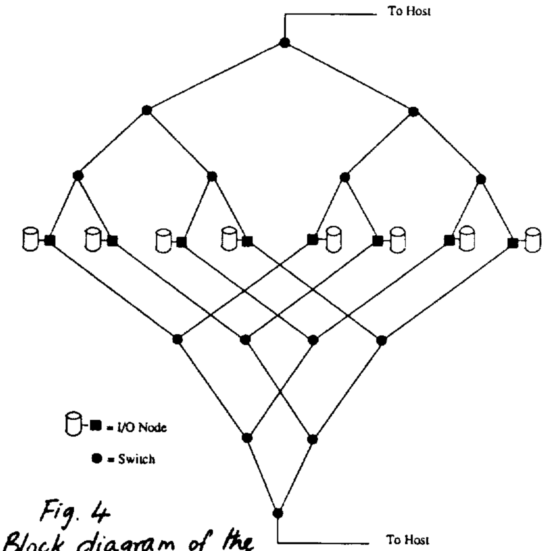


Fig. 4
Block diagram of the RDBMS

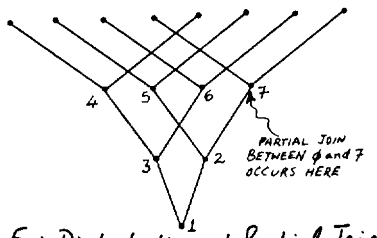


Fig. 5 : Distribution of Partial Joins between Node ϕ and other nodes. using $\nu_0$.

# RELIABILITY CONSIDERATIONS IN LARGE-SCALE COMPUTING SYSTEMS*

W. Najjar and J.-L. Gaudiot[†]

[†]University of Southern California
Department of EE-Systems, MC 0781
Los Angeles, CA 90089

Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292

*ABSTRACT: The technological availability of large scale multiprocessor systems raises new issues in the design of highly fault-tolerant architectures. The hardware redundancy inherently available in such systems can be used for increased throughput, faster execution and improved reliability. This paper addresses the issue of scalability in the reliability analysis of large scale degradable homogeneous multiprocessors. It is shown that the amount of processor-hours a realistic system can deliver is upper bounded independently from the number of processors. The results demonstrate that graceful degradation in large-scale systems is not scalable: an increase in the number of processors must be matched by a significant increase in the coverage factor in order to maintain the same performance and reliability levels.*

## 1 Introduction

The advent of large-scale multiprocessing, where several hundred processors cooperate on the same computation, has placed the issue of fault-tolerance and reliability analysis for computing systems under a new light. A large number of Processing Elements not only increases the processing speed, but also the likelihood that one or more elements will fail. Therefore, the overall system reliability becomes a key issue in the design, implementation and performance analysis of large-scale systems. Infact, the overall failure rate increases with the number of processors and with the complexity of the underlying interconnection network with respect to the failure rate of a single processor using a comparable technology. This increase is at least proportional to the number of processors and results in a decrease of the expected up-time of the system. The focus of this paper, therefore, is on analyzing the *scalability* of large-scale gracefully degradable computing systems. The main motivation behind this analysis stem from the following considerations: would the reliability of such systems impose a limit on the number of processors that can be put to cooperate on one problem? Our analysis shows indeed that the amount of reliable processor-hours a large-scale gracefully degradable system is upper-bounded.

Traditional techniques of reliability and performability analysis are used to evaluate the asymptotic behavior of measures such as the *mean-time-to-failure*, and the *mission-time*. The concept of *computational reliability*, as introduced by Beaudry [1] is used as a tool to evaluate the measure of *reliable computational work* as functions of the *number of processors*.

## 2 System and Fault Models

The model under consideration is that of a large-scale, homogeneous multiprocessor. The computation is, initially, uniformly partitioned among $N$ identical processing elements. The system is assumed to support graceful degradation. Upon the detected failure of a processor its computational load is picked up by another processor or set of processors with near uniform load partitioning. A distributed fault-tolerance algorithm relies on the detection and isolation of faults and system recovery and reconfiguration. The ability of a system to gracefully degrade hinges on the *combined* success of these step. The failure of any step can potentially result in a total system failure. The cumulative probability of success of these three steps is expressed by the *coverage factor* [2]. The analysis in this paper is based on the following simplifying and strongly optimistic assumptions:

- No communication costs among processors.
- Fail proof communication links.
- No overhead associated with recovery and reconfiguration procedures.

Although these assumptions are unrealistic, they are justifiable in an analysis of asymptotic behavior. Based on such simplifying assumptions, this analysis will determine upper-bounds that are never reached in practice. The system is modeled by a continuous-time Markov chain (CTMC), shown in Figure 1, [3, 4]. Since our analysis will focus on gracefully degradable systems, we will not consider system repair and therefore the CTMC is acyclic. In this model $P_i(t)$ is the occupation probability of *state* $i$, $i$ being the number of failed processors, $i = 0, \ldots, D-1, F$; where $F$ is the state of total system failure. $D$ is the number of allowable degradation states, expressed as a function of $N$. $c$ is the coverage factor, which is the probability of successful recovery from a single failure. The state

*This paper is based upon research supported by the National Science Foundation under Grant No. CCR-8603772 (USC/Department of Engineering - Systems), and by the Office of Naval Research, Arlington, VA under Contract No. N00014-86-K-0311 (USC/Information Sciences Institute).

CH2649-2/89/0000/0507$01.00 © 1988 IEEE

probabilities can be derived as:

$$\begin{aligned} P_0(t) &= e^{-N\lambda t} \\ P_i(t) &= c^i \binom{N}{N-i} (e^{-\lambda t})^{(N-i)}(1 - e^{-\lambda t})^i \\ & \qquad\qquad\qquad\qquad i = 1, \ldots, D-1 \end{aligned}$$

The reliability $R(t)$ is simply the probability of being in any one of the states $i = 0, \ldots, D-1$. Therefore, $R(t) = \sum_{i=0}^{D} P_i(t)$ and the *mean time to failure* ($MTTF$), which is the expected time to first failure, is

$$MTTF = \int_0^\infty R(t)dt = \int_0^\infty \sum_{i=0}^{D-1} P_i(t)dt = \frac{1}{\lambda c}\sum_{i=1}^{D-1}\frac{c^i}{i} \qquad (1)$$

The Mission Time, $MT$, is defined for a given minimum reliability $R_{min}$ as the time interval during which the reliability is larger than $R_{min}$ i.e., $R(MT) = R_{min}$.

Unless otherwise noted, in the rest of this discussion, we will assume a fully degradable system. This means that the system allows graceful degradation for up to $N-1$ failures, in other words, $D = N-1$. The unit-time will be taken as $1/\lambda = MTTF_1$ (i.e, the MTTF of a single processor) and a value of $R_{min} = 0.99$.

## 3 Time-Based Analysis

The expression for $MTTF$ in Equation 1 indicates that increases in $N$ have diminishing effects on the value of the expected time to first failure. In fact, an increase from $N$ to $N+1$ processors results in a minimal increase in $MTTF$:

$$MTTF(N+1) - MTTF(N) = \frac{c^{N+1}}{N+1}$$

since $c < 1$, the increase becomes insignificant for large values of $N$. The values of $MTTF$ are plotted in Figure 2 as a function of $N$ for different values of $c$. The series in Equation 1 is not convergent but has a logarithmic behavior. Therefore there is no asymptotic limit to $MTTF$. However, for all practical purposes, the mean time to failure can be considered constant for sufficiently large $N$ given a value of the coverage factor. Let $N_k$ be the value of $N$ at the knee of the curve in Figure 2, then $MTTF(c, N) \approx MTTF(c, N_k)$ for $N > N_k$. From the above analysis, we can conclude that the effects of the probability of successful recovery, as expressed by the coverage factor $c$, on the mean time to failure, $MTTF$, increases with an increase in the number of processors. Also that, for larger systems, the mean time to first failure is a constant function of the coverage factor and is independent of the number of processors.

The values of $MT$ are plotted in Figure 3 as a function of $N$ for different values of $c$. These curves show that for a given value of $c$, there exists a value of $N$, denoted by $N_p$, at which the $MT$ is maximal. It is clear from these curves that for smaller values of $N$ ($N < N_p$) the inherent redundancy of the system provides a higher mission time. As $N$ increases ($N > N_p$) the higher failure rate dominates and reduces the mission time. Furthermore, as $c$ is increased, the value of $N_p$ also increases. From this analysis we can deduce

- The peak value of $MT$ is significantly larger than that of a single processor. For example $MT(N_p, 0.999) \geq 93MT(1)$.
- As the number of processors is increased, i.e., $N > N_p$, we observe a decrease in $MT$ that is inversely proportional to $N$, $MT(2N, c) = 0.5MT(N, c)$.
- While the peak value of $MT$ for a multicomputer system can be significantly larger than that of a single processor, the reverse becomes true for very large values of $N$, $MT(1) = 10MT(1024, 0.99)$.
- For $N > N_p$, a 10 fold decrease in $(1-c)$ (the probability of failed recovery) results in a 10 fold increase in the $MT$ for the same number of processors. For example, $MT(128, 0.999) = 10MT(128, 0.99)$. In other words, the mission-time is inversely proportional to $(1-c)$.

Because of the cumulative effects of the probability of successive recovery, the reliability of the system, after the $i^{th}$ failure, is constrained by $R(t) \leq c^i$. Let $K'$ be defined such that $c^{K'} = R_{min}$. Let $K$ be the integer value of $K'$

$$K = \left\lfloor \frac{\log R_{min}}{\log c} \right\rfloor \qquad (2)$$

Given the definition of $MT$, $K'$ is the expected number of failures in the interval $[0, MT]$ constrained by the condition that $D \geq K'$. In other words, if the number of processors is large enough, $K$ failures are sufficient to reach $R(t) = R_{min}$. From this we can deduce that when the number of allowed degradation states is sufficiently large (i.e., $D \gg K$), the necessary condition to reach the minimum reliability level ($R(t) = R_{min}$) and therefore the mission-time is that $K$ processors fail. Since the rate of failures is proportional to the number of processors, the time interval $[0, MT]$ is inversely proportional to $N$. On the other hand, for $\epsilon << 1$ and $x = 1-\epsilon$, we can use $\log x \approx 1 - x$. Therefore, for $D \gg K$ $MT \propto \frac{1}{(1-c)N}$. This proportionality expression implies that in order to maintain a constant mission-time, any increase in the number of processors must be matched by an equivalent decrease in the probability of *failed recovery*: $(1-c)$.

## 4 Computation-Based Analysis

In this section we present an evaluation of performance and reliability of large-scale degradable systems based on the notion of *computational work*. There is no formally defined unit of computational work. In this analysis we will use *processor-hours* as units of computational work. Another related unit of computational work is machine instructions. Any computational task is characterized by a certain amount of computational work, measured in processor-hours. When this task is executed over several processors, the execution time is reduced, but the amount of processor-hours required for that computation is kept constant if the speed-up is linear. For non-linear speed-ups the amount of required processor-hours increases due to added overhead.

Let $T_n$ be the execution time of a given computation over $n$ processors. $S_n$ is the attainable speed-up defined by $S_n = \frac{T_1}{T_n}$. $S_n$ is equivalent to the number of *effective processors* (i.e the number of virtual processors fully utilized by the given computation). We define $CW(N, t)$ as the amount of *effective computational work* a system will deliver for a given computational speed-up.

$$CW(N, t) = \int_{\tau=0}^{t} \sum_{i=0}^{D-1} S_{(N-i)} P_i(\tau) d\tau \qquad (3)$$

Based on the model described in section 2, we define $PH(N,t)$ as the amount of processor-hours a system, with initially $N$ processors, can deliver up to time $t$, as:

$$PH(N,t) = \int_{\tau=0}^{t} \sum_{i=0}^{D-1} E_{(N-i)}(N-i)P_i(\tau)d\tau \qquad (4)$$

For a computation that exhibits linear speed-up (i.e. $S_n = n$) we have: $CW(N,t) = PH(N,t)$. Note that $PH(N,\infty)$ is the *mean computation before failure* (MCBF) and $CW(N,t)$ is the integral of the *computational availability*, $a_C(t)$, as defined in [1]. Both $PH(N,t)$ and $CW(N,t)$ are *expected* values of the processor-hours and computational work measures.

From this definition we can prove that the amount of computational work a purely degradable system can deliver is upper bounded and that the upper bound is independent of the initial number of processors.

**Theorem 1** *$\forall N$ and $c < 1$ $\exists$ $PH_{max}$ such that $PH(N,t) < PH_{max}$, $\forall t$.*

Extensive proofs of this theorem can be found in [5]. The conclusion from Theorem 1 is that no matter how large the initial number of processors is, there is an upper bound on the amount of processor-hours that are obtainable when $c < 1$. This upper bound is determined by $c$ only and is reached asymptotically. $PH_{max}$ is therefore the upper limit on the *mean computation before failure* (MCBF). Therefore increasing the system size does not increase the amount of expected computational work the system can deliver before total failure.

### 4.1 Reliable Processor-Hours

The measure of *reliable processor-hours*, $RPH$, is defined as the amount of processor-hours available while the reliability is maintained above a given minimum, i.e,

$$RPH(N,c) = PH(N,MT) = \int_{t=0}^{MT} \sum_{i=0}^{D-1} (N-i)P_i(t)dt \qquad (5)$$

The results of evaluating the $RPH$, according to Equation 5, are presented in Figure 4 for $D = N - 1$, for values of $c$ in the range $[1, 0.99]$, and for $R_{min} = 0.99$. The values of $RPH(N,c)$ are expressed in $processor - hours$ where the unit time is taken as $1/\lambda$. Two observations can be made:

1. for a given value of $c$, there exists a value of $N$, $N_{ph}(c)$, beyond which an increase in $N$ will not increase the amount of reliable processor-hours.
2. the values of $N_{ph}$ and $RPH_{max}$ increase with increasing values of $c$.

Theorem 1 states that the expected amount of processor-hours in the interval $[0,\infty]$ is upper bounded by $PH_{max}$. These results show that the expected amount of processor-hours in the interval $[0,MT]$ is also upper bounded by $RPH_{max}$. $RPH_{max}$ is therefore the maximum expected amount of computational work the system can deliver subject to the constraint of $R(t) \geq R_{min}$. In the next section we present an analytical derivation of $RPH_{max}(c)$. The maximum value of $RPH(N,c)$, $RPH_{max}(c)$, can be derived analytically by using the expression for the expected number of failures in the interval $[0,MT]$, $K$, as $RPH_{max}$ $RPH' = \frac{1-c^K}{1-c}$. These results show that there is no increase in reliable computational work when $N$ is increased above $N_{ph}$ for a given value of $c$. This confirms the results obtained in the $MT$ based evaluation. It appears, therefore, that for a given value of the coverage $c$ there exists an optimal value of $N$, $N_{opt}$, that would maximize the mission-time $MT$ and the amount of reliable processor hours.

### 4.2 Reliable Computational Work

$RPH$ evaluates the amount of reliable processor-hours potentially available from the system. The fraction of $RPH$ that is actually used by a computation depends on the speed-up $S_n$ of the computation. Similarly to $RPH$ we define $RCW$ as $RCW(N,c) = CW(N,MT)$ $RCW$ is therefore the amount of *effective reliable computational work* a system can deliver with respect to a given computation while $R(t) \geq R_{min}$. In evaluating $RCW$, we will take as example a sub-linear speed-up case where $S_n = \frac{n}{\log n}$. The results, plotted in Figure 5, show that there exists a value of $N$ denoted by $N_r$ at which the value of $RCW$ is maximal. Figure 6 shows the plot of both $RPH$ and $RCW$ versus $N$ for $c = 0.995$ and $S_n = \frac{n}{\log n}$. This implies that as the system size is increased over $N_{ph}$, the probability of a computation not completing reliably decreases if the speed-up of the computation is sub-linear. This result has implications on the scalability of graceful degradation. For a large-scale gracefully degradable system to be scalable, any increase in the system size should be matched by an increase in the quality of the recovery scheme, i.e., the coverage factor, in order to maintain the same performability level.

## 5 Summary and Conclusions

In this paper we have addressed the issue of scalability in large-scale gracefully degradable systems. The objective being to assess the limitations that reliability considerations would impose the number of processors in a massively parallel. Traditional measures of reliability, such as mission-time and mean-time-to-failure, as well as measures of computational reliability have been evaluated as functions of the number of processors. It was shown that the mean-time-to-failure is, for all practical purposes and for large system sizes, a constant function of the coverage factor and is independent of the number of processors. The analysis of the mission-time shows that, for a given value of the coverage factor, there exists a value of the number of processors at which the mission-time is maximal. As the system size is increased beyond this value, the mission-time becomes a decreasing function of the number of processors. The measure of processor-hours was defined as the amount of potential computational work. This measure was shown to be upper-bounded and the upper-bound independent of the initial number of processors. For computations with linear speed-up, it is shown that the amount of reliable computational work is constant for large system sizes. When the speed-up is not linear, this amount is a decreasing function of the number of processors. Therefore, for large system sizes and same technology, increasing the number of processors results in a decrease of the expected amount of reliable computational work the system can deliver. It is therefore demonstrated that graceful degradation in large-scale fault-tolerant systems is not scalable. In order to preserve the same performance and reliability level an increase in the number of processors should be matched by a decrease of the same magnitude in the probability of failed recovery.

## References

[1] M.D. Beaudry. Performance-related reliability measures for computing systems. *IEEE Transactions on Computers*, C-27(6):540–547, June 1978.

[2] W.G. Bouricius, W.C. Carter, D.C Jessep, P.R. Schneider, and A.B. Wadia. Reliability modeling for fault-tolerant computers. *IEEE Transactions on Computers*, C-20(11):1306–1311, November 1971.

[3] K.S. Trivedi. *Probability and Statistics with Reliability, Queueing and Computer Science Applications.* Prentice-Hall, Englewood Cliffs, N.J., 1982.

[4] D.P. Siewiorek and R.S. Swartz. *The Theory and Practice of Reliable System Design.* Digital Press, Bedford, Mass., 1982.

[5] W. Najjar. *Fault-Tolerance and Reliability Analysis of Large-Scale Multicomputer SystemsJuly.* PhD thesis, University of Southern California, July 1988.

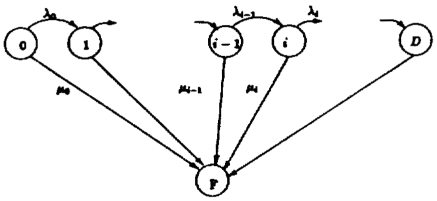


Figure 1: Markov model of failures

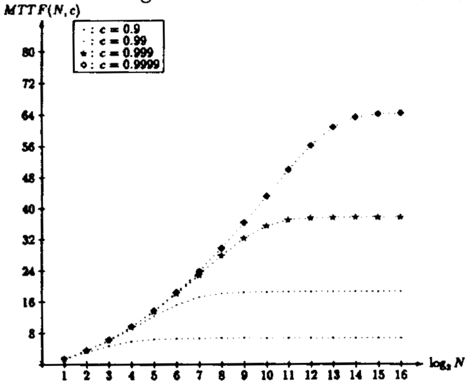


Figure 2: *MTTF* as function of $N$ and $c$

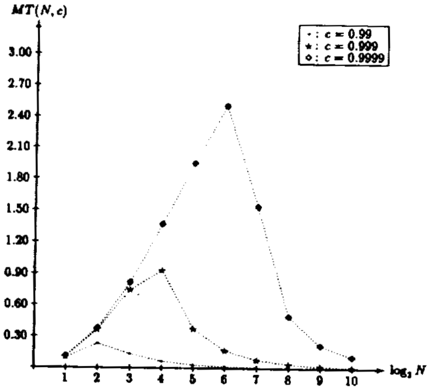


Figure 3: *Mission Time* as function of $N$ and $c$, $(D = N - 1)$

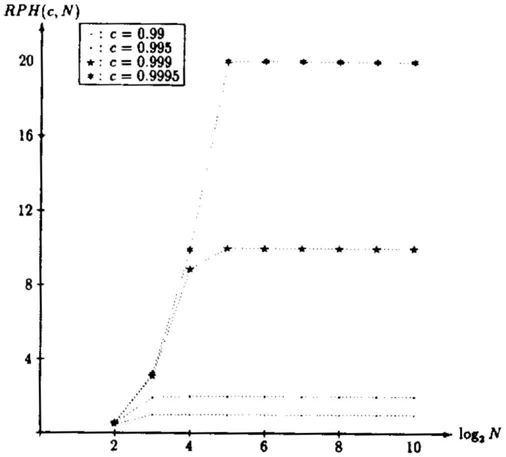


Figure 4: *RPH* as function of $N$ and $c$, $D = N - 1$

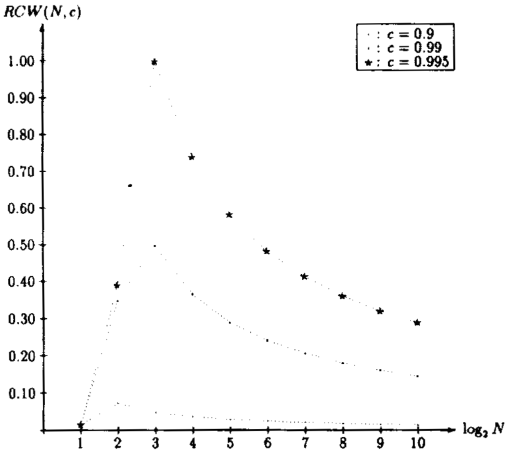


Figure 5: *RCW* as function of $N$ and $c$ for $S_n = \frac{n}{\log n}$

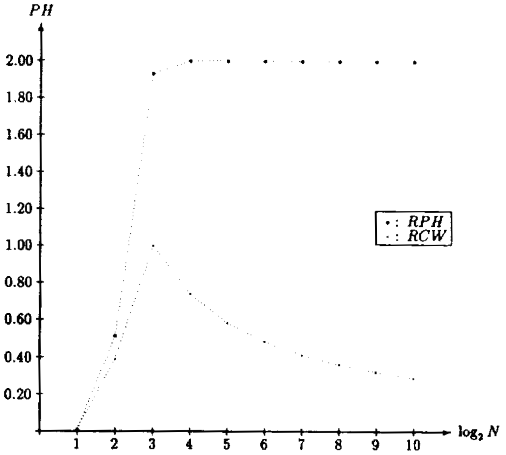


Figure 6: *RCW* and *RPH* as function of $N$, $c = 0.995$ and $S_n = \frac{n}{\log n}$

# Design Considerations for a Pyramidal Cellular Logic Processor

*Joseph J. Pfeiffer, Jr.*

Department of Computer Science
New Mexico State University
Las Cruces, NM

## *ABSTRACT*

A number of cellular array processors have been successfully constructed over the last several years. In general, these processors have concentrated on interconnection topologies and processing element microarchitectures to the exclusion of instruction set and high level language issues. Even in those cellular processors with explicit hardware support for particular operations (*e.g.* multiplication support in MPP), issues of actual instruction set design and support are secondary to low-level microarchitectural details.

Continued advancement of cellular processors requires consideration of the processor as a component in the overall context of executing parallel algorithms, and so requires consideration of issues including parallel high level languages, user-level assembly languages, and efficient microarchitectures. Additionally, high-bandwidth input and output must be combined with efficient data structure conversion hardware to provide a mechanism for communication of mid-level image data between the cellular processor and external processors for mid-level and high-level processing.

This paper describes a project involving all these levels, providing an integrated environment for pyramidal problem solving. The use of a top-down design approach, beginning with algorithms, then language development, and finally architecture and microarchitecture, has resulted in an exceptionally clean and well-integrated processor for pyramid operations.

## 1. Introduction

While a number of parallel computers, such as ILLIAC IV, were built by the late 1960s, the first modern cellular array processor constructed was the ICL DAP (Ref 6 ). This processor included a number of features common to most of its successors, including bit-serial processing elements (PEs) and a grid interconnection pattern. Also in common with most of its successors, it possessed a number of special features; in the case of DAP, the most noteworthy was the presence of row/column highways for communication of data throughout the array.

While DAP was extremely fast for its time, it also suffered from a malady common to most of its successors: communication of instructions and data between the host and the PE array was very cumbersome, resulting in underutilization of the PE array. More recent processors have generally shown similar strengths and weaknesses. In general, much more attention has been paid to interconnection topologies and PE designs than to host/PE array integration.

A second weakness in cellular processors has been their Input/Output systems. Typically, the IO rates in these processors have been far below that necessary to maintain efficient processing in the PE array. Some systems (notably MPP) have addressed this problem through high-bandwidth IO subsystems, operating asynchronously (Ref 1 ). Even these systems, however, are weak in converting data between formats which are useful for the cellular processor and formats which are useful for the host.

A current project under way at NMSU is the development of an architecture for computer vision which will successfully integrate low-level, intensity array processing with high-level, graph structure processing. Issues of concern to this project include data structure conversion between low-level and high-level representations (Ref 4 ), graph structure representations suitable for parallel manipulation (Ref 5 ), model-based reasoning (Ref 2 ), and development of architectures for graph structure manipulation. The ultimate goal is the development of a unified architecture capable of structural analysis of images at realtime rates.

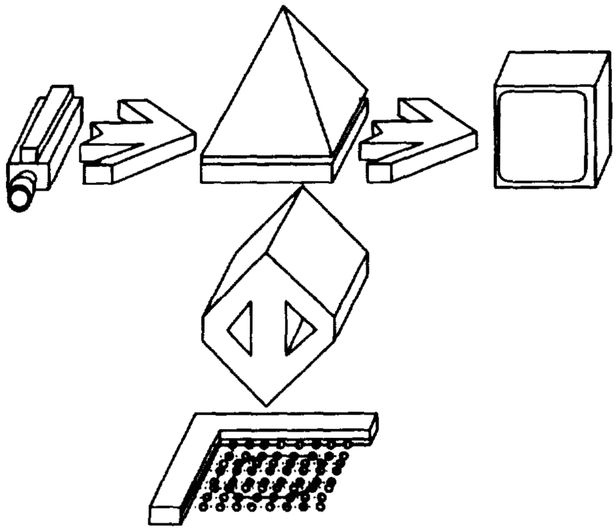


**Figure 1: Integrated Computer Vision Architecture**

A view of the eventual system is shown in Figure 1.

In this paper, we discuss a massively parallel component of the overall project: a pyramid processor for performing low level image processing. The processor is designed from the ground up with the goal of providing a set of SIMD pyramid operations useful to programs written in a high level language, for eventual feature extraction. The approach taken begins with a consideration of the operations required to support typical vision operations, and embeds these operations in the C programming language as parallel constructs. A compiler for this extended C (called HCL, for Hierarchical Cellular Logic) defines the set of low-level pyramid operations required for a pyramidal cellular processor. Finally, this instruction set defines the microarchitecture of the PE's, and the capabilities of the controller.

At the same time, study of mid-level processing algorithms, and analysis of the properties of edges in an image, results in the definition of fast parallel hardware for searching edge maps in order to create graph structures for high-level processing.

This paper will consider each of these issues in turn. Section 2 deals with the parallel operations of HCL. Section 3 describes three candidate instruction sets which were evaluated for PCLIP II, supporting stack, enhanced stack, and two-operand instructions. Section 4 describes the micrarchitecture of PCLIP II, and the impact of the microarchitecture on the final choice of instruction set. Section 5 discusses the IO subsystem.

CH2649-2/89/0000/0511$01.00 © 1988 IEEE

Finally, Section 6 provides some conclusions.

## 2. HCL: A High-Level Language for Low-Level Image Processing

Algorithm development for pyramid processors in the early 1980's led to the definition of a notation for operations on pyramids, called Hierarchical Cellular Logic (Ref 7 ). The distinguishing features of this notation are its pattern match operators, the definition of grey-level operations (such as addition) from binary operators (such as exclusive OR), and the ability to use the results of prior operations to selectively mask subsets of the pyramid. The notation was based on two pattern match operators, and the full set of bitwise Boolean operators. These features were inspired by the characteristics of bit-serial parallel processors (Ref 8 ), and by the needs of pyramid algorithms.

The set of operators supported was formalized when the language was embedded in the C programming language, resulting in a pyramid programming language called HCL (Ref 3 ). An example of a statement in HCL, calculating the Sobel edge detector, is shown in Figure 2.

```
dy = (nw_pixel(in_image) +
       2*n_pixel(in_image) +
       ne_pixel(in_Image)) -
      (sw_pixel(in_image) +
       2*s_pixel(in_image) +
       se_pixel(in_Image));
```

**Figure 2: HCL Code Calculating Gradient Component**

Development of the HCL compiler provided a set of fundamental pyramid operations, and architectural features, required to support the language. In the next section, we discuss instruction set development for HCL.

## 3. Instruction Set Support for HCL

The PCLIP II instruction set implements precisely the features needed to support HCL. As the language is based on SIMD extensions to C, the instruction set has taken a form very similar to conventional, serial instruction sets, with the two differences that (1) operations are performed on parallel objects and (2) there are no provisions for flow of control. The processor is being developed for use as a specialized functional unit in an image processor supporting a unified instruction set for scalar and parallel operations. In this environment, scalar data and flow of control will be handled elsewhere, and so are not required in the pyramidal processor. Three instruction sets have in fact been developed and evaluated: a pure stack machine, an enhanced stack machine, and a two operand machine.

Historically, the stack machine was created first, as a "Version 0" architecture which could be used to determine the routing and processing characteristics of HCL programs. The enhanced stack machine was developed next as a result of simulations of the pure stack machine, in order to reduce routing overhead. The two operand instruction set was then designed to simplify the microcontroller. This third instruction set is now being pursued for further development.

Rather than discuss the three instruction sets and their performances in the order they were evaluated, however, it is more instructive to present the instruction sets first, and then compare them. Three quantitative measures of the efficiency of the instruction set are shown for this benchmark. The number of instructions acts as a measure of the work required for the instruction dispatcher to process the code, while code size gives a measure of the bandwidth which will need to be supported between the dispatcher and the pyramid. The execution speed is, of course, the critical measure of the effectiveness of the architecture. This speed estimate is based on the PE microarchitecture, described in Section 4. A qualitative comparison of the microcontroller complexity is used.

### 3.1. Instruction Set Descriptions

In the following section, the three instruction sets are described. Along with each description, the code required to execute the HCL statement is presented as an example.

### 3.2. Stack Machine Instruction Set

The first instruction set developed is a nearly pure stack machine, corrupted only by the presence of instructions popping the stack into the E register, and adding the C register to the top of stack. The code required in this instruction set to implement the HCL statement is shown in Figure 3.

### 3.2.1. Enhanced Stack Machine

To obtain the second instruction set, three enhancements are added to the stack machine instruction set:

- Combined routing with push operations
- Performing arithmetic on the top of stack without requiring push operations
- Visible P register, with this register added to the set of possible sources and destinations for register transfer instructions

Code for this enhanced stack machine is shown in Figure 4.

### 3.2.2. Two Operand Instruction Set

The third instruction set simplifies the microcontroller by moving responsibility for stack management from it to the instruction dispatcher. The code sample for the two operand instruction set is shown in Figure 5.

### 3.3. Instruction Set Comparisons

In this section, the three candidate instruction sets are compared on four criteria: execution rate, number of instructions, code size, and microcontroller complexity.

| Size | Cycles | Instruction | |
|---|---|---|---|
| 3 | 8 | push | (8) in_image |
| 5 | 16 | and_match | x 1xx xxx xxx xxxx |
| 3 | 8 | pushint | 1 |
| 3 | 8 | push | (8) in_image |
| 5 | 16 | and_match | x x1x xxx xxx xxxx |
| 1 | 8 | lshft | |
| 1 | 16 | add | |
| 3 | 8 | push | (8) in_image |
| 5 | 16 | and_match | x xx1 xxx xxx xxxx |
| 1 | 16 | add | |
| 3 | 8 | push | (8) in_image |
| 5 | 16 | and_match | x xxx xxx 1xx xxxx |
| 3 | 8 | pushint | 1 |
| 3 | 8 | push | (8) in_image |
| 5 | 16 | and_match | x xxx xxx x1x xxxx |
| 1 | 8 | lshft | |
| 1 | 16 | add | |
| 3 | 8 | push | (8) in_image |
| 5 | 16 | and_match | x xxx xxx xx1 xxxx |
| 1 | 16 | add | |
| 1 | 16 | sub | |
| 1 | 8 | pop | dy |

**Figure 3: Pure Stack Machine Code Sample**

### 3.3.1. Quantitative Comparisons

The original motivation for the design of the pure stack machine was to completely separate data manipulation from data movement. This was to locate instruction pairs that might profitably be combined into single instructions. Analysis of the code sample (see Figure 3) shows two such pairs. First, as expected, there are a very large number of push and pop instructions. Second, nearly every push instruction is followed immediately by a pattern match operation. In addition to these, the pattern

| Size | Cycles | Instruction | | |
|---|---|---|---|---|
| 4 | 16 | push | nw | (8) in_image |
| 4 | 16 | push | n | (8) in_image |
| 2 | 8 | lshft | 1 | |
| 1 | 16 | adds | | |
| 4 | 16 | add | ne | (8) in_image |
| 4 | 16 | push | sw | (8) in_image |
| 4 | 16 | push | s | (8) in_image |
| 2 | 8 | lshft | 1 | |
| 1 | 16 | adds | | |
| 4 | 16 | add | se | (8) in_image |
| 1 | 16 | subs | | |
| 4 | 8 | pop | dy | |

**Figure 4: Enhanced Stack Machine Code**

| Size | Cycles | Instruction | | | | |
|---|---|---|---|---|---|---|
| 6 | 16 | mov | nw | (8) | in_image | Stack |
| 4 | 1 | clr | | (1) | Stack+8 | |
| 6 | 14 | mov | n | (7) | in_image | Stack+9 |
| 6 | 16 | add | c | (8) | Stack+8 | Stack |
| 6 | 16 | add | ne | (8) | in_image | Stack |
| 6 | 16 | add | sw | (8) | in_image | Stack |
| 4 | 1 | clr | | (1) | Stack+8 | |
| 6 | 14 | mov | s | (7) | in_image | Stack+9 |
| 6 | 16 | add | c | (8) | Stack+8 | Stack |
| 6 | 16 | add | se | (8) | in_image | Stack |
| 6 | 16 | sub | c | (8) | Stack+8 | Stack |
| 6 | 16 | mov | c | (8) | Stack | dy |

**Figure 5: Two Operand Code**

match operations are typically used to route data from a particular neighbor, which does not require the full flexibility of the pattern match instruction. Not apparent in this code sample is a fourth area for improvement: pattern matching algorithms (such as region filling) also show an excessive amount of communications between memory and the PE.

The largest difference between the first two instruction sets is the combination of push and routing instructions with arithmetic instructions. In the code sample, this provides nearly all of the performance improvement: folding the instructions together reduced the instruction count and reduced PE-memory traffic, while adding a lookup table for routing reduced the code size.

The performance measures for the second and third instruction sets are nearly identical. Typical algorithms show very similar numbers of instructions, and processing rates. The code size for the two operand machine is much larger, however. These performance criteria are shown together in Figure 6.

| Machine | Instructions | Size | Cycles |
|---|---|---|---|
| Stack Machine | 22 | 62 | 264 |
| Enhanced Stack Machine | 12 | 35 | 168 |
| Two Operand | 12 | 68 | 158 |

**Figure 6: Quantitative Performance Comparisons**

### 3.3.2. Complexity of Microcontroller

The first two instruction sets require the microcontroller to maintain the expression stack. Due to the bit-serial nature of the instruction set, two stacks are required. The Pyramid Stack, consisting of the actual operands, is stored in bit-addressed pyramid memory. The Controller Stack, holding the length of the operands in the operand stack, is stored in the controller. The two stack pointers (Controller Stack Pointer and Pyramid Stack Pointer) are also maintained in the controller.

The microcontroller for the second instruction set is more complex than for the first one, due to the necessity of maintaining routing pattern lookup tables. This difference is not nearly as significant as the difference in the size of the instruction set. The enhanced stack machine requires all of the instructions present in the pure stack instruction set, as well as its own enhancements.

The two operand instruction set uses a substantially simpler microcontroller. First, the stack maintenance hardware is not required. Second, as stack operations are simply special cases of two operand instructions, the size of the instruction set is effectively reduced to the size of the pure stack instruction set.

Extending this instruction set to a three operand instruction is also being considered. The approach here is to continue development of the two operand instruction set, while leaving space in the op code list for later implementation of the additional instructions.

## 4. The PCLIP II Processing Element

A microarchitecture has been developed for execution of the instruction set. This microarchitecture was developed with the single goal in mind of providing a maximally efficient hardware and microcode implementation of the pyramid instruction sets described above.

Since the three instruction sets developed perform nearly identical operations, the PE microarchitectures are also be identical. The only difference between them in terms of operations supported is the presence of a visible P register in the second and third. This register is also required for routing in the first instruction set, although it is not visible there. The major differences between the instruction sets lie in the addressing mechanisms used, and so do not affect the PE design.

### 4.1. Processing Elements

The PEs are required to support the operations defined by the instruction set described above. This translates into three specific requirements:

- Direct implementation of HCL pattern matching operations
- Support for C, E, and P registers
- Efficient bit-serial arithmetic

The resulting microarchitecture (Figure 7) is in many ways a conventional bit-serial PE, with optimizations in place for these requirements.

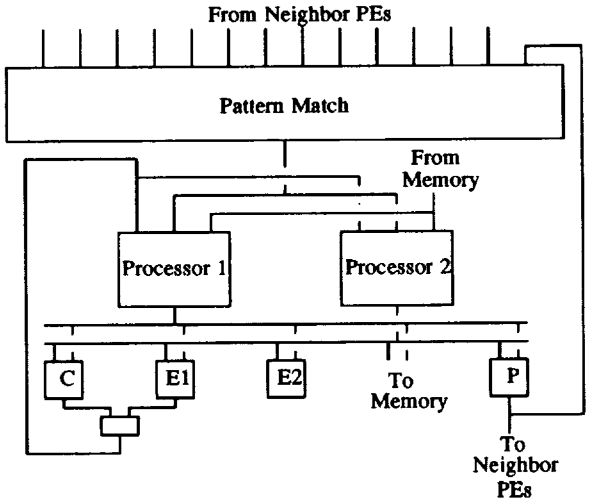


**Figure 7: Processing Element Microarchitecture**

Routing is provided by a pattern matching unit which directly implements the AND_Match operation of HCL. The matcher is provided with a vector of fourteen pattern elements, each containing a 0, 1, or X. Each 0 or 1 is matched against its corresponding element in the neighborhood of the

PE; an X pattern element is ignored. The output of the pattern matcher is a 1 if all of its non-X elements match. HCL's OR_Match can also be executed, by modifying the pattern contents and the functions to be performed by the boolean processors.

Two independent processors are each capable of evaluating any of the 256 boolean functions on three inputs. A typical use of the processors would be to generate sum and carry simultaneously in an addition. The outputs of the processors may also be independently routed to any of the registers in the PE or to memory.

This microarchitecture is able to perform simple two operand arithmetic instructions, such as addition or subtraction, at a rate of two cycles/bit, with a memory utilization of 75%.

## 5. Communications

Attention has been paid to two areas of communications. First, the instruction dispatcher must be capable of sending instructions to the pyramid microcontroller, and scalar results and status information must be returned. Second, it must be possible to transmit a massive amount of data into and out of the pyramid, and to effectively communicate this data with other subprocessors. In this section, we will briefly describe the mechanisms provided in PCLIP II for these classes of communications.

### 5.1. Instruction and Scalar Data Communication

Communication between the microcontroller and the instruction dispatcher is provided through two mechanisms. First, an instruction queue receives instructions from the dispatcher. The Instruction Register is double buffered, to permit fetching from the queue with no overhead. A second channel is used to communicate data back to the dispatcher. The returned data can be either the contents of a Non Zero Result register or a Scalar Result Return register. To synchronize the instruction dispatcher with the microcontroller and obtain this data, the instruction dispatcher sends a "send NZR" or "send scalar" instruction to the microcontroller, and then waits for the requested data to become available by attempting to read the communication channel.

The NZR register reflects the contents of the P registers in the pyramid. One NZR bit corresponds to each level of the pyramid. Whenever any enabled PE has a nonzero result, the NZR bit corresponding to that PE's level in the pyramid is set to one.

The SRR register operates under control of the microcontroller. The least significant bit of the SRR is loadable with the contents of the P register of the root of the pyramid. This provides a mechanism for loading scalar results of algorithms into the controller, and from there to the dispatcher and scalar processor.

### 5.2. High Bandwidth Input/Output and Mid-level Vision Support

One of the primary design goals of PCLIP II is developing an environment in which IO may take place efficiently, and at rates capable of supporting real-time vision processing. This includes two types of IO: raster IO, for transfer of image data in and out of the pyramid, and graph structure IO, for transferring edge maps in a compressed format suitable for use with graph structure processors.

The raster IO mechanism is very similar to that used by MPP, with simplifications permitted by the development of Video RAM memory. Interleaved VRAM is used as staging memory, with byte-serial data provided by a camera or other source. The VRAM shift registers are used to convert this data to row-serial data for PCLIP II. Shift registers built into the bottom level of the pyramid are used to transfer data between VRAM and the pyramid. These shift registers operate asynchronously with the pyramid, permitting simultaneous IO and processing.

The interfaces between the pyramid shift registers and the VRAM are also capable of searching the data passing through them for edge endpoints, and generating the addresses of these endpoints. These endpoints are then transferred to high-level, graph structured processors. In order to support vertical, horizontal, and diagonal edges, four IO shift planes are used in graph structure IO. Mechanisms also exist for the high level processors to transmit coordinates to line drawing hardware, which is able to translate them to a line drawing and transmit them to the pyramid base level. This mechanism is described in greater detail in Ref 4.

Maximum IO rates are two gigabits/second for raster IO, or up to 24 gigabytes/second for graph structure IO. These IO rates scale linearly with investment in IO hardware and cycle-stealing overhead in the processor; the initial implementation will only support 128 megabits/second for raster IO and three gigabytes/second for graph structure IO. These rates for graph structure IO are peak rates assuming unrealistically dense edge maps; more realistic numbers are perhaps ten percent of this.

This IO structure provides an interface to high-level processors which eliminates the need for searching through images to develop graph representations of edge maps. Instead, the graph structures can be developed directly, resulting in superior performance on the high-level vision tasks.

## 6. Conclusions

The design of PCLIP II has been guided from its very inception by a small number of basic principles. Most important of these has been a view of the processor as a component in the overall context of a vision computer, and a language-driven design.

This approach has been successful in producing a design with a very clean, well integrated, and efficient instruction set. For simple operations, the processor is able to operate within a factor of two of the bandwidth of memory. In addition, communication with other components of the vision processor, especially graph structure processors for cognitive processing, is greatly enhanced.

## References

[1] Gilmore, P. A., K. E. Batcher, M. H. Davis, R. W. Lott, and J. T. Burkley, "Massively Parallel Processor," GER-16684, Goodyear Aerospace, Akron, Ohio, July, 1979.

[2] Hartley, Roger T., Michael J. Coombs, and Eric Dietrich, "An Algorithm for Open-World Resaoning using Model Generation," in *Proceedings of the Second Rocky Mountain Conference on Artificial Intelligence*, pp. 193-203, Boulder, Colorado, 1987.

[3] Pfeiffer, Joseph J. Jr., "HCL: a language for low level image analysis," *Journal of Parallel and Distributed Computing*, (in press).

[4] Pfeiffer, Joseph J. Jr., "Integrating high level and low level computer vision," in *Proceedings of the 1985 IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, pp. 119-125, Miami Beach, FL, November, 1985.

[5] Pfeiffer, Joseph J. Jr. and Carol A. Soderlund, "Solid Modelling Data Structures for Computer Vision," in *Proceedings of the 22nd Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, California, November, 1988.

[6] Reddaway, S. F., "DAP - A distributed array processor," in *Proceedings of the First Annual Symposium on Computer Architecture*, pp. 61-65, Florida, 1973.

[7] Tanimoto, Steven L., "A hierarchical cellular logic," University of Washington Technical Report 83-10-06, October, 1983.

[8] Tanimoto, Steven L. and Joseph J. Pfeiffer, Jr., "Data Processing System Having a Pyramidal Array of Processors.," United States Patent Number 4,622,632, November 11, 1986.

# MICROCODE GENERATION FOR THE CONTROL OF A MASSIVELY PARALLEL COMPUTER

Tom Phillips, J. Bret Michael, and Zahi Abuhamdeh
School of Information Technology and Engineering
George Mason University
Fairfax, Virginia, 22030, U.S.A.

## ABSTRACT

A software organization concerning the control of an SIMD machine is explained. It is argued that the bulk of an algorithm adapted to an SIMD architecture should be executed on a microsequencing controller which is directly connected to the SIMD machine. Typical systems have a main control unit which calls low level primitives on a controller which is directly connected to an SIMD machine. An algorithm can be autonomously run on a microsequencer directly connected to an SIMD machine. Two algorithms are applied to the two major SIMD controller configurations and their performances are examined. To demonstrate how the software organization can be applied to a system, a compiler has been written, for the GAM Pyramid, that generates microcode from a high-level language with parallel data structures. The microcode executes on a microcode sequencer unit which is directly connected to a pyramid architecture SIMD machine.

Keywords: SIMD, Microcode Sequencer Unit, Compiler, Convolution Algorithm, MPP, GAM Pyramid.

## INTRODUCTION

There are two major philosophies governing the control of SIMD machines. The first is to divide control among two or more controllers which are connected in a hierarchical fashion. The lowest level controller (typically a microcode machine connected directly directly to the SIMD machine) executes primitive instructions such as addition or logical expressions and is not capable of supporting a large program; only small subroutines can be run.. Its parent controller executes higher level code and calls primitives residing on the low-level controller. There is usually some form of connecting hardware that eliminates controller wait time (such a queue in the case of the MPP [1]).

The second paradigm is to execute most if not all code on a low-level controller connected directly to the SIMD machine. A higher-level controller would be used to process external I/O and/or act as a gateway to other systems. This approach requires a sophisticated controller capable of executing different instruction types in parallel. One way to accomplish this is to have the low-level controller execute very versatile microcode.

The latter design is advocated. Examples of a possible implementation are given and an actual compiler for implementation are given and an actual compiler for the GAM pyramid's microsequencer is presented.

## NOTATION

The notations used to represent parts of a hypothetical microcode instruction are described below.

### Indirection

Indirection is handled by: [r], where r is the register used. Thus [r] represents the memory location pointed to by r.

### Looping

DO r1,r2 TIMES block is executed until r1 = 0
LOOP decrements r and loops back up if r1 > 0
The DO looping structure will automatically use r1 as a looping variable, first loading r1 with r2, then looping until r1 = 0.

### Program Flow Control

Program flow control consists of the following instructions:

BR k = branch to x on the next clock cycle
BRS = branch to subroutine on next cycle
CASE r1,r2 = branch to [r1+r2] on next cycle
DRT = execute two RT's on the next cycle
RT = return from subroutine (pop PC from stack)
CCASE r1,r2 = conditional case
CBR k = conditional branch to k
CBRS k = conditional branch to subroutine
CDRT = CRT,RT
CRT = conditional return from subroutine

### Arithmetic Control

Arithmetic control is based on a three-ALU configuration. See Figure 1. Formats are based on:

ADD source, source, destination

Any source may be represented by a register or a constant (k) that would appear in the microcode instruction.

LADD r1,r2,r3 = left add (may use indirection on any register.
RADD r1,r2,r3 = right add (may use indirection on any register)
CADD r = center add (may use indirection on r)

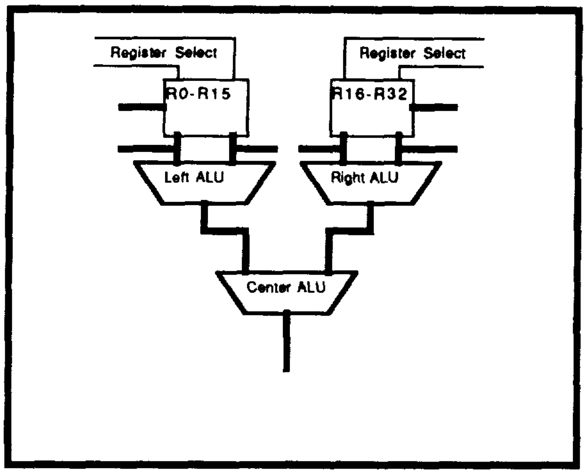


Figure 1. Three ALU Configuration.

### Condition Codes

The controller should have various condition lines such as ALU results (e.g., overflow, zero, etc.) or status lines (e.g., host ready, SIMD status, and so on).

CCSEL = condition code select (e.g., CCSEL 5 would select condition code 5 as the condition for conditional branches).

### Mutual Exclusion

Parallel Sets:
{BR,BRS,CBR,CBRS,RT,CRT,DRT,CDRT,CASE, CCASE}, {CCSEL}, {LADD}, {RADD}, {CADD}, {DO,LOOP}, {SIMD control instructions}

Elements from the same set are mutually exclusive and cannot be included in the same microcode instruction.

## SYSTEM COMPARISONS

Two routines written for the MPP (a 128 x 128 bit-serial array) are now presented to show the differences between using a hierarchical control system and a single microsequencer to execute large portions of code (note that the hypothetical microsequencer is assumed to have a sufficient amount of data memory and/or a large register set).

The first routine is that of convolution (see Program 1). It is important to note that every line of code involves manipulating the SIMD machine. Therefore, if it were to be executed by itself, it would make sense to contain it within one machine that controls the SIMD computer.

Program 2 shows how the convolution routine (residing in a hierarchical control system) would be called from a hypothetical FOR loop which executes one thousand times. Parameters are passed from the high-level controller to the low-level controller on each line within the convolution routine. That means that the actual convolution routine resides on the high-level controller while simple instructions such as ADD and SHIFT reside in the low-level controller.

In contrast, Program 3 demonstrates that no overhead at all is required to execute the convolution routine one thousand times. The DO instruction uses register seventeen to loop one thousand times. This looping structure takes no extra cycles. Thus, the entire routine may reside on a microsequencer, freeing the host and eliminating a need for complicated inter-communications hardware.

```
Program conv(row_index,col_index);
type
        sarray = Parallel Array[0..127,0..127] of     0..131071;

procedure conv33( a: sarray; var temp: sarray; k,s: integer);
var
        ashift : sarray;
begin
        temp := a * k;
        ashift := shift( a, 1, 0);
        temp := temp + ashift * s;
        ashift := shift(a, -1, 0);
        temp := temp + ashift * s;
        ashift := shift(a, 0, 1);
        temp := temp + ashift * s;
        ashift := shift(a, 0, -1);
        temp := temp + ashift * s;
        ashift := shift(a, 1, -1);
        temp := temp + ashift * s;
        ashift := shift(a, -1, 1);
        temp := temp + ashift * s;
        ashift := shift(a, 1, 1);
        temp := temp + ashift * s;
        ashift := shift(a, -1, -1);
        temp := temp + ashift * s;
end;
```

Program 1. Convolution Example.

```
a := 1;
k := 8;
s := -1;
section := 1;
for (i = 1 to 1000)
        conv33(a,b,k,s)
end.
```

Program 2. Convolution--Two Stage

```
a := 1;
k := 8;
s := -1;
section := 1;
load_register_18_with      1000
DO 17,18 TIMES         instruction
                       instruction1
                       instruction2
                          ...
LOOP                   instruction N
```

Program 3. Convolution--Single Stage Method.

The second example involves Program 4, the power factor routine [2]. There is considerably more code than convolution, yet it shares the same portability as convolution.

The first few lines (the nested ifs) can be seen as initialization code. A normal variable, stage, must be checked before any instructions can be given to the SIMD machine. However, once the initial tests are

```
Program Test(row_index,col_index);
type
          pli8 = Parallel Array[0..127,0..127] of 0..255;

          iarray = parallel array[0..127,0..127] of 0..65535;

procedure powergen(var power:pli8; var tn2, gamma,
               stage:integer; var id:pli8);

var       k, tk : pli8;
          twotoi, twotox : integer;
          i, x : integer;

begin
          if stage = 1 then power := 0
           else begin
             if stage = gamma then
                    where id >= tn2 do
                            power := power + 1
                    else
                    begin
                     x := gamma - stage;
                     twotox := 1;
                     for i := 1 to x do
                            twotox := twotox * 2;
                     k := id div twotox;
                     power := 0;
                     twotoi := 2;
                     twotoi := tn2 div twotoi;
                     i := 1;
                     while i < stage do begin
                            tk := k mod twotoi;
                            where (tk >= (twotoi div 2)) do
                              power := power + x;
                            twotoi := twotoi * 2;
                            i := i + 1;
                            x := x div 2;
                    end;
          end;
 end;
end;
```

Program 4. Power Factor Example.

complete, it is possible to start executing the bulk of the code (in the case that stage <>1 and stage <> gamma).

The first two lines simply do integer calculations. Then a FOR loop appears with a scalar multiplication inside. The FOR loop can be a DO loop, and the integer calculations are initialization code for any machine (although both could be done in one cycle on the proposed microsequencing control unit). Next there are more integer calculations followed by an array manipulation. A clever compiler (or observant programmer) could mesh these instructions into an optimum piece of code executing in minimum time. A hierarchical controller system would still need to pass parameters from the high-level controller to the low-level even if the sequential code and parallel code were executed concurrently (i.e., a communications overhead will always exist). Finally a WHILE statement appears that is basically a FOR loop in disguise. The microsequencer could simply put a DO loop with a CBR (testing if i is zero or negative) in the first microcoded instruction. Other code inside the WHILE can be optimized in a similar fashion to that already presented. Therefore, this entire routine could reside on a microsequencer connected directly to an SIMD machine.

## GAM PYRAMID COMPILER

In an effort to implement the microcode paradigm, we developed a compiler that generates microcode from a high-level language with parallel data structures for the GAM Pyramid. The microcode executes on a microcode sequencer unit based on AMD 2900 chips and is connected directly to a pyramid architecture SIMD machine (the GAM Pyramid [3]).

## CONCLUSIONS

Two algorithms were applied to the two major SIMD controller configurations and their program flow was examined. The results of this examination showed that the bulk of an algorithm adapted to an SIMD architecture can be executed on a microsequencing controller which is directly connected to the SIMD machine, thus eliminating the need for expensive and/or complicated inter-communications hardware which would normally be used. This means a microsequencer could execute large routines and even entire programs with little or no host interaction, leaving the host free to do external I/O, act as a user interface, or solve unrelated problems while the microsequencer solves parallel problems using the SIMD machine. A compiler that generates microcode from a high-level language with parallel data structures for the GAM Pyramid is being used to demonstrate the feasibility of this approach. The microcode executes on a microcode sequencer unit which is directly connected to a pyramid architecture SIMD machine. At this time, however, the GAM pyramid has received an additional level (it now has six levels) and the microsequencer is temporarily disconnected until the new interfaces are thoroughly tested.

## REFERENCES

[1] **Theory of MPP Hardware Operation.** Goodyear Aerospace Corporation, Akron, OH, 1983.

[2] Reeves, Anthony P., and Gutierrez, Maria. "On Measuring the Performance of a Massively Parallel Processor." **Proceedings of the 1988 International Conference on Parallel Processing**, 1988.

[3] Schaefer, D. H., Ho, P., Boyd, J., Vallejos, C., "The GAM Pyramid." **Parallel Computer Vision**, L. Uhr, Ed., Academic Press, Boston, 1987, pp. 15-42.

The authors would like to thank Ping Ho of Science Applications Research, Lanham, MD for writing and benchmarking Program 1, the convolution example.

# A MULTI-LAYERED G-NETWORK FOR MASSIVELY PARALLEL COMPUTATION

Teresa Haynes Rice
East Tennessee State University
Johnson City, TN 37614, USA

Ratan K. Guha
University of Central Florida
Orlando, FL 32816, USA

## ABSTRACT

A new network design, the Multi-layered G-network, is proposed. The Multi-layered G-network is obtained by interconnecting copies of the G-network in parallel. The design is suitable for large interconnection networks and has the following desirable characteristics: efficient routing, small number of links and simple connections. Moreover, the routing performance is not altered by a faulty link or node - an important consideration in any fault tolerant design.

Keywords: G-network, Fault-tolerance, Speedup, Degree, Cycle, Subgraph.

## INTRODUCTION

This paper presents the Multi-layered G-network obtained by interconnecting copies of the G-network (Ref. 1-2) in parallel. The design is suitable for large interconnection networks and has the following properties: efficient routing, small number of links, simple connections and a high level of fault tolerance.

## THE CONSTRUCTION

Each layer of this network is a copy of the G-network where the $\gamma$ special nodes in copy i are labeled $1_i, 2_i, \ldots, \gamma_i$. Here i, $0 \le i \le h-1$, is the layer number. Between each pair of special nodes, there are two nodes of degree two adjacent to both of the special nodes. Each special node $a_i$ is connected to $a_{(i+1)\bmod h}$. Observe that the nodes $a_0, a_1, \ldots, a_{h-1}$ induce a cycle subgraph.

One layer of the Multi-layered G-network with $\gamma = 4$ is shown in Figure 1, with connections to other layers indicated by dotted lines. Each layer has $\gamma^2$ nodes and $2\gamma^2-2\gamma$ links. An additional $\gamma h$ links are required to connect the layers, giving a total of $2h\gamma^2-h\gamma$ links. There are $h(\gamma^2-\gamma)$ degree two nodes and $h\gamma$ degree $2\gamma$ nodes.

## ROUTING AND FAULT-TOLERANT PROPERTIES

The maximum number of routing steps (hops) required between any two nodes in the same layer is four since each layer is a G-network (Ref. 1-2). This fact is independent of the number of processors in the network.

**Theorem 1**
**The maximum number of hops required between any two nodes in the Multi-layered G-network is** $\lfloor h/2 \rfloor+4$.
**Proof**
In the worst case situation a source node can reach a special node in the same layer in one hop. From that special node it takes at most $\lfloor h/2 \rfloor$ hops to reach the layer containing the destination node. It then takes at most three additional hops to reach the destination node. ■

We note that the routing performance dependent only on the number of layers h. Given a fixed number of layers, h, the maximum number of routing steps required is constant. The following theorems demonstrate the network's fault tolerance by showing that this characteristic remains unchanged when any single link or node fails.

**Theorem 2**
Any single link can fail in the Multi-layered G-network and the maximum number of routing steps required remains $\lfloor h/2 \rfloor+4$.
**Proof**
Every node can still reach a special node. If the faulty link has both its endnodes in the same layer, then an argument similar to the one in the proof to Theorem 1 yields the result. If the faulty link is between layers, then in the worst case situations routing between layers still

CH2649-2/89/0000/0519$01.00 © 1988 IEEE

requires $\lfloor h/2 \rfloor$ hops and the combination of routing steps on both the source and destination layers total at most four. ■

**Theorem 3**
If any single node in the Multi-layered G-network fails, the maximum number of routing steps required for any two active nodes to communicate remains $\lfloor h/2 \rfloor+4$.

**Proof**
If a degree two node fails, each active node is either a special node or still has two links to special nodes, and each pair of special nodes on the same layer is still joined by at least one path of length two. If a special nodes fails, each active degree two node can still reach a special node in one hop and each pair of active special nodes on the same layer is still joined by two paths of length two. Hence in either case routing between any two active nodes can be accomplished as before. ■

## CONCLUDING REMARKS

Using the appropriate choices for $\gamma$ and h and comparing the Multi-layered G-network to the popular interconnection networks: the Illiac Mesh, the Barrel Shifter and the Hypercube in terms of the number of links and the maximum number of routing steps between any pair of nodes, it is observed that the G-network has fewer links than each of the others while providing better routing. For example, when the number of nodes in the network is 4096 with $\gamma$ = 32 and h = 4, the Multi-layered G-network has 8064 links and requires at most six hops for any pair of nodes to communicate; whereas, the Illiac has 8192 links and requires at most 63 hops, the Barrel Shifter has 47104 links and requires at most 6 hops, and the Hypercube has 24576 links and requires at most 12 hops. Thus, in this example, the Multi-layered G-network has fewer links than each of the others and gives the speedups in routing performance over the Illiac, Barrel Shifter and Hypercube networks of 63/6, 6/6, and 12/6, respectively. (Note that the Multi-layered G-network requires at most four hops between any pair of nodes when h = 1.) In general, it can be shown that as the number of processors approaches infinity, the speedup of the G-network over each of the others and the difference in the number of links approaches infinity.

Moreover, we have shown that the Multi-layered G-network can withstand the occurrence of a faulty link or node and the maximum number of hops required between any pair of nodes remains unchanged. Recall that the maximum degree of $2\gamma$ is independent of the total number of processors. Hence the Multi-layered G-network possesses the following desirable properties: fault tolerance, low cost, fast routing, and simple connections with a maximum degree dependent only on the number of processors per layer.

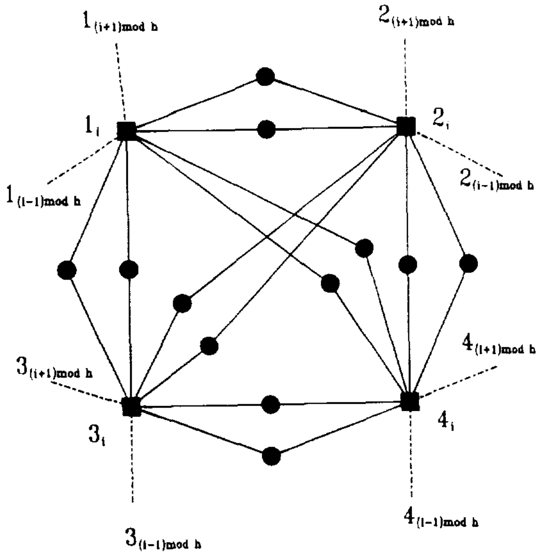

$1_{(i+1)\mathrm{mod}\ h}$
$2_{(i+1)\mathrm{mod}\ h}$
$1_i$
$2_i$
$1_{(i-1)\mathrm{mod}\ h}$
$2_{(i-1)\mathrm{mod}\ h}$
$3_{(i+1)\mathrm{mod}\ h}$
$4_{(i+1)\mathrm{mod}\ h}$
$3_i$
$4_i$
$3_{(i-1)\mathrm{mod}\ h}$
$4_{(i-1)\mathrm{mod}\ h}$

**Figure 1.** One layer of the Multi-layered G-network with $\gamma$ = 4.

## REFERENCES

1. Rice, T. H., R. K. Guha, R. C. Brigham, and R. D. Dutton, "The G-network and Its Inherent Fault Tolerant Properties", Technical Report CS-TR-88-10, Department of Computer Science, University of Central Florida.

2. Rice, T. H., "On k-$\gamma$-Insensitive Domination", Ph.D. dissertation, University of Central Florida, 1988.

# A SEQUENCED HYPERCUBE TOPOLOGY FOR A MASSIVELY-PARALLEL DATABASE COMPUTER

*Naphtali Rishe, Doron Tal, Qiang Li*

School of Computer Science
Florida International University–
The State University of Florida at Miami
University Park, Miami, Florida 33199

## Abstract

The architecture of a massively parallel multi-processor and multi-disk database computer is presented. The interprocessor communication network has a hypercube topology. The architecture requires selection of linear ordering of the nodes of a network of processors. A method is developed and presented here which can arrange the nodes in sequences efficient for management of data. Among the features of the produced sequences is that the size of the sequence can grow as the size of the hypercube grows, without changing the existing sub-sequences.

**Keywords:** Massively parallel architecture, Interprocessor network, Hypercube, Sequencing, Database computer.

## 1. Introduction

The massively parallel database machines offer a way, perhaps the only way, to meet the ever increasing demands of information processing. Most of the contemporary approaches seek to achieve this goal by increasing the size of the processor, memory and disk, and also by employing a large number of processors. The primary purpose of doing so is to increase the parallel processing power and the parallel secondary storage accessing power. However, in the overall structures of the typical systems, the processing units are one group, and the secondary storage units, possibly with some preprocessing power, are another. The processing unit group and the secondary storage group are connected by communication channels. The system's formation is still "Processors–I/O Channels–Secondary Storage". The traditional I/O channel bottleneck is still present. This problem remains no matter how many processors we add to the machine, because the throughput is dominated by the relatively poor performance of the disk and channel. One can see the analogy between the processor/disk split and the processor/memory split that leads to the von Neumann bottleneck.

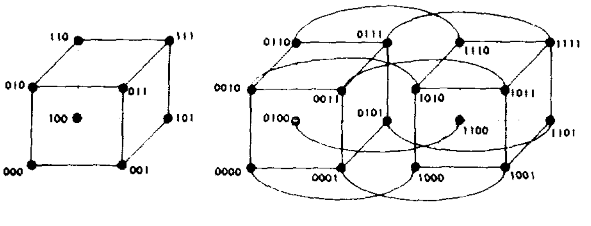
These shortcomings have motivated the proposal of a new architectural concept called Linear-throughput Semantic Database Machine (LSDM). We discuss here only those concepts of LSDM relevant to this paper, and more details can be found in [7]. LSDM consists of thousands processors coupled with disks. Each processor-disk unit consists of one or a few fairly powerful processors, a dedicated memory module and a small capacity disk, e.g., a 20 megabyte disk. The processor-disk units are linked into a tightly coupled network.

The processor-disk units comprising the machine work simultaneously on different segments of the same query and on concurrent queries as well, offering two levels of parallelism. The hosts receive streams of users' requests. Each request is dispatched to the processor having the best control of the relevant fragments of the database. The processor then decomposes the request into smaller operations and communicates them to subcontracting processors. The processors related to a request can communicate with each other to get the data necessary to carry out the operations concurrently. With the completion of a request, the results are sent to the host nodes and eventually back to the users.

Our database machine uses the Semantic Binary Database Model [5]. The Semantic Binary Database Model represents information of an application's world as a collection of elementary facts of two types: unary facts categorizing objects of the real world and binary facts establishing relationships of various kinds between pairs of objects. The purpose of the model is to provide a simple natural data-independent flexible and non-redundant specification of information.

In order to fully utilize the parallel processing power of the proposed architecture, it is essential to have an appropriate data structure so that the processor-disk units have balanced load. In our implementation of the Semantic Binary Database Model, we store the entire database, including all indexing information, in one logical file. (The file is organized in a B-tree like structure.) The file is partitioned into segments of the size that can be stored in small disks. There is no logical difference between one segment and another. This important property allows us to

This research has been supported in part by a grant from the Florida High Technology and Industry Council.



a A 3-Dimensional Hypercube

b A 4-Dimensional Hypercube

Figure 1: Hypercubes

partition the database based solely on the consideration of maximum parallelism. In addition, the data is stored with some small but intelligent duplications which greatly increase the locality of data accessing[6]. This implies a much lower data traffic between the processor-disk units, resulting in a higher degree of parallelism.

The n-cube topology has been chosen to connect the network. The n-cube [2,4,1,9] is defined as follows: Each of the $2^n$ nodes is labeled from 0 to $2^{n-1}$ by a unique binary string of length $n$. Two nodes are connected iff they differ in exactly one bit position. Figure 1 shows a 3-dimensional and a 4-dimensional hypercube.

## 2. The sequencing problem

### 2.1 Database machine sequencing problem

An essential problem is how to map a linear file onto the hypercube topology which is non-linear. Practical considerations, such as heavy data flow between consecutive file fragments, dictate that any two nodes of the network containing consecutive file fragments should be directly connected in the network. Therefore, we need a Hamiltonian path or loop (i.e. a path going through every node exactly once) in the hypercube so that the $i$-th partition of the file can be associated with the $i$-th node of the path. This is our *Requirement 1.* In addition, when more than one database is stored in the hypercube architecture in an overlapped fashion (i.e., each database is partitioned among all the nodes), the databases should be stored in non-coinciding paths to avoid unbalanced data traffic load. Therefore, many paths are needed.

There are many sequences that satisfy *Requirement 1.* At least one such sequence is guaranteed to exist because every hypercube has a Hamiltonian path since the hypercube satisfies the known criterion for Hamiltonian cycle: the graph does not contain the so-called theta-subgraph. As we shall show later, there are very many sequences which satisfy *Requirement 1.* We shall describe another requirement in term of sub-hypercubes.

Let $H$ be a hypercube of dimension $d$. Let $f$ be a part of a bit-string (binary number) of length $d$, i.e. for some positions in the bit-string it assigns bit values. For example, $f$ gives the second and the fifth bits of an 8-dimensional hypercube node as follows: **?0??1???**, where **?** stands for 0 or 1. The sub-hypercube defined by $f$ is the set of all nodes of $H$ having the pattern $f$. There are as many nodes in the sub-hypercube as there are the possibilities to fill the question marks. Notice, that the nodes can be relabeled so that the sub-hypercube would itself be a hypercube. Its dimension is the number of question marks in $f$. For example, the nodes

00000 00001 00100 00101 01000 01001 01100 01101

form a 3-subcube for $f$=**0??0?** in the 5-cube; for $f$=**00???** the sub-hypercube is:

00000 00001 00010 00011 00100 00101 00110 00111.

Now we shall define a hierarchy of sub-hypercubes. Consider, for example the hierarchy of sub-hypercubes of the 3-cube as defined by the tree of $f$-patterns in Figure 2.

In the above example, we first varied bit #2, then #1, then #3, and received a hierarchy of sub-hypercubes consisting of 1 cube, 2 square, 4 segments, and 8 points. There are as many such hierarchical families of sub-hypercubes as there are orders in which to vary the bits. A more formal definition follows.

Let $p$ be a permutation of position numbers from 1 to $d$. E.g., $p = (2,1,3)$ in the above example. The hierarchical family specified by $p$ is the set of the following sub-hypercubes of dimensions 1 to $d$. For every $D$, for every $f$-pattern assigning constant bit values to positions $p[1], p[2], ..., p[d-D]$, the sub-hypercube specified by $f$ is in the family. E.g., for the above example, the family consists of:

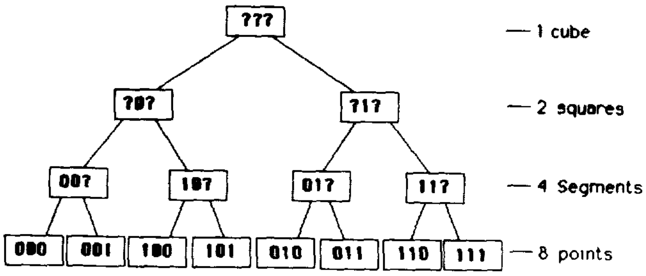


Figure 2: A Sub-hypercube Hierarchy

- the 2-dimensional sub-hypercubes specified by the $f$-patterns assigning constant bits to position $p[1] = 2$, i.e. to patterns **?0?** and **?1?**;
- the 1-dimensional sub-hypercubes specified by the $f$-patterns assigning constant bits to positions $p[1] = 2$ and $p[2] = 1$, i.e. to patterns **00?**, **10?**, **01?**, and **10?**;

- and so forth.

*Requirement 2* of the sequence is that when a hierarchical family of sub-hypercubes is given, every sub-hypercube of the family should comprise a contiguous sub-sequence. This requirement has several purposes. First, the hypercube can be expanded without changing the existing logical sequence. Second, since a sub-hypercube can be easily identified in the sequence as a consecutive sub-sequence, data backup, trouble shooting and module replacement are much easier. Third, because of the simple mapping between the sub-hypercubes and the sequence, the connections between the positions in the sequence are expected to follow a systematic pattern, which will facilitate the analysis and simulation of the system.

There can be many sequences, depending on which sub-hypercubes are required to be consecutive in the sequence. For example, for the hierarchical family with $p = (1,2,3)$, the sequence is:

000 001 011 010 110 111 101 100;

for the family with $p = (2,3,1)$, the sequence is:

000 001 101 100 110 111 011 010

In the first sequence, the nodes of the square 000 001 011 010 are consecutive, and in the second they are not. It is the opposite for the square 000 001 101 100. (A square is a 2-dimensional sub-hypercube.)

### 2.2 Generalization of the sequencing problem

The sequencing problem in a hypercube has more general applications. Many applications need to map sequential data structures into a hypercube. Some applications, involve sequential operations between nodes of a hypercube, e.g., scanning. Sometimes, several logical sequences are needed at the same time.

The following is a general definition of the requirements. Let $H$ be a $d$-dimensional hypercube. Let $p$ be a specification of a hierarchical family of sub-hypercubes. Let $L = 2^d - 1$. A sequence $N(0), N(1), ..., N(L)$ of all the nodes of $H$ is sought, satisfying:

1. For all $i$ in $0..L-1$, $N(i)$ and $N(i+1)$ are adjacent in $H$. Also, $N(L)$ is adjacent to $N(0)$.
2. For every sub-hypercube $S$ of dimension $D$ in the family specified by $p$, for some $i$, the sub-sequence $N(i), N(i+1), ..., N(i+2^{D-1})$ is the sub-hypercube $S$.

## 3. The sequencing method

A problem equivalent to a subset of our problem has been solved in the Control Theory. That solution is known as the Gray Code [3] for sequencing of binary numbers. If it is adapted to our hypercube problem, we would have *Requirement 1* satisfied, as well as a portion of *Requirement 2*. The Gray Code would be consecutive for only one hierarchical family of sub-hypercubes, while we need to be able to have an arbitrary family as a parameter to sequencing. We call the family which happens to be consecutive in the Gray Code, "The natural family". For 5-dimensional hypercube, the natural family is specified by $p = (1,2,3,4,5)$. As another example, the followings are two of the sequences generated for a 4-cube with $p = (1,2,3,4)$ and $p = (3,1,4,2)$:

- 0000 0001 0011 0010 0110 0111 0101 0100 1100 1101 1111 1110 1010 1011 1001 1000
- 0000 0100 0101 0001 1001 1101 1100 1000 1010 1110 1111 1011 0011 0111 0110 0010

In the above sequences, every 4 nodes form a 2-cube, and every 8 nodes form a 3-cube.

We have solved the general case of the problem. Our sequencing algorithm follows. The proof of its correctness is in [8].

A parameter to the algorithm is an array $p$ of position numbers which is a specification of a hierarchical family of sub-hypercubes.

We shall describe our algorithm in terms of a binary tree $T$. The tree $T$ has the following properties. It is a full binary tree; each node of $T$ has a label. The root has label "1". For any node, its label is greater by 1 than that of its parent. All the leaves have the label "$d$", where $d$ is the dimension of the hypercube. All the nodes at the same level have equal labels. Figure 3a shows an example of the tree $T$ for $d = 4$.

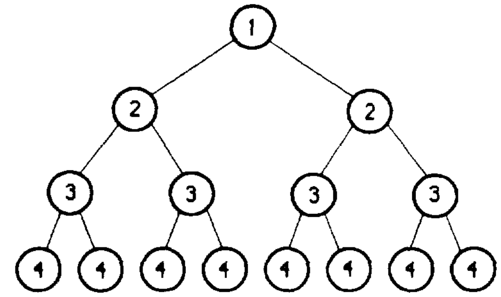


Figure 3a The Tree For a 4-Cube

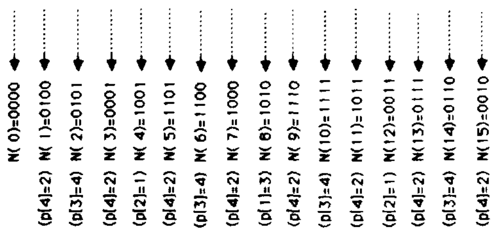


Figure 3b The Sequence Generated For p=(3,1,4,2)

We will use the $2^d - 1$ nodes of the tree, plus one additional node, to generate the $2^d$ nodes of the hypercube sequence. In the algorithm, a variable $N$ is used. The initial value of $N$ is the first binary number, $N(0)$, in the sequence to be generated. The rest of the binary numbers of the hypercube are generated while traversing $T$ in the inorder order. When a node of $T$ is visited, one bit of the contents of $N$ is flipped. If the label of the tree node is a number $i$, then the bit position to be flipped is $p[i]$. Every change in the value of $N$ generates the next binary number in the output sequence. The bits are numbered from left to right. For example, if $N = 00011, p = (1,3,2,4,5), i = 3$, then $p[i] = 2$; we flip bit #2 to receive the next $N = 01011$. Since the output sequence will be a cycle, the first binary number $N(0)$ can be an arbitrary node of the hypercube.

```
1. $N := 0$ (Let 0, i.e. a string of $d$ zero-bits, be
   the first number in the sequence, without loss of
   generality.)
2. repeat
      (a) Get the next node from $T$ according to
          the inorder traversal. Let $i$ be the label
          of the node.
      (b) $N := N$ XOR $2^{d-p[i]}$ (flip the $p[i]$-th bit
          of $N$ to get the next hypercube node
          number in the sequence).
   until all the nodes of the tree have been traversed.
```

*Example.* Using the tree of Figure 3a, the algorithm generates the sequence shown in Figure 3b for the sub-hypercube family defined by $p = (3,1,4,2)$.

Our program implementing the algorithm does not physically create a tree, but rather performs analytical calculation. The algorithm is linear in the number of nodes of the hypercube.

## 4. Discussion

The proposed system can lend itself to a stand alone database computer as well as a backend connected to hosts. Several processors can be identified as the hosts or interfaces to user hosts.

One of the primary goals of our architecture is to achieve a high throughput and approximate linearity of the throughput in the degree of parallelism. The degree of parallelism is the number of processor-disk units. The throughput is measured as the average number of transactions per time unit. The linearity is to be achieved for a typical transaction load comprised of a large number of relatively small, localizable queries and transactions. Under the current design (several thousands of processor-disk units), a conservative estimation of the throughput is more than 3000 simple queries per second per thousand processor-disk units, provided that the host interfaces have the same or higher throughput.

Currently, we are implementing our architecture on a network of 32-bit INMOS transputers and 20 megabyte Winchester disks.

## Acknowledgement

The authors gratefully acknowledge the advice of David Barton, Nagarajan Prabhakaran, Istvan Ereny.

## References

[1] S. Heller. *Directed Cube Network: A Practical Investigation.* Technical Report 253, MIT, 1985.

[2] K. Hwang and F. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, 1984.

[3] R.M. Klein. *Digital Computer Design*, pages 33–34. Prentice-Hall, Inc., Englewood Cliffs, NJ., 1977.

[4] F.P. Preparata and J. Vuillemin. The cube-connected cycles: a versatile network for parallel computation. *Communications of the ACM*, 24(5):300–309, May 1981.

[5] N. Rishe. *Database Design Fundamentals: A Structured Introduction to Database and a Structured Database Design Methodology.* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[6] N. Rishe. *A File Structure for Semantic Database.* Technical Report SCS TR 88-001, Florida International University, 1988.

[7] N. Rishe, D. Tal, and Q. Li. The architecture for a massively parallel database machine. *The Euromicro Journal*, Aug 1988. In press.

[8] Naphtali Rishe, Doron Tal, and Qiang Li. *A Sequenced Hypercube Topology for a Multi-disk Multi-processor Database Computer.* Technical Report 88-006, Florida International University, Miami, FL., 1988.

[9] L.G. Valiant and G.J. Brebner. Universal schemes for parallel communication. In *STOC, ACM Conference Proceedings*, Milwaukee, 1981.

# AN ARCHITECTURE FOR THE IMPLEMENTATION OF A PARALLEL MARKER PROPAGATION SYSTEM

Howard Schneider
Department of Psychiatry
Cité de la Santé de Laval Hospital
Laval, Quebec H7M 3L9 Canada

## ABSTRACT

An architecture is proposed which allows the construction of massively parallel (i.e., multi-billion node, multi-trillion linked) marker propagation systems. The nodes used in this architecture allow the best intersection of markers to be found by comparing the contents of the INTERSECTION REGISTER with the descending values on the CONTROL WORD BUS. The physical implementation consists of a silicon structure which is composed of 20,000 functional planes of silicon-based circuitry, interconnected by 5 billion vertical interconnecting wires. Any node can be linked directly with any other node.

## INTRODUCTION

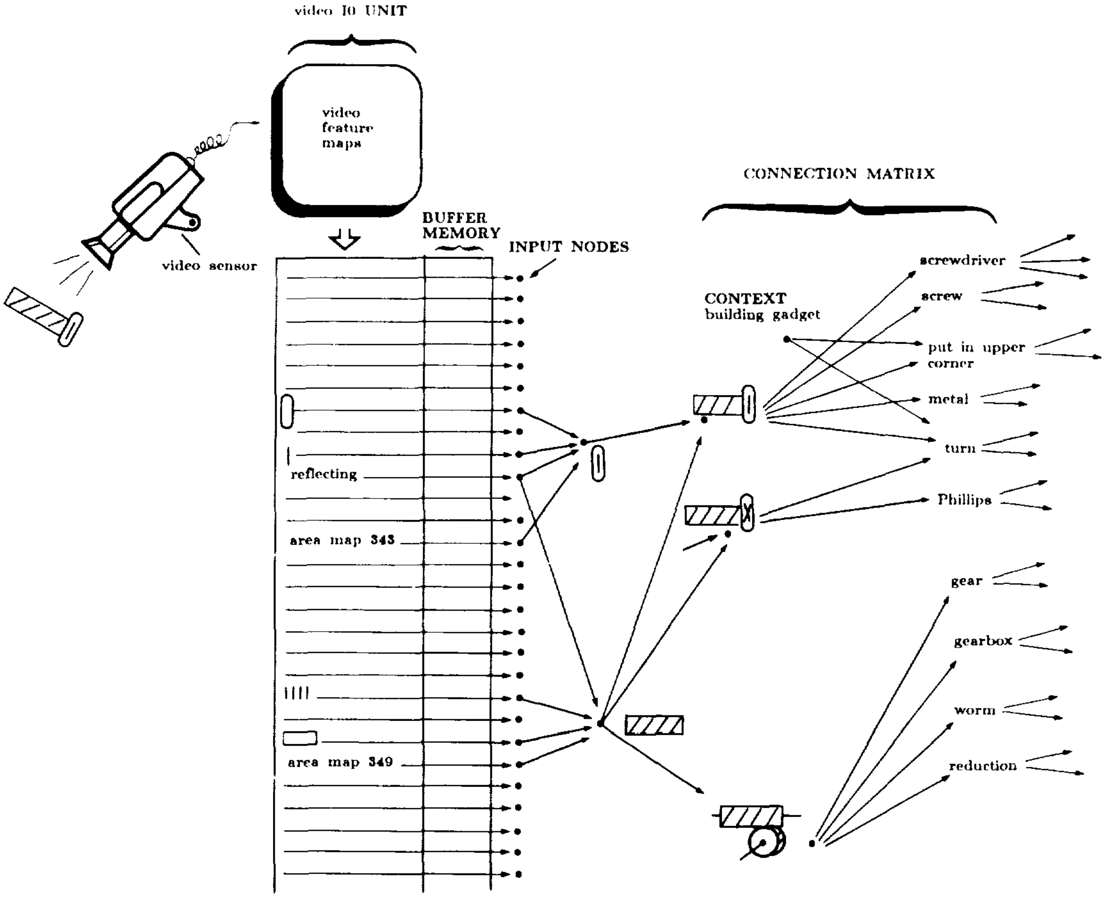
The parallel marker propagation system is an extremely elegant and powerful means of parallel computation. The parallel marker propagation system, also known as the semantic network, allows knowledge to be represented in such a way that a data base can be quickly searched for items with desired qualities regardless of the size of the data base. Through the process of spreading activation an intersection of markers quickly occurs at a node corresponding to the inputs presented to the system.

A portion of a parallel marker propagation system is shown in Figure 1. In this simplified example, the preprocessed outputs of an image detector are attached to the parallel marker propagation system. In this illustration the image detector is focused on a simple screw. The features of this image activate nodes in the parallel marker propagation system which correspond to these features. Markers are propagated from these nodes resulting in a better intersection at the node corresponding to the simple, single slot screw than at the nodes corresponding to the Phillips screw or the worm reduction gear. A node which reflects the context of the situation, in this case building a certain gadget, is also activated and markers are propagated from it. As a result even better intersections occur at the nodes corresponding to "put in upper corner" and "turn." Not only has the parallel marker propagation system allowed the rapid identification of the image viewed by the image detector, but it has quickly retrieved the appropriate information that the object in question should be put in the upper corner of the gadget being built and should be turned.

Since described by Quillian (1986), parallel marker propagation systems have been developed further by many researchers. Readily available references include Brachman (1979), Fahlman (1979), and Touretzky (1986). However, very little work has actually been done on ways of actually implementing large-scale parallel marker propagation systems. Hillis (1985) writes that "the Connection Machine architecture was originally developed to implement the marker-propagation programs for retrieving data from semantic networks (Fahlman 1979)." While the Connection Machine architecture does allow interconnection of processors (i.e., "nodes") to one another, it greatly sacrifices efficiency for flexibility. The computational power of the individual processors in the Connection Machine vastly exceeds the requirements of the nodes in almost any type of parallel marker propagation system. As well, one would optimally like to have direct connections between nodes instead of the indirect connections established through the intercommunications network of the Connection Machine.

CH2649-2/89/0000/0525$01.00 © 1988 IEEE

Figure 1. Parallel Marker Propagation System Recognizing a Visual Object

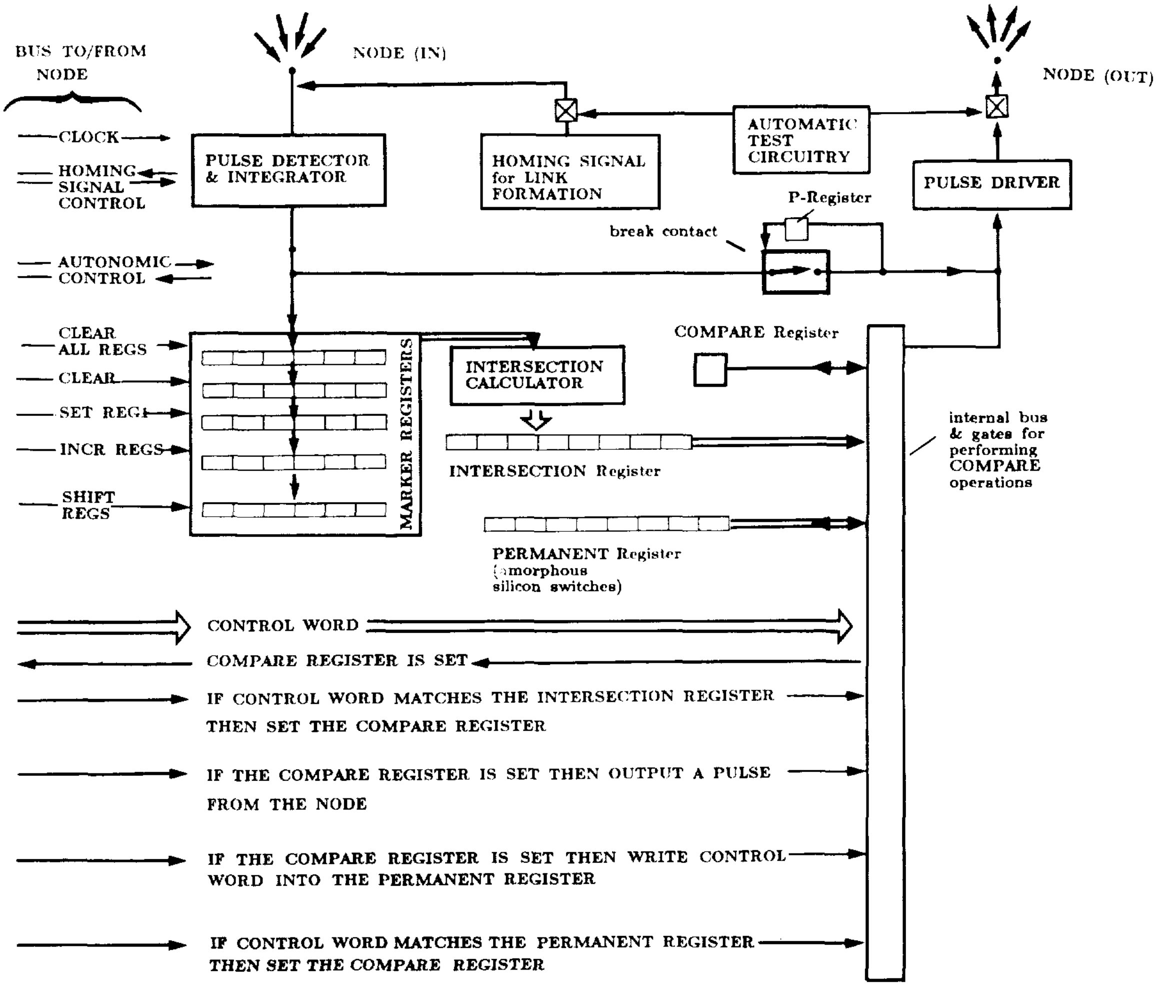


## ARCHITECTURE

This paper describes an architecture which implements a parallel marker propagation system with several *billion* nodes and several *trillion* connections between the nodes. The reason in describing such a futuristic architecture is not to say that this is the way a massively parallel marker propagation system will one day be implemented, but to point out that such a massively parallel system has the potential of actually being built. I have long been interested in massively parallel (i.e., multi-billion node) marker propagation systems as both a means of explaining certain brain functions and as the basis of a machine which will possess many of the skills humans have (Schneider–1987). As such, it is important to show that such massively parallel systems are indeed realizable.

The internal structure of the nodes used in the architecture being described is shown in Figure 2. The node corresponding to the best intersection of markers can be found by comparing the contents of the INTERSECTION REGISTER with descending values on the CONTROL WORD BUS while keeping the IF CONTROL WORD MATCHES THE INTERSECTION REGISTER THEN SET THE COMPARE REGISTER line high (i.e., true). In such a scheme the first node to set the COMPARE REGISTER represents the node with the highest value in the INTERSECTION REGISTER, i.e., the node corresponding to the best intersection of markers. If for example, the IF COMPARE REGISTER IS SET THEN OUTPUT A PULSE FROM THE NODE line is high, propagation of markers can then originate from this node.

Figure 2. Node Structure



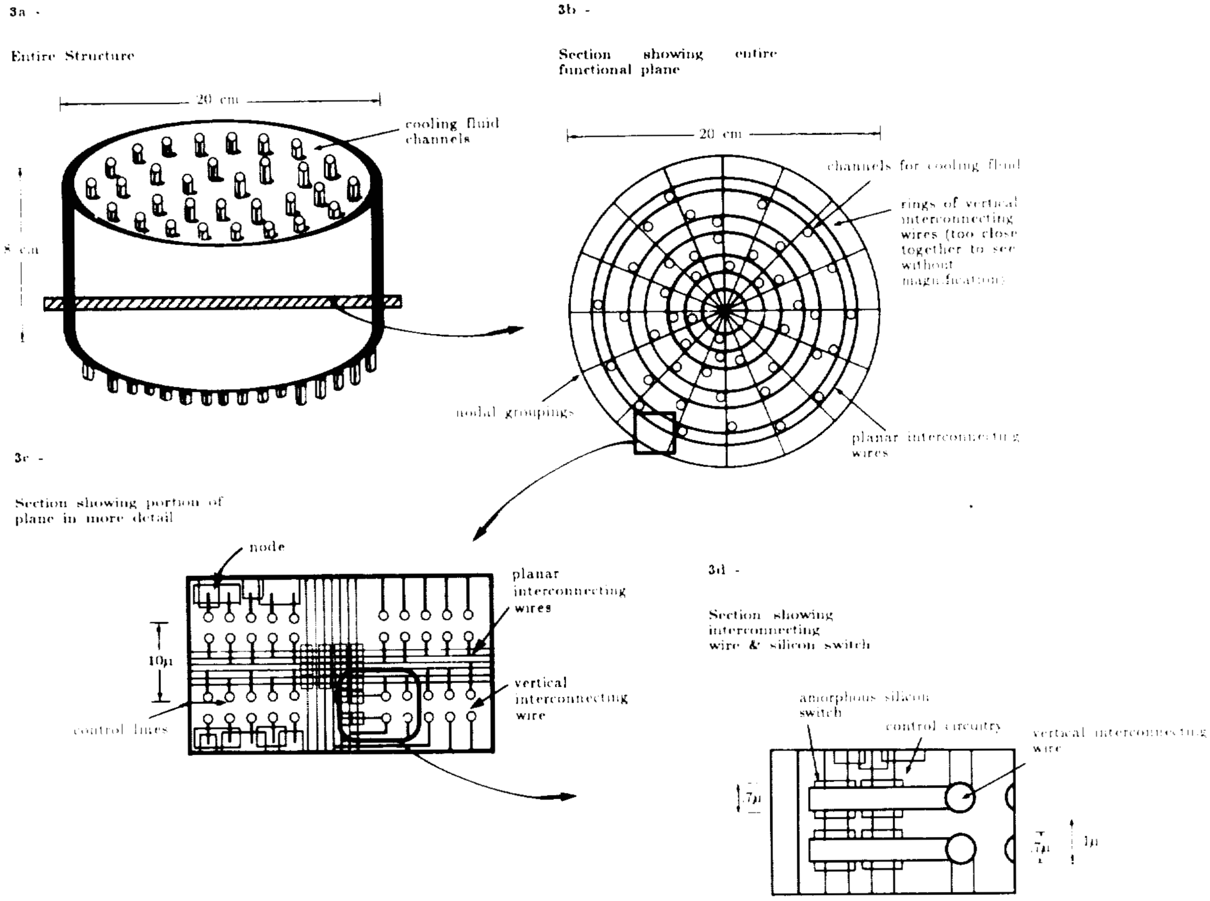
BUS TO/FROM NODE
NODE (IN)
NODE (OUT)
CLOCK
HOMING SIGNAL CONTROL
AUTONOMIC CONTROL
PULSE DETECTOR & INTEGRATOR
HOMING SIGNAL for LINK FORMATION
AUTOMATIC TEST CIRCUITRY
PULSE DRIVER
P-Register
break contact
CLEAR ALL REGS
CLEAR
SET REGS
INCR REGS
SHIFT REGS
MARKER REGISTERS
INTERSECTION CALCULATOR
COMPARE Register
INTERSECTION Register
PERMANENT Register (amorphous silicon switches)
internal bus & gates for performing COMPARE operations
CONTROL WORD
COMPARE REGISTER IS SET
IF CONTROL WORD MATCHES THE INTERSECTION REGISTER THEN SET THE COMPARE REGISTER
IF THE COMPARE REGISTER IS SET THEN OUTPUT A PULSE FROM THE NODE
IF THE COMPARE REGISTER IS SET THEN WRITE CONTROL WORD INTO THE PERMANENT REGISTER
IF CONTROL WORD MATCHES THE PERMANENT REGISTER THEN SET THE COMPARE REGISTER

## PHYSICAL IMPLEMENTATION

The physical implementation of the architecture is shown in Figure 3. The silicon structure shown in Figure 3a provides the several hundred meters squared of silicon-based circuitry required to implement a system with five billion nodes and five trillion links. The structure is composed of twenty thousand planes, the geometric term notwithstanding, of functional silicon-based circuitry. These planes are connected by five billion vertical interconnecting wires running the height to the structure thus allowing any node to be linked to any other node. Most of the surface area on each plane is used for wiring and for linking nodes. By applying a voltage somewhat higher than that used to propagate markers the insulating amorphous silicon layer between the intersection of two selected wires is nonreversibly transformed into conducting crystalline silicon thereby linking the two wires.

Although the technology to produce wafer-scale silicon circuitry connected in the third dimension is only in its infancy, there is much commercial incentive to believe that the technology to produce a working silicon structure as shown in Figure 3 will exist within the next decade. Metal-organic chemical vapor deposition and molecular beam epitaxy are two currently available technologies which in conjunction with other integrated circuit fabrication technologies allow one to build up layers of functional planes. A structure could be built up whereby planes of silicon circuitry are separated by insulating planes. By etching down to the layer below or by preventing deposition of the insulating plane in certain predetermined regions of each plane and depositing a metal in this region, vertical interconnection wires could link the entire structure. It is interesting to note that by using a molecular beam epitaxy process, integrated circuits with over two thousand layers have already been successfully constructed by Bell Laboratories (1986).

Interspersed among the planes of functional silicon circuitry are designated autonomic planes. The silicon structure is a heterogeneous substance composed of materials with different temperature coefficients. Not only is it necessary to remove the excess heat produced so that no part of the structure is damaged by an excessive temperature, but it is also necessary to remove the heat in a fashion such that the temperature gradients which may arise are tolerable ones. One function of the autonomic planes is to control such heat production and removal. These planes also serve to monitor the control bus going to every node and will disconnect a group of nodes from the bus should a problem be detected. This prevents, for example, a defective grounded bus line from rendering the entire system nonfunctional. The autonomic planes serve numerous other functions which allow successful operation of the silicon structure shown in Figure 3. For example, long lengths of vertical interconnecting wires less than a micron apart will induce false signals in each other. Thus, at every autonomic plane the positions of the wires relative to each other are switched.



3a -
Entire Structure
20 cm
cooling fluid channels
8 cm
3b -
Section showing entire functional plane
20 cm
channels for cooling fluid
rings of vertical interconnecting wires (too close together to see without magnification)
nodal groupings
planar interconnecting wires
3c -
Section showing portion of plane in more detail
node
planar interconnecting wires
10μ
vertical interconnecting wire
control lines
3d -
Section showing interconnecting wire & silicon switch
amorphous silicon switch
control circuitry
vertical interconnecting wire
.7μ
.7μ
1μ

Figure 3. Physical Implementation of the Architecture

In summary, an architecture is proposed here which allows the construction of massively parallel (i.e., multi-billion node, multi-trillion linked) marker propagation systems.

## REFERENCES

Bell Laboratories (1986) The World's Fastest Semiconductor Built From the Atoms Up, *Technology Review* 89(8):20–21.

Brachman, R.J. (1979) On the Epistemological Status of Semantic Networks. In Findler (ed.) *Associative Networks,* New York: Academic Press, pp. 3–50.

Fahlman, S.E. (1979) *NETL: A System for Representing and Using Real-World Knowledge,* Cambridge: MIT Press.

Hillis, W.D. (1985) *The Connection Machine,* Cambridge: MIT Press.

Quillian, M.R. (1968) Semantic Memory. In Minsky (ed.) *Semantic Information Processing,* Cambridge: MIT Press.

Schneider, H. (1987) Simulation of Parallel Marker Propagation Systems, *Proceedings of the 1987 Summer Computer Simulation Conference,* pp. 674–680, San Diego, CA: Society for Computer Simulation.

Touretzky, D. (1986) *The Mathematics of Inheritance Systems,* Los Altos, CA: Morgan Kaufmann.

# LN-TREE: A FAULT-TOLERANT TREE ARCHITECTURE

K. Y. Srinivasan and A. K. Sood

Department Of Computer Science and Engineering, University Of Toledo
Department Of Computer Science, George Mason University

## ABSTRACT

Tree structures have been widely used in the design of distributed systems. One distinct advantage of the tree architecture is the O(logN) speed of information exchange between any two nodes of an N node system. Further, the tree architectures can naturally map several important class of problems that can be described as divide-and-conquer algorithms. In this paper we present the analysis and design of a highly reliable tree structure; the LN-tree. The LN-tree is formed by augmenting the simplex binary tree with redundant nodes and links. The reliability of the proposed structure is evaluated and compared with previously proposed augmented tree architectures. The results of comparison show that the LN-tree is more reliable than the existing fault-tolerant tree structures.

Keywords: Fault-Tolerance, Reliability, Hierarchical Architecture.

## I. INTRODUCTION

A tree by definition is a minimally connected graph. Although the tree can naturally map several important class of problems, it is susceptible to single node or link failures. For instance, the failure of a single node or a link may invalidate the operation of the whole tree. Consequently, several researchers have addressed the problem of providing redundancy to the basic tree in terms of extra links and nodes with a view to increasing the fault-tolerance of the basic tree. Two basic approaches have been proposed in the literature. The two approaches differ in terms of what constitutes an acceptable level of performance in the event of faults. Some of the pioneering work in the design of fault-tolerant tree structures has been done by Hayes [Hay 76]. The following definitions allow us to formally describe the two approaches presented in the literature for the design of fault-tolerant tree structures. Without loss of generality, we restrict faults to node failures.

**Definition I:**

Let S be the system graph. A k-fault F in S is the removal of k nodes { $x_1, x_2, \ldots, x_k$ } from S. All links connected to these nodes are also removed. The resulting system graph under faults will be denoted by $S^F$.

**Definition II:**

A simplex system $S^0$ is a non-redundant system that can not tolerate any faults.

In the first approach [Des 78], [Hor 81], [Grey84], it is assumed that S is k-fault-tolerant with respect to $S^0$ if, for every k-fault in S, the graph $S^F$ is connected. Under the above definition of fault- tolerance, it is possible to design fault-tolerant tree structures solely through the addition of redundant links to the simplex tree structure. This approach permits performance degradation in the event of faults. The system is considered open or repairable. Hence, performance degradation is allowed during periods of outages (node failures). Under the above assumptions, a temporary degradation in performance is considered acceptable as long as the reliability of the complete tree remains high.

In the second approach proposed by Hayes [Hay 76], a more stringent constraint is placed on $S^F$ for S to be considered fault-tolerant with respect to $S^0$. In this approach it is assumed that S is k-fault- tolerant with respect to $S^0$ if, for every k-fault in S, the graph $S^F$ has a subgraph isomorphic to $S^0$. Here the objective is to preserve the original tree structure fully, in spite of faults: by reconfiguring the tree with the standby nodes and links. At least theoretically this ensures that the overall system performance suffers no degradation in the event of faults. The chief concern of this design approach is augmenting the basic tree structure for fault-tolerance while keeping the number of redundant nodes and links to a minimum for the required level of fault-tolerance. Hayes [Hay 76] first posed the above problem and proposed an algorithm for generating optimal k-fault-tolerant structures for a class of symmetrical hierarchical trees. The work of Hayes was extended by Kwan et al, [Kwan 81]. The algorithm proposed by Kwan et al, is not only optimal with respect to the number of redundant nodes and links, but also with respect to the fan-in of the nodes.

Raghavendra et al, [Rag 83] have proposed two augmented tree structures neither of which is optimal in the sense defined by Hayes. Both schemes can tolerate multiple faults, as long as the faults occur in different parts of the tree. In the first scheme proposed, there is a spare node for each level of the tree. This structure can tolerate up to m node failures (for a tree of level m) as long as each level has no more than one node failure. We shall refer to this tree as the RAE-tree. The second scheme proposed by raghavendra et al, [Rag 83] is an extension of the first scheme. In this scheme a spare node is provided for every $2^i$ nodes for some value of i. There is a variety of schemes possible depending upon the value of i chosen (the value of i may be different for each level of the tree). Hassan et al, [Has 86] have proposed a modular fault-tolerant binary tree structure. The approach uses modular fault-tolerant building blocks to construct the complete binary tree. Each module is a one-fault-tolerant building block which constitutes a three node two level subtree for the complete binary tree. In this paper we will refer to this tree as the module-tree.

In this paper we present the design and analysis of a new fault-tolerant tree structure; the LN-tree. The LN-tree is formed by augmenting the simplex binary tree with redundant links and nodes. The LN-tree maintains a rigid tree structure in the event of faults. In the following section we present the architecture of the LN-tree. In section 3 we present the reliability analysis for the proposed architecture. Finally, we present conclusions in section 4.

CH2649-2/89/0000/0531$01.00 © 1988 IEEE

## 2. DESCRIPTION OF THE TOPOLOGY

In this section we propose a scheme for augmenting the simplex tree with redundant nodes and links. We will refer to the proposed architecture as the LN-tree (tree with redundant links and nodes). Consider a simplex binary tree of m levels, m>1. Observe that each level i (i>1) of the simplex tree can be partitioned into $2^{i-2}$ partitions each having two nodes. In the proposed architecture each such pair of nodes is replaced by a fault-tolerant module. The structure of a 15 node LN-tree is shown in Fig. 1. Each fault tolerant module consists of three nodes and the required switching circuitry to effect reconfiguration in the event of faults. The structure of the proposed fault-tolerant module is presented in Fig. 2. Each fault-tolerant module has two input and four output links. All links are assumed to be undirected. At any given time only two of the three nodes of the module are active. The two input lines to the module are switched to the currently active nodes through a 2X3 cross bar switch, NW1. The six outputs of the three nodes are switched through a 6X4 cross bar switch (NW2) to the output of the module. Both NW1 and NW2 are controlled by one of the active nodes. The three nodes are also connected to a diagnostic bus. A virtual token ring is set up on the diagnostic bus to support distributed self diagnosis. At any given time the highest numbered active node in the module is the module controller (node having control over the switching networks NW1 and NW2).

In a hierarchical system such as the tree, it is reasonable to expect that the complexity of nodes in different levels of the tree will be different. Hence, we feel that the partitioning proposed in the design of LN-tree is natural. Each fault-tolerant module of the LN-tree can be implemented as a VLSI system made up of three homogeneous processing elements and the associated reconfiguration circuitry. Compare this to the partitioning used in the module-tree, where each fault-tolerant module is made up of nodes belonging to two adjacent levels of the tree along with the associated reconfiguration circuitry. If nodes of different levels are logically different, the partitioning proposed in the design of module-tree may lead to VLSI implementation problems. Another advantage of the proposed scheme is that the maximum degree of any node is 3 (considering only the data links connected to each node). This degree of connectivity in our scheme is much lower than the module-tree whose maximum degree is 5 (the maximum degree in the RAE-tree is 7).

The redundant nodes in the LN-tree are not active unless there is a fault. The proposed structure can tolerate multiple node failures as long as the failed nodes belong to different parts of the tree. Further, the number of node failures a level can tolerate depends on the level number. For example, levels one and two of the LN-tree can each tolerate a node failure. Level i of the LN-tree, $2<i\leq m$, can tolerate at most $2^{i-2}$ node failures (each module in level i can tolerate a fault). As all the redundant nodes are used in standby mode, the performance of the LN-tree is same as that of the simplex binary tree.

## 3. RELIABILITY ANALYSIS

In this section we evaluate the reliability of the LN-tree. We perform reliability analysis under the following assumptions:

1. The reliability of links is large compared to the reliability of nodes. Further, it is assumed that the reliability of a fault-tolerant module is a function of node reliabilities only.
2. The structure under faults is considered functional if the functional nodes can be reconfigured to maintain a rigid tree structure of specified depth.
3. The failure process (of nodes) is a Poisson process.

Consider a LN-tree of depth m. Let $R_i$ be the reliability of level i of the LN-tree. The system reliability of the LN-tree can be expressed in terms $R_i$ and is given by the following expression:

$$R_{SYS} = \prod_{i=1}^{m} R_i$$

Since level one can tolerate a single node failure, the reliability of level one, $R_1$ is given by:

$$R_1 = R^2 + 2R(1-R)$$

where R is the reliability of a single node. If λ is the failure rate of a node, then $R=e^{-\lambda t}$. Consider level i, $1<i\leq m$, of the LN-tree. Level i of the LN-tree is made up of $2^{i-2}$ fault-tolerant modules. If $R_{MODULE}$ is the reliability of a fault-tolerant module, then, the reliability of level i of the LN-tree is given by

$$R_i = (R_{MODULE})^{2^{i-2}}$$

as each of the $2^{i-2}$ fault-tolerant modules of level i (i>1) must survive for level i to survive. Each fault-tolerant module has three nodes and can tolerate a single node failure. Hence, the reliability of a fault-tolerant module, $R_{MODULE}$ is given by the following expression:

$$R_{MODULE} = R^3+3R^2(1-R)$$

Since each level of the system has to survive for the entire system to survive, we have

$$R_{SYS} = \begin{cases} R^2+2R(1-R) & \text{for } m=1 \\ (R^2+2R(1-R)) \prod_{i=2}^{m} R_i & \text{for } m>1 \end{cases}$$

In the preceding analysis we have assumed perfect coverage. If the probability of successful recovery after the occurrence of a fault is C, the system reliability under imperfect coverage conditions is given by

$$R^C_{SYS} = \begin{cases} R^2+2RC(1-R) & \text{for } m=1 \\ (R^2+2RC(1-R)) \prod_{i=2}^{m} R^C_i & \text{for } m>1 \end{cases}$$

where $R^C_{SYS}$ is the system reliability under imperfect coverage conditions and $R^C_i$ is the reliability of level i, i>1 of the LN-tree under imperfect coverage conditions. $R^C_i$ is given by

$$R^C_i = R^C_{MODULE}{}^{2^{i-2}}$$

where $R^C_{MODULE}$ is the reliability of a fault-tolerant module under imperfect coverage conditions and is given by

$$R^C_{MODULE} = R^3+3R^2C(1-R)$$

We consider a four level LN-tree for reliability computations. Under the assumptions of perfect coverage and uniform node reliability (failure rates of all the nodes assumed to be identical), we study the effect of individual node reliability on the system reliability. The reliability plot of the LN-tree is given in Fig. 3. For purposes of comparison we have also included the pertinent data for the simplex tree, module-tree and the RAE-tree. In Table 1 we present the contribution of the reliability of each level tó the system reliability for the LN-tree. It is noted that the reliability of the LN-tree is higher than that of the module-tree and the RAE-tree.

In order to study the effect of coverage, we evaluate the system reliability under imperfect coverage conditions. The reliability plot under imperfect coverage conditions is given in Fig. 4. In Table 2 we present the reliability of each level of the LN-tree under imperfect coverage conditions. It is noted that the reliability of the LN-tree is higher than that of the module-tree and the RAE-tree even under imperfect coverage conditions. In Table 3, we present the variation of system reliability with respect to the coverage factor. It is noted that the reliability of the LN-tree is not very sensitive to the variations in the coverage factor for the values of C considered. A 10 percent drop in the coverage factor results in a system reliability deterioration of about 4.2 percent.

In a hierarchical system such as the tree, it is reasonable to expect the nodes of different levels to have different failure rates. For example, one would expect the root node to have a low failure rate, as all transactions to the tree is through the root node. Again, as the leaf nodes perform very basic operations in a typical tree architecture, it is reasonable to expect a low failure rate for the leaf nodes (reliability of a node is a function of the complexity of the node). We study the reliability of a non-homogeneous tree under the assumption that each level of the tree is made up of homogeneous nodes. The results are presented in Table 4. The values of failure rate and coverage chosen for each level is presented in Table 5 (the failure rates presented is for a node belonging to the particular level). It is observed that in order to achieve high reliability for LN- trees, it is necessary to balance the reliabilities in each level of the tree. Specifically, the reliability of nodes in the highest level of the tree will have to be higher than the other nodes in the tree since the system reliability is limited by the reliability of the highest level of the tree.

## 4. CONCLUSIONS

The binary tree has long been recognized as a natural interconnection structure for describing several hierarchical computations. We have presented a new fault- tolerant binary tree architecture: the LN-tree. The LN-tree maintains a rigid tree structure even under node failures. In the design of the LN-tree we have used Hayes definition of fault-tolerance. Although the proposed structure is not optimal with respect to the constraints defined by Hayes, the LN-tree of m levels can tolerate multiple node failures as long as the failed nodes are in different parts of the tree. The LN-tree has been designed using fault-tolerant building blocks. As fault detection and reconfiguration is local to each fault-tolerant module, simple fault detection and reconfiguration is possible. The reliability of the LN-tree was evaluated and compared with other comparable architectures proposed in the literature.

## REFERENCES

[Avi71] A. Avizienis, "Fault-Tolerant Computing-An overview," IEEE Computer, Vol.4, pp. 5-8, Jan. 1971.

[Avi82] A. Avizienis, "The Four-Universe Information System Model For The Study Of Fault-Tolerance," FTCS-12, pp. 6-13, 1982.

[Ber73] C. Berge, Graphs and Hypergraphs. North Holland, 1973.

[Des78] Despain, A et al.,"X-Tree: A Tree Structured Multiprocessor Computer Architecture," Proc. 5th Annu. Symp. Comput. Archit., pp. 144-151, Apr. 1978.

[Grey84] Grey et al., "A Fault-Tolerant Architecture for Network Storage Systems," FTCS-14, pp. 232-239, 1984.

[Has85] A. S. M. Hassan et al., "A Modular Approach To Fault-Tolerant Binary Tree Architectures," FTCS-15, pp. 344-349, 1985.

[Has86] A. S. M. Hassan et al., "A Fault-Tolerant Modular Architecture For Binary Trees," IEEE Trans. on Comp., pp. 356-361, April 1986.

[Hay76] J. P. Hayes, "A Graph Model For Fault-Tolerant Computing Systems," IEEE Trans. on Comp., 25(9), pp.875-884, June 1976.

[Hor81] Horowitz et al., "The Binary Trees as an Interconnection Network: Applications Multiprocessor Systems, and VLSI," IEEE Trans. Comput., vol. C-30, no 4, pp. 247-253, April 81.

[Kwa81] C. L. Kwan et al., "Optimal Fault-Tolerant Realization Of Some Classes Of Hierarchical Tree Systems," FTCS-11, pp. 176- 178. June 1981.

[RAG82] C. S. Raghavendra, "FAULT TOLERANCE IN COMPUTER COMMUNICATION ARCHITECTURES," Ph.D Dissertation, Dept. of Computer Science, University of Califórnia, Los Angeles, 1982.

[Rag83] C. S. Raghavendra et al., "Fault-Tolerance In Binary Tree Architectures," FTCS 13, pp. 360-364, 1983.

[Rag85] C. S. Raghavendra et al., "Reliable Loop Topologies for Large Local Computer Networks," IEEE Trans. on Comp., pp. 46-55, Jan. 1985.

[Tri84] K. S. Trivedi, "Reliability Evaluation for Fault-Tolerant Systems," in Mathematical Computer Performance and Reliability, North Holland, pp. 403-414, 1984.

Table 1. Reliability of a four level LN-tree.

$\lambda = 0.1$   $C = 1.0$

| t | $R_{SYS}$ | $R_1$ | $R_2$ | $R_3$ | $R_4$ |
|---|---|---|---|---|---|
| 0.2 | 0.991 | 0.999 | 0.998 | 0.997 | 0.995 |
| 0.4 | 0.967 | 0.998 | 0.995 | 0.991 | 0.982 |
| 0.6 | 0.930 | 0.996 | 0.990 | 0.980 | 0.961 |
| 0.8 | 0.882 | 0.994 | 0.983 | 0.966 | 0.934 |
| 1.0 | 0.827 | 0.990 | 0.974 | 0.949 | 0.902 |

Table 2. Reliability of a four level LN-tree under imperfect coverage.

$\lambda = 0.1$   $C = 0.98$

| t | $R_{SYS}$ | $R_1$ | $R_2$ | $R_3$ | $R_4$ |
|---|---|---|---|---|---|
| 0.2 | 0.982 | 0.998 | 0.997 | 0.995 | 0.990 |
| 0.4 | 0.951 | 0.996 | 0.993 | 0.986 | 0.973 |
| 0.6 | 0.908 | 0.994 | 0.987 | 0.974 | 0.949 |
| 0.8 | 0.855 | 0.991 | 0.979 | 0.958 | 0.919 |
| 1.0 | 0.797 | 0.987 | 0.969 | 0.940 | 0.884 |

Table 3. Variation of system reliability with respect to coverage.

$\lambda = 0.1$   $t = 0.2$

| C | $R_{SYS}$ |
|---|---|
| 1.00 | 0.991 |
| 0.98 | 0.982 |
| 0.96 | 0.974 |
| 0.94 | 0.965 |
| 0.90 | 0.948 |

Table 4. Reliability of a non-homogeneous LN-tree.

| t | $R_{SYS}$ | $R_1$ | $R_2$ | $R_3$ | $R_4$ |
|---|---|---|---|---|---|
| 0.2 | 0.999 | 1.0 | 0.999 | 0.999 | 0.999 |
| 0.4 | 0.999 | 1.0 | 0.999 | 0.999 | 0.999 |
| 0.6 | 0.998 | 1.0 | 0.999 | 0.999 | 0.999 |
| 0.8 | 0.998 | 1.0 | 0.999 | 0.998 | 0.999 |
| 1.0 | 0.997 | 1.0 | 0.999 | 0.998 | 0.999 |

Table 5. Fault coverage and failure rate for each level.

| Level | Coverage | Failure rate |
|---|---|---|
| 1 | 1.00 | 0.0001 |
| 2 | 1.00 | 0.0100 |
| 3 | 0.98 | 0.0100 |
| 4 | 0.95 | 0.0001 |

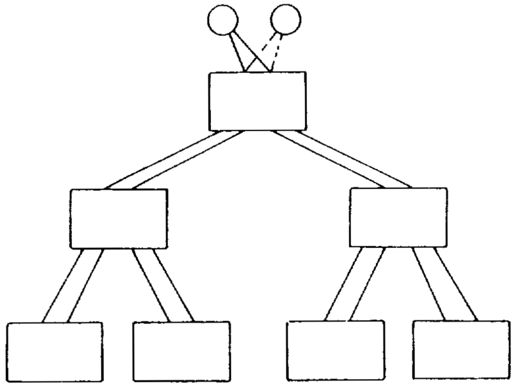


**Figure 1. A four level LN-tree.**

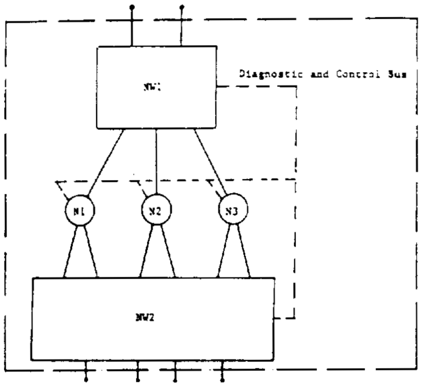


**Figure 2. Fault-tolerant module used in an LN-tree.**

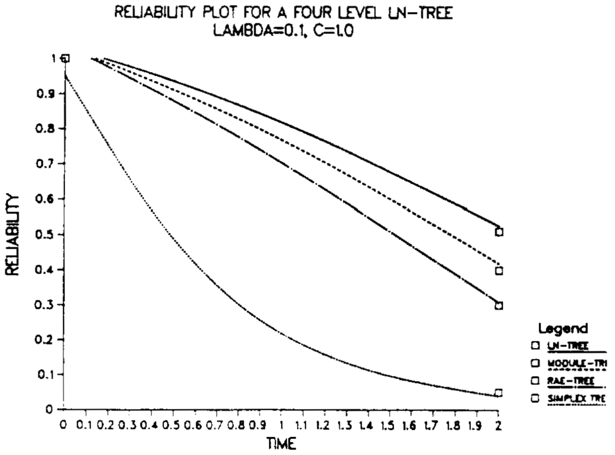


Figure 3. Reliability plot of an LN-tree under perfect coverage.

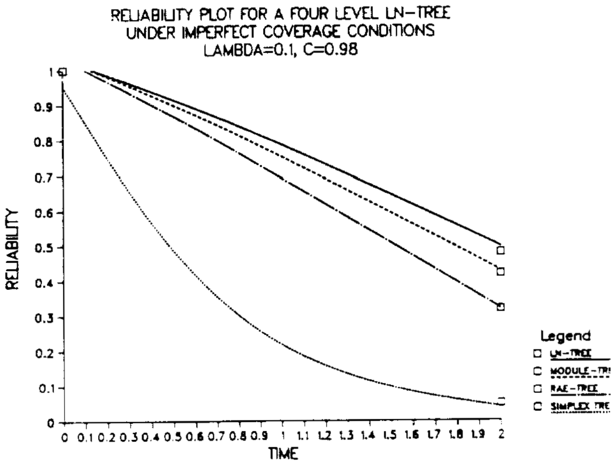


Figure 4. Reliability plot of an LN-tree under imperfect coverage.

# A SLIDING MEMORY PLANE ARRAY PROCESSOR*

M.H. Sunwoo and J.K. Aggarwal

Computer and Vision Research Center
The University of Texas at Austin
Austin, TX 78712

## Abstract

In this paper, we describe a new mesh-connected array processor for low level vision tasks. Several disadvantages of existing mesh-connected array processors, such as (*i*) communication overhead between processing elements (PEs), (*ii*) data input/output (I/O) overhead, and (*iii*) complicated interconnections are alleviated. The new *Sliding* Memory Plane (SliM) array processor achieves higher speeds than existing mesh-connected array processors. In this architecture inter-memory communication as well as inter-PE communication can take place without interrupting PEs. In other words, during computation, the contents of all register cells on the sliding memory plane can be shifted simultaneously and in the same direction to the neighboring cells. In addition, the duplicated I/O planes can provide buffering capability. Since communication, I/O and computation occur at the same time, the communication and I/O overhead will be greatly diminished. The performance of the SliM shows remarkable improvement over existing machines. The SliM array processor is a fine-grained bit-parallel SIMD architecture.

## 1. INTRODUCTION

Several disadvantages of existing mesh-connected array processors, such as communication overhead between processing elements (PEs), data input/output (I/O) overhead, and complicated interconnections limit efficient implementation of low level vision tasks. In order to alleviate these disadvantages, and to achieve higher speeds than existing mesh-connected array processors, a new *Sliding* Memory Plane (SliM) array processor is proposed for low level vision tasks.

Most operations needed in low level computer vision tasks are neighborhood operations which transform the value of each pixel into a new value calculated from itself and its neighboring pixels. Such operations can be accomplished with a high degree of concurrency by using mesh-connected array processors which are well suited to the structure of image data (*spatial* characteristic) [1]. Many mesh-connected array processors have been proposed [2]. Examples of such architectures include MPP [3-5], CLIP [6-8], DAP [9], LIPP [10], etc.

These array processors have major limiting factors towards speed-up. During processing, almost all communications are *localized.* In other words, a great deal of local communication occurs between neighboring PEs. This communication overhead is a significant problem on existing mesh-connected array processors [1][5][10]. Moreover, when the size of a window (neighborhood operator) is larger than 3 x 3, this overhead may seriously degrade performance.

To reduce communication overhead, the LIPP architecture was proposed [10]. However, this scheme has several drawbacks. Complicated gate logic circuits for multiplexing and special purpose RAM (Random Access Memory) and processors are needed [2]. In addition, if the size of window is larger than 3 x 3, or if other types of windows (circular, diamond, rectangular, etc), which are frequently used in low level vision [11-12], are employed, communication overhead may become worse.

In order to alleviate these disadvantages (communication overhead, data I/O overhead, complicated interconnections), and to achieve higher speeds than existing mesh-connected array processors, the new massively parallel SliM array processor is proposed in this paper for low level vision tasks. In this architecture inter-memory communication as well as inter-PE communication can take place without interrupting PEs. In other words, during computation, the contents of all register cells on the sliding memory plane can be shifted simultaneously and in the same direction to the neighboring cells. In addition, the duplicated I/O planes can provide buffering capability without interrupting PEs. Since communication, I/O and computation occur at the same time, communication and I/O overhead can be overlapped with computation, and communication and I/O overhead are significantly diminished. Moreover, four communication links for each PE are sufficient, instead of the eight, thus greatly simplifying connectivity. The SliM array processor is a fine-grained bit-parallel SIMD architecture.

The remainder of this paper is organized as follows. In section 2, the SliM array processor is introduced. The overall scheme and the structure of a PE are presented. In section 3, the analytical model of the SliM is established and compared with those of existing machines. In section 4, the applications to low level vision tasks are discussed. The performance evaluation based on the analytical model is described. The performance of the SliM for low level vision tasks shows remarkable improvement over existing machines. Finally, section 5 contains concluding remarks.

## 2. ARCHITECTURE

In this section, we describe the new architecture for low level vision, and compare its features with those of existing mesh-connected array processors.

### 2.1 Overall System

The logical block diagram of the *Sliding* Memory Plane (SliM) array processor is shown in Fig. 2.1.1. The processor plane consists of *N* x *N* processors. The sliding memory plane *S* consists of *N* x *N* register cells. The top row of the sliding memory plane is connected to the bottom row to form a wrap-around mesh connection scheme. Similarly, the leftmost column is connected to the rightmost column. The processors can process the data in the sliding memory plane *S*.

*The research was supported in part by IBM.

CH2649-2/89/0000/0537$01.00 © 1988 IEEE

In the SliM array processor, as in most mesh-connected array processors, grid like communication links among PEs are used for inter-PE communication. These grid communication links are also used for inter-memory communication in order to reduce communication overhead. The I/O planes, $D$ and $D'$, are exclusively used for input and output, whereas the sliding memory plane, $S$ is used for data communication (parallel data movement). The I/O processor (IOP) can load the input data in a row-parallel (or column-parallel) manner into the I/O shift register plane $D$ or $D'$. Of course if sensory array is used for $D$ or $D'$, image-parallel I/O can be achieved. After being loaded, the data in $D$ or $D'$ is shifted into the sliding memory plane $S$ in one unit cycle time (parallel shift).

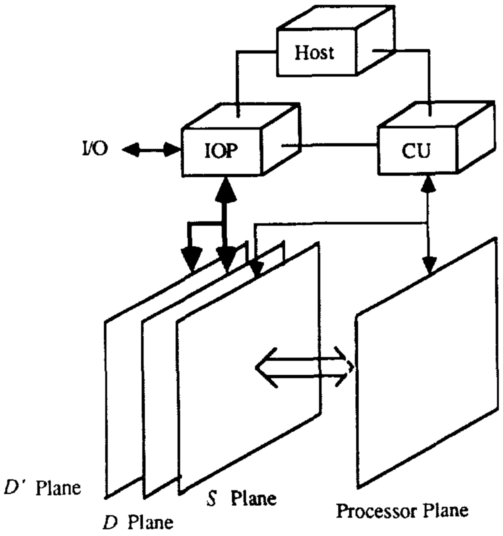


Fig. 2.1.1 Sliding Memory Plane (SliM) Array Processor

The role of the Host is to control the CU and the IOP. The CU consists of three subunits; the I/O control subunit, the processor control subunit and the sliding memory plane control subunit. The CU can broadcast not only instruction set to processors but can also broadcast control signals to the sliding memory plane at the same time instant. The CU controls the data movement in $S$.

While processors process the data in $S$ from $D$, the IOP can unload the output data from $D'$, load another input data into $D'$. The output data in $D'$ are those which were previously processed and shifted from $S$. While processors process the data in $S$ from $D'$, the IOP can unload the output data from $D$, load another input data into $D$, alternately. This buffering capability allows I/O to be overlapped with computation. The details of a PE will be described in the next subsection.

### 2.2 A Processing Element

A PE shown in Fig. 2.2.1 consists of a ALU (Arithmetic Logic Unit), its local memory, registers, multiplexers (MUXs) and a demultiplexer (DMUX). The register $s$ is an element of the sliding memory plane $S$ shown in Fig. 2.1.1. Similarly, $d$ and $d'$ are elements of the I/O planes $D$ and $D'$. $s$ is connected to the neighboring registers (North, East, West or South) via a multiplexer. Thus, a PE is also connected to its four neighboring PEs. $d$ and $d'$ are connected only to their left and right neighboring registers. This scheme provides inter-memory communication as well as inter-PE communication as described below.

While the ALU processes the pixel in $s$ from $d$, a neighboring pixel can be shifted into $s$, and I/O operation can occur through $d'$ register. Again, while the ALU processes the new pixel shifted into $s$, another neighboring pixel can be shifted into $s$, and I/O operation can occur through register $d'$. These operations can be executed by all PEs at the same time instant. Each operation can be controlled separately by the I/O control subunit, the sliding memory plane control subunit or the processor control subunit. Thus, I/O overhead and inter-PE data communication overhead can be overlapped with computation. All pixels are moved into the neighboring location register cells at the same time and in the same direction. In addition, direct inter-PE communication can be provided like in existing array processors.

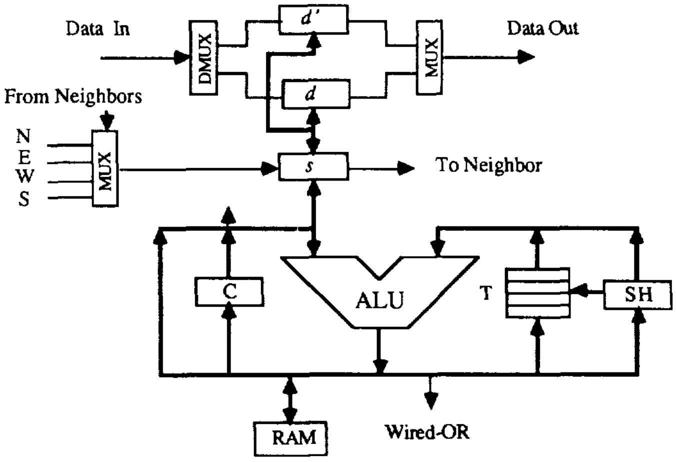


Fig. 2.2.1 A Processing Element

The other major components of a PE consist of an ALU, a shift register (SH), a condition register (C), four registers (T's) and a small memory module which is used for local data storage. The 8-bit ALU provides boolean functions as well as arithmetic functions. SH performs arithmetic and logic shifts. C provides conditional operations and control of the MUX for neighboring communication. $s$ stores the data transferred from neighboring PEs. For direct inter-PE communication, ALU stores the data to be transferred into $s$ register. During this communication, ALU can perform other operations. The memory is a one-byte wide RAM.

Most operations in image processing are performed on grey-level rather than binary data [7-8]. Hence, the SliM array processor uses bit-parallel processing rather than bit-serial processing, because the former is better suited to low level vision and the speed of bit-parallel is faster than that of bit-serial. Each register cell contains one pixel. The communication link between neighboring PEs is, however, one-bit wide so that area of VLSI can be efficient. The thick lines represent bit-parallel paths, while the thin lines represent bit-serial paths.

The SliM is different from existing mesh-connected array processors, in that the Sliding memory plane ($S$) provides parallel data movement during computation, and the I/O planes, $D$ and $D'$, provide data buffering capability. In the SliM, additional communication links for NE, SE, NW and SW neighbor cells are not needed, because these contents can be accessed after two sliding memory plane shifts. Thus, the SliM can significantly reduce the serious disadvantages of existing array processors, that is, inter-PE communication overhead, I/O overhead and complicated interconnections. In general, communication time may be invisible, because computation time is longer than communication time.

Since the movement of sliding memory plane is programmable and flexible, any shape and any size of window may be employed on the sliding memory plane with very little or no communication overhead. In contrast, the existing array processors may suffer from performance degradation. If the window size is larger than 3 x 3 or the shape of window is not square, performance degradation becomes worse.

## 3. ANALYTICAL MODEL

Neglecting the time for program loading to the CU, the total processing time ($T_A$) consists mainly of three time components : data I/O time ($T_{I/O}$), computation time ($T_{CP}$), and inter-PE communication time for data exchange ($T_{PP}$). These times are functions of algorithm length ($L$), image size ($I$) and the number of the PEs employed ($N$). If the size of the image is larger than the size of the array processor, then the size of a subimage becomes $N$. The total number of subimages ($n_s$) is $\left\lceil \frac{I}{N} \right\rceil$. Thus, the total time to process a whole image can be expressed by

$$T_A = n_s(T_{I/O} + T_{CP} + T_{PP}) \qquad (2.3.1).$$

The I/O time using column parallel (or row parallel) for a subimage is expressed by

$$T_{I/O} = 2\, n_{i/o}\, t_{i/o} \qquad (2.3.2.a)$$

where $n_{i/o}$ is the number of columns in a subimage, and $t_{i/o}$ is the I/O control subunit cycle time for one column. Of course if $D$ and $D'$ are the actual sensor arrays, then all pixels can be loaded in one cycle.

The computation time for a subimage is expressed by

$$T_{CP} = n_i\, t_i \qquad (2.3.2.b)$$

where $n_i$ is the number of instructions to execute a specific algorithm, and $t_i$ is the processor instruction cycle time.

The inter-PE communication time for a subimage is expressed by

$$T_{PP} = n_b\, n_c\, t_c \qquad (2.3.2.c)$$

where $n_b$ is the number of bits to be transferred to neighboring PEs while executing the algorithm, $n_c$ is the number of neighboring communications for a specific algorithm, and $t_c$ is the communication time for one bit between neighboring PEs. Since the width of communication link is one bit, the number of bits to be transferred must be considered instcad of the number of bytes.

Therefore, the total time for a whole image is again expressed by

$$T_A = 2n_s n_{i/o} t_{i/o} + n_s n_i t_i + n_s n_b n_c t_c \qquad (2.3.3).$$

Since the SliM has buffering capability, I/O can be overlapped with processing. The total processing time reduces to

$$T_A = \begin{cases} n_s T_{CP} + n_s T_{PP} & \text{if } n_s T_{CP} + n_s T_{PP} \geq n_s T_{I/O} \\ n_s T_{I/O} & \textit{otherwise} \end{cases} \qquad (2.3.4).$$

In addition, the SliM has inter-memory communication capability during computation. Thus, communication can be overlapped with computation. This further reduces total processing time. The total time $T_A$ can be expressed as follows

$$T_A = \begin{cases} n_s T_{CP} & \text{if } n_s T_{CP} + n_s T_{PP} \geq n_s T_{I/O} \text{ and } n_s T_{CP} \geq n_s T_{PP} \\ n_s T_{PP} & \text{if } n_s T_{CP} + n_s T_{PP} \geq n_s T_{I/O} \text{ and } n_s T_{CP} < n_s T_{PP} \\ n_s T_{I/O} & \textit{otherwise} \end{cases} \qquad (2.3.5).$$

As shown in (2.3.5), the total time of the SliM can be expressed by one of three time components. In general, computation time is larger than I/O time or inter-PE communication time. Hence, on the SliM the total time is composed of only pure computation time, if the above conditions are satisfied. In contrast, the total time may be expressed by Eq. (2.3.3) in existing mesh-connected array processors. In some tasks, inter-PE communication cannot be fully overlapped, and some portions of $T_{PP}$ may still exist. Then, $T_A$ is expressed by $n_s T_{CP} + (1 - \rho) n_s T_{PP}$, if $n_s T_{CP} + (1 - \rho) n_s T_{PP} \geq \rho n_s T_{PP}$, where $\rho$ is the overlapped portion of $T_{PP}$ with $T_{CP}$.

## 4. APPLICATIONS AND PERFORMANCE EVALUATION

The SliM is suitable for low level computer vision tasks, where there are excessive data exchanges between neighbors, because it can perform neighborhood operations with little or no communication overhead. For instance, 2-D convolution, median filtering, average value, template matching, thresholding, zero-crossing, etc., are suitable applications for the proposed architecture. Edge detection can be performed by using 2-D convolution algorithms. Gradient, Laplacian, difference of Gaussians, and Sobel operator are some of examples. After convolution of image with these operators, edges can be efficiently detected by the SliM. Parallel algorithms for 2-D convolution and median filtering are illustrated.

### 4.1 Convolution

A parallel 2-D convolution algorithm [12] is highly suited for implementation on the SliM architecture. For a image convolution using 3 X 3 window, 9 multiplications, 8 additions, and 8 inter-PE communication steps are required on existing mesh-connected array processors. Since the time to shift contents of the sliding memory plane is much less than the time to execute one multiplication, the parallel data movement can be completed within the computation time. If the SliM is employed, 9 multiplications, and 8 additions, are needed, regardless of the size of image. Inter-PE communication overhead is overlapped, and I/O overhead can also be overlapped if the computation time is larger than the I/O time. Therefore, the total processing time for a whole image consists of computation time (i.e., 9 multiplications and 8 additions). The 3 X 3 convolution window and the movement of the sliding memory plane are shown in Fig. 4.1. If the movement starts at the center pixel and ends at the southwest pixel through a clockwise direction, the movement of the contents in the sliding memory plane is O → S → E → N → N → W → W → S → S. The direction of the movement is a Hamiltonian path which starts at any node and visits every node only once. Any kind of window shape and size can be employed with little or no communication overhead.

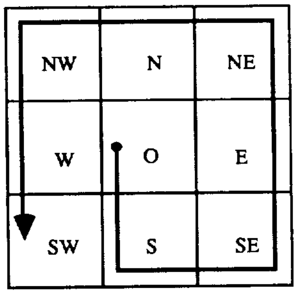


Fig. 4.1 3 x 3 Square Window and Movement of Sliding Memory Plane

Before we discuss performance issue, we briefly describe the figures of MPP. Then, we make assumptions for performance evaluation of the SliM. In MPP, one memory access and several operations can be executed together within one cycle (100 ns). The actual memory access time is about 50 ns [4]. Thus, several operations can be merged into one instruction [3-4]. The performance evaluation of the SliM is based on the following conservative assumptions. First, memory access time (1 byte) is 100 ns, and is defined as a nominal cycle time. Only one memory access without any operation can be executed in one cycle. Two operations can be merged into one instruction if there is no conflict, and this instruction can be executed within one cycle. Thus, $t_i$ is 100 ns. In practice, the time for 1-bit memory access is equal to the time for 1-

byte memory access. We assume most of the operations like addition, shift, compare, etc., to be executed within a half cycle except several operations like multiplication, division, etc. Second, 8-bit passing to a neighboring PE is completed within one cycle time. Third, a multiplication of two 8-bit integers requires 8 additions and 7 shifts. If two operands are from registers, 8 cycles are needed. But if two operands are from memory and the result is stored into memory, 12 cycles are needed.

To simplify the performance evaluation, the size of the SliM is assumed to be greater than or equal to the size of the image. In addition, assume that window coefficients are broadcasted from the CU. Since *Sliding*, i.e., inter-PE communication, can occur during execution of operation, Sliding and operation statements are put on the same line. In the following description of the algorithm, wherever there are more than one statement on the same line, then these statements can be considered to be overlapping. The general algorithm is shown as follows, in which $s$ represents a register on the Sliding memory plane $S$ and $T$ is a set of registers in a PE.

```
$T \leftarrow s*w_0$;        $s \leftarrow$ a neighboring pixel;     /* Sliding */
for i ← 1 until window_size - 1 do
    $T \leftarrow T + s*w_i$;    $s \leftarrow$ a neighboring pixel;     /* Sliding */
```

As shown above, window_size multiplications and window_size - 1 additions must be required. If the window size is 3 x 3 and the number of bits per pixel is 8, it requires $9*n_m + 8$ cycles, where $n_m$ is the number of cycles for two 8-bit integer multiplication. If we assume $n_m$ to be 8, the total time is 8.0 μsec.

### 4.2 Median Filtering

In general, median filtering requires a sorting algorithm after all neighboring pixels are collected. Since the sorting algorithm itself may take long and sorting and collecting neighbors cannot occur simultaneously, median filtering is a time consuming task.

A new parallel 2-D median filtering algorithm is similar to the parallel 2-D convolution algorithm with respect to the window operation and the movement of the sliding memory plane. The ordered singly linked list is used for median filtering. Each PE has its own list. After shifting the sliding memory plane, each PE can access its neighboring pixel, and insert it into its list in order. While this insertion is taking place, the sliding memory plane can be shifted. If the time for insertion into the list is larger than the time for shifting, communication overhead is overlapped, and only the time for creating the ordered singly linked list of neighboring pixels is needed. After making the list, the median value in each list can be easily found simultaneously. The total processing time thus consists of only the time for making the list. In contrast with the widely used median filtering algorithms which consist of collecting and sorting procedures, sorting procedure is not required, and collecting procedure is invisible in the new algorithm. The worst case is in which every time the pixel received from a neighbor is greater than the pixels in the list. For a 3 x 3 window and 8-bit/pixel, the estimated time is 12.7 μsec in the worst case.

Performance evaluation for other tasks, i.e., Gassusian, zero-crossing, threshold, average value, Sobel operator, etc., shows remarkable improvement over existing mesh-connected array processors. Due to page limitation, the details are omitted.

## 5. CONCLUSION

As discussed, remarkable performances can be achieved by the SliM. There are several reasons why the performance of the SliM is better than that of existing machines. First, inter-PE communication and I/O overhead can be overlapped with computation. Second, bit-parallel processing is faster than bit-serial processing. Third, on existing machines PEs store pixels into memory, after pixels are received. During processing, the pixels stored in memory must be accessed for computation. This memory access overhead is significant. In contrast, the $S$ plane on the SliM contains the pixels which can be transferred to neighbors, and directly accessed by the ALU. Moreover, a set of registers ($T$) can be efficiently used. Thus, overhead for memory access can be reduced.

In summary, the proposed array processor, the SliM array processor, for low level vision alleviates the drawbacks of existing array processors. Performance degradation due to these drawbacks is minimized to allow higher throughput than existing array processors. The SliM is a fine-grained bit-parallel SIMD architecture. The simplicity and the regularity of the architecture, and the straightforward control strategy make this architecture highly suitable for VLSI implementation.

### Acknowledgement

We would like to thank Dr. N. Nandhakumar for his valuable suggestions and comments. We thank V. Chaudhary for reading and commenting.

### REFERENCES

[1] F.A. Gerritsen, "A Comparison of the CLIP4, DAP and MPP Processor-Array Implementations," *Computing Structures for Image Processing*, Edited by M.J.B. Duff, Academic Press, pp. 15-30, 1983.

[2] T.J. Fountain, "A Survey of Bit-Serial Array Processor Circuits," *Computing Structures for Image Processing*, Edited by M.J.B. Duff, Academic Press, pp. 1-13, 1983.

[3] K.E. Batcher, "Design of a Massively Parallel Processors," *IEEE Trans. Comput.* Vol. C-29, pp. 836-840, Sep. 1980.

[4] K.E. Batcher, "Bit-Serial Parallel Processing Systems," *IEEE Trans. Comput.* Vol. C-31, pp. 377-384, May 1982.

[5] T. Kushner, A.Y. Wu, and A. Rosenfeld, "Image Processing on MPP," *Pattern Recognition*, Vol. 15, No. 3, pp. 121-130, 1982.

[6] M.J.B. Duff, "Review of the CLIP image processing system," in *Proc. National Computer Conference*, 1978, pp. 1055-1060.

[7] T.J. Fountain, "Plans for the CLIP 7 Chip," *Integrated Technology for Parallel Image Processing*, Edited by S. Levialdi, Academic Press, pp. 199-214, 1985.

[8] T.J. Fountain, K.N. Matthews, and M.J.B. Duff, "The CLIP7A Image Processor," *IEEE Trans. PAMI Vol. 10, No. 3, pp. 310-319, May 1988.* Academic Press, 1986.

[9] S.F. Reddaway, "DAP - A Distributed Processor Array," in *Proc. First Annual Symposium on Computer Architecture*, 1973, pp. 61-65.

[10] T.S. Ericsson, and P-E. Danielsson, "LIPP - A SIMD Multiprocessor Architecture for Image Processing," in *Proc. 10th Annual Int. Symp. Comput. Architect.*, 1983, pp 395-400.

[11] A. Rosenfeld and A.C. Kak, *Digital Picture Processing*, Academic Press, 1982.

[12] S.-Y. Lee and J.K. Aggarwal, "Parallel 2-D Convolution on a Mesh Connected Array Processor," *IEEE Trans. PAMI*, Vol. PAMI-9, pp. 590-594, July 1987.

# A Study of the Generalized Multiple Bus-Connected Parallel Computer

Chia-Jiu Wang
University of Colorado
Department of Electrical and Computer Engineering
Colorado Springs, CO 90933-7150

C. Wu and Victor P Nelson
Auburn University
Department of Electrical Engineering
Auburn, AL 36849

ABSTRACT

The advancement of optical interconnection and microelectronics packaging technology have made it possible to use optical fibers as media to carry information between processors. In this paper, we study an architecture called the generalized multiple bus-connected parallel computer (GMBPC). The GMBPC is suitable for optical interconnection and wafer scale packaging applications. A special case of the GMBPC operating as a data flow machine has been studied and fabricated by Hughes. The GMBPC operates as a message passing MIMD machine in this investigation. The performance of the GMBPC is modeled by generalized stochastic Petri nets (GSPN). To lessen the exponential distribution assumption used in the GSPN model, Monte Carlo simulation technique has been used to predict the performance probabilistically. For application, two-dimensional fast Fourier transform algorithms for the GMBPC are derived. The performance of solving multi-dimensional Poisson's equation by relaxation and fast Fourier transform methods on the GMBPC with different parameters of GMBPC and problem sizes has also been analyzed.

Keywords: Message passing, Petri nets, Monte Carlo simulation, Normalized Processing Power, Posisson Equation.

## 1. INTRODUCTION

As the number of interconnected processing elements increases, the interconnection among them plays an increasingly important role in influencing the performance of the multicomputer system. A network configuration is considered ideal if it possesses a direct communication link between any two interacting processing elements. To provide a reasonable communication bandwidth, various mesh-connected types and many other schemes are described [1-5]. Among the interconnection schemes bus systems are inexpensive and easy to implement but have limited bandwidth, fanout and lack fault tolerance. Meanwhile, computer architectures are driven by available technologies and application needs. Recently the optical interconnection technolog [6] has shown that the optical fiber can be used to transmit optical signals between two silicon VLSI chips recessed into a silicon carrier. A small gallium arsenide transceiver chip electrically connected to the VLSI chip is employed to convert the electrical signals from the silicon chip into into light signals and vice versa. Optical interconnects offer the combination of large bandwidth and large fanout. The bus-connected structure suffers from limilted bandwidth and fanout. The fanout limitation can be improved to some extent by consuming larger silicon area to construct drivers with higher driving capability, but the bandwidth limitation of the bus-connected structure is not easy to overcome unless other material rather than electrical wires is used to carry signals. Therefore, the combination of optical interconnects and the bus-connected structure seems to be a promising approach to construct a parallel computer, because the properties of optical interconnects can remove the limitations in the bus-connected system and keep the merits of the bus-connected structure. In this paper we investigate a generalized multiple bus-connected parallel computer (GMBPC) assuming the buses are made of optical fibers. In section 2 the GMBPC is introduced. The stochastic performance modeling of the GMBPC is presented in section 3. Application examples such as the two-dimensional (2D) fast Fourier transform (FFT) and the multidimensional Poisson's equation are given in section 4. Conclusion and discussion are given in section 5.

## 2. GENERALIZED MULTIPLE BUS-CONNECTED PARALLEL COMPUTERS

The generalized multiple bus-connected parallel computer (GMBPC) is defined as a four-tuple denoted as GMBPC = (N, K, L, D) where N and K represent the total number of processing elements (PE's) and the total number of buses in the GMBPC respectively, L is the number of processing elements within a bus, and D represents the dimension. The relation between N, K, L, D can be formulated as $L = N^{1/D}$, $K = (N^{1/D})^{D-1}D$. Figure 1 shows a case of GMBPC = (64, 48, 4, 3). A special case of the GMBPC operating as a data flow machine has been studied and fabricated by Hughes, known as Hughes data flow multiprocessor for real-time radar signal processing [7]. The GMBPC is assumed to be a message passing MIMD machine throughout this paper. In order to get a symmetrical structure, we always select N and D properly to make L an integer.

## 3. STOCHASTIC PERFORMANCE MODELING

The behavior of the GMBPC is modeled as a generalized stochastic Petri net (GSPN) graph. From the GSPN graph the performance of the GMBPC is derived analytically. The Monte Carlo simulation technique is also used to investigate more cases which can't be easily modeled by the GSPN model.

### 3.1 GSPN Model of the GMBPC

The generalized stochastic Petri nets (GSPN) have been used to model computer systems. The results of this modeling scheme have been verified by experiments and reporeted in many literatures [8,9,10]. We apply the GSPN modeling scheme to model the behavior of the GMBPC. In order to apply the GSPN modeling principles, the states of each PE in the GMBPC are classified as follows: 1) Active: The PE is executing programs in its own private memory, 2) Communicating: The PE is exchanging messages with other cooperating PE's through buses, 3) Queued: The PE is waiting to access a bus.

Modeling the GMBPC at the message-passing level corresponds to the identification of processing periods (of average length $1/\lambda$) which require no bus access, and data transfer periods (of average length $1/\mu$) which do require bus access.

CH2649-2/89/0000/0541$01.00 © 1988 IEEE

### 3.2 Modeling Unit Structure and Assumptions

Since the GMBPC is a regular and symmetrical structure, we apply the GSPN model to a GMBPC having only one dimension. In other words a one-dimensional GMBPC is used as an unit structure for performance modeling. Another reason of modeling the unit structure rather than the entire GMBPC is to keep the reachability set from becoming unmanageably large, reducing computational complexity. The modeling unit structure is shown in Figure 2. It is assumed that the workloads are uniformly distributed among all PE's, resulting in uniform processing loads and communication loads. The data processing period follows an exponentially distributed random variable with rate $\lambda$, and the message transfer period also follows an exponentially distributed random variable with rate $\mu$.

### 3.3 Performance Estimation

For a GMBPC = (64, 48, 4, 3), the corresponding GSPN graph of the modeling unit structure is shown in Figure 3. This GSPN model comprises 13 places and 12 transitions. The uniform workload assumption implies that $\lambda_1 = \lambda_2 = \lambda_3 = \lambda_4 = \lambda$ and $\mu_1 = \mu_2 = \mu_3 = \mu_4 = \mu$. The immediate transition rate r is determined by bus acquisition mechanisms. Since all the transition rates can be categorized into three groups, each group representing a specific function of the GMBPC, the 12 transitions can be reduced to 3 transitions. The number of places can also be reduced by combining places $P_1$, $P_4$, $P_7$, $P_{10}$ into one place, representing active state. By the same method we can reduce the rest places except $P_{13}$ into two places, representing the communicating state and the queued state. The simplified GSPN model is shown in Figure 4. With the simplified model the performance of larger size GMBPC can be modeled and estimated by placing larger number of tokens in the place $P_1$. Because the bus is assumed to be made of optical fibers, the bus speed is expected to be much higher than PE data processing speed. We consider three cases which are $\lambda/\mu$ = 25, $\lambda/\mu$ = 50 and $\lambda/\mu$ = 100 respectively. The performance is measured by a index, P, called normalized processing power, which is the ratio of the average number of active PE's over the total number of PE's in the system. Table 1 presents P versus L for the three cases. The data shown in

| | L | 2 | 4 | 6 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|
| $\lambda/\mu$=100 | P | 0.99 | 0.99 | 0.98 | 0.98 | 0.98 | 0.98 | 0.98 |
| $\lambda/\mu$=50 | P | 0.98 | 0.98 | 0.97 | 0.97 | 0.97 | 0.97 | 0.97 |
| $\lambda/\mu$=25 | P | 0.96 | 0.95 | 0.95 | 0.94 | 0.94 | 0.93 | 0.93 |

Table 1. P vs. L for GMBPC's with D=1 (N=L)

Table 1 demonstrates that even L = 12, the processing power still remains above 90% for the three cases. This is a very good performance but it represents only a one-dimensional case. In the following we apply the Monte Carlo simulation scheme to estimate the GMBPC performance with higher dimensions and more PE's within a bus. The GSPN model will become very complex and difficult to solve when modeling a multidimensional GMBPC.

### 3.4 Monte Carlo Simulations

In Monte Carlo simulations we analyze the following cases:

1) one dimension GMBPC with L up to 50 PE's per bus,
2) two dimension GMBPC with L up to 50 PE's per bus,
3) three dimension GMBPC with L up to 50 PE's per bus,
4) the data processing period and the data transfer period are exponentially distributed random variables with different average values,
5) the data processing period and the data transfer period are uniformly distributed random variables with different average values.

In Monte Carlo simumlations we assume all the I/O ports of the PE can be enabled concurrently. The transputer [11] and the FAIM-1 [12] are two examples which show that all the I/O ports of a PE can be enabled simultaneously. Tables 2, 3 and 4 show the simulation results for three cases with different distribution functions.

| | N | $2^2$ | $4^2$ | $6^2$ | $10^2$ | $20^2$ | $30^2$ | $40^2$ | $50^2$ |
|---|---|---|---|---|---|---|---|---|---|
| $\lambda/\mu$=100 | P | 0.99 | 0.94 | 0 94 | 0.90 | 0.84 | 0.79 | 0.74 | 0.69 |
| $\lambda/\mu$=50 | P | 0.94 | 0.90 | 0.88 | 0.82 | 0.74 | 0.65 | 0.57 | 0.52 |
| $\lambda/\mu$=25 | P | 0.89 | 0.84 | 0.79 | 0.72 | 0.56 | 0.45 | 0.38 | 0.32 |

Table 2. P vs. N for a GMBPC with D=2
(data processing and transfer periods follow the exponential distribution function.)

| | N | $2^2$ | $4^2$ | $6^2$ | $10^2$ | $20^2$ | $30^2$ | $40^2$ | $50^2$ |
|---|---|---|---|---|---|---|---|---|---|
| $\lambda/\mu$=100 | P | 0.99 | 0.95 | 0 94 | 0.91 | 0.85 | 0.80 | 0.75 | 0.70 |
| $\lambda/\mu$=50 | P | 0.94 | 0.92 | 0.89 | 0.83 | 0.74 | 0.65 | 0.58 | 0.52 |
| $\lambda/\mu$=25 | P | 0.89 | 0.85 | 0.80 | 0.72 | 0.57 | 0.46 | 0.38 | 0.32 |

Table 3. P vs. N for a GMBPC with D=2
(data processing and transfer periods follow the uniform distribution function.)

| | N | $2^3$ | $4^3$ | $6^3$ | $10^3$ | $20^3$ | $30^3$ | $40^3$ | $50^3$ |
|---|---|---|---|---|---|---|---|---|---|
| $\lambda/\mu$=100 | P | 0.99 | 0.92 | 0 91 | 0.85 | 0.77 | 0.70 | 0.63 | 0.57 |
| $\lambda/\mu$=50 | P | 0.94 | 0.87 | 0.74 | 0.74 | 0.64 | 0.52 | 0.44 | 0.37 |
| $\lambda/\mu$=25 | P | 0.89 | 0.77 | 0.58 | 0.62 | 0.42 | 0.30 | 0.23 | 0.18 |

Table 4. P vs. N for a GMBPC with D=3
(data processing and transfer periods follow the exponential distribution function.)

The data obtained from Monte Carlo simulations agree quite well with the data from GSPN model in the same situations.

## 4. APPLICATIONS

The GMBPC has been studied in solving two-dimensional and three-dimensional Poisson's equations. Algorithms designed for the GMBPC machine in solving 2D FFT problems are also derived.

### 4.1 Definitions and Assumptions

1. The I/O ports of each PE can be enabled concurrently.
2. The communication time is modeled by $t_{st} + t_{comm}$ per data transfer, where $t_{st}$ is an overhead due to address calculation, control of data flow and synchronization and any other overhead for accessing the bus, $t_{comm}$ varies as the GMBPC structure varies. For the reason of simplicity, $t_{st}$ is assumed to be a constant. (Based on the data provided in [11], $t_{st}$ is asssumed to be 5000ns throughout this paper.) $t_{byte}$ represents the one byte transfer time over the optical serial bus.
3. Each datum is a single precision floating point complex number with 8 bytes in length (four for the real part, four for the imaginary part) for the FFT algorithm and a double precision floating point real number for the Jacobi method..

### 4.2 a 2D FFT problem for the GMBPC

We first introduce the FFT algorithm [13] designed for the two-dimensional GMBPC, then the algorithm for the three- dimensional GMBPC will be presented.

Alg. 1 : 2D $n^2$-point FFT on the GMBPC=($L^2$,2L,L,2), n>$L^2$.

step 1) partition $n^2$ points into $L^2$ domains, so that each domain has $n/L^2$ complete sets of one-dimensional data. Each set of data is ready for 1D n-point FFT processing.

step 2) assign a domain to a PE consecutively, so that each PE will have an unique data domain.

step 3) All PE's simultaneously perform 1D n-point FFT on their assigned data domains, i.e. performing 1D n-point FFT $n/L^2$ times each PE.

step 4) Each PE partitions the transformed data into $L^2$ groups with $(n^2/L^2)(1/L^2)$ data in each group.

step 5) All the PE's on the same row swap L groups of data between each other, then all the PE's on the same column do the same data swap afterwards. (Although all rows and columns can swap data at the same time through the dedicated row and column buses, column (row) data swap can not start until after the end of row (column) data swap because some data are not available.)

step 6) Each PE performs $n/L^2$ times 1D n-point FFT.
step 7) Finish.

In step 5 the data swap is to redistribute partially transformed data to the appropriate PE's for another dimension 1D FFT processing. Based on the above algorithm, we calculate the speedup of solving 2D FFT problems on a two-dimensional GMBPC = $(L^2, 2L, L, 2)$. The communication time for each data transfer is $T_{sc} = t_{st} + t_{byte}8(n^2/L^2)1/L$. The total communication time is $2L(L-1)T_{sc}$. A single PE performing a 2D $n^2$-point FFT needs execution time $2nT_{FFT}$, where $T_{FFT}$ represents a n-point FFT execution time on a single PE. $T_{FFT}$ is proportional to the execution time of $5n(\log_2 n)$ arithmetic operations for complex input data. Assume $t_p$ represents the average one arithmetic operation time, so $T_{FFT}=5n(\log_2 n)t_p$. The speedup S is

$${}_{2DFFT}S_{2DGM}=2nT_{FFT}/(2nT_{FFT}/L^2+2L(L-1)T_{sc}).$$

The above speedup equation is shown in Figure 5.

**Alg.** 2: 2D $n^2$-point FFT on the GMBPC=$(L^3, 3L^2, L, 3)$, $n>L^3$

step 1) partition $n^2$ points into $L^3$ domains, so that each domain has $n/L^3$ complete sets of one-dimensional data. Each set of data is ready for 1D n-point FFT processing.

step 2) assign a domain to a PE consecutively, so that each PE will have an unique data domain.

step 3) All PE's simultaneously perform 1D n-point FFT on their assigned data domains, i.e. performing 1D n-point FFT $n/L^3$ times each PE.

step 4) Each PE partitions the transformed data into $L^3$ groups with $(n^2/L^3)(1/L^3)$ data in each group.

step 5) For the PE's on the same shared bus, $L^2$ groups of data in each PE are swapped between each other. This applies to all PE's and buses in the GMBPC. For a GMBPC having three dimensions (x,y,z) the data swap within either x,y or z dimension can be done simultaneously, but at any time only one dimension data swap is allowed because of the unavailability of some data.

step 6) Each PE performs $n/L^3$ times 1D n-point FFT.
step 7) Finish.

Based on the above algorithm, we calculate the speedup of solving a 2D FFT problem on a GMBPC = $(L^3, 3L^2, L, 3)$. The communication time for each data transfer is $T_{sc} = t_{st} + t_{byte}8(n^2/L^3)1/L$. The total communication time is $3L(L-1)T_{sc}$. The speedup S is

$${}_{2DFFT}S_{3DGM}=2nT_{FFT}/(2nT_{FFT}/L^3+3L(L-1)T_{sc}).$$

Figure 5 presents the speedup versus N for different dimension GMBPCs in solving a 2D FFT problem. Two bus speeds, 800Mbits/sec and 80Mbits/sec are considered. 800Mbits/sec is a realizable bus speed via optical interconnects [6] in the very near future. (Throughout the rest of this paper, a bus with speed = 800Mbits/sec implies this is an serial optical bus.) It shows that for a 2D $4096^2$ point FFT problem a GMBPC with D=2, bus speed=800Mbits/sec has slightly better performance than the one with D=3, bus speed=800Mbits/sec for N < 1000. A GMBPC with D=3 and bus speed=80Mbits/sec has worse performance than one with D=2 and bus speed=800Mbits/sec. It concludes that there are a number of parameters such as the structure of GMBPC, bus speed, PE data processing speed, cost of interconnection network and problem sizes to be considered to achieve the desired speedup.

### 4.3 Solving the Poisson Equation on the GMBPC

In the following we solve a three-dimensional Poisson's equation on GMBPC with different parameters.

**Example** 1: Solving a three-dimensional Poisson's equation with $n^3$ points in size on the GMBPC =$(L^2, 2L, L, 2)$ and n > L. The Jacobi iterative solution technique [14,15] is used in this example. Although the Jacobi method has not been used in practice, it is the basis for understanding the fast converging methods and represents the same complexity of communication. It is assumed that $n^3$ points are equally partitioned into $L^2$ processors. Therefore each PE has equal amount of work in computation. The amount of data for each transfer between two adjacent PEs is $n^2/L$. The communication time per data transfer is $T_{sc}= t_{st} + t_{byte}8n^2/L$. Since all the I/O ports in the PE can be enabled concurrently, the total communication time per iteration is $[2(L-2)+2]T_{sc}$. Based on the INMOS IMS T800 data book [11], the computation time per point per iteration is $t_{comp}$ = 2915ns [14]. The total computation time for the $n^3$ points is $T_{comp} = n^3 t_{comp}$. The cost of communicating convergence checking information on the GMBPC is insignificant because it involves only one number from each PE in each dimension, and is hence neglected. The speedup S is

$${}_{3DPO}S_{2DGM}= T_{comp}/(T_{comp}/L^2+2(L-1)T_{sc})$$

Speedup analysis for this case is presented in Figure 6.

**Example** 2: Solving a three-dimensional Poisson's equation with $n^3$ points in size on the GMBPC =$(L^3, 2L^2, L, 3)$, and n>L. The Jacobi method is also used in this example. Since the $n^3$ points are evenly partitioned into $L^3$ processors, the amount of data for each transfer between two adjacent PE's is $n^2/L^2$. The communication time per data transfer is $T_{sc}= t_{st} + t_{byte}8n^2/L^2$. Since all the I/O ports in the PE can be enabled simultaneously, the total communication time per iteration is $[2(L-2)+2]T_{sc}$. The computation time required for each point is still $t_{comp}$. The total computation time for $n^3$ points is $T_{comp}= n^3 t_{comp}$. The speedup S is

$${}_{3DPO}S_{3DGM}= T_{comp}/(T_{comp}/L^3+2(L-1)T_{sc})$$

Figure 6 presents the speedup versus the number of PE's in GMBPC's with D=2, or 3 and different bus speeds. It has shown that given N PE's in total, a GMBPC with D=2, bus speed=800Mbits/sec, or D=3, bus speed=80Mbits/sec produces the same performance; and the performance of a GMBPC with D=3, bus speed=800Mbits/sec is only slightly better than the just mentioned two cases. Therefore the choice of the D depends on the cost of interconnection network and desired speedup.

In the following we solve the same Poisson's equation but using Fast Fourier Transform method [16,17] rather than Jacobi method.

**Example** 3: Solving a two-dimensional Poisson's equation with $n^2$ points in size and zero Dirichlet boundary conditions by FFT method on the GMBPC. First, transform the Poisson's equation from real space to Fourier space. Second, perform $4n^2$ arithmetic operations. Lastly, transform the data in Fourier space back to real space. The speedup for the GMBPC with D=2 and D=3 are :

$$_{2DPO}S_{2DGM} =$$

$$(4nT_{FFT}+4n^2t_p)/[4nT_{FFT}/L^2+4n^2t_p/L^2+4L(L-1)T'_{sc}]$$

where $T'_{sc} = t_{st}+t_{byte}8(n^2/L^3)$

$$_{2DPO}S_{3DGM} =$$

$$(4nT_{FFT}+4n^2t_p)/[4nT_{FFT}/L^3+4n^2t_p/L^3+6L(L-1)T''_{sc}]$$

where $T''_{sc} = t_{st}+t_{byte}8(n^2/L^4)$

The speedup comparison between different dimension GMBPC's in solving a 2D Poisson's equation by FFT method is presented in Figure 7. It shows that an optical-interconnected GMBPC with D=2 has better performance than a non-optical-interconnected GMBPC with D=3. It also indicates that when D=2 the speedup seems to get flat much faster than when D=3 that provides more communication paths.

## 5. CONCLUSION AND DISCUSSION

A bus-connected multiple parallel computer has been studied in this paper. The bus-connected parallel computers are inexpensive and easy to implement, and the limited bandwidth and fanout can be eliminated by using optical buses. The GSPN model and Monte Carlo simulations show that a GMBPC can achieve 70% of its full processing power even N=2500 and D=2. A speedup comparison in solving a 2D FFT problem and a 3D Poisson's equation on GMBPC's with different dimensions has shown that an optical-interconnected GMBPC can indeed generate speedup closing to the number of PE's in a GMBPC. The relation between speedup, L, and D studied in this paper and the cost of interconnection network can provide a guide for the design of GMBPCs as message passing machines.

**ACKNOWLEDGEMENT** - The author C. Wu likes to thank Auburn University Grant-in-Aid program.

## REFERENCES

[1] R. Miller et al., "Computational Geometry on a Mesh Connected Computer," Proc. 1984 Int'l. Conf. Parallel Processing, pp. 66-73.

[2] C.S. Raghavendra and V.K.P. Kumar, "Permutations on ILLIAC IV-type Network," Proc. 1984 Int'l. Conf. Parallel Processing, 1984

[3] A.P. Reeves, "A Systematically Designed Binary Array Processor," IEEE trans. Comp., April 1984.

[4] T. Hoshino et al. "Highly Parallel Processor Array 'PAX' for wide Scientific Applications," Proc. 1983 Int'l Conf. Parallel Processing, 1983, pp. 95-105.

[5] G.A. Anderson and E.D. Jensen, "Computet Interconnection Structures: Taxonomy, Characteristics, and Examples." ACM Computing Surveys, Vol. 7, Dec. 1975. [6] L.D. Hutcheson and P. Haugen, " Optical Interconnects Replace hardware," IEEE Spectrum, March 1987.

[7] R. Vedder, M. Campbell and G. Tucker, "The Hughes data flow multiprocessor," Proc. of the 5th Int'l Conf. on Distributed Sys.

[8] M.A. Marsan et al. "A Class of Generalized Stochastic Petri Nets for Performance Evaluation of Multiprocessor Systems," ACM Trans. Comp. Sys. May,1984. [9] M.A. Marsan et al. "Modeling Bus Contention and Memory Interference in a Multiprocessor Systems," IEEE Trans computers, Jan. 1983, pp. 50-71.

[10] M.K. Molloy, "Performance Analysis Using Stichastic Petri Nets," IEEE Trans. Computers, Sep. 1982.

[11] INMOS, Transputer Ref. Manual, Prentice Hall, 1988.

[12] J.M. Anderson et al.,"The Architecture of FAIM-1," IEEE Computer, Jan. 1987, pp. 55-65.

[13] Gorin, A.L. et al.,"Computing the Two-Dimensional Discrete Fourier Transform on the ASPEN Parallel Computer Architecture," Proc. Int'l Conf. Parallel Processing, 1987, pp. 921-923.

[14] Reed, D.A. et al.,"Stencils and Problem Partionings : Their Influence on the Performance of Multiple Processor Systems," IEEE Trans. Computers, C-36, 1987

[15] Ortega, J.M. and R.G. Voigt,"Solution of Partial Differential Equations on Vector and Parallel Computers," SIAM Review, 1985.

[15] Hockney, R.W.,"A Fast Direct Solution of Poisson's Equation using Fourier Analysis," J. of the ACM,1965, 12, pp. 95-113

[17] Hockney, R.W.,"Characterizing Computers and Optimizing the FACR(1) Poisson-Solver on Parallel Unicomputer ," IEEE Trans. Computers, 1983, C-32, pp. 333-341.

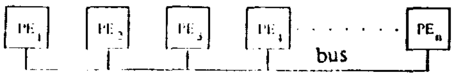


Figure 2. The modeling unit structure

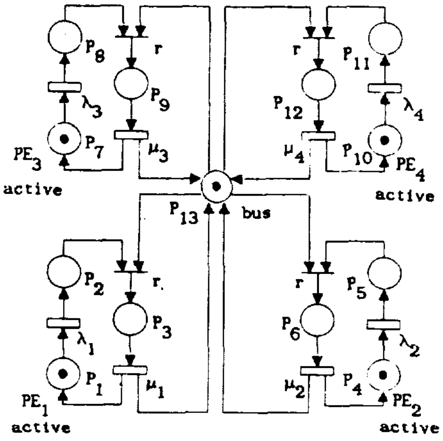


Figure 3. The GSPN for a modeling unit structure of a GMBPC = (64, 48, 4, 3)

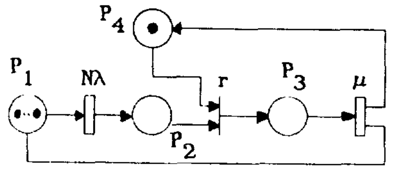


Figure 4. The simplified GSPN for a one-dimensional GMBPC = ( N, 1, N, 1)

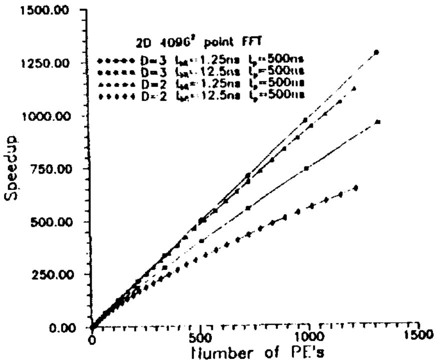


Figure 5. Speedup comparison between different dimension GMBPC in solving a 2D FIT problem

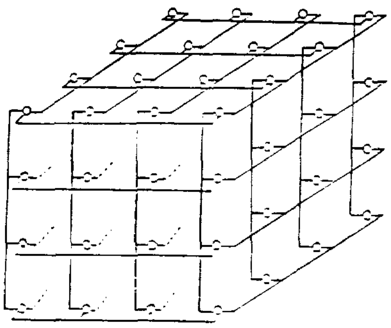


Figure 1. The GMBPC = (64, 48, 4, 3)

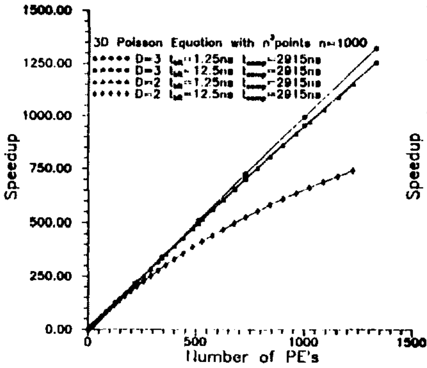


Figure 6. Speedup comparison between different dimension GMBPC in solving a 3D Poisson equation by Jacobi method

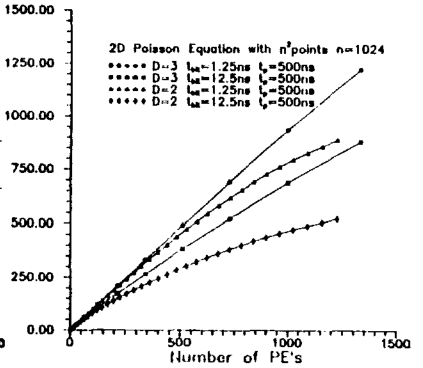


Figure 7. Speedup comparison between different dimension GMBPC in solving a 2D Poisson equation by FFT method

# Neurocomputing with Optical Pipeline Networks [1]

Zhiwei Xu
Department of Electrical and Computer Engineering
Rutgers University, Piscataway, NJ 08854.

Kai Hwang
Department of Electrical Engineering
University of Southern California, Los Angeles, CA 90089.

### Abstract

With the recent advances in neural networks, there has been a great deal of research in neurocomputer architectures based on electronic and optical technologies. This paper presents techniques for realizing connectionist style neurocomputations on Opcom, an optical computer architecture based on the pipeline networking concept. Special attention is paid to the efficient implementation of learning and search procedures.

## 1 Introduction

With the recent advances in neural networks [4,7], there has been a great deal of research in architectural studies of neurocomputers, based on electrical or/and optical technologies [2]. The authors have recently proposed an optical computer architecture, called Opcom [12], which is based on the pipeline networking concept [5] and maps nicely into the characteristics of optical interconnections and gate array structures [6]. The primary operation mode of Opcom is massively parallel and pipeline processing at the gate level.

In this paper, we extend our research in [12] to the area of neurocomputing. More specificly, our interest is in connectionist style computations in digital neural networks [1,3,9]. Special attention is paid to the implementation of the two most important functions of any neural network: learning and search (retrivial).

The architecture of Opcom is described in Section 2. The implementation of neural networks in Opcom and the search process is discussed in Section 3. The implementation of a supervised learning procedure, the Pocket Algorithm [1], is shown in Section 4. Our research results are summerized in Section 5.

## 2 The Architecture of Opcom

We will only briefly discuss features of Opcom that are relevant to this paper. Details of architecture, instruction set, and optical implementation can be found in [12].

The functional architecture of Opcom is shown in Fig.1. The Host provides an interface to the outside world and can be implemented using any micro- or mini-computer. The Array of Cells is a rectangular array of optical cells. Each cell, as shown in Fig.2, can be viewed as a NOR gate cascaded with a D flip-flop. Such a cell provides one bit storage and processing power and can be implemented by an optical bistable device [8,11]. The Interconnect Unit provides dynamically reconfigurable interconnections among the cells. It can be implemented by computer generated holograms or optical crossbar networks [6,10,11].

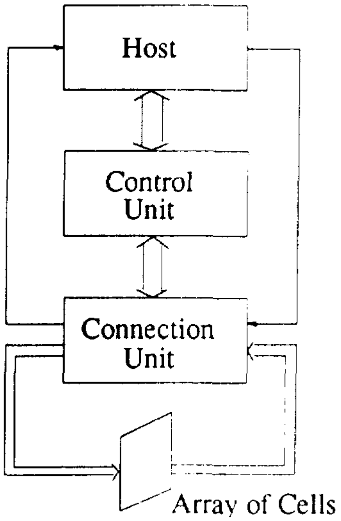
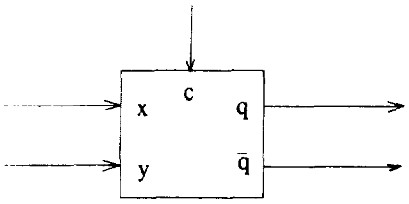


Figure 1. The Logical Arthitecture of Opcom

[1] This research was supported in part by an NSF grant DMC-84-21022, by an AFOSR grant 86-0008, and by a Henry Rutgers Research Fellowship.

CH2649-2/89/0000/0545$01.00 © 1988 IEEE



$$\text{new } q = \bar{c}q + c(x + y)$$
$$\text{new } \bar{q} = \bar{c}\bar{q} + c\overline{(x + y)}$$

Figure 2. The Functional Description of a Cell

The Control Unit executes instructions of Opcom. Only two instructions are needed in this paper. The execution of a "CONNECT $p$" instruction will reconfigure the Interconnect Unit to set up inter-cell connections as specified by $p$. A "START $n$" instruction will enable all cells involved for $n$ clock periods. There are no conventional data transfer or arithmetic/logic operations in Opcom. Instead, any computation is carried out in the following fashion: First, a optical circuit is constructed by executing a CONNECT instruction; then the execution of a START instruction propagates operand data through the circuit to generate the desired result data. Thus the basic operation mode of Opcom is massively parallel and pipelining at the gate (cell) level.

# 3 Implementation of Neural Networks in Opcom

Because there are many different definitions of neural networks and neurocomputing, we will define below what we mean by "connectionist style neural computations" [1,3,9]. A *neural network* consists of a number of *neurons* interconnected by unidirectional *synapses*. A network is *cyclic* (also called *fee˷back*) if it contains directed cycles, otherwise it is *acyclic* (also called *feedforward*). We will focus on feedforward networks in this paper, although the techniques presented can also be applied to feedback networks.

Each synapse from neuron $n_i$ to neuron $n_j$ has an integer ***weight*** $w_{ij}$. Each neuron $n_i$, as shown in Fig.3, has an *activation* (also called *state*) $a_i$, which can have values -1, 0, or 1. The activation $a_i$ is determined by the following equations:

$$S_i = \textstyle\sum_{j=1}^{n} w_{ij}a_j \qquad a_i = T(S_i) = \begin{cases} 1 & \text{if } S_i > 0 \\ 0 & \text{if } S_i = 0 \\ -1 & \text{if } S_i < 0 \end{cases}$$

A neuron can be implemented in Opcom using a bit slice circuit as shown in Fig.3. Using ones-complement binary number system, each $w_{ij}$ (and $S_i$) can be represented by an $r$-bit binary number $w_{ij}^r w_{ij}^{r-1} ... w_{ij}^1$ (and $S_i^r S_i^{r-1} ... S_i^1$), if the range of possible integers is from $-2^{r-1}$ to $2^{r-1}$. Each activation $a_j$ can be represented by two bits $a_j^2 a_j^1$, with 0=00, -1=10, and 1=01. A multiplication operation $w_{ij}a_j$ can be realized by a three-gate circuit in a bit-sliced fashion. In fact, the $k$-th bit of the product is equal to $w_{ij}^k a_j^1 + w_{ij}^k a_j^2$, which needs one OR gate and two AND gates. Consequently, a bit-sliced multiplier can be implemented as shown in Fig.4 using three optical cells of Fig.2.

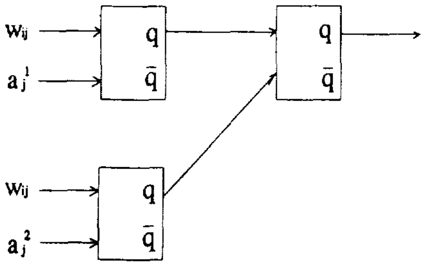


Figure 4. The Multiplication Circuit

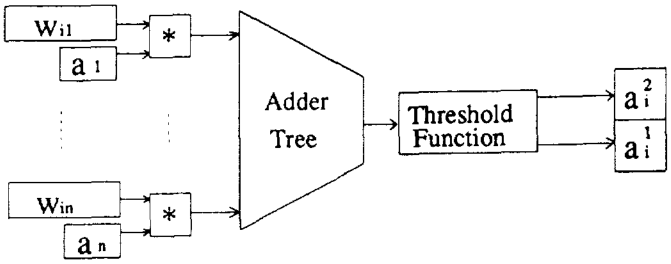


Figure 3. An Optical Circuit for Implementing a Neuron

In [12], we presented an integer adder that needs 11 optical cells. The $n$ products from the bit slice multipliers can be summerized using $n - 1$ adders organized as a ($log_2 n$)-stage adder-tree to produce $S_i$. The threshold function $T(S_i)$ can be implemented with 4 optical cells as shown in Fig.5. A simple calculation shows that a neuron with $n$ inputs can be implemented by $(r + 14)n - 5$ optical cells. A neural network with $m$ neurons can be implemented using $O((r + 14)mn)$ cells.

A basic function performed by any neural network is *search*. That is, given some (maybe partially specified) input, find an output that best matches the input. In a feedforward neural network, such an output can always be produced by propagating the input through the network. If the longest path in the network has length $L$, the propagation time will be $rL(4 + \log_2 n)$ clock periods, where $r$ is the word length of integer and $4 + \log_2 n$ is the number of stages of cells in Fig.3.

Due to the acyclic property, a feedforward network can perform $N$ searches in a pipelined fashion: First, the Control Unit executes a "CONNECT $p$" instruction, where $p$ is the interconnection pattern of all the $O((r + 14)mn$ optical cells involved. The Interconnect Unit is reconfigured to provide all inter-cell connections. Secondly, a "START $t$" instruction is executed, where $t$ has the value $4L + L\log_2 n + rN$. This instruction enables a search to be performed very $r$ clock periods.

## 4 Learning in Opcom

Another basic function of a neural network is *learning*. That is, we need to set the weight attached to each synapse by applying some training examples to the neural network. Many learning schemes have been developed in the past, including supervised procedures, reinforcement procedures, and unsupervised procedures. For a survey of various learning procedures, the reader is referred to [3].

In this paper, we show how the Opcom architecture can efficiently implement a supervised learning procedure, called the Pocket Algorithm [1], which has been successfully integrated into the design of connectionist expert systems. The basic idea of this algorithm is as follows: Initially, all weights are zero. A number of training examples are applied to each neuron in a network. For each training input $E$, the neuron's actual response is compared with the desired output $C$. If they do not match, the weights are updated by adding the product $C \cdot E$. For a detailed description of the Pocket Algorithm, the reader is referred to [1].

This learning procedure for each neuron can be implemented by $O((r+24)n)$ optical cells as illustrated in Fig.6. Because of the feedback connections in Fig.6, no pipelining is possible. That is, a training example can not be sent to the optical learning circuit until the previous training example has propagated through the entire circuit. This problem can be solved by using the pipeline networking techniques presented in [5]. We first insert some optical cells as noncompute delays into each cyclic path in Fig.6 to ensure that all cyclic paths have the same length $q$. Then we use the same circuit to carry out the learning procedures of $q$ neurons in a pipelined fashion. This is possible because in the Pocket Algorithm, neurons learn their weights independently. With this scheme, if there are $N$ training examples, a network of $m$ neurons can finish learning in $O(qrN)$ clock periods using $m/q$ optical learning circuits of Fig.6, where $r$ is the integer word length.

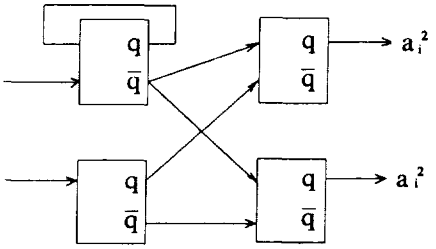


Figure 5. The Threshold Circuit

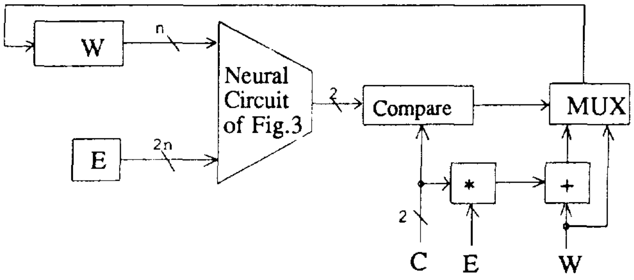


Figure 6. Sketch of an Optical Learning Circuit

## 5 Summary of Research Results

Three topics of research naturally arise in building a neurocomputer that efficiently supports search and learning: new instruction sets, innovative architectures, and proper technologies. In this paper, we present our solution to each of the three problems. The conventional data transfer and arithmetic/logic instructions found in a traditional sequential computer are not suitable for efficient implementation of neurocomputing. In contrast, our CONNECT instruction supports massive, reconfigurable interconnections and the START instruction enables massively parallel and pipeline operations. All these are properties essential in a neural network or a connectionist model.

The Opcom architecture also matches with a neural network model. There are no separate memory and CPU as in most traditional computers. Instead, Opcom can be viewed as a set of dynamically interconnected optical cells, each has one bit processing and storage capability. Note that a neural network is defined as a set of dynamically interconnected neurons, each has limited processing power (some threshold operation) and storage capability (state or activity). Advances in optical technology promise reconfigurable interconnections, massive parallelism, and gate level pipelining, which make the Opcom architecture feasible.

A traditional computer has millions or even billions of gates. However, most time only a very small portion of the hardware is involved in operation, while the major portion of hardware is idle. On the contrary, we have shown in this paper that due to the mode of massively parallel and pipeline operation at gate level, Opcom can have a much higher hardware utilization rate. This, coupled with the fast switching time of optical devices (in the order of nanoseconds), could lead to very impressive performance. In fact, using a single circuit of Fig.6, which has a gate count smaller than a microprocessor, a network of 1000 neurons could be implemented to learn 1000 training examples in less than a second.

## References

[1] S. I. Gallant. "Connectionist Expert Systems". *Comm. of ACM*, 31(2):152–169, 1988.

[2] R. Hecht-Nielsen. "Neurocomputing: Picking the Human Brain". *IEEE Spectrum*, 25(3):36–41, 1988.

[3] G. E. Hinton. "Connectionist Learning Procedures". 1987. Tech. Rept. CMU-CS-87-115, Carnegie Mellon University.

[4] J. J. Hopfield and D. W. Tank. "Computing with Neural Circuits: A Model". *Science*, 233(8):625–633, 1986.

[5] K. Hwang and Z. Xu. "Pipeline Nets for Compound Vector Supercomputing". *IEEE Trans. on Computers*, TC-37(1):, Jan. 1988.

[6] B. K. Jenkins, P. Chavel, R. Forchheimer, A. A. Sawchuk, and T. C. Strand. "Architectural Implications of a Digital Optical Processor". *Applied Optics*, 23(19):3465–3474, 1984.

[7] R. P. Lippman. "An Introduction to Computing with Neural Nets". *IEEE ASSP*,4–22, April 1987.

[8] N. Peyghambarian and H. M. Gibbs. "Optical Bistability for Optical signal Processing and Computing". *Optical Engineering*, 24(1):68–73, 1985.

[9] D. E. Rumelhart and J. L. McClelland (Eds.). *Parallel Distributed Processing: Explorations in the Microstructures of Cognition.* MIT Press, Cambridge, Mass., 1986.

[10] A. A. Sawchuk, B. K. Jenkins, C. S. Raghavendra, and A. Varma. "Optical Crossbar Networks". *IEEE Computer*, 388–392, Aug. 1987.

[11] A. R. Tanguary, Jr. "Materials Requirements for Optical Processing and Computing Devices". *Optical Engineering*, 24(1):2–18, 1985.

[12] Z. Xu, K. Hwang, and B. K. Jenkins. "Opcom: An Architecture for Optical Computing Based on Pipeline Networking". *Proc. of Twentieth Annual Hawaii Int'l. Conf. on System Sciences*, 1:147–156, Jan. 1987.

# SECTION IV: LANGUAGES

## Part 1: Oral Presentations

# Massively Parallel Data Optimization

Kathleen Knobe[*] Joan D. Lukas[†] Guy L. Steele Jr.[‡]

**Abstract**

An optimizing compiler for a data parallel programming language can significantly improve program performance on a massively parallel computing system by incorporating new strategies for allocating array elements to processors. We discuss techniques for automatic layout of arrays in a Fortran compiler supporting Fortran 8x array features and targeted to the Connection Machine computer system. Our goal is primarily to minimize the costs of moving data between processors and secondarily to minimize memory usage. Improved array layout may allow communications operations to be eliminated or to be replaced by more specialized communications operations having lower cost. We exhibit and discuss thoroughly a typical example of a code fragment that can be improved by a factor of 2 in memory consumption and a factor of 20 in speed.

## 1 Introduction

An optimizing compiler for a data parallel programming language can significantly improve program performance on a massively parallel computing system by incorporating new strategies for allocating array elements to processors.

We have designed and implemented a Fortran compiler for the Connection Machine® computer system at COMPASS on behalf of Thinking Machines Corporation. Work on improvements and extensions is currently in progress. The data optimizations described here are under design for incorporation into a future version of the compiler.

The language implemented is Fortran 77, extended by Fortran 8x array features, including some features (`FORALL` and vector-valued subscripts) now listed in the "removed extensions" section of the latest ANSI proposed standard for Fortran [3]. These features suit the data parallel computing style, which associates (at least metaphorically) one processor with each data element.

The Connection Machine system [6,12] supports the data parallel style by providing thousands of hardware processors that can operate on as many data elements simultaneously. A full Connection Machine system includes 65,536 physical processors, each with its own memory. Each processor can perform all of the usual arithmetic and logical operations on integers and floating-point numbers stored within its own memory. Parallel data structures are spread across the processors, with a single element stored in each processor's memory.

The Connection Machine is accessed through a front end, which provides the programming environment. The front end holds scalar data, and also controls execution of the data parallel program. Program steps involving parallel data are passed over an interface to the Connection Machine, where they are broadcast for simultaneous execution by all the processors.

Interprocessor communication is implemented by a high-speed routing network. In a `send` operation, each processor contains data to be sent and a pointer to a processor (possibly itself) that is to receive it. In a `get` operation, each processor contains a pointer to a processor containing data to be copied back to the first processor. A `send` is faster than a `get`, but a `get` allows data from one processor to be copied to many other processors at once.

The Connection Machine Model CM-2 singles out certain patterns of communication for special hardware and microcode support. Cartesian grids of any number of dimensions can be embedded within the boolean hypercube structure used by the router. Array elements that are neighbors along any dimension are allocated to processors that are neighbors within the hypercube structure. As a result, a single-position shift along any axis of such a grid can be performed much faster than the general case of the `send` instruction. Such operations are called NEWS operations (for North-East-West-South); the simplest one is called simply `get-from-news`.

Complex operations on Cartesian grids are also directly supported as single Connection Machine instructions. These include `spread` and `scan`. A `spread` operation can take any row of a matrix and copy it into all the other rows. A `scan` operation takes a combining operation (such as `add`, `max`, or `logior`) and performs a parallel prefix computation [9,7] on each row of a matrix. For example, `scan-with-add` computes running totals for each row. Both `spread` and `scan` generalize to columns instead of rows and to any number of dimensions.

As a rough rule of thumb, if a 32-bit addition (to be performed by all 65,536 processors) takes 1 time unit then

- a 32-bit NEWS transfer takes 4 units
- a `spread` takes 25 units
- a `send` takes 80 units
- a `get` takes 180 units

One unit is approximately 16 microseconds. These figures are only useful rough estimates; actual times depend to some extent on the virtual processor configuration.

Our goal is to minimize the cost of interprocessor communication. Appropriate allocation of data to processors frequently

[*] Compass, Inc. (formerly Massachusetts Computer Associates, Inc.)

[†] University of Massachusetts at Boston and Compass, Inc.

[‡] Thinking Machines Corporation

CH2649-2/89/0000/0551$01.00 © 1988 IEEE

allows a general communication operation (such as send) to be replaced by a faster one (such as get-from-news) or eliminated altogether.

## 2 Examples

As an example of the optimization issues addressed here, consider this code fragment:

```
10    B(1:5,1:N) = B(1:5,1:N) * S
20    L11(1:N) = 1.0 / B(1,1:N)
30    U2(1:N) = B(2,1:N) * L11(1:N)
40    U3(1:N) = B(3,1:N) * L11(1:N)
50    U4(1:N) = B(4,1:N) * L11(1:N)
60    U5(1:N) = B(5,1:N) * L11(1:N)
```

(This fragment is loosely based on a fragment of a vectorized tridiagonal solver subroutine. For emphasis we have written out all subscripts explicitly.)

The simplest and most naïve way to arrange the data is to lay out each array with one element per processor, starting at processor zero. See figure 1. This is the "canonical" layout used by the compiler in the absence of other criteria. This places L11 and each U in the same set of processors so they align, but B is spread across the processors in a manner that does not necessarily bear any useful relationship to the other arrays. Alignment of L11 and each U with the sections of B requires motion.

This strategy requires a total of 8 memory slices: one for B, one for each U, one for L11, and a temporary used to hold a moved copy of L11 or B($i$,1:N) in each of statements 30–60. The total communication cost depends on which of two strategies is used for statements 30–60. If L11 is moved to align with B($i$,1:N), and then the result of the multiplication is moved to U$i$, a total of 9 send operations will be required (1 for statement 20, and 2 each for statements 30–60). If B($i$,1:N) is moved to align with L11 and U$i$, then only 5 send operations are required in all. The time for motion will thus be either 720 or 400 units.

In summary, the total cost of lines 10–60 for the naïve allocation strategy is

| | | |
|---|---|---|
| Total time for operations | = | 6 |
| Total time for motion | = | 720 or 400 |
| Total number of memory slices | = | 7 |

An obvious hypothesis is that aligning the one-dimensional arrays with some row of B, say the first, would reduce communication (figure 2). This eliminates one communication step (that between L11 and B(1,1:N)), but is not the best possible improvement. Observe that each send operation can be replaced by a sequence of NEWS moves, copying values from row to row of B, giving

| | | |
|---|---|---|
| Total time for operations | = | 6 |
| Total time for motion | = | 80 or 40 |
| Total number of memory slices | = | 7 |

In this particular case it might be better to lay out the first dimension of the array B as a serial axis rather than a parallel axis; that is, successive elements of B along that dimension will occupy successive locations within the same processor rather than the same location within different processors. See figure 3. In this manner no communication is required for any of the assignment statements, and no temporary locations are required. On the other hand, the multiplication of B by S in line 10 now requires 5 separate computational steps. The cost of lines 10–60 is

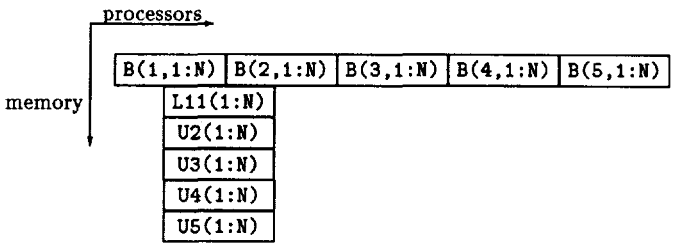


Figure 1: Naïve Layout of Some Arrays

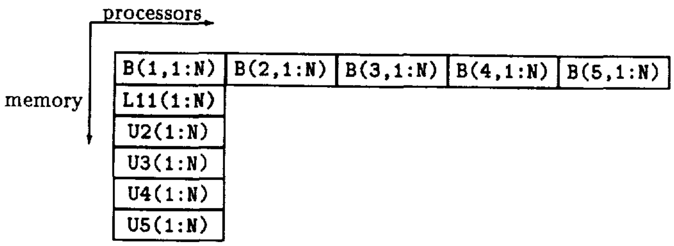


Figure 2: Aligned Layout of Some Arrays

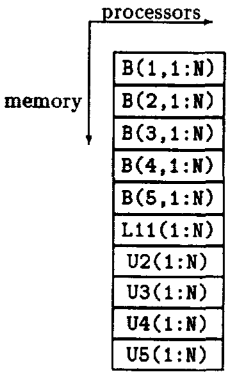


Figure 3: Sequential Layout of Some Arrays

| | | |
|---|---|---|
| Total time for operations | = | 10 |
| Total time for motion | = | 0 |
| Total number of memory slices | = | 10 |

Another possibility is the layout shown in figure 4. Here B is laid out in parallel and each U is aligned with the appropriate section of B. Copies of L11 are still required, so one extra memory slice is needed for a temporary.

| | | |
|---|---|---|
| Total time for operations | = | 6 |
| Total time for motion | = | 40 or 80 |
| Total number of memory slices | = | 7 |

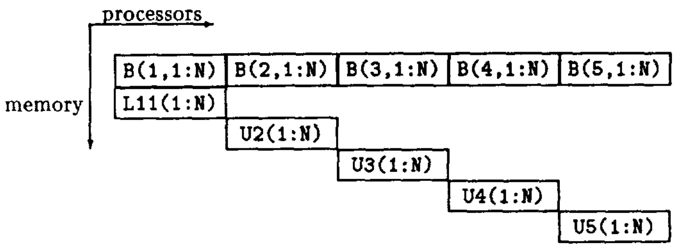


Figure 4: Skewed Layout of Some Arrays

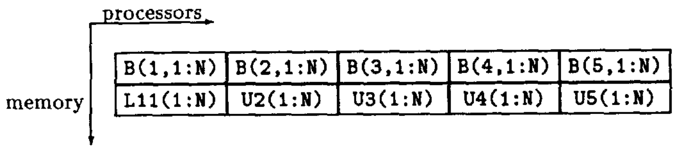


Figure 5: Compressed Layout of Some Arrays

This layout has the same time and space possibilities as in figure 2, with the NEWS costs of the two motion strategies reversed. But this layout leads to an even better one.

Note that the U arrays are now in disjoint processor sets and can share a single slice of memory (figure 5). Then the four multiplications can be collapsed into a single multiplication to be carried out for all of U2, U3, U4, and U5 simultaneously, resulting in

| | | |
|---|---|---|
| Total time for operations | = | 3 |
| Total time for motion | = | 40 or 80 |
| Total number of memory slices | = | 3 |

This example illustrates a need to optimize the *static* layout of user-declared arrays. Sometimes *dynamic* layout is required, as in the following example.

```
      DO 10 J=1,M
      TEMP(1:N) = A(J,1:N)
      A(J,1:N) = B(J,1:N)
      B(J,1:N) = TEMP(1:N)
   10 CONTINUE
```

In this case it is desirable not to assign TEMP to one fixed set of locations. Its uses on different iterations do not interact (there is no dependence among them), so TEMP may be allocated to a different set of operations on each iteration. In other words, the compiler can in effect rewrite the code to behave something like this

```
      DO 10 J=1,M
      TEMP(J,1:N) = A(J,1:N)
      A(J,1:N) = B(J,1:N)
      B(J,1:N) = TEMP(J,1:N)
   10 CONTINUE
```

## 3 Overview of Solution

Data optimization processing takes place after the Internal Representation (IR) of the source code has been created to expose all source level computations and after this IR has been optimized. IR requiring exact locations, such as for data motion or context setting, has not yet been exposed. The algorithms take machine-specific time and space cost functions as parameters and can therefore be applied to a variety of machines by using different cost functions.

The first phase of data optimization is the Align phase, which determines preferred relative alignments among occurrences of named array sections based on dependences. Resolving conflicts among these preferences indicates a need for data motion. The Inter phase determines the best location for intermediate results, tracking multiple copies of array sections produced by conflict resolution in Align and by Inter's own processing of previously analyzed statements and subexpressions.

When Align is unable to allocate sections according to their preferred relative alignment, motion is required. The Motion phase determines what data is to be moved, which motion instructions are needed and their initial placement in the generated code. A Mini-Vectorizer is then activated to perform vectorization-like transformations on the code introduced by previous phases. The exact allocation of array occurrences is now determined and code is exposed in the IR for context setting and motion. Standard global optimization is then performed on the newly added code.

The major compiler phases that are unique to data optimization are Align, Inter, Motion, and the Mini-Vectorizer. These are discussed in more detail in the remainder of this paper.

## 4 Align

The align phase determines the allocation of occurrences of named arrays and their sections. Align analyzes usage patterns in the source, noticing situations where the best allocation of one array occurrence depends on the allocation of another array occurrence. We call these situations *preferences.*

An **Identity Preference** is between corresponding dimensions of a definition and use of the same array and indicates a preference to align identical elements of the array in the same processors for the two occurrences.

A **Conformance Preference** is between corresponding dimensions of occurrences of different arrays that are operated on together and indicates a preference to align corresponding elements of distinct arrays in the same processors.

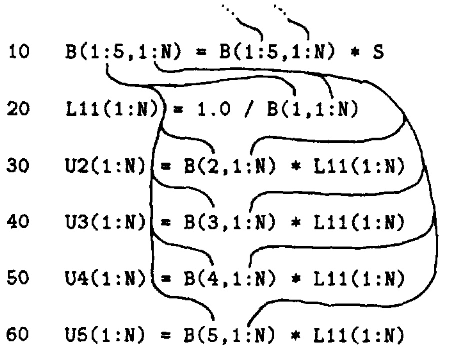
An **Independence Anti-Preference** is associated with a single dimension of an array occurrence (not a relationship between them) and indicates a preference to store the distinct elements along that dimension in different processors.

The three characteristics above refer to allocation across processors. There is, of course, a **Uniqueness Requirement** that each dimension must have a unique effect on the storage of the array elements over the entire memory space of the machine but this effect could well be with respect to the storage within processor memory. Therefore, even with this requirement, the value in a particular dimension may legitimately have no effect on the allocation of an array across processors.

These preferences are discussed below with their implications for the Connection Machine. Some implications for other architectures are presented as well.

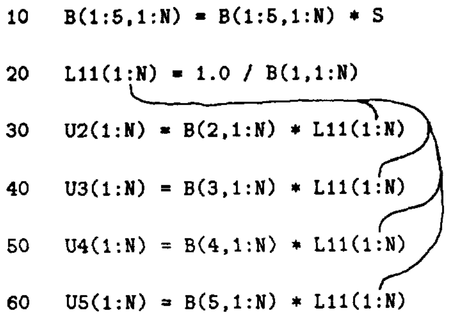
**Identity Preference:** This preference is between corresponding dimensions of a definition and a use of the same array if there is a true dependence between them. Note that the allocation of a definition and use of the same array may be determined independently if there is no true dependence between them. (Also note that output dependences and anti-dependences have no impact on allocation.)

The identity preferences in our example are shown below



```
10   B(1:5,1:N) = B(1:5,1:N) * S
20   L11(1:N) = 1.0 / B(1,1:N)
30   U2(1:N) = B(2,1:N) * L11(1:N)
40   U3(1:N) = B(3,1:N) * L11(1:N)
50   U4(1:N) = B(4,1:N) * L11(1:N)
60   U5(1:N) = B(5,1:N) * L11(1:N)
```

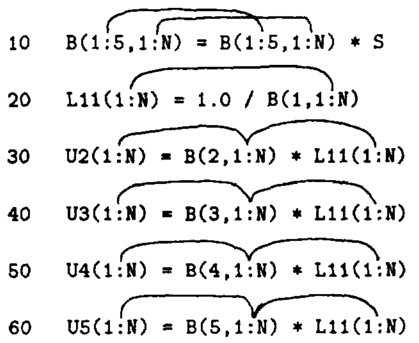
and



```
10   B(1:5,1:N) = B(1:5,1:N) * S
20   L11(1:N) = 1.0 / B(1,1:N)
30   U2(1:N) = B(2,1:N) * L11(1:N)
40   U3(1:N) = B(3,1:N) * L11(1:N)
50   U4(1:N) = B(4,1:N) * L11(1:N)
60   U5(1:N) = B(5,1:N) * L11(1:N)
```

On the Connection Machine, an identity preference means that if the allocation of the array at the definition is not identical to the allocation at the use, motion will be required. On a MIMD machine this preference might be used when partitioning an application among the processors. On a MIMD machine an identity preference means that if the definition and the use are not executed on the same processor, there is a cost for synchronization, and, if the MIMD processors have local rather than shared memory, there is also a cost for data motion.

**Conformance Preference:** This preference is between corresponding dimensions of two arrays that are operated on together. The conformance preferences in our example follow:



```
10   B(1:5,1:N) = B(1:5,1:N) * S
20   L11(1:N) = 1.0 / B(1,1:N)
30   U2(1:N) = B(2,1:N) * L11(1:N)
40   U3(1:N) = B(3,1:N) * L11(1:N)
50   U4(1:N) = B(4,1:N) * L11(1:N)
60   U5(1:N) = B(5,1:N) * L11(1:N)
```

In the assignment to U2 there is a conformance preference between the second dimension of B and the only dimension of L11. If these two dimensions are not allocated identically, motion to align will be required. *Restrictions* may be associated with each conformance preference. For example, the conformance preference in the statement 30 holds only when the first subscript of B is 2. Statement 40 has the same preference between the second dimension of B and L11, but the restriction holds when the first subscript of B is 3.

On the Connection Machine system, motion will be minimized if dimensions associated by conformance preferences are allocated so that corresponding elements align. On a machine with memory bank contention, a conformance preference means that delays will be minimized if we allocate these two sections or at least the corresponding elements of the sections in distinct memory banks. On a MIMD machine with local memory, a conformance preference means that the two sections or at least the pairs of elements determined by this correspondence must reside in the same processor to minimize data motion.

**Independence Anti-Preference:** Both the identity preference and the conformance preference attempt to reduce motion. This anti-preference attempts to preserve maximum parallelism. Without this preference, the other preferences can always be resolved by storing large sections or even entire arrays in a single processor. While such a resolution will result in no data motion, it may reduce parallelism and increase memory requirements. (In our example, loss of parallelism occurs if we break the independence anti-preference on the first dimension of B and store this dimension serially within the processor memory, as shown in figure 3. With this allocation, a single operation on all of B must be transformed into five operations, one on each row of B.) The independence anti-preference is a characteristic of a specific dimension occurrence if that dimension contains a potentially parallel subscript, that is if the subscript is not scalar. The independence anti-preference is included so the algorithm will not always decide to minimize motion at the expense of memory and parallelism but will make an explicit choice based on cost by associating a cost with each independence anti-preference.

The relationships described above are represented as arcs in a graph whose nodes are dimensions of occurrences of arrays and array sections. Such a graph can be *consistent* or *in conflict*. It is consistent if all the preferences can be satisfied. It is in conflict if preferences contradict each other. In the above example the preference graph contains a conflict. The second dimension of B must align with L11 where the first subscript of B is 1 and also where it is 2. If the first dimension of B is stored across the processors, the conformance preferences cause a conflict. If the first dimension is stored down processor memory, the independence preference causes a conflict. The graph is therefore in conflict.

If the preference graph is consistent, no motion need occur at runtime. Conflict resolution involves either the introduction of motion or, if a dimension is to be stored down the processor memory, additional memory requirements and possibly loss of parallelism. When a conflict is broken, a preference arc is removed from the preference graph. Removal of a conformance or identity arc implies the necessity of data motion. To enable the Motion phase to insert this motion, Align records information about the removed arcs. Removal of an independence arc results in increased storage requirements per processor and possible reduction in parallelism, but no data motion is required.

Align processing begins by building the preference graph for each basic block. If the graph is in conflict, this will involve conflict resolution. Sometimes resolving a conflict can be accomplished in several ways which may differ significantly in their cost. Costs are associated with the arcs in the graph and Align attempts to find the least cost resolution. Costs may include the cost of motion, the cost of memory, and the cost of lost parallelism.

When the block is made consistent, Align processing then proceeds up the control tree, a hierarchical representation of the control flow of the program with basic blocks at the leaves. At each stage in a bottom-up control tree walk, the consistent graphs for the subnodes of a control node are combined by taking their union and then including arcs representing the relationships that cross between the subnodes. When joining control nodes, new identity preferences may be introduced. Conformance preferences and independence anti-preferences occur strictly within control nodes. Conformance preferences arise only within statements and therefore no additional conformance preferences can be introduced by joining the preference graphs of the control nodes on the way up the control tree.

Since the subgraphs (graphs for the children of the control node) are known to be conflict free, conflict resolution is performed incrementally by processing only the new identity preference arcs added between the subgraphs.

Since small local graphs are processed and then incrementally combined, the processing time is significantly faster than that of an equivalent global algorithm. Since it starts at the leaves of the control tree and works outward, the algorithm gives priority to preferences arising in inner loops.

The conflict in our example may be broken in a number of ways. Two are indicated below:

1. Store the first dimension of the B array down the processors so that one copy of L11 can align with all the appropriate sections of B (figure 3). This resolution breaks the independence preference.
2. Store B across the processors and align L11 with B(1,1:N) and each of U2, U3, U4, and U5 with the corresponding section of B (figure 4). This removes the conformance preference between L11 and the sections of B.

Option 1 uses more memory but incurs no loss of parallelism and results in *no* motion for execution of lines 10–60. This option results in

| | | |
|---|---|---|
| Total time for operations | = | 10 |
| Total time for motion | = | 0 |
| Total number of memory slices | = | 10 |

This may be the optimal result. On the other hand, the advantages of this option may well be offset by the necessity of either motion to serialize B from its parallel definition or executing the assignment defining B once for each value of the subscript in dimension 1. In any case, adoption of this strategy will be unenlightening for the rest of the paper. Option 2 will be assumed. The effect of this decision will be that for each statement the LHS and the B section are now aligned. Motion will be required to align these with L11.

There are a number of possibilities for this motion. At a minimum, L11 will be moved to align with each LHS at a cost of 1, 2, 3, and 4 NEWS moves, respectively. The total motion time is ten times the unit cost of a NEWS move. At worst, each B section will be moved to align with L11, the operation performed there, and the result moved back to align with the LHS. This would require a total of 20 NEWS moves. This option results in

| | | |
|---|---|---|
| Total time for operations | = | 6 |
| Total time for motion | = | 40 or 80 |
| Total number of memory slices | = | 7 |

# 5 Inter

Once allocation of occurrences of named arrays has been determined, the compiler must deal with the proper allocation of array temporaries arising from operations, that is, it must decide where the operations will be performed. If Align was able to satisfy all preferences, then all operands in a statement as well as the destination of an assignment will have the proper alignment. In this situation the optimal location for each operation will be obvious. In many cases, however, some conflicts will have arisen, causing Align to break preference arcs. If the operands are not aligned, two or more possible locations for temporaries will exist.

The Inter phase, which chooses optimal locations for array temporaries, uses a dynamic programming algorithm over an expression tree. The locations considered are those of the named sections in the same statement. For each temporary, Inter considers each of these locations along with the cost of moving operands to each location to align them for operations. Let $M(L_1, L_2)$ denote the cost of moving an array from location $L_1$ to location $L_2$. This cost is determined by the Connection Machine instruction(s) needed to effect the move. Then, for each operation $\oplus$ and each location $L$, the cost of performing $x \oplus y$ in $L$, denoted $C(x \oplus y)$, is the minimum cost, over all considered pairs of locations $L_1$ and $L_2$, of computing $x$ in $L_1$ and then moving it to $L$ added to the cost of computing $y$ in $L_2$ and moving it to $L$. That is,

$$C(x \oplus y, L) = \min_{L_1, L_2}\Big(C(x, L_1) + M(L_1, L) + C(y, L_2) + M(L_2, L)\Big)$$

For an assignment statement $x = y$ where $x$ is in location $L_x$, the cost is given by

$$C(x = y) = \min_{L}\Big(C(y, L) + M(L, L_x)\Big)$$

The statement level cost equation drives the computation for interior nodes of the right-hand side $y$.

Note that when an operand is moved (copied) from one location to another, an additional copy of it is available for use in subsequent statements. Inter keeps track of these multiple copies in determining minimum costs. The algorithm for tracking copies uses a mechanism similar to that of value numbering. A copy remains valid until an assignment to any of the elements in the original array section is made. Tracking of multiple copies can be done across control flow as well as within basic blocks.

To return to our example, let us assume that Align has decided to allocate B across the processors and to align L11 with B(1,1:N) in location $L_1$ and each U*i* with the appropriate section of B in location $L_i$. Then statement 20 requires no alignment.

The operation will take place in $L_1$. For statement 30, the cost is given by

$$\min_{L}\Big(C(\texttt{B(2,1:N) * L11(1:N)}, L) + M(L, L_2)\Big)$$

The locations to consider for $L$ are $L_1$ (holding L11) and $L_2$ (holding U2 and B(2,1:N)). For $L = L_1$, the cost is

$$C(\texttt{B(2,1:N) * L11(1:N)}, L_1) + M(L_1, L_2)$$

To perform the multiplication in $L_1$ would require moving B(2,1:N) to $L_1$, at a cost of one NEWS move. The total cost of the assignment would then be 2 NEWS moves. For $L = L_2$, the cost of performing the operation in $L$ is one NEWS move and the cost of moving the result to $L_2$ is 0. Thus the total cost is one NEWS move. This option is clearly preferable.

As a result of the choice just made, there will now be two copies of L11, the original in $L_1$ and a new one in $L_2$. For statement 40, there are now three possible locations: the two locations holding L11, and also $L_3$, which holds U3 and B(3,1:N). An analysis similar to the one performed above will determine that the lowest cost will be obtained by performing the operation in $L_3$. In order to do this, L11 will be copied to $L_3$. The cost of moving L11 from its original location to $L_3$ is 2 NEWS moves while the cost of moving from the copy at $L_2$ is 1 NEWS move. Thus this copy will be used.

As the algorithm continues for the remaining statements, each multiplication will be carried out in $L_i$, with the most recent copy of L11 at $L_{i-1}$ being copied at a cost of 1 NEWS move.

At the end of the Align phase, the cost of our example depended on an arbitrary choice of locations for the intermediate operands. Inter assures the best choice and, in fact, does better than this by tracking multiple copies.

| | | |
|---|---|---|
| Total time for operations | = | 6 |
| Total time for motion | = | 16 |
| Total number of memory slices | = | 7 |

Although it is not relevant to our example, we describe an additional optimization performed by Inter. The cost of computation for a complex arithmetic expression may sometimes be reduced if reordering of operations is permitted. When such reordering is permitted, a set of adjacent addition and subtraction nodes can be viewed as an $n$-ary addition, with some operands preceded by a unary minus. The same is true for a sequence of multiplication and division operations. Inter will exploit this fact by regrouping operands to minimize the cost of motion among them. Operands in the same location are combined before joining them with other operands. For example, if statement 30 in our example had instead been

```
30   U2(1:N) = B(1,1:N) * B(2,1:N) * L11
```

then L11 would first be multiplied by B(1,1:N) in location $L_1$. The result would then be copied to $L_2$ for multiplication by B(2,1:N) and assignment to U2.

## 6 Motion

After Align and Inter have determined where each occurrence of an array or section will reside, the Motion phase determines which references require motion, the position of the motion code in the IR, the type of motion required and the amount of data to be moved.

Motion is required:

- when Align had an identity or conformance preference that it had to break (breaking an independence anti-preference does not require motion)
- when Inter decides to move a named section
- when Inter decides to move an intermediate result

Motion will attempt to collect motions in a basic block to the same place within the block so that they can be combined to minimize cost. For example, two disjoint sets of values may be more cheaply sent together than separately. Also one motion may totally subsume another. If this is the case, only the larger need be performed. An identity preference may exist because data independence of the subscripts cannot be proved at compile time. In such cases, Motion may insert a scalar test, which will be executed at runtime and possibly save the cost of motion.

In our example, the motions of L11 is of the right form to be converted to a **spread** but it is not moved far enough for conversion to be cost effective. For this example Motion simply inserts motion code after the definition of L11, having no additional effect on the costs.

## 7 Mini-Vectorizer

Upon completion of the previous phases, a number of opportunities for vectorizer-like transformations that further improve performance are uncovered and are handled by this phase. These transformations are with respect to motion code and therefore cannot be performed until allocation and motion have been determined. The following transformations are processed by this phase.

- Motion is associated with a particular loop in a loop nest. If loops are interchanged the total cost of the motion may be affected, sometimes significantly.
- If a loop itself is not vectorizable and a series of definitions or uses of individual elements in an array remain, motion between the front end and the Connection Machine is required. The direction of the motion depends on whether the references are uses or definitions. The mini-vectorizer may be able to transform the motion from elemental motion within a loop to array motion outside the loop even if the operations are not vectorizable.
- If all arrays are stored in locations determined by their shapes, the only factor determining when arrays can share a slice of memory is their lifetimes. With the allocation scheme based on preferences, even if two arrays are live at the same time, they can be allocated to the same memory address if they share no processors.
- The same operation performed on distinct sections of the same array or on different arrays that do not share processors can sometimes be converted to a single operation on all sections at once.

The last case above occurs in our example. The four sections of B are being multiplied by different copies of L11. In order to convert this to a single multiply, we must prove that

- The four locations (set of processors) for the operations are distinct. (They may, however, have array elements in common such as L11.)
- No two LHSs may contain the same array element.
- The same operation is used on each section.

We allocate a single memory address for all copies of L11, a single memory address for U2, U3, U4, and U5, and perform a single multiplication. This transformation saves seven slices of memory and three multiplications.

The algorithms described above have produced two possibilities for this code sequence, with costs of

| | | |
|---|---|---|
| Total time for operations | = | 10 |
| Total time for motion | = | 0 |
| Total number of memory slices | = | 10 |

and

| | | |
|---|---|---|
| Total time for operations | = | 3 |
| Total time for motion | = | 16 |
| Total number of memory slices | = | 3 |

Recall that the numbers for the naïve allocation strategy were

| | | |
|---|---|---|
| Total time for operations | = | 6 |
| Total time for motion | = | 720 or 400 |
| Total number of memory slices | = | 7 |

On this fragment data layout optimization can either

- improve memory usage by a factor of 2 and execution speed by a factor of 20 or more or
- increase memory requirements and time for arithmetic operations while eliminating all data motion.

## 8 Generalized Common Subexpressions

Generalized common subexpression elimination is an optimization performed by the Inter phase. It is not relevant to our example and therefore was not presented above, however, it can have a significant effect on the cost of motion for for a number of applications and on a variety of machines and is therefore included here.

If a SIMD operation is performed on a section or sections and subsequently the same operation is performed on different sections of the same array(s), the two operations can be combined. This is not exactly a common subexpression because the sections may not be the same, but, except for the context (the set of enabled processors), the operations are, in fact, identical. By adjusting the context, the expressions can be combined. The operations involved may be either local per processor computations or they may involve motion operations. Finding common motion operations is the more interesting (and cost effective) case.

Consider the following assignment:

```
A(2:N,2:N) = A(2:N,2:N) + A(1:N-1,2:N)
&              + A(2:N,1:N-1) + A(1:N-1,1:N-1)
```

Neglecting boundary conditions, the nature of this assignment may be captured in the following schematic form:

```
A = A + west(A) + north(A) + northwest(A)
```

By rewriting `northwest(A)` as `north(west(A))` and then noting that `north` distributes over `+`, we get

```
A = (A + west(A)) + north(A + west(A))
```

Viewed this way the two occurrences of `A + west(A)` can be thought of as a kind of common subexpression, but our informal notation, because it glosses over the boundary conditions, hides the fact that different sets of values are involved. If we cast this back into correct Fortran (using a temporary variable because Fortran 8x arbitrarily does not permit one to subscript an expression), we find that they are not quite common subexpressions after all:

```
TEMP = A(2:N,2:N) + A(1:N-1,2:N)
A(2:N,2:N) = TEMP + TEMP(?,?)       !wrong
```

A solution of this general form will work, but only if the common subexpression is *generalized* to contain the *union* of the set of values required by all uses. Thus TEMP will contain more values than either of its uses requires:

```
TEMP = A(2:N,1:N) + A(1:N-1,1:N)
A(2:N,2:N) = TEMP(1:N,2:N) + TEMP(1:N,1:N-1)
```

This is worthwhile when the cost of computing the union of two (or more) sets of values will be cheaper than computing each set separately. This frequently occurs in codes where various shifted sections of a single array are to be combined.

To notice opportunities for this optimization we need to remove attention from the specific elements being operated on and focus instead on the array names of the operands and on their offsets. Specifically, $A \oplus B$ and $A' \oplus B'$ are general common subexpressions if

- $A$ and $A'$ are sections of some array, and $B$ and $B'$ are sections of some array
- the relationship between the corresponding subscripts of $A$ and $A'$ is the same as the relationship between the corresponding subscripts of $B$ and $B'$
- flow analysis indicates that the arrays involved have not been altered in the interim

If the operation is a motion operation as opposed to a local computation, then $B$ and $B'$ may be viewed as the target set of processors.

## 9 Related Work

There is a great deal of other literature on compiling Fortran for parallel computers. Much of this work addresses techniques for extracting parallelism from programs that may have been written in a sequential style, for example [2,5,8]. We use some of these parallelizing techniques in our "mini-vectorizer" phase, but regard them as complementary rather than central to our work.

Allen and Kennedy [2] further point out that appropriate program transformations may not only improve opportunities for parallelism but reduce synchronization overhead. This happens not to apply to our particular practical problem, because the SIMD nature of the Connection Machine architecture obviates the need for explicit synchronization, but might well be relevant to an adaptation of our data layout techniques to an architecture with asynchronous parallelism.

We pursue questions of data structure transformation rather than program transformation. Our main goal is to determine data layouts that minimize communications overhead in architectures with non-shared memory, once opportunities for parallelism have already been made explicit. The IVTRAN compiler [10,11] addressed similar concerns of array layout; our work differs in considering a richer space of layout possibilities and in handling a larger set of communications primitives spanning a spectrum of functionality/speed tradeoffs.

Crystal [4] also deals with questions of data layout. It accepts and transforms programs in a language based on recursion equations, and targets systolic architectures. Crystal also emphasizes the detection of common subexpressions so that results computed in one processor may be shared with other processors. However, Crystal apparently finds only standard cse's that are costly. The notion of generalized common subexpressions covers more cases.

## 10 Summary and Conclusions

In an earlier paper [1] on the Connection Machine Fortran compiler, it was demonstrated that the Connection Machine is a natural target for compilation of Fortran 8x array constructs. However, careful attention to allocation of data across processors according to usage patterns is necessary in order to exploit the full benefits of the Connection Machine architecture. Naïve allocation strategies give rise to the need for large amounts of expensive motion code in order to align data for parallel operations.

The compilation approach presented in this paper performs analysis of usage patterns and determines the allocation strategy for each occurrence of an array section. The potential performance impact of this compilation technology is measured in orders of magnitude rather than percentages. The problem as well as the solution apply to any SIMD machine with local memory. Aspects of the analysis apply to a variety of architectures where the presence of data in the right place at the right time can significantly improve performance.

## References

[1] Albert, Eugene, Knobe, Kathleen, Lukas, Joan D., and Steele, Guy L., Jr. Compiling Fortran 8x array features for the connection machine computer system. In *Symposium on Parallel Programming: Experience with Applications, Languages and Systems.* ACM SIGPLAN (Jul 1988).

[2] Allen, John R., and Kennedy, Ken. Automatic loop interchange. In *Proceedings of the ACM SIGPLAN 84 Symposium on Compiler Construction.* Association for Computing Machinery (June 1984), 233–246.

[3] *Draft Proposed Revised American National Standard Programming Language Fortran,* ANSI X3.9-198x edition. American National Standards Institute, Inc. (Washington, D. C., 1987).

[4] Chen, Marina C. A parallel language and its compilation to multiprocessor machines or VLSI. In *Proceedings of the Thirteenth Symposium on Principles of Programming Languages.* Association for Computing Machinery (St. Petersburg Beach, Florida, January 1986), 131–139.

[5] Gottlieb, R., Kimball, K., Jaskiewicz, T., and Swift, R. A new way to speed up a supercomputer. *Electronics 58,* 30 (July 1985), 56–58.

[6] Hillis, W. Daniel. *The Connection Machine.* MIT Press (Cambridge, Massachusetts, 1985).

[7] Hillis, W. Daniel, and Steele, Guy L., Jr. Data parallel algorithms. *Communications of the ACM 29,* 12 (December 1986), 1170–1183.

[8] Irigoin, F., and Triolet, R. Supernode partitioning. In *Proceedings of the Fifteenth Symposium on Principles of Programming Languages.* Association for Computing Machinery (San Diego, January 1988), 319–329.

[9] Kruskal, C. P., Rudolph, L., and Snir, M. The power of parallel prefix. *IEEE Transactions on Computers C-34,* 10 (October 1985), 965–968.

[10] Millstein, Robert E. Control structures in Illiac IV Fortran. *Communications of the ACM 16,* 10 (October 1973), 621–627.

[11] Presberg, David L., and Johnson, Neil W. The Paralyzer: IVTRAN's parallelism analyzer and synthesizer. *ACM SIGPLAN Notices 10,* 3 (March 1975), 9–16.

[12] *Connection Machine Model CM-2 Technical Summary.* Thinking Machines Corporation (Cambridge, Massachusetts, April 1987).

# Testing Shared-Memory Parallel Programs

Andrew H. Sung

Computer Science Department
New Mexico Tech
Socorro, NM 87801

## Abstract

The problem of testing shared-memory, synchronous parallel programs is addressed. We use a simple, Pascal-based general purpose parallel language for SIMD machine programming and investigate the testing problem in this context. A scheme for classifying PRAM programming errors is proposed. Several approaches of designing testing coverage criteria for PRAM programs are presented. Various testing coverage metrics are defined and their properties are discussed. Also presented is a complete parallel program testing methodology which incorporates test generation tools for serial programs. This methodology allows easy implementation of testing procedures for parallel programs.

Keywords: Software Testing, Testing Coverage Criteria, Parallel Programming, SIMD, Shared-Memory Machines, PRAM, Programming Errors.

## 1. Introduction

With the increasing availability of parallel computers and supercomputers, the timely development of methodologies for building and validating parallel software is of crucial importance. Clearly, the tremendous computing power offered by parallel machines can be fully utilized only if we can effectively implement and validate the software for such machines. However, while much has been done on programming systems and environments for parallel computers and supercomputers, little research has focused on validation techniques for parallel programs.

The software validation problem for parallel machines is harder than and different from that for supercomputers. For vectorized/ concurrentized programs executed on supercomputers, validation is easier as we can generally assume a correct vectorizing/optimizing compiler and validate the vectorized code by validating the serial source code or to choose already validated serial code to begin vectorization with. Except for possible round-off errors (due to the fact that vectorization and concurrentization of serial code might lead to different round-off error accumulation and therefore give different answers), this simple approach should be sufficient given that the serial software validation techniques are fairly well established and have been used extensively.

For parallel programs which are executed on machines containing a large number of processors, the validation problem is more difficult due to the lack of obvious approaches. In this paper we address the issues involved in the validation of parallel programs for shared-memory SIMD machines. Specifically, we investigate the problem of testing programs for the most powerful model of SIMD parallel computers, the shared-memory, Parallel Random Access Machine or PRAM [2,15]. Our aim is to take an approach general enough to be applicable to a variety of high-level programming methods for SIMD machines. PRAM is chosen because it is a general model for developing and analyzing synchronous parallel programs [2,15].

We propose a scheme for classifying errors. Two aspects of PRAM programming, processor activation (the sequence of sets of processors activated—PAS) and processor coordination (the sequence of read/write/computation activities of active processor—PCS) are formally described, and PRAM programming errors are characterized in terms of them. Thus, PRAM programming errors are classified as either ***processor activation errors***, ***processor coordination errors***, or ***computation errors***. This error classification scheme refines the widely used concept of computation and domain errors for serial programs and provides a framework for analyzing parallel programming errors.

We then propose a hierarchy of testing coverage criteria for PRAM programs based on input-driven structures like processor activation, processor coordination, read-write sequence, etc. Coverage criteria based on flowgraphs and dataflow information are also discussed. The relative strengths of these criteria are analyzed.

Finally, we present a methodology for implementing testing procedures for PRAM programs. This incorporates a parallel to serial translator, a path finder, and a serial program testing tool. The idea is to choose an appropriate coverage criterion, interactively analyze the test result and select the next path to be tested, and use the translator and path finder to assist in test data generation. As the translator and path finder translate descriptions of paths within a parallel program to

CH2649-2/89/0000/0559$01.00 © 1988 IEEE

descriptions of paths within an equivalent serial program, serial program test generation tools are utilized to generate tests. Using this methodology, parallel program testing and debugging tools can be implemented economically.

The rest of the paper is organized as follows: In section 2 a general purpose parallel programming language for PRAM programming (with minor modification, it can also be used for interconnection SIMD machine programming) is presented. Section 3 deals with classification of PRAM programming errors. In section 4 we define families of testing coverage criteria for parallel programs. In section 5 a methodology for testing parallel programs is proposed.

## 2. A Language for SIMD Programming

A general purpose high-level language for SIMD machines should provide means to express parallelism, and to represent data organization and transfer. The rest of this section discusses those additional features with brief explanations.

### 2.1 Data Representation and Organization

In a PRAM, all processors (or PEs) are connected to the global or shared memory. A master processor which controls all PEs is called the control unit (CU). There are three types of variables: PE or local variables, bound locally to each PE; shared or global variables, bound to the shared memory and accessed by all PEs; and CU variables, bound to the CU. Parallel data objects which reside in PEs' local memory or PE objects are identified by the keyword *pe* preceding the data type in their declarations, and variables in the memory of CU (CU objects) are declared as usual. The keyword *shared* is used in front of the global data in the shared memory.

### 2.2 Control of Parallelism

PEs are requested statically using a *network* declaration. The language does not concern itself with the actual number of processors available. A simple *par* <range> *do* <statement> construct is used to represent the activation of different subsets of PEs whose data are referenced in the statement. Those activated PEs execute the statement following *do* in parallel.

Conditional selection of PEs is specified by *where* <predicate> *do* <statement> {*elsewhere* <statement>}, where the predicate involves parallel data in PEs. All the PEs where the predicate evaluates to true execute the statement following *do* while the PEs where it is false execute the statement following *elsewhere*, with the statement in the *do* clause executed before that in *elsewhere*. The *elsewhere* clause is optional.

### 2.3 Data Transfer through Shared Memory

All data transfers among PEs and between PE and the CU are achieved through writing into and subsequent reading from the shared memory.

This completes the list of our extensions to Pascal. The programming language has been used to code a representative set of parallel algorithms from the literature. Two example programs are given below (see ref. [2] for algorithms).

**Program 1**

```
procedure OR (n : integer; var y : boolean);
{Find the OR of n bits in O(1) time on CRCW
 PRAM with common write.  }
const
    max = 256;
network
    pe [1..max] : PRAM;
var
    i : integer;
    x : pe boolean;
    M : shared array [1..max] of boolean;

begin
{Initialization, read input n bits
 into first n cells of shared memory}
{1}   par i := 1 to n do
      begin
{2}   x[i] := M[i];
{3}   where x[i] = 1
{4}   do M[1] := 1
      end;
{5}   y := M[1]
end;
```

**Program 2**

```
procedure MAX (n,m : integer; var maxi: integer);
{Find the maximum of n integers in O(log n) time
 using EREW PRAM.  Assume n = 2^m.  }
const
    p = 256;
network
    pe [1..p] : PRAM;
var
    i, incr, step : integer;
    temp, big : pe integer;
    M : shared array [1..2*p] of integer;

begin
{Initialization, read n inputs into first n cells of
 shared memory, and set all other cells to -∞ }
{1}   par i := 1 to n do
{2}   big[i] := M[i];
{3}   incr := 1;
{4}   for step := 1 to m do
      begin
{5}       par i := 1 to n do
```

```
        begin
{6}       temp[i] := M[i+incr];
{7}       big[i] := max (big[i], temp[i])
        end;
{8}       incr := 2*incr;
{9}       par i:= 1 to n do
{10}      M[i] := big[i]
      end;
{11}  maxi := M[1]
end;
```

Note that there are three models of PRAM—Exclusive Read Exclusive Write (EREW), Concurrent Read Exclusive Write (CREW), and Concurrent Read Concurrent Write (CRCW)—according to whether concurrent reads and writes are allowed; and there are three submodels of CRCW PRAM, depending on how write conflicts are resolved [2,15]. Programs 1 and 2 above work on different models. We will, however, not be concerned with the difference between the models.

## 3. Errors in PRAM Programs

The widely used classification of computation and domain errors [7] for serial programs is not directly applicable to parallel programs. In the following sections, we identify two classes of errors characteristic of parallel programming for PRAM, namely, incorrect selection and activation of PEs and erroneous reading or writing sequences.

Suppose * denotes any aspect of a parallel program P. We use SPEC(P,*(d)) to denote the correct * of P on input d according to the specification of P. (Equivalently, SPEC(P,*(d)) is the * of the hypothetical, correct version $\hat{P}$ of P on input d.) Further, let IMP(P,*(d)) denote the implemented or actual * of P on d, obtainable through an analysis and/or execution of P. Intuitively, P contains an error whenever SPEC(P,*(d)) differs from IMP(P,*(d)) for any instance of * on some input d. (When P is understood, we use SPEC(*(d)) for SPEC(P,*(d)) and IMP(*(d)) for IMP(P,*(d)).)

### 3.1 Processor Activation Error

Let P be a parallel program with input domain D. Suppose that P takes t steps to process d $\in$ D, using q logical PEs. (Note that q and t are usually functions of the input size. Program 2, for example, takes q = n PEs and t = 4m + 3 = O($\log_2$n) steps to find the maximum of n numbers, assuming n = $2^m$ and ignoring the initialization step.) An input d causes P to activate a fixed sequence of PEs. More specifically, let PAS(d) = $<A_1, A_2, \ldots , A_t>$, a t-tuple, be the "Processor Activation Sequence" of P on input d, where each $A_i$ ($1 \leq i \leq t$) is a subset of {c,1,2,...,q} and contains the indices of active PEs at step $i$ during P's execution on d (assume that the CU is indexed with c, and the q PEs are numbered 1,2,...,q). We call each of this subset an "activation snapshot". Further, let PA(d) = {A | A is a component of PAS(d)} be the set of all activation snapshots of P on input d.

One prominent class of errors in SIMD programming is caused by processors being incorrectly activated/deactivated, i.e., a wrong set of PEs being active at some step during the execution of the program. The discrepancy between SPEC(PAS) and IMP(PAS) characterizes this type of errors. That is, a program is said to contain a "processor activation error" whenever SPEC(PAS) $\neq$ IMP(PAS), i.e., SPEC (PAS(d)) $\neq$ IMP (PAS(d)) for some d $\in$ D.

As an example of a processor activation error, consider statement {3} of program 1 erroneously written as "*where* x[i] <> 1", this causes the incorrect subset of PEs to be activated at statement {4}, and thus is a processor activation error.

### 3.2 Processor Coordination Error

Another notable feature of PRAM programming is processor coordination. Since the processors of a PRAM communicate through a shared memory, readings and writings are coordinated to achieve interprocessor communication; in addition, there are algorithms which specifically exploit the concurrent write features of PRAM. Let P, D, t, q be the same as above. Define PCS(d), the "Processor Coordination Sequence" of P on input d, as $<C_1, C_2, \ldots ,C_t>$, where each $C_i$, called a "PC snapshot", is as follows: (1) If the $i$th step in the computation is a reading step, i.e., active PEs perform reading from the shared memory, then $C_i = [s_1, s_2, \ldots, s_r]$ such that $s_j \subseteq A_i$ for $1 \leq j \leq r$ and $\cup_{j-1}^{r} s_j = A_i$, indicating that $r$ groups of PEs are reading from $r$ disjoint shared memory locations in parallel. PEs in the same group are reading simultaneously from the same location in shared memory. (2) If the $i$th step in the computation is a writing step, i.e., active PEs perform writing into the shared memory, then $C_i = \{s_1, s_2, \ldots, s_r\}$ such that $s_j \subseteq A_i$ and $\cup_{j-1}^{r} s_j = A_i$, indicating that $r$ groups of PEs are writing into $r$ disjoint shared memory locations in parallel. PEs in the same group write simultaneously into the same shared memory location. (3) If the $i$th step performs an operation other than read or write, then $C_i = |a_1, a_2, \ldots, a_r|$ such that $a_j \in A_i$, i.e., each $a_j$ is the index of a single PE, and $\{a_1, a_2, \ldots, a_r\} = A_i$, i.e., $C_i$ and $A_i$ contain the same subset of PEs. In short, within a PCS(d), [-,-, ..., -] denotes a read step and {-,-, ..., -} denotes a write step, with PEs performing simultaneous reads or writes grouped together, and | -,-, ..., -| denotes a computation step. Also, let PC(d) = {C | C is a component of PCS(d)} be the set of all PC snapshots for input d.

Difference between SPEC(PCS), the correct processor coordination sequence and IMP(PCS), the actual coordination sequence indicates errors. Specifically, we say P contains an "processor coordination error" if SPEC(PAS) = IMP(PAS) but SPEC(PCS) $\neq$ IMP(PCS), i.e., SPEC(PAS(d)) = IMP(PAS(d)) for all d $\in$ D and SPEC(PCS(d)) $\neq$ IMP(PCS(d)) for some d $\in$ D. Thus, processor coordination errors are attributed solely to erroneous PE coordination or erroneous reads or writes, since the PE activation sequence is correct during P's computation on d.

As an example, suppose statement {4} of program 1 were erroneously written as "*do* M[i] := 1", then the active PEs would write to different locations in the shared memory while simultaneous writes are intended; this is a processor coordination error.

### 3.3 Computation Error

The dichotomy of processor activation error and processor coordination error for PRAM programs can be interpreted as a refinement of the concept of domain errors for serial programs [7]. The other type of errors of serial programming, i.e., computation errors, can surely occur in parallel programs too. We say P contains a "computation error" if SPEC(PCS(d)) = IMP(PCS(d)) for all inputs d $\in$ D but P(d) $\neq$ $\hat{P}$(d) for some d $\in$ D, where P(d) and $\hat{P}$(d) are, respectively, the outputs of P and $\hat{P}$ on input d. Computation errors are, therefore, caused solely by incorrect assignment statements, incorrect arithmetic operations, etc., when P is given some input d, because both the sequence of PE activations and sequence of processor coordinations are correct during the computation of P on d. (Note that SPEC(PCS) = IMP(PCS) implies SPEC(PAS) = IMP(PAS).)

As an example of a computation error, suppose statement {4} of program 1 were mistakenly written as "*do* M[1] := 0", then error is caused by writing an incorrect value into M[1] and thus a computation error.

One desirable property of our classification is that it makes the three categories of errors mutually exclusive. Also observe that, according to this classification, programs that are functionally correct cannot contain computation errors; however, this does not necessarily imply that such programs are free of processor activation errors or processor coordination errors. Consider this example of a performance bug: Suppose statement {4} of program 2 were mistakenly written as "*for* step := 1 *to* n *do*", the program will still correctly output the maximum value for every input, and so the error will not be detected by merely checking the output values. Nevertheless, the program contains an obvious error, and in our classification, a processor activation error.

## 4. Parallel Program Testing Criteria

An essential component of a program testing methodology is a test coverage or adequacy criterion. An ideal criterion should have the following properties: (1) applicable, in the sense that coverage of the criterion can be effectively monitored; further, a test set satisfying the criterion can be effectively constructed. (2) reliable, in that a test set satisfying the criterion has high probability of detecting errors. (3) cost effective, in that the cost of generating and running a test set satisfying the criterion is acceptable; this implies that the criterion requires only a reasonable number of test cases.

In this section, we present several approaches of defining test coverage criteria for parallel programs and discuss their properties.

### 4.1 Flowgraph Based Criteria

Three commonly used measures for serial program testing are statement, branch, and path coverage. To apply these as well as some other structural testing coverage criteria to parallel programs, a flowgraph representation of parallel programs is needed. For this purpose, we can use any graphical representation for parallel programs based on a reasonable extension of serial program flowgraphs. We say statement i is "executed" by PE j, if statement i involves a variable of PE j. For example, if the input to program 1 is "00101", then statement {4} of program 1 is executed by PE 3 and 5 simultaneously.

To apply the statement, branch, and path testing criteria to parallel programs, we consider a parallel statement "covered" if and only if it has been executed by at least one PE. Thus, the test data of one input "1" can minimally satisfy statement coverage testing of program 1; however, the input "0" does not exercise statement {4} and therefore does not satisfy statement coverage. The test set consisting of two input data, "0" and "1", satisfy branch coverage of program 1, etc. The advantage of these simple criteria is their consistency with the corresponding serial testing criteria; that is, like any structural testing criterion for serial programs, the "structure" of a parallel program can be analyzed from its control flowgraph alone. Thus, many testing tools developed for serial software can be utilized for parallel software with minor modification.

### 4.2 Input Based Coverage Criteria

With the presence of new programming language constructs and a variable number of active PEs in parallel programs, the criteria defined in section 4.1 for parallel programs are apparently not adequate. As the execution of a PRAM program entails various PE activation/deactivation and interprocessor communication through shared memory, testing coverage criteria which explicitly require test cases to exercise such events would probably be more effective.

#### 4.2.1 Path Coverage Testing

Let P be a parallel program with s executable elementary statements, numbered 1 through s. (An elementary statement is one that can be executed in one step.) Define Path(d), the path of P exercised by input d, to be

$(l_1,A_1)(l_2,A_2)...(l_t,A_t)$, where $1 \leq l_i \leq s$ $(1 \leq i \leq t)$, and $l_i$ represents the ID of the statement being executed at the $i$th step; $A_i$ is the $i$th coordinate of PAS(d), denoting the set of active PEs at the $i$th snapshot during the computation of P on d. This definition of path of parallel programs is compatible with that of serial programs; however, as it represents an actual path traversed on a specific input, all infeasible paths are excluded.

Once paths are defined for parallel programs, the corresponding path testing criterion can be formulated as follows: Select enough test cases to exercise all paths. This criterion is not practical as most programs contain an infinity of paths.

### 4.2.2 Criteria Based on PAS and PCS

Based on the concept of processor activation and processor coordination, a family of testing coverage criteria can be defined. These criteria, along with the path coverage criterion of the previous section, are based on input-driven structures of the program and are not based solely on the program or its specification.

(1) PCS (Processor Coordination Sequence) Testing: Select test set $T \subseteq D$ such that $\{PCS(d) \mid d \in D\} \subseteq \{PCS(d) \mid d \in T\}$.
PCS testing requires that every possible processor coordination sequence during P's execution over all input points be exercised at least once.

(2) PAS (Processor Activation Sequence) Testing: Select $T \subseteq D$ such that $\{PAS(d) \mid d \in D\} \subseteq \{PAS(d) \mid d \in T\}$.
PAS testing requires that every possible processor activation sequence during the computation of P over its entire input domain be driven at least once.

(3) PC (Processor Coordination) Testing: Select test set $T \subseteq D$ such that $\{PC(d) \mid d \in D\} \subseteq \{PC(d) \mid d \in T\}$.
PC coverage testing requires that every possible processor coordination during the execution of P over all input points be invoked at least once.

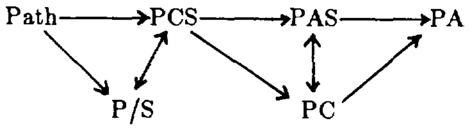
(4) PA (Processor Activation) Testing: Select $T \subseteq D$ such that $\{PA(d) \mid d \in D\} \subseteq \{PA(d) \mid d \in T\}$.
PA testing requires that every possible processor activation snapshot during the computation of P over its entire input domain be tested at least once.

(5) P/S (Processor/Statement) Testing: Select $T \subseteq D$ such that $\{E(d) \mid d \in D\} \subseteq \{E(d) \mid d \in T\}$, where E(d) the executability matrix, is a $q \times s$ matrix such that

$$E(d)[i,j] = \begin{cases} 1, & \text{if } X(d)\,[i,j] \geq 1 \\ 0, & \text{if } X(d)\,[i,j] = 0 \end{cases}$$

where X(d), the profile matrix, is a $q \times s$ matrix such that X(d)[i,j] = number of executions of statement j by processor i, when P is run on input d. P/S testing requires a test set T containing enough test data such that if it is possible for processor i to execute statement j when P is given some input $d \in D$, then there is a $d \in T$ which drives processor i to execute statement j.

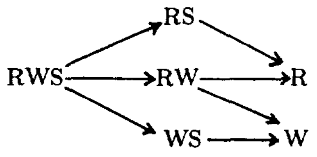
For two testing coverage criteria C1 and C2, C1 (strictly) subsumes C2 or C1 > C2, if every test set T that satisfies C1 also satisfies C2 but not vice versa. C1 and C2 are incomparable or C1 <> C2, if neither one subsumes the other [16]. The following lattice diagram shows the relative strengths of the path testing criterion and criteria (1) - (5) above.



### 4.2.3 Criteria Based on Read and Write

Reading from and writing into the shared memory are conspicuous features of PRAM programs. So an alternative approach to define test coverage criteria based on input-driven structures is to concentrate on the read/write events of PRAM programs. Let RWS(d) = $<X_1, X_2, ..., X_t>$, where each $X_i$, called a "Read/Write snapshot", is as follows: If the $i$th step in the computation is a read or write step, then $X_i = C_i$ = the $i$th coordinate of PCS(d); otherwise, $X_i = \varnothing$. That is, RWS(d) concentrates on the read/write portions of PCS(d). Let RW(d) = {X | X is a component of RWS(d)} be the set of all R/W snapshots. Similarly, RS(d) and WS(d) consist of, respectively, the read and the write portions of PCS(d); R(d) and W(d) are respectively the set of all reads and the set of all writes.

Corresponding to the definitions above, a hierarchy of six test coverage criteria can be formulated. Their relative strengths are indicated in the diagram below. Compared with the criteria in section 4.2.2, the relationship are PCS > RWS and PC > RW, most other pairs of metrics are incomparable.



### 4.3 Data Flow Based Testing

Useful data flow testing metrics will likely be necessary components of complete testing methods for parallel programs, especially because the correctness of parallel programs depends on correct data flow interactions even more critically than serial programs do. In this section, we present three simplifying measures to apply the data flow based, serial program testing metrics to parallel programs.

### 4.3.1 Criteria Based on PE Variables

In this approach, we consider each PE variable in a parallel program as a single variable rather than a collection of variables bound to different PEs; thus, PE variables can be treated in the same way as arrays of serial programs. In program 2, for example, definition of PE variable "big" at statement {7} is considered live at {10}, definition of variable "temp" at {6} is live at {7}, etc. With this unification of PE variables, liveness of each (CU and PE) variable definition at each statement can be determined statically. Therefore, algorithms that were designed for computing this for serial programs (see [14]) can be used to obtain the same information for parallel programs; and anomaly detection tools based on data flow information for serial programs can be utilized for parallel programs with minimal modification.

With the concept of liveness of definitions extended to parallel programs, concepts such as "data environment", "data context", and "ordered elementary data context" for serial programs [13] can be similarly extended to parallel programs. With theses extended definitions, data flow based static testing coverage metrics for serial programs can be applied directly to parallel programs.

### 4.3.2 Criteria Based on CU Variables

In this approach, we concentrate on the data flow activities of CU variables and ignore that of PE variables. In most parallel programs, CU variables are used to control the execution of loops which contain parallel statements or to store results of computation. Exercising a particular program path to test a particular data flow activity of CU variables inevitably exercises and tests other constructs.

For example, a *for* loop index (CU) variable is defined (initialized) before the loop is first entered, it is used (incremented) at the end of the loop, and used (in a predicate) to decide if traversal is to be continued at the beginning of the loop. Therefore, a metric such as "requiring that the definition-use chain of length 2 of every *for* loop index variable be tested at least once" necessarily forces the body of the *for* loop to be tested at least once.

### 4.3.3 Combined Criteria

Static program analysis methods based on combining data flow and control flow information have been proposed recently in ref. [9]. The approach is to use a program dependence graph to model the data and control dependences between program instructions. Taking the simplifying measures of unifying PE variables or concentrating on CU variables, we can extend the concept of program dependence graph and apply it to parallel programs. Thus, algorithms and tools based on program dependence graphs and developed for serial programs [10] can be used for parallel programs too. Testing coverage metrics based on combination of data dependence and control dependence, e.g., requiring each control dependence and each data dependence to be tested at least once, can also be defined.

### 4.3.4 Discussion

There are obvious advantages of using static data flow metrics for measuring testing thoroughness for parallel programs: these metrics are essentially the same as that for serial programs, and therefore data flow analysis tools for serial programs can be readily used. The major disadvantage, however, is that these strategies are rather weak. For examples, it can be shown that the metric of testing all live definitions from the data environment of every instruction or block of [13] is incomparable with the branch coverage criterion; the example CU variable based metric in section 4.3.2 is strictly subsumed even by branch coverage. For many parallel software applications requiring high level of reliability such as in real time systems, such structural metrics are not adequate.

## 5. Practical Testing Strategies

In this section, we present a parallel program testing methodology which incorporates the coverage metrics defined in section 4. A preprocessor consisting of a translator which translates parallel programs into serial programs and a path finder which, for a path of a parallel program, finds the corresponding path in the serial program is the major component of this methodology. The testing procedures also incorporate a serial program test data generation mechanism.

### 5.1 Flowgraph Based Coverage Testing

To perform a statement (or branch) testing of a parallel program P, a set of paths S = $\{p_1, p_2, ..., p_k\}$ is first selected to cover all statements (or branches) of P. P and S are then input to the translator and path finder. The outputs are an equivalent serial program $\hat{P}$, and the corresponding paths $\hat{S} = \{\hat{p}_1, \hat{p}_2, ..., \hat{p}_k\}$. Once $\hat{P}$ and $\hat{S}$ are obtained, a preferred serial program test generation method can be used to derive the desired test set T = $\{d_1, d_2, ..., d_k\}$. Then, the test data can be executed with P. With proper instrumentation (for example, inserting counters at each decision to decision or d-d path to monitor branch coverage [8]), the above procedure can also be carried out incrementally.

To carry out path testing, it is necessary to first apply an equivalence relation to partition the infinite set of paths into finite equivalence classes. (For example, using Howden's boundary-interior method [6] extended to parallel programs.) Then the preprocessor can be used to guide test data generation until a representative path from each class has been covered by a test case.

The method described above can also be used to conduct testing with a static data flow coverage criterion like the ones described in section 4.3.

### 5.2 Input Based Coverage Testing

The input based structure coverage criteria of section 4.2 are not directly applicable as they may require an infinity of test cases. To perform a testing to cover a particular input based structure Q (e.g., Q = PCS, then Q coverage testing is PCS testing), it is therefore necessary to find an appropriate equivalence relation among instances of Q to reduce the corresponding infinite criterion to a finitary, satisfiable one. (Formal methods of defining equivalence deserves further investigation, even though it seems unlikely that a generic and meaningful equivalence relation among parallel program paths can be found, due to the large variety of data structures, control structures, and interprocessor communication patterns that can be present. To reduce the domain of Q into finite classes, a useful equivalence relation will have to be defined from the specification and the program by the tester.)

After an appropriate equivalence relation is found and applied to an infinite input based criterion Q, it becomes finitary and thus satisfiable, because a test data which covers an instance of Q covers all equivalent instances. Let's use $Q_c$ to denote this practical Q coverage criterion. The testing of a parallel program P can then proceed according to the following procedure:

(1) P is instrumented to monitor the coverage of $Q_c$. An equivalent serial program $\hat{P}$ is then produced by the translator.

(2) Choose an input d from P's input domain, say randomly.

(3) Execute d with P (or $\hat{P}$). The result is analyzed to determine the path and the instance of Q which were exercised by d.

(4) Determine the next path *p* of P to be traversed. This path, of course, is to exercise an instance of Q which has not been covered by previous test data.

(5) Input *p* and $\hat{P}$ to the path finder to produce the corresponding serial path $\hat{p}$ of $\hat{P}$.

(6) A pre-selected serial software test data generator is used with $\hat{P}$ and $\hat{p}$ to generate the next test case d.

(7) Repeat steps (3) - (6) until the criterion $Q_c$ is satisfied.

The method described above for input based coverage testing incorporates the essential aspect of Kundu's method of generating tests for serial programs [11], namely, reversing the roles of path selection and test data generation.

### 5.3 Input Space Partitioning

Simple heuristics can often be used to partition the input space into "equivalence classes" for test data selection. In program 2, say, we can define equivalence in terms of the degree of parallelism exhibited during execution. As an example, the input "n=1, m=0" causes the *for* loop beginning at statement {4} to be skipped altogether and thus no parallelism is exhibited. The input "n=2, m=1" causes the *for* loop to be traversed exactly once and hence the *par* statement at {5} executed with two PEs exactly once, thus a minimal parallelism is exhibited. The input "n=4, m=2" causes the *for* loop to be traversed twice and hence the *par* statement at {5} will be executed twice by four PEs, thus a "nonminimal" parallelism is exhibited. All inputs with n $\geq$ 4 can be classified as equivalent in terms of exercising nonminimal parallelism, and thus the three described test data would suffice. Usually, however, more thorough testing is needed, and the partition method can be refined to a desired level to include more test cases.

Consider program 1 for another example. The algorithm has O(1) time complexity and therefore the method of partitioning based on exhibited parallelism is not applicable. Then we may consider the input of n bits as representing an integer, and define two inputs to be equivalent if they are congruent modulo a certain number, e.g., consider inputs x and y to be equivalent iff $x \equiv y \bmod 20$ if 20 test cases are planned. This is one form of random testing.

### 5.4 Use of Serial Programs

Much research effort has gone into building compilers which detect parallelism in serial programs and generate parallel codes for the target parallel machine. Conversely, a parallel-to-serial translator can be implemented to make some of the tools developed for serial programs useful in the parallel domain.

For synchronous parallel programs, the same input should always produce the same output, therefore they can be translated, one-to-one, into serial programs. As the serial program gives exactly the same result for every input point as the corresponding parallel version, one is functionally correct if and only if the other is. Thus the parallel program can be validated by testing either the original or the translated serial program. For example, a "parallel program mutation testing system" may be implemented by simply combining a translator and a serial program mutation system [3]. This is a straightforward and readily adoptable testing methodology.

This method, however, has several drawbacks. Firstly, it does not consider the structural aspects unique to parallel programs, hence could ignore some of the likely sources of errors. Secondly, as explained in section 3, programs which contain errors might still perform the correct input-output transformation. As it is hard to obtain timing information about the parallel program by simulation, this method would fail to detect most of the errors that are not evidenced by incorrect output values.

### 5.5 Preprocessor: Translator & Path Finder

We have implemented the preprocessor for the execution and testing of parallel programs written in the Pascal-based parallel language. The preprocessor consists of two components—the translator and the path finder. The translator takes parallel programs as its input, and produces corresponding serial programs as well as data files for the path finder. The path finder takes a set of path specifications for a parallel program as its input and produces the corresponding paths in the translated serial program. Implementation details of the preprocessor is discussed in [4].

## 6. Conclusion

We have presented a scheme for classifying PRAM programming errors and proposed various approaches of deriving testing coverage metrics for shared-memory parallel programs.

The approaches based on extending the serial methods can be used to define testing metrics which are easily understood and applicable. On the other hand, they tend to define weak metrics due to the fact that they overlook the features unique to parallel programming.

The criteria based on input-driven structures, combined with the proposed translator/path finder approach to testing provide a viable methodology. The advantages of this methodology are [1] practical—the preprocessor, consisting of the translator and path finder, is easy to implement and runs in linear time, [2] economical—serial program test generation tools, which programmers/testers are already familiar with, can be "plugged in", thus our software investment in testing/debugging tools for serial programs can be utilized in a parallel environment, and [3] immediately usable—it can be readily used while various programming methodologies and new techniques for parallel program validation are developed.

Testing has been the most widely used method for validating serial software and there is a growing attention to the theory and techniques of testing [5]. In view of the increasing applications of parallel computing and the lack of effective methodologies for validating parallel software, the presented approaches and methods provide useful directions for further research and development.

## References

[1] Allen, J.R. and K. Kennedy, *A Parallel Programming Environment*, Computer Science Dept. Technical Report TR84-8, Rice Univ., July 1984.

[2] Baase, S, *Computer Algorithms, 2nd Edition*, Addison-Wesley, 1988.

[3] Budd, T.A., *Mutation Analysis: Ideas, Examples, Problems and Prospects*, in B. Chandrasekaran and S. Raddichi, eds., Computer Program Testing, North-Holland (1981) 129-148.

[4] Fang, M. and A.H. Sung, *A Preprocessor for Testing Parallel Programs*, Computer Science Technical Report, New Mexico Tech, 1988.

[5] Hamlet, R. et al., *Special Section on Software Testing*, Comm. of the ACM, vol. 31, no. 6 (1988) 662-695.

[6] Howden, W.E., *Methodology for the Generation of Program Test Data*, IEEE Trans. on Computers, vol. C-24 (1975) 554-560.

[7] Howden, W.E., *Reliability of the Path Analysis Testing Strategy*, IEEE Trans. on Software Engineering, vol. SE-2, no. 3 (1976) 208-215.

[8] Huang, J.C., *An Approach to Program Testing*, ACM Computing Surveys, vol. 7, no. 3 (1975) 113-128.

[9] Korel, B, *The Program Dependence Graph in Static Program testing*, Information Processing Letters, vol. 24 (1987) 103-108.

[10] Korel, B. and J. Laski, *A Tool for Data Flow Oriented Program Testing*, in SoftFair II - 2nd Conf. on Software Development Tools, Techniques, and Alternatives (1985) 34-38.

[11] Kundu, S., *SETAR - A New Approach to Test Case Generation*, INFOTECH State of the Art Report, Software Testing, Infotech Intl. Ltd (1979) 163-186.

[12] Laski, J.W., *On Data Flow Guided Program Testing*, ACM SIGPLAN Notices, vol. 17, no. 9 (1982) 62-71.

[13] Laski, J. and B. Korel, *A Data Flow Oriented Program Testing Strategy*, IEEE Trans. on Software Engineering, vol. SE-9, no. 3 (1983) 347-354.

[14] Muchnick, S.S. and N.D. Jones, *Program Flow Analysis : Theory and Applications*, Prentice-Hall, 1981

[15] Quinn, M.J., *Designing Efficient Algorithms for Parallel computers*, McGraw-Hill, 1987.

[16] Rapps, S. and E.J. Weyuker, *Data Flow Analysis Techniques for Program Test Data Selection*, Proc. of 6th Int'l Conf. on Software Engineering (1982) 272-278.

# On the Expansion, Analysis, and Mapping of Conventional Programs into Code for Bit Level Processor Arrays

*Jose A. B. Fortes* †

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

## Abstract

Massively parallel bit level processor array architectures are becoming increasingly popular due to their suitability for VLSI implementation as well as because they can take advantage of parallelism at all levels of granularity (down to bit-level). However, the programming of such machines requires from the user the ability to understand and exploit low level parallelism and considerable familiarity with the low level details of the processor array architecture. This paper discusses a possible solution to this problem via the usage of a software package capable of taking a program expressed in a conventional sequential language and mapping it into code executable by the processor array. It describes a first attempt at implementing such a tool and discusses in detail all of its components and the techniques used. Based on past experience with this experimental tool, aspects of a planned new version of the same tool are also discussed.

## I. Introduction

Existing and proposed solutions to the problem of using bit-level processor arrays to execute conventional programs for sequential machines are described and discussed in this paper. An existing software package, RAB (Reconfiguration Algorithm for Bit-level arrays) [10] [12], developed at Purdue, is described in detail as well as experience with its usage. A new planned version of RAB is briefly described with emphasis on the differences with respect to RAB and how it will incorporate lessons learned with RAB.

Processor arrays with simple 1-bit processing elements have become a pervasive form of massively parallel processing architecture. A non-exhaustive list of commercially available machines of this type includes DAP[1], MPP[2], the Connection Machine [3], and GAPP[4]. Commonly perceived advantages of these architectures are, among others, their suitability for large and dense VLSI implementations, the ability to support parallelism at all levels of granularity (down to the bit level) and the fact that they match many algorithms and data structures of interest (e.g., image processing and arrays of data). However, as for other novel parallel processing architectures, the programming of processor arrays often requires considerable thought and skill if highly efficient codes are sought. This is particularly true when complex tasks are to be implemented using concurrent low-level operations, as in the case of the bit-level processor arrays mentioned above. Several approaches to this problem have been used which include:

- subroutine libraries accessed through a standard high-level language which have been optimized for a particular architecture, as in the *DAP* processor [1];
- parallel languages, which are often architecture or machine dependent [5];
- microcoded routines to handle standard word level operations in a general way, without using bit level optimizations.

These approaches lack portability among different machines and sometimes ignore optimizations possible at the bit level. It is often difficult to prove the optimality of a given mapping using these methods. An alternative approach (to that of putting the programming burden solely on the programmer) is to use software capable of transforming a program expressed in a familiar sequential programming language into processor array code. Such large program is akin to an optimizing compiler in the sense that it must detect and exploit parallelism but, in addition, it must be able to reexpress high-level computations as combinations of low-level bitwise operators and data movements. In the past, related research efforts have been pursued in this direction [6] and several methodologies, which were intended for word level processor arrays, are potentially applicable to bit level arrays. This paper reports on the techniques used in the major components of RAB, a large program developed for the purpose mentioned above. It also discusses aspects of a second version of RAB, yet to be implemented, and the improved techniques and approaches to be used in it.

A block diagram relating the main components of RAB is shown in Figure 1 and is briefly explained next. The input to RAB consists of C programs which describe word level algorithms. These algorithms correspond to nested **for** loops with static behavior. RAB first expands the computations in the input program into bit level operations. This expansion phase uses a library of macro expansions to replace each word level computation with a bit level implementation of the arithmetic operations. This phase is followed by data dependence and broadcast analysis using a technique called Dependence Arc Set Analysis[7]. The result of this analysis is a formal description of the internal structure of the bit level algorithm. This structural information is used to generate an algorithm transformation which yields a restructured algorithm suitable for mapping onto a bit level processor

†This work was supported in part by the National Science Foundation under Grant DMC-8419745 and in part by the Innovative Science and Technology Office of the Strategic Defense Initiative Organisation and was administered through the Office of Naval Research under contract no. 00014-85-k-0588.

CH2649-2/89/0000/0567$01.00 © 1988 IEEE

array. The mapping may be a full design of an algorithmically defined array or full (partial) mapping for a fixed (variable) size array corresponding to the fourth level of modules in Figure 1. The last two modules, code generation and code optimization, comprise the phase in which code is generated for a particular target architecture and optimized using a standard compaction technique.

The separate phases of RAB, shown in Figure 1, are now described in more detail.

## II. RAB - Reconfiguration Algorithm for Bit-Level arrays

### II.1 Program Specification

RAB accepts a subset of the C language as input. This subset consists of nested **for** loops with static behavior, which are likely candidates for mapping into processor arrays given their repetitive and regular structure. Pointers and function calls cannot be used, and the nested loops must meet the following requirements:

- the lower and upper bounds of the outermost loop must be integer constants.
- the bounds of the nested loops must be linear expressions of the outer loop variables or integer constants.
- the step of each loop must be one.
- no two loops can have the same nesting level.
- arrays of any dimensions are allowed; the range of each dimension must be an integer constant.
- the boolean expression of a conditional statement must be a linear expression of the outer loop variables.
- all subscript expressions used when referencing elements of arrays must be linear expressions of the outer loop variables.

Algorithms which satisfy these requirements include matrix-matrix and matrix-vector multiplication, FIR and IIR filtering, convolution, and many others used in digital signal and image processing. Many algorithms which fall outside this class can be transformed to satisfy the above constraints using such techniques as loop fusion and normalization [8].

The following is an example of a program which satisfies the requirements listed above:

```
for(j1 = 1;  j1 <= N1; j1++){
   for(j2 = 1;  j2 <= N2; j2++){
      y[j1] = y[j1] + w[j2] * x[j1+j2-1]
   }
}
```

This program computes a sequence of outputs $y[j_1]$ as the convolution of a sequence of inputs $x[j_1+j_2-1]$ with weights $w[j_2]$.

### II.2 Bit Level Expansions

This first stage of RAB implements a source-to-source mapping [9] of the word level 'C' code into an equivalent bit level form. To some extent it represents a *devolution* [9] in that a higher level language feature is replaced by its meaning in terms of lower level language features. Several expansions for a given arithmetic expression are possible, but only a small library of expansions is currently implemented. These expansions exist for addition, multiplication, division, and subtraction, and all possible pairwise combinations of these operations. An example of a bit-level expansion for the convolution algorithm is given below.

```
for(j1 = 1; j1 <= N1; j1++){
for(j2 = 1; j2 <= N2; j2++){
for(j3 = 1; j3 <= N3; j3++){
for(j4 = 1; j4 <= N4; j4++){
cy[j1][j2][j3][j4] = (w[j2][j3] & sum[j1][j3+j4-1]) | (cy[j1][j2][j3][j4-1] &
(w[j2][j3] | sum[j1][j3+j4-1]))
sum[j1][j3+j4-1] = ( w[j2][j3] ^ sum[j1][j3+j4-1] ^ cy[j1][j2][j3][j4-1] &
x[j1+j2-1][j4]) | (sum[j1][j3+j4-1] & ~x[j1+j2-1][j4])
}
}
}
}
```

In the expansion shown above, the two statements in the body of the loop compute the carry and sum bits of the weighted sum implemented by the convolution program. All variables have been expanded/renamed so that each denotes a bit of the corresponding original word level variable. The two additional loops result from the need for operating on different bits of variables being multiplied in the original program.

It is possible to expand a word level 'C' program into two forms: one form can actually be directly compiled into executable code allowing the user to see the bit level expanded version of the program running; the other form (an example of which is shown above) is directly input to the analysis stage, where dependence analysis is performed on the expanded code.

One of the main advantages of the expansion process is that it exposes lower level operations to the view of the optimizer, allowing for more improvement to the generated code. This is somewhat analogous to the optimizations that are possible to an intermediate form in a typical optimizing compiler.

### II.3 Program Analysis (Dependence Analysis)

The techniques used in the mapping phase of RAB require specific structural information about the expanded program. They assume that the algorithm is characterized by its index set $J^n$ (i.e., the set of n-dimensional vectors whose entries correspond to the value of each loop index for the iterations executed by the nested loop program) and a matrix of dependencies D where every column corresponds to the vector difference between the indices of dependent computations. In other words, if a computation with index $j \in J^n$ (or, equivalently, computation *at* j) depends on a computation at $j' \in J^n$ then $j - j'$ is a column of D. A computation at j depends on a computation at j' if data generated at j' is used at j (this is called a 'true' data dependence or flow dependence) or the same data is generated at j and j' (called output dependence) or data used at j is modified at j' (called anti-dependence). The mapping phase of RAB is concerned only with true data dependencies but the analysis phase can detect all three types of dependencies. Currently, RAB requires the user to remove all dependencies other than flow dependencies from the expanded program or from the output of the analysis phase. This could also be done automatically through variable renaming and expansion[8] but is not currently implemented in RAB. The analysis phase is based on Kuhn's Dependency Arc Set Analysis (DASA) technique [7] and is discussed in more detail next.

DASA is used to detect dependencies between statements referencing arrays. In order for a dependence to exist between two statements referencing arrays, the following conditions must be satisfied:

1. the array references in the two statements must have the same name.
2. given that condition 1 is satisfied, the functions which specify the subscripts of the array references must take the same value for some index value(s).
3. the index value(s) for which condition 2 is satisfied must belong to the iteration space.

The analysis module is invoked when the parser detects that condition 1 holds and DASA is used to verify conditions 2 and 3.

DASA utilizes five relations (represented as convex sets) to gather information about the dependence arcs and to determine whether conditions 2 and 3 are satisfied. Dependencies are considered in relation to the Cartesian product of the loop indices and the nesting level of the dependence. Two of the five relations, T and H, characterize the control structure of the loops surrounding the tail statement (i.e., the point where data is generated) and the head statement (i.e., the point where data is used) of the dependence, respectively. Two other relations, $S_g$ and $S_u$, respectively, correspond to the indexing function of the generated and used arrays referenced in the head and tail statements. The fifth relation, $F_{th}$, represents the lexicographical ordering used to execute the nested loop program. These relations are represented as convex sets in a matrix format that is easily implemented and manipulated in software. As an example, it is shown below how such relations are represented for the variable cy in the convolution example; the explanation of this representation follows its description. The following is actual output from RAB when used in the debugging mode.

```
FOR GEN    VAR: cy INDEX:  [j1][j2][j3][j3]

FOR USAGE  VAR: cy        INDEX:  [j1][j2][j3][j4-1]

ORIGINAL MATRIX

 -1   0   0   0   0   0   0   0  -1
  0  -1   0   0   0   0   0   0  -1
  0   0  -1   0   0   0   0   0  -1
  0   0   0  -1   0   0   0   0  -1
  1   0   0   0   0   0   0   0  30
  0   1   0   0   0   0   0   0  25
  0   0   1   0   0   0   0   0  10
  0   0   0   1   0   0   0   0  20
  0   0   0   0  -1   0   0   0  -1
  0   0   0   0   0  -1   0   0  -1
  0   0   0   0   0   0  -1   0  -1
  0   0   0   0   0   0   0  -1  -1
  0   0   0   0   1   0   0   0  30
  0   0   0   0   0   1   0   0  25
  0   0   0   0   0   0   1   0  10
  0   0   0   0   0   0   0   1  20
  1   0   0   0  -1   0   0   0   0
  0   1   0   0   0  -1   0   0   0
  0   0   1   0   0   0  -1   0   0
  0   0   0   1   0   0   0  -1  -1
 -1   0   0   0   1   0   0   0   0
  0  -1   0   0   0   1   0   0   0
  0   0  -1   0   0   0   1   0   0
  0   0   0  -1   0   0   0   1   1
  1   0   0   0  -1   0   0   0  -1
```

In the above representation, each row corresponds to an inequality of the form $[a_t\ b_h][j_t\ j_h]^T \leqq b$ where $a_t$ and $a_h$ are vectors that multiply the index $j_t$ of the tail statement and the index $j_h$ of the head statement; b is some constant. Because $j_t$ and $j_h$ are implied, each row shown above represents only $a_t$ (the first four entries) $a_h$ (the second four entries) and b (the last entry). The first eight rows represent the upper and lower bounds for the loops surrounding the tail statement; the following eight rows do the same for the head statement (In this example the upper bounds for $j_1$, $j_2$, $j_3$ and $j_4$ are 30, 25, 10 and 20, respectively; the lower bounds are all 1.) The next eight rows represent the fact the indices used to reference cy in the tail and head statements differ only in $j_4$ (by one). The last row represents the lexicographical ordering in terms of the index $j_1$. Another three similar matrices are also considered by DASA where the lexicographical ordering is represented in terms of two, three and four loop indices. These matrices are not shown here.

If a solution space results from the convex analysis of the intersection of the relations, T, H, $F_{th}$, and $S_g$ composed with $S_u^{-1}$, then a dependence exists for the conditions defined by the forward relation $F_{th}$. For example, the set of inequalities shown above is inconsistent but that where the lexicographical ordering is represented in terms of the four loop indices is consistent and detects that a dependence is associated with the variable cy (the associated dependence vector is shown in the next paragraph). Further details about DASA can be found in [7] and [9]. DASA is also used to analyse data broadcasts. A broadcast is required when a data item is used by more than one computation at the same instant of time. To avoid a broadcast it suffices to schedule those computations to be executed at different instants of time so that the data item can be "buffered" in one computation and later passed to the other. This is similar to a data dependence except for the fact that the dependence is "undirected", i.e., the order in which computations use the data can be reversed. According to the above considerations, the vector difference between computations that use the same data item is referred to as a *buffering vector* and DASA is used to detect such vectors as well as dependency vectors. The synthesis phase of RAB takes into account the fact that either the detected buffering vector or its complement can be used to identify a valid mapping.

As an example of the final output of the analysis phase, the dependence matrix for the expanded convolution is as follows:

$$D=\begin{bmatrix}0 & 0 & 0 & 1 & 1 & 0\\ 0 & 0 & 1 & 0 & -1 & 0\\ 0 & 1 & -j_3 & 0 & 0 & 1\\ 1 & -1 & j_3 & 0 & 0 & 0\end{bmatrix} \quad \text{for } j_3 = 1,\ldots,N_3 \text{ given } N_3 < N_4.$$

Column 1 of this dependence matrix corresponds to the used variables cy[$j_1$][$j_2$][$j_3$][$j_4-1$] and the input variables w[$j_2$][$j_3$], columns 2 and 3 correspond to the used variables sum[$j_1$][$j_3+j_4-1$], and the last three columns corresponds to the input variables w[$j_2$][$j_3$] and x[$j_1+j_2-1$][$j_4-1$].

The distance vectors for the generated data items are used in the synthesis phase to preserve the semantics of the program; the buffering vectors (or their complements) for the input data items are included in the dependence matrix in order to assign different execution times for the computations requiring the same variable. The next section describes the methodology used to generate an algorithm transformation for a variable size array.

## II.4 Time and Space Mappings

The synthesis phase of RAB utilizes a well known transformation methodology sometimes referred to as the data dependency method[16]. This methodology generates a transformation matrix, T which maps the index points of the bit-level algorithm into the space-time domain (i.e., processors and instants of time). The first row of T, $\pi \epsilon Z^{(1 \times n)}$, (Z denotes the set of integers) corresponds to the time mapping; the last two rows, denoted $S \epsilon Z^{(2 \times n)}$, correspond to the the space mapping.

### Time Mapping

The linear time mapping, $\pi \in Z^{(1 \times n)}$ maps the index set of the algorithm into the unidimensional time space, $\pi : J^n \longrightarrow t$. Given $\pi$, the time of execution of a computation indexed by j is:

$$f(\bar{j}) = \left\lceil \frac{\pi\bar{j} + O}{\text{disp } \pi} \right\rceil \qquad (1)$$

where $\text{disp } \pi = \min\{\pi\bar{d}_i,\ \bar{d}_i \in D\}$ and $O = -\min\{\pi j : j \in J^n\} + 1$.

The constant O in (1) forces the first computations to be executed at time t=1. The parameter disp $\pi$ represents the number of parallel arithmetic computations executed in each processing element. In RAB the value of disp $\pi$ is restricted to one because this is representative of the array used with RAB (GAPP) and some other available architectures (i.e., MPP, DAP, CLIP). Given that disp $\pi = 1$, the total execution time of an algorithm is

$$\left\{1 + \sum_{i=1}^{n} \left|\pi_i\right| (N_i - L_i)\right\}\delta \qquad (2)$$

where $N_i$ and $L_i$ correspond to the upper and lower bounds of the index $j_i$ respectively and $\delta$ represents the number of clock cycles needed for the execution of the body of the nested loop program. To insure that the ordering determined by $\pi$ is valid, the restriction that $\pi d_i > 0$ for all $d_i \epsilon D$ is imposed. This insures that all dependencies are respected, i.e., a data item is never used before its generation.

The time transformation $\pi$ is found by trying to minimize the function (2) which is monotonic with respect to the entries of $\pi$. Due to the monotonicity of the function, a heuristic approach similar to the one presented in [11] is used to generate $\pi$. The absolute value of the entries of $\pi$ are progressively increased (starting from zero) and all possible combinations of their signs are considered with the exception of those obtained by negating previously generated $\pi$'s. The validity of each of the $\pi$ is checked and the valid time transformations are ordered according to the execution time . Possible $\pi$'s, which might result from further increases in the absolute value of the entries of a particular $\pi$, need not be considered because the monotonicity property mentioned above insures that they result in execution times larger than the known minimum. The ordered list of $\pi$'s is also used to generate space transformations as discussed next.

### Space Mapping

The space mapping determines what processor is used to execute any given computation at the instant of time determined by the time mapping. It must be selected so that processors can exchange any data item within the time interval that separates its generation and usage. A structural model of the target processor array is used in this phase and it consists of a q-dimensional index set $L^q$ (q=1 or 2) and a matrix of interconnection primitives P.

Each point $\bar{\ell} \epsilon L^q$ corresponds to the relative location of a processing element in the systolic array. The matrix of interconnection primitives is such that if $\bar{p} \epsilon P$ then for any $\bar{\ell} \epsilon L^q$, $\bar{\ell}$ is connected to $\bar{\ell}' = \bar{\ell} + \bar{p}$ if $\bar{\ell}' \epsilon L^q$ and $\bar{\ell}$ is connected to an input-output port if $\bar{\ell}' \notin L^q$. In other words, $L^q$ and P define the topology of the processor array. In addition to this, RAB needs additional information for other phases. For timing purposes and to evaluate $\delta$ in expression (2), it uses the longest time $\tau$ taken by any basic operation executed by the processors of the array (in many arrays all basic operations take the same time to execute). The time $\delta$ results from counting the number of basic operations used for the bit-level computations in the body of the expanded loop program and multiplying it by $\tau$. The set R of resources available in each processing element is used in the microcode optimization phase as discussed in section II.5.

For the GAPP (6 × 12) processor array chip the index set is given by

$$L^2 = \left\{(\ell_1, \ell_2) : 1 \leqq \ell_1 \leqq 12,\ 1 \leqq \ell_2 \leqq 6\right\}$$

and the matrix of interconnection primitives is

$$P = \begin{bmatrix}0 & 1 & 0 & 0 & -1\\ 0 & 0 & 1 & -1 & 0\end{bmatrix}.$$

The set R of shareable resources available in each processing element of GAPP consists of its four 1-bit registers, one RAM port and one ALU, and the worse case execution time $\tau$ of a basic operation is assumed to be three clock cycles (one clock cycle to place data in proper registers and execute a "full add" operation, and two clock cycles to place data in the proper register for communication purposes).

In mapping an algorithm into a systolic array, the main goal is to insure that the data communication between processors can be accomplished using the given interconnection primitives. In other words, if a computation performed by processor $\ell_1$ at time $t_1$ depends on data generated by processor $\ell_2$ at time $t_2$, then there must be a composition of interconnection primitives that connects $\ell_2$ to $\ell_1$ in time $t_1 - t_2$. The composition of interconnection primitives is given by the matrix $K \in Z^{(r \times m)}$. To insure that a direct path is taken for the movement of data, the entries in a column of K are restricted to have the same sign. Given these parameters, the spatial transformation S must satisfy the following set of diophantine equations

$$SD = PK \tag{3}$$

where $S \in Z^{(q \times n)}$, $D \in Z^{(n \times m)}$, $P \in Z^{(q \times r)}$, and $K \in Z^{(r \times m)}$.

The sum of column i of the K matrix represents the total number of data movements for the corresponding data item associated with column i of the dependence matrix. This sum is bounded above by $r\pi d_i$, where $r$ represents the ratio between computation and communication delays in the array. This upper bound represents the time between computations that is available for communication according to the schedule defined by $\pi$. Only one intercommunication primitive for each unique data link is included in the P matrix, i.e., even if a data link is bidirectional only one primitive corresponding to one of the directions is included in P. Consequently, the matrix of interconnection primitives used with RAB for the GAPP architecture contains only the first three columns of the P matrix.

If no solution exists to (3), another $\pi$ is selected from the ordered list with minimal increase in execution time. If solutions exist to (3), the transformation matrices (composed of an S and the corresponding $\pi$) are ordered according to the AT (area × time) criteria. The first transformation matrix in the ordered list for which a conflict does not occur is chosen. A conflict occurs when two or more computations are mapped into the same PE to be executed at the same time. In other words given two computations indexed by j′ and j″, a conflict occurs when $T\,j' - T\,j'' = 0$ or

$$T(\bar{j}' - \bar{j}'')=0 \tag{4}$$

where $\bar{j}' - \bar{j}''$ represents the conflict vector. The conflict vectors are generated using an analysis scheme similar to the one used with the generation of buffering vectors for input variables. If the conflict vector exists within the given iteration space, the corresponding T is disregarded and the next transformation matrix in the ordered list is checked. The procedure is continued until a conflict-free algorithm transformation can be found for the partial mapping of the bit-level algorithm into the variable size array. The conflict-free transformation matrix for the example convolution algorithm is (for $N_1=N_2=N_3=3$, $N_4=5$)

$$T = \begin{bmatrix} 5 & 4 & 2 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

for time = $27\delta$ and space = 21 PE's. This transformation optimizes AT where area corresponds to the number of PE's since only one GAPP chip is required. A different mapping for the same convolution program which results from using a different expansion and time and space mappings is reported in [12].

## II.5 Code Generation and Optimization

The current version of RAB does not have code generation implemented in software. The designer must hand code the algorithm generated in the mapping phase. However, RAB provides a capability for compacting the microcode in order to minimize its running time. This is done by detecting what processor instructions can run simultaneously without violating data dependencies and machine dependencies. While data dependencies result from the structure of the algorithm (see section II.3) machine dependencies (also called resource dependencies) result from the existence of limited resources in each processing element. A given statement $S_j$ is machine dependent on a statement $S_i$ if $S_i$ precedes $S_j$ and $S_i$ uses resources required for the execution of $S_j$.

Machine dependencies can be divided into two categories: explicit machine dependencies and implicit machine dependencies. Explicit machine dependencies result from the apparent limitations of the architecture. For example, statement $S_j$ is explicitly machine dependent on statement $S_i$ if both statements require a write to two different RAM locations and the given architecture only has one RAM port. Implicit resource dependencies are inherent in the semantics of the instructions. For example, in a GAPP array, the ALU of each PE always executes a "full add" operation every clock cycle, regardless of the instruction being executed. As a consequence, the architecture of each PE exhibits implicit resource dependencies with the use of the calculated variables sm, bw, and cy (which denote sum, borrow, and carry respectively). Thus, if a statement explicitly uses a calculated variable, then it will always depend on the previous statement.

An optimizer specialized for the GAPP array performs code compaction using a modified version of a technique developed by Ramamoorthy ([13], [14]) known as Precedence Partitioning. The straight-line microcode is parsed in a sequential manner placing used and generated variables in a symbol table. If a used variable is encountered, the optimization function checks the symbol table to see if this variable has been generated in a previous statement resulting in a data dependence (in this case all types of dependencies are checked, i.e., flow-, output- and anti-dependencies). The resources required for the parallel execution of two statements are also pairwise checked if they are data independent. If the required resources exceed the resources available, then a machine dependence exists between the two statements. The dependencies are represented in a ((v-1) × v) connectivity matrix, where v is the number of statements in the straight-line code. The element $c_{ij} = 1$ if statement j is dependent on statement i and 0 otherwise. The precedence partitioning algorithm uses this matrix to partition the set of computations into independent blocks by locating columns containing zeros, allocating the corresponding statements to a block and deleting the row corresponding to the allocated statement. The partitions are executed serially but the statements within the same partition block are executed in parallel. An example of the precedence partition for straight line code is given below. An example using GAPP instructions would require detailed knowledge about the GAPP architecture, which is beyond the scope of this paper.

For the following straight line code

(1) $A = B + C$
(2) $D = A + E$
(3) $F = D + E$
(4) $G = H + I$

the connectivity matrix is given by

$$C = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

The following partitions result from this matrix: $\{ 1, 4 \}$, $\{ 2 \}$, $\{ 3 \}$.

## III. Past Experience and Future Work

Experience and insights provided by the development and usage of RAB showed some of limitations of the approaches used for the purposes described above. However, it also revealed opportunities for new improved techniques that can be applied to the problems that RAB attempts to solve. RAB was developed as a first attempt to demonstrate the viability of a tool that uses algorithm transformations to support the systematic mapping of high-level programs into bit-level processor arrays. The techniques used represent the state-of-the-art in that field as of 1985. Since then significant progress has been made in this area. Additionally, several idiosyncrasies of RAB limit the possibility of easily incorporating such new techniques, considering different processor array architectures and testing the validity of the concepts in a wide range of programs. Finally, as expected, experience with RAB revealed many insufficiencies of the known techniques and generated many new interesting questions that must be addressed if a truly useful tool with an acceptable level of generality is to be implemented. This prompted the author and his co-workers to start the development of a new software package which improves on RAB in the following regards:

*Bit-level expansions:* Currently, the optimality and generality of the mappings provided by RAB is greatly affected by how bit-level expansions are derived from word-level operations. This is due to the very large number of expansions that are possible for any single word-level operation. Because RAB needs to have a copy of each possible expansion explicitely stored (i.e., the expansions are stored and not generated when needed), it is infeasible to consider every possible expansion. In addition, assuming that all possible expansions were available, RAB has no means of knowing or estimating which expansion yields in the best design or mapping. As a consequence, RAB needs to to be fully executed for every single possible expansion in order to guarantee optimality. This is also too expensive in practice.

To deal with the problems mentioned above, automatic programming and program transformation techniques are being considered which allow for the on-line generation of expansions from a reduced number of "basic" stored expansions. The "derived" expansions are the result of applying transformations to the basic expansions by exploiting properties such as commutatitivity, distributivity and associativity of the bit-level operations. In order to estimate or identify the best expansion, some dependence analysis is performed and attached to each expansion. By analysing relevant characteristics of data dependences, it is possible to arrive at relative performance of different expansions and, furthermore, guide the generation of additional expansions with improved characteristics. The knowledge of the dependence structure of the expansion used is also beneficial in the analysis phase, as discussed later.

RAB was intended solely for numerical computing. It is possible to use the same approach for symbolic processing, image processing or other specialized application as long as the expansions are provided for the basic operations that characterize the "algebra" of the application. These expansions will be included in the new version.

*Program Analysis (Dependence Analysis):* Currently, RAB analyzes the full expanded program by using the DASA technique which yields structural and dependence information about the program as a convex set. The inequalities that define this convex set are then used to derive a matrix where each column is a dependence vector. This process has two disadvantages. First, the analysis is unnecessarily complex in the sense that no attempt is made in RAB to exploit the fact that the "complex" expanded program is the result of a "simple" known expansion applied to a "simple" program. Secondly, information about the structure of the program is lost in the process of generating the matrix of dependencies. In addition, in order to be able to use a matrix representation for dependencies, it is often necessary to "fuse" loops or consider only those expansions where loops are "fused". The problems mentioned above are avoided in the new approach in the following ways. The dependence analysis of the expanded program is obtained by combining the results of the dependence analysis of the original word-level program and the dependence analysis of the expansion. The computational cost of this approach is smaller than that of analysing the full program. With regard to the generation of the dependence matrix, this feature will remain in the new approach only for the purpose of interaction with the user. However, for purposes of optimization in the synthesis phase, convex set information is used instead. In addition, loop fusion is not needed explicitly, and, instead, constraints in the form of inequalities are sufficient to convey how loops are arranged.

*Mapping and design phase:* Currently, RAB is fairly inflexible in the type of constraints that it can take into account for mapping and design purposes. The same can be said about the optimization criteria. This is due to the fact that different constraints are represented differently and, to a certain extent, built into the code. The same can again be said of the objective functions to be optimized. The new approach provides a unique form for the representation of any type of constraint (in essence, a set of inequalities) and optimization procedures that are independent of what the constraints and objective functions are (within certain limits which have to do with the convexity of the objective function and the space of feasible solutions). Therefore, constraints and optimization criteria can be easily added, deleted or changed. By having the expansion, analysis and synthesis phases of RAB use the same convex set representation, a more modular, efficient and coherent mapping and design procedure will hopefully result. In cases where specialized optimization procedures are used often (e.g., to optimize execution time) this approach can be complemented by software modules which implement such procedures or related heuristics. For example, several different nested loop programs were processed by RAB and this revealed that the search spaces for the optimal time and space mappings were extremely large, and the execution time of this code was excessive. This led to the consideration of projection techniques for space mappings that are now described.

With projection techniques, an $n$-dimensional *iteration vector* $\upsilon$ is employed in determining the space mapping [17]. The iteration vector is the right null vector of the space transformation matrix S, that is $S\upsilon = \underline{0}$. It can be interpreted geometrically as the direction of the projection of the index set onto the processor space. Two computations indexed by $\underline{i}$ and $\underline{j}$ are mapped into the same processing element if and only if $\underline{i} = \underline{j} + \alpha\upsilon$, for some integer $\alpha$. For a given processor space, it is unique if the following two conditions are imposed:

1) The greatest common divisor of the elements of $\upsilon$ must be one,
2) the leading element of $\upsilon$ must be positive.

For a given iteration vector $\upsilon$ that satisfies these conditions, many S matrices may be defined that form a basis for the same processor space. Searching for an iteration vector that defines the space mapping rather than an S matrix avoids this redundancy. The topology of the processing element space remains fixed, and only the labelling of each processing element is different. If the linear schedule $\pi$ has been determined, the iteration vector $\upsilon$ must satisfy $\pi\upsilon^T \neq 0$, if the mapping matrix T is to be nonsingular. This follows from the fact that if $\pi\upsilon^T = 0$, the $\pi$ matrix will be in the row space of S. Hence, computations to be executed at the same time will be mapped into the same processing element. The number of processing elements can be used as a measure of the optimality of a given space mapping.

*Code generation and optimization:* Future work will focus on a retargetable code generation module for RAB. A first step in this effort is an intermediate language that is appropriate for a variety of highly parallel processor arrays. The basic problems of instruction selection, register allocation, storage allocation, and machine-dependent optimizations that are important to code generation for uniprocessors are also important to code generation for processor arrays. Additional complexity is present in that the code generator must separate the program into serial and parallel segments. The instructions to execute the parallel segments must use the parallel execution resources of the machine efficiently. In uniprocessors, some optimization of machine code is almost always possible by looking at only a small portion of a program, but this characteristic does not hold for parallelization of machine code. It is necessary to look at a large part of the whole program to insure that the parallelized version of the program is correct and equivalent to the serial version.

## References

[1] R.W. Hockney and C.R. Jesshope, **Parallel Computers: Architecture, Programming and Algorithms,** Adam Hilger Ltd.: Bristol, 1981, pp.178-192.

[2] K.E. Batcher, "Bit-Serial Parallel Processing Systems," *IEEE Trans. on Computers,* Vol. C-31, No. 5, pp. 377-384.

[3] W.D. Hillis, **The Connection Machine,** MIT Press: Cambridge, MA, 1985.

[4] R. Davis and D. Thomas, "Systolic Array Chip Matches the Pace of High-Speed Processing," *Electronic Design,* Oct. 31, 1984.

[5] A.P. Reeves, "Parallel PASCAL and the Massively Parallel Processor," in **The Massively Parallel Processor,** ed. J.L. Potter, MIT Press: Cambridge, MA, 1985.

[6] J.A.B. Fortes, B.W. Wah, and K.S. Fu, "Systematic Approaches to the Design of Algorithmically Specified Systolic Arrays," *IEEE Int. Conf. Acoustics, Speech, and Signal Processing,* 1985, pp.300-303.

[7] R.H. Kuhn, "Optimization and Interconnection Complexity for Parallel Processors, Single Stage Networks, and Decision Trees," PhD dissertation, Dept. of Comp. Sci., U. of Ill. at Urbana-Champ., 1980.

[8] M.J. Wolfe, "Optimizing Supercompilers for Supercomputers," PhD dissertation, Dept. of Computer Science, U. of Illinois at Urbana-Champaign, October, 1982.

[9] D.B. Loveman, "Program Improvement by Source-to-Source Transformation," *J. ACM,* Vol. 24, No. 1, pp. 121-145, January 1977.

[10] V.E. Taylor, "RAB: A Tool for Systematically Mapping Numerical Algorithms into Bit-level Processor Arrays," M.S.E.E. Thesis, School of Electrical Engineering, Purdue University, W. Lafayette, Indiana, December 1986.

[11] M.T. O'Keefe and J.A.B. Fortes, "A Comparative Study of Two Systematic Design Methodologies," *1986 Int. Conf. Parallel Proc.,* St. Charles, IL, pp.672-675.

[12] V.E. Taylor and J.A.B. Fortes, "Using RAB to MAP Algorithms into Bit-Level Systolic Arrays," *2nd. International Conference on Supercomputing,* May 1987, Sta. Clara, California, pp.227-236.

[13] C.V. Ramamoorthy, "Connectivity Considerations of Graphs Representing Discrete Sequential Systems," *IEEE Transactions on Electronic Computers,* October 1965, pp. 724-727.

[14] C.V. Ramamoorthy and M.J. Gonzalez, "A Survey of Techniques for Recognizing Parallel Processing Streams in Computer Programs," *1969 Fall Joint Computer Conference,* Vol.35, Montvale, N.J., AFIPS Press, pp.1-15.

[15] A. Schrijver, **Theory of Linear and Integer Programming.** John Wiley and Sons: Chichester, UK, 1986.

[16] D.I. Moldovan and J.A.B. Fortes, "Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays," *IEEE Trans. on Computers,* Vol. C-35, No. 1, January 1986, pp. 1-13.

[17] S.K. Rao, "Regular Iterative Algorithms and their Implementations on Processor Arrays," PhD dissertation, Stanford University, 1985.

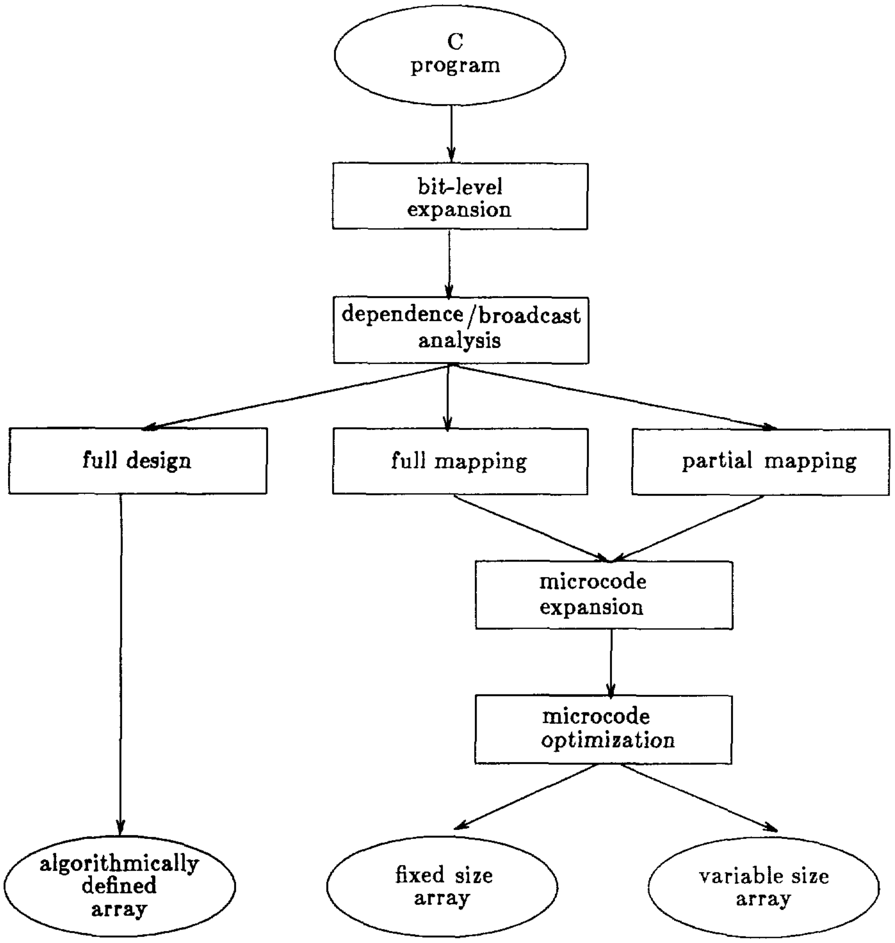


Figure 1. Flow diagram of RAB.

# COMPILING COLLECTION-ORIENTED LANGUAGES ONTO MASSIVELY PARALLEL COMPUTERS

Guy E. Blelloch[1]
Gary W. Sabot[2]

(extended abstract)

November 28, 1988

## Abstract:

This paper covers some issues and techniques for compiling collection-oriented languages for use on parallel and vector machines. Collection-oriented languages are centered around data structures which represent collections of elements and include operations for manipulating the collections. Many of the collection-oriented languages, such as PARALATION LISP, CONNECTION MACHINE LISP (CM-LISP), and SETL, allow the creation of nested collections, collections whose elements are themselves collections. Nested collections are most conveniently manipulated through the nesting of parallel constructs, which is analogous to the nesting of serial loops. However, parallel and vector machines operate most efficiently when operating on long vectors of atomic values (flat collections). This paper describes techniques for translating nested parallelism into flat parallelism.

This paper discusses a compiler that translates a subset of PARALATION LISP into the instruction set of a virtual machine. The instructions of the virtual machine are simple vector instructions which could be further translated to a broad class of architectures including vector machines, single instruction parallel machines, and multiple instruction parallel machines. Another translator has also been implemented that translates the virtual machine instructions to instructions for the Connection Machine (CM-2), a massively parallel single instruction computer. The quicksort example in this paper has been executed on the CM-2.

Keywords: paralation lisp, scan, parallel prefix, compiler, Connection Machine.

## 1 Introduction

Collection-oriented languages, such as PARALATION LISP [8], CM-Lisp [11], SETL [9], NIAL [7], APL [5,6] or APL2 [1], are based on data structures which represent collections of elements, and operations for manipulating the collections as a whole, such as multiplying two arrays or taking the intersection of two sets. The collections can be unordered, such as *xets* in CM-lisp and *sets* in SETL, or ordered, such as *fields* in PARALATION LISP, *xectors* in CM-LISP *tuples* in SETL and *arrays* in NIAL, APL and APL2. The techniques discussed in this paper are relevant to both ordered and unordered collections. We believe that the collection-oriented languages are excellent languages for cleanly implementing a broad set of applications.

Conventional von Neumann languages, such as Pascal and Fortran, also support collections, usually in the form of an array data type. However, the *operators* of collection-oriented languages are different because they focus on manipulating collections as a whole. For example, multiplying all elements of a collection by some constant, sorting the elements, or summing all the elements into a single value are basic collection-oriented operations. Because the operations of collection-oriented languages operate on whole collections of elements, these languages tend to be much higher level than conventional languages. This typically leads to code that is clearer, easier to write and more concise[1]. Also, because the code does not include specifics on how to implement the various operations, it leads to code which is more easily mapped onto a broader set of architectures. On the other hand, collection-oriented languages have historically been hard to compile to run as efficiently on serial machines as conventional languages, and have therefore never gained great acceptance.

In this paper we cover some issues and techniques for compiling collection-oriented languages onto parallel and vector machines. We are particularly concerned with the languages that allow nested collections. In such languages, each element of a collection can itself be a collection. Of the above mentioned languages all but APL allow such nested collections. The compilation techniques mostly involve translating operations on nested collections into a flat

[1] MIT Artificial Intelligence Laboratory and Thinking Machines Corporation. Author's current address: MIT AI Lab, Cambridge, MA 02139

[2] Harvard University and Thinking Machines Corporation. Author's current address: Thinking Machines Corportion, 245 First Street, Cambridge, MA 02142

[1] Unfortunately, APL has given the computer community the impression that collection-oriented languages are difficult to interpret. This is not the fault of the semantics of the language but rather of the cryptic syntax.

form—operations on vectors of atomic values. When operating on nested collections, there are often two types of concurrency available: the concurrency available from the operation itself when applied to one of the nested collections, and the concurrency available since the operation can be applied to each collection independently. By translating into a flat form we can take advantage of both types of concurrency in a homogeneous way using a simple vector model with very little runtime support.

Although the techniques discussed in this paper could be used for many collection-oriented languages with nested collections, the techniques will be presented in the framework of a compiler for PARALATION LISP [8]. PARALATION LISP consists of a new data structure and a small set of operators that are added into COMMON LISP. The data structure, the field, is an ordered collection much like an array, but related fields are grouped into paralations to describe and exploit locality. The parallel computation operator, **elwise**, can be applied to any COMMON LISP code, and executes the code over all of the elements of a set of fields. **elwise** can be nested. The remaining two operators perform general communication between paralations. The first operator, **match**, encapsulates a communication pattern into a **mapping**. The second operator, <-, transfers a field according to a mapping.

The compiler translates to a language called Scan-Vector Lisp (SV-LISP). This language is a subset of COMMON LISP with the addition of a set of instructions for manipulating vectors of atomic values. These vector instructions can be broken into four classes: the *elementwise* instructions, the *permutation* instructions, the *scan* instructions, and the *vector-scalar* instructions. The *elementwise* instructions execute an elementary arithmetic or logical operation, such as $+, -, *$, or, and not, over the indices of the input vectors. The *permutation* instructions permute the elements of a vector based on another vector of indices. The *scan* instructions execute a prefix operation on a vector; the name scan comes from APL. The *vector-scalar* instructions are used to create vectors and to extract scalar elements from a vector.

Figure 1 illustrates the organization of the compiler discussed in this paper. The discussion of the compiler is broken into two main parts: the mapping of PARALATION LISP data structures onto SV-LISP data structures and the translation of PARALATION LISP code onto SV-LISP code. Figure 2 illustrates how the compiler fits into a larger system that compiles into the instruction set of an actual target machine, the CM-2. The particular vector instructions of SV-LISP were chosen because they can be further translated to a large variety of architectures. This allows the front end of the compiler to be ported to many computers, including vector machines such as the CRAY or Convex machines, single instruction parallel machines such as the CM-2, and multiple instruction parallel machines such as the Alliant. We have implemented a simple back end that compiles to the CM-2 instruction set.

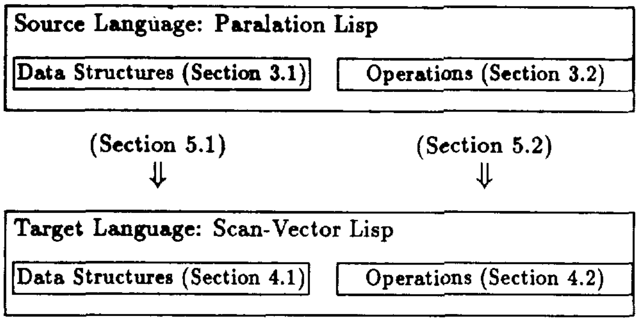

Source Language: Paralation Lisp
Data Structures (Section 3.1)
Operations (Section 3.2)
(Section 5.1)
(Section 5.2)
Target Language: Scan-Vector Lisp
Data Structures (Section 4.1)
Operations (Section 4.2)

Figure 1: Organization of the compiler. To translate PARALATION LISP onto scan-vector lisp both the data structures and the operations must be translated.

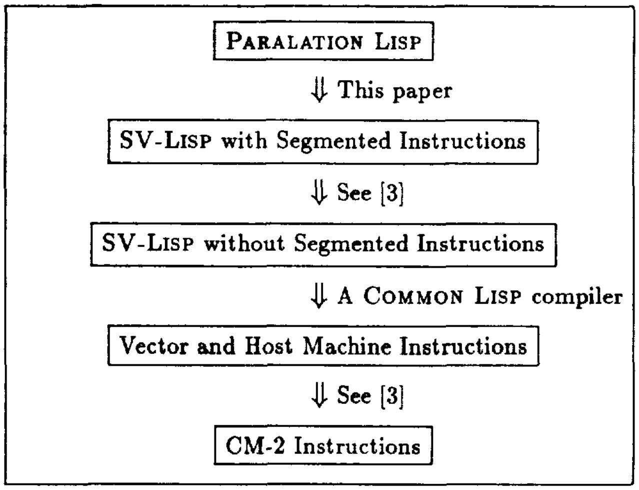

PARALATION LISP
This paper
SV-LISP with Segmented Instructions
See [3]
SV-LISP without Segmented Instructions
A COMMON LISP compiler
Vector and Host Machine Instructions
See [3]
CM-2 Instructions

Figure 2: This figure shows the organization of the full compiler that translates PARALATION LISP into CM-2 instructions. This paper focuses on the first step.

The purpose of implementing this compiler was to experiment with the ideas of compiling nested parallelism and to see what issues are involved in compiling a collection-oriented language onto a vector model. The compiler is therefore by no means complete: it only supports a small, but important, subset of PARALATION LISP. The compiler, however, produces very efficient output code. The CM-2 code generated by the quicksort example described in Section 2 is only a factor of two slower than the fastest sorting routine for the CM-2, and is faster than a version written in *LISP[12].

We will discuss some of the important techniques used by the PARALATION LISP compiler, including the compilation of nested parallel conditionals, and the compilation of nested parallel communication. The compilation of a quicksort algorithm will be used as the central example. The most interesting and important elements of the compiler are:

- The representation used for the PARALATION LISP collections, fields.

- The creation of two versions of each function, one to be called at top level and one to be called nested inside the PARALATION LISP **elwise** form.

- The stepping-up and stepping-down manipulations which are used when entering and leaving the PARALATION LISP apply-to-each form.

- The implementation of the *if* statement.

After a illustrative example (quicksort), the remainder of this paper defines the source language and the target language in detail and then describes the transformations performed by the compiler.

## 2 Quicksort Example

This section is an overview of the issues addressed by our compiler that uses quicksort as a concrete example.

The basic idea of quicksort [4] is to start with a sequence (an ordered collection) of values. If the sequence is already sorted, it returns the sequence; and if not sorted, it executes the following steps:

1. Pick a random pivot value from the sequence

2. Determine for each element whether it is less than the pivot or greater-or-equal to the pivot.

3. Split the set into two sequences such that all elements less than the pivot are in one set, and all elements greater-or-equal to the pivot are in the other.

4. Call the quicksort routine recursively on each sequence.

5. Append the two sequences returned from the recursive call.

There are two ways in which we can parallelize this routine. Firstly, when we split the set in two, we can execute the recursive call on each set independently and therefore in parallel. Secondly, the process of selecting a pivot, and then splitting based on the pivot can be executed in parallel. Ideally we would want to take advantage of both these kinds of parallelism, which are discussed in Section 2.1.

To take a closer look, we can step through the execution of the actual PARALATION LISP quicksort code. Only the results of the program will be discussed in this example; the language constructs are explained from a more general point of view in the next section. The program is written in a verbose style to facilitate line by line analysis (that is, temporary results are saved in named variables even though they are only needed once).

```
   ;; count the number of unique values in a field
   ;; of keys
1  (defun value-count (keys)
2    (length (<- keys :with #'arb
3                     :by (collapse keys))))
4
5  ;; sort a set of keys
6  (defun qsort (keys)
7    (if (> (value-count keys) 1)
8        (let* ((pivot-value
9                  (elt keys (random (length keys))))
10              (side (elwise ((key keys))
11                      (if (< key pivot-value)
12                          0
13                          1)))
14              (sub-data
15                (collect keys
16                         :by (match #f(0 1) side)))
17              (sorted-sub-data
18                (elwise (sub-data)
19                  (qsort sub-data))))
20         (expand sorted-sub-data))
21       keys))
```

The **value-count** function defined in lines 1–3 takes a collection of keys and uses a paralation library function to count how many unique values occur in it. When there is only 1 unique value, all of the values are the same, and are therefore properly sorted. This is the base case (the termination condition) of the recursive **qsort** function.

**qsort** begins by checking for its base case in line 7. If there is only 1 kind of value in **keys**, it does not need to be sorted, and line 21 returns **keys** as the result. In general, the test for the base case fails, and a random **pivot-value** is then picked. For example, if **qsort** is called on:

keys = [7 9 2 11 19 6 12]

**value-count** would have returned 7, and perhaps 9 would be selected as the random pivot value. Next, lines 10–13 calculate a flag value, **side**. Each value compares itself to the pivot value, returning 0 if it is less than the pivot and 1 otherwise:

side = [0 1 0 1 1 0 1]

In the next step, lines 14–16, paralation library functions are used to split the keys into two collections based on the two distinct values (0 and 1) in **side**. Each collection is nested in a larger collection:

sub-data = [[7 2 6] [9 11 19 12]]

At this point, although a nested collection is being used, they do not seem necessary. A flat vector of the split numbers, along with some representation of the lengths of the subcollections, captures the state information equally well. In fact, that is the basis of our compiler's representation of nested collections. Simply put, a collection of collections can be represented by two collections: one containing the data and another containing the lengths of the subsets. For example, **sub-data** can be represented with the following two flat vectors:

[7 2 6 9 11 19 12]
[3 4]

In the next step, it becomes clear why it is desirable to hide this representation from the user: Lines 17–19 contain a parallel recursive call that creates yet another level of collection nesting. For example, the recursive qsort call that operates on the left half might cause a split based on a pivot value of 7 while the other call might split based on a pivot value of 12:

call 1's sub-data = [[2] [7 6]]
call 2's sub-data = [[9 11] [19 12]]

Viewed together from the macroscopic viewpoint, the two parallel recursive calls operate on a collection of collections of collections, and the nesting of collections grows a level deeper with each recursion, even as the number of qsort invocations that are executing in parallel doubles. A programmer using a collection oriented language does not have to be concerned about such details, any more than a C programmer must worry about allocating automatic (local) variables on a stack. Our compiler, and programmers using vector models directly, must deal with these issues explicitly.

Inductively assuming that qsort works, we can return to line 20 of the top level call to tt qsort. It uses a library function to transform a collection of sorted subcollections:

sorted-sub-data = [[2 6 7] [9 11 12 19]]

into the final result by flattening the collection of collections into a single collection.

qsort result = [2 6 7 9 11 12 19]

## 2.1 Two Kinds of Parallelism

The parallelism involved in a pivoting and splitting can trivially be exploited in parallel by a vector model. The parallelism involved in the parallel recursive calls to qsort, although apparent to the programmer, cannot be directly expressed using flat vector operators. By flattening the nested collections, we can take advantage of both kinds of parallelism on flat parallel hardware.

We now consider how this quicksort can be implemented to run in parallel. We consider two types of parallelism. First, the value-count routine, the comparison of the pivot-value, the collect and the match can all be implemented in parallel as discussed later in Section 5.2. We call this the *intraroutine parallelism*. This type of parallelism seems natural for vector or SIMD architectures. The second kind of parallelism occurs on lines 18–19 where quicksort is called recursively twice. Each of these calls can run in parallel. We call this the *interroutine parallelism*. It seems more suited to coarse-grained MIMD architectures.

If we took advantage of intraroutine parallelism but ignored interroutine parallelism, the code would execute rapidly in the first stages, where the vectors of data to be sorted are large, but would become very inefficient in the final stages of quicksort. There would be many invocations of quicksort that would have to be run separately, and each would be operating on a vector that was small compared to the number of processors available. The original PARALATION LISP compiler and the CM-LISP compiler both took this inefficient, but easy to implement, approach.

On the other hand, one can imagine an implementation on a coarse grained machine that only took advantage of the interroutine paralleism. Such an approach would be efficient in the later stages of quicksort, but it would be inefficient in the early stages. Small numbers of processors would be responsible for operating on relatively large vectors.

An important goal of the compiler discussed in this paper is to take advantage of both kinds of parallelism in a efficient and simple way and map them onto a strictly SIMD model. The compiler does this and compiles the quicksort into a routine which has an expected complexity of $O(\lg n)$ calls to a set of simple data parallel operations such as permute, scan, and elementwise arithmetic and logical operations.

# 3 Source Code: Paralation Lisp

This section summarizes the PARALATION LISP language; for more details the reader should see [8]. PARALATION LISP consists of a new data structure, three primitive operators, and a set of other operators built on the primitive operators, all added to COMMON LISP.

## 3.1 Data Structures

The data objects permitted in PARALATION LISP are all the standard COMMON LISP data objects with one additional object, the *field*. The *field*, is a linear-ordered collection of elements. A field can be heterogeneous and the elements can be any PARALATION LISP value—including another field—thus allowing nested collections. Here are some examples of fields.

A homogeneous field:

#F(7 9 2 11 19 6 12)

A nested homogeneous field:

#F(#F(4 8 3) #F(9 1 12 7) #F(2 9))

A heterogeneous field:

#F(7 #F(4 Nil 3) T "horse")

A structure field:

$$\#F\left(\boxed{\begin{array}{l}u:u_{00}\\ v:v_{00}\end{array}}\ \boxed{\begin{array}{l}u:u_{01}\\ v:v_{01}\end{array}}\ \boxed{\begin{array}{l}u:u_{02}\\ v:v_{02}\end{array}}\right)$$

## 3.2 Operators

The three additional primitive operators are an iteration operator and two field operators. The iteration operator, elwise, is used to iterate any PARALATION LISP code, including another elwise, over all elements of a field. The two primitive field operators, **match** and **<-**, perform communication among the elements of fields: **match** encapsulates a communication pattern into a *mapping*, and **<-** transfers a field according to a *mapping*. Several other operations are supplied by PARALATION LISP but can be defined in terms of **match** and **<-**. All the COMMON LISP sequence functions can also be used on fields.

The operators of PARALATION LISP that are needed for the compilation examples are outlined below. The ideas behind paralations and mappings, which are both important concepts of the language, are not discussed because they are not germane to a discussion of compiler issues.

**Elwise:** The **elwise** operator is used to apply a body over each element of a field, or set of fields.

```
(elwise bindings
   body)
```

executes the body for each element of each field in the bindings and execute the body. So, for example, the form:

```
(elwise ((a A)
         (b B))
   (+ a b))
```

will pairwise add the elements of A and B and return a field of the results (the body in this example is simply a function call to +). Each binding of an **elwise** must be from the same paralation; the result returned by **elwise** is a new field in that same paralation. The **elwise** body can include any valid PARALATION LISP form.

**Match and Move:** The match operation takes two key fields as arguments; one from a source paralation and one from a destination paralation. It returns a *mapping*. A mapping can be thought of as a bundle of one-way arrows that connect certain sites of the source paralation to certain sites of a destination paralation. Two sites are connected if their key field values are equal. A mapping is an encapsulated communication pattern.

The <- (move) function accepts a mapping and a field from the source paralation of the mapping as its arguments. <- simply pushes this source data field into the tails of the mappings arrows, causing a field in the destination paralation to pop out at the other end of the mapping. The elements in this field are calculated based upon what arrived over the arrows. When several arrows leave a single source site, a concurrent read takes place. When many arrows collide at a single destination site, the multiple incoming values are reduced into a single value by repeatedly applying a user-specified, two-argument combining function. (Combining will not be needed for the examples presented in this paper, but it is an important part of the paralation model.) Finally, when a destination site receives no incoming values, a value is taken from a user-specified default field in the destination paralation.

**match** creates mappings which describe communication patterns; <- makes use of mappings, and includes the functionality of both concurrent read and combining. We now outline some of the operations that can be defined on top of the match and <- operations.

**Vref:** The **vref** operation "sums" the elements of a field according to any binary operator. So, for example:

```
(vref #F(7 4 1 11 2 6) :with 'max)
  ⇒  11
```

**Collapse and Collect:** The **collapse** operator takes a set of *keys* and generates a mapping in which all elements with equal valued keys are mapped to the same position. The **collect** operator takes a mapping and a field and appends all the elements which are mapped into the same position into a subfield. This can be implemented using a <- with a combiner of **concatenate**. The **collect** operator returns a field of fields. As an example of **collect** and **collapse** consider the following operation:

```
(let ((A #F(a0 a1 a2 a3 a4 a5))
      (B #F(k0 k1 k0 k2 k1 k1)))
  (collect A :by (collapse B)))
  ⇒  #F(#F(a0 a2) #F(a1 a4 a5) #F(a3))
```

**Expand:** The **expand** operator takes a field of fields and appends all the subfields into a single field. So, for example, **expand** applied to the *result* field given above returns:

```
(expand (collect A :by (collapse B)))
  ⇒  #F(a0 a2 a1 a4 a5 a3)
```

## 3.3 Restrictions

The compiler implements a small enough subset of PARALATION LISP that the subset is more concisely described by what it does include rather than what it does not include.

The subset only supports homogeneous fields and the data type of each elements of a field must be either an integer, boolean, field or structure. Since the elements can be fields, the subset supports nested fields. Many other data types, such as floating-point numbers or characters, would be easy to add but were left out for the sake of simplicity.

The subset supports the following sequence operations on fields: `elt, length, sort, reduce,` and `concatenate`. Other sequence operators would be easy to add but these were the only ones we needed for our test code.

The subset supports the three primitive operations of PARALATION LISP, **elwise**, **match** and **<-**. However, it only knows how to **match** integer and boolean keys. The subset supports most of the operations on integer and boolean values inside an **elwise**. It also supports nested operations on fields. For example, any of the above mentioned sequence functions can appear in an **elwise**. The only conditional the subset supports is the *if* special form, and it places the restriction that the results returned from both the then-expression and the else-expression must be of the same type. The reasons for these restrictions are discussed in [3].

# 4 Target Code: Scan-Vector Lisp

We now describe the target code of the compiler, SV-LISP (Scan-Vector Lisp) [3]. SV-LISP is a small subset of COMMON LISP with the addition of a new data type, the *pvector*, and a set of instructions for manipulating this data type, the *pvector instructions*. The pvector data type—a vector of atomic values—is much more primitive than the *field* data type of PARALATION LISP since it neither allows nested collections nor collections of structures. Likewise, the pvector instructions—which include pairwise adding the elements of two pvectors or permuting the atomic elements of a pvector—are more primitive than the match and <- instructions of PARALATION LISP.

Section 4.3 describes segmented versions of all the pvector instructions. The segmented versions break a vector into contiguous segments and operate independently within each segment and are used by the compiled code for manipulating nested fields. All the segmented versions of the pvector instructions can be translated into a small number of calls to the unsegmented versions [3]. This translation is implemented as a second phase of the compiler. The compiler first translates into SV-LISP with segmented pvector instructions, and then translates this into SV-LISP with only the unsegmented pvector instructions (see Figure 2).

Translating PARALATION LISP to SV-LISP rather than directly onto a parallel machine, such as the CM-2, has some important advantages. First, it separates the novel techniques of compiling collection-oriented languages onto a set of vector instructions from standard compiler techniques. The novel techniques, such as flattening nested parallelism, are used when translating from PARALATION LISP into SV-LISP while the standard techniques, such as compiling recursive routines, are used when translating from SV-LISP into actual machine instructions. Second, SV-LISP is relatively easy to port to new machines. Assuming a COMMON LISP compiler exists for a machine, PARALATION LISP can be ported to that machine simply by implementing a subroutine for each of the pvector instructions, and interfacing these subroutines into COMMON LISP.

The pvector data type and the pvector instructions of SV-LISP were selected because they can be implemented efficiently and straightforwardly on a broad variety of parallel and vector machines [3]. The pvector data type can be mapped onto a parallel machine by placing one element of the vector in each processor of the machine. If a vector is longer than the number of processors, multiple elements can be placed on each processor. The pvector instructions can then manipulate the pvectors in parallel: each processor manipulates one value.

## 4.1 Data Structures

COMMON LISP Data Types

Integers, Booleans, Structures

Additional Data Types

Integer-Pvectors, Boolean-Pvectors

Figure 3: The data types of SV-LISP.

SV-LISP has five data types. Three of these come from COMMON LISP: integers, booleans and structures. Two of them are new: boolean pvectors and integer pvectors. Pvectors are arbitrarily long linear-ordered collections of atomic values. The values are booleans in boolean pvectors and integers in integer pvectors.[2] Every pvector can have a different length and the only operations that can create or manipulate the pvector data types are the pvector instructions discussed in Section 4.2. If one was to implement a complete PARALATION LISP rather than the subset discussed in this paper, SV-LISP would need to be augmented with some other types such as floating-point numbers and floating-point pvectors.

## 4.2 Operations

Figure 4 lists the operations of SV-LISP. These operations are broken into two classes, operations from COMMON LISP and the pvector instructions. The COMMON LISP operations are defined in the COMMON LISP reference manual [10]. In this section we define the pvector instructions.

COMMON LISP Operations

Special Forms and Macros:
if, defstruct, defun, let, let*, progn, setq

Scalar Arithmetic and Logical Operations:
+, −, and, or, =, <, ...

Pvector Instructions

Elementwise Instructions:
p+, p−, p-and, p-or, p= , p<, p-select, ...

Permutation Instructions:
permute, select-permute

Scan Instructions:
+-scan, max-scan, min-scan, or-scan, and-scan

Vector-Scalar Instructions:
insert, extract, distribute, length

Figure 4: The operations of SV-LISP.

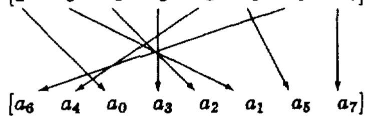
[2] The term pvector is used instead of vector so as not to confuse it with the COMMON LISP vector data type—a linear-ordered collection whose elements can be of any type.

**Elementwise Instructions:** Each elementwise instruction operates on equal-length vectors, producing a result vector of the same length. The element $i$ of the result is an elementary arithmetic or logical primitive—such as $+, -, *$, or or **not**—applied to element $i$ of each of the input vectors. For example:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| A | = | [5 | 1 | 3 | 4 | 3 | 9 | 2 | 6] |
| B | = | [2 | 5 | 3 | 8 | 1 | 3 | 6 | 2] |
| A p+ B | = | [7 | 6 | 6 | 12 | 4 | 12 | 8 | 8] |
| A p× B | = | [10 | 5 | 9 | 32 | 3 | 27 | 12 | 12] |

**Permute Instructions:** The permute instruction takes two vector arguments—a *data* vector and an *index* vector—and permutes each element in the data vector to the location specified in the index vector. For example:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| A (data vector) | = | $[a_0$ | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | $a_6$ | $a_7]$ |
| I (index vector) | = | [2 | 5 | 4 | 3 | 1 | 6 | 0 | 7] |



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| C ← permute(A, I) | = | $[a_6$ | $a_4$ | $a_0$ | $a_3$ | $a_2$ | $a_1$ | $a_5$ | $a_7]$ |

It is an error for more than one element to contain the same index; by definition a permutation is a one-to-one mapping. SV-LISP also includes a **select-permute** instruction which permutes elements between vectors of different lengths by masking out certain elements and placing defaults in certain positions.

**Scan Instructions:** A scan instruction executes a scan operation on a vector.[3] Scan is exceedingly useful in parallel algorithms and can be implemented very efficiently in parallel hardware [2].

The scan operation takes a binary associative operator $\oplus$ with identity 0, and an ordered set $[a_0, a_1, ..., a_{n-1}]$ of $n$ elements, and returns the ordered set $[0, a_0, (a_0 \oplus a_1), ..., (a_0 \oplus a_1 \oplus ... \oplus a_{n-2})]$. For example:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| A | = | [5 | 1 | 3 | 4 | 3 | 9 | 2 | 6] |
| +-scan(A) | = | [0 | 5 | 6 | 9 | 13 | 16 | 25 | 27] |
| max-scan(A) | = | [0 | 5 | 5 | 5 | 5 | 5 | 9 | 9] |

SV-LISP only includes +, **maximum**, **minimum**, **or** and **and** as operators for the scan instructions.

**Vector-Scalar Instructions:** SV-LISP includes four instructions that take both scalar and vector arguments. The **extract** instruction extracts a scalar value from a vector based on a scalar index. The **insert** instruction inserts a scalar value into a vector based on a scalar index. The **distribute** instruction generates a vector with a scalar copied across the whole vector. The **length** instruction returns the length of a vector. For example:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| A | = | $[a_0$ | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5]$ |
| V | = | v | | | | | |
| I | = | 3 | | | | | |
| L | = | 5 | | | | | |
| extract(A, I) | = | $a_3$ | | | | | |
| distribute(V, L) | = | [v | v | v | v | v] | |

### 4.3 Segmented Pvector Instructions

A vector can be partitioned into contiguous segments of elements by keeping a second vector which specifies the length of each segment. For example:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Vector | = | [5 | 1 | 3 | 4 | 3 | 9 | 2 | 6] |
| Segment Descriptor | = | [2 | 4 | 2] | | | | | |
| Segmented Vector | = , | [5 | 1] | [3 | 4 | 3 | 9] | [2 | 6] |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| A | = | [5 | 1] | [3 | 4 | 3 | 9] | [2 | 6] |
| B | = | [1 | 0] | [2 | 0 | 3 | 1] | [0 | 1] |
| I | = | [0 | 3 | 1] | | | | | |
| S (segment descriptor) | = | [2 | 4 | 2] | | | | | |
| +-scan(A, S) | = | [0 | 5] | [0 | 3 | 7 | 10] | [0 | 2] |
| permute(A, B, S) | = | [1 | 5] | [4 | 9 | 3 | 3] | [2 | 6] |
| extract(A, I, S) | = | [5 | 9 | 6] | | | | | |

Figure 5: Examples of the segmented versions of the pvector instructions.

The segmented versions of the pvector instructions take extra arguments that specify how the vectors are segmented and operate independently within each segment (see Figure 5). The segmented version of the permutation primitive bases its indices relative to the beginning of each segment so values permute *within* a segment. The segmented version of the scan instructions restart at the beginning of each segment. The segmented version of the elementwise operations are unchanged.

## 5 Translation

We now discuss how PARALATION LISP is translated into SV-LISP. In keeping with the rest of the paper, we first describe data structures and then describe operations.

### 5.1 Data Structures

In collection-oriented languages, different mappings of the high-level collections onto the target architecture can give rise to orders of magnitude differences in the efficiency of code on the architecture. A compiler must therefore pay special attention to how the mappings effect the efficiency of code. This section discusses how the compiler maps the collections of PARALATION LISP, fields, onto the primitive data structures of SV-LISP, pvectors. The mapping discussed allows a particularly efficient manipulation of nested fields by the vector instructions of SV-LISP. The representation of nested fields is based on segments as introduced

[3] The term *scan* is taken from APL [5]. Many other terms have been used for this operation but *scan* seems to be the first and most concise. The term *prefix computation* is often used in the computer science theory community.

in Section 4.3 and allows the generated code to operate over all subfields in parallel.

All fields are constructed from the *pfield* structure—a COMMON LISP structure type with two slots. The first slot stores a *segment-descriptor*, which describes the length or segmentation of the field (see Section 4.3).[4] The second slot stores the actual values. This slot contains a pvector if the field contains only atomic values, contains another pfield if the field is nested, and contains a user-defined structure if the field is a field of user-defined structures. We now discuss each of these cases. We only consider homogeneous fields since the subset of PARALATION LISP we are considering only supports homogeneous fields.

**Simple Field:** To represent a simple field—a field whose elements are all atomic—we use a single pfield structure. The first slot contains a definition of a single segment. The second slot contains a pvector with the values of the field. For example:

$$\texttt{\#F(}a_0\ a_1\ a_2\ a_3\ a_4\texttt{)}$$

$$\Downarrow$$

```
pfield
   segdes: [5]
   values: [a0 a1 a2 a3 a4]
```

We use a pfield structure instead of using a pvector directly since it allows us to check if two equal length fields belong to the same paralation.[5] Using the pfield structure is also useful for the stepping-up and stepping-down manipulations discussed in Section 5.2.

**Nested Field:** We represent a nested field—a field whose elements are themselves fields—by nesting the pfield structures and using segments of a single pvector to represent each subfield. For example:

$$\texttt{\#F(\#F(}a_{00}\ a_{01}\texttt{)}\ \texttt{\#F(}a_{10}\ a_{11}\ a_{12}\texttt{)}\ \texttt{\#F(}a_{20}\texttt{))}$$

$$\Downarrow$$

```
pfield
   segdes: [3]
              pfield
   values:       segdes: [2 3 1]
                 values: [a00 a01 a10 a11 a12 a20]
```

In this example, the *segdes* slot of the inner pfield describes the segmentation of the *values* slot. This technique can be applied recursively to represent a nesting of arbitrary depth. A field nested $n$ deep can be represented with $n$ segment-descriptor structures and $n$ pfield structures.

As mentioned in the introduction, the purpose of representing nested fields with a single value pvector is to get both the parallelism on operations within each bottom level field and the parallelism over all the bottom level fields.

**Structure Field:** We represent a structure field—a field whose elements are each a user defined structure—by pulling the structure out from inside the field. For example:

$$\texttt{\#F}\left(\boxed{\begin{matrix}u:u_{00}\\ v:v_{00}\end{matrix}}\ \boxed{\begin{matrix}u:u_{01}\\ v:v_{01}\end{matrix}}\ \boxed{\begin{matrix}u:u_{02}\\ v:v_{02}\end{matrix}}\right)$$

$$\Downarrow$$

```
pfield
   segdes: [3]
              uv-structure
   values:       u: [u0 u1 u2]
                 y: [v0 v1 v2]
```

In this example, the field of three uv-structures is mapped onto a single uv-structure whose slots contain a pvector with the values of all three of the original uv-structures.

Figure 6 illustrates a final example of a field with both nesting and structures.

A mapping is represented by a pair of pvectors that capture the site to site equality relationship defined by the original key fields supplied to `match`. The pvectors are canonicalized to make the use of the mapping by `<-` more efficient [8].

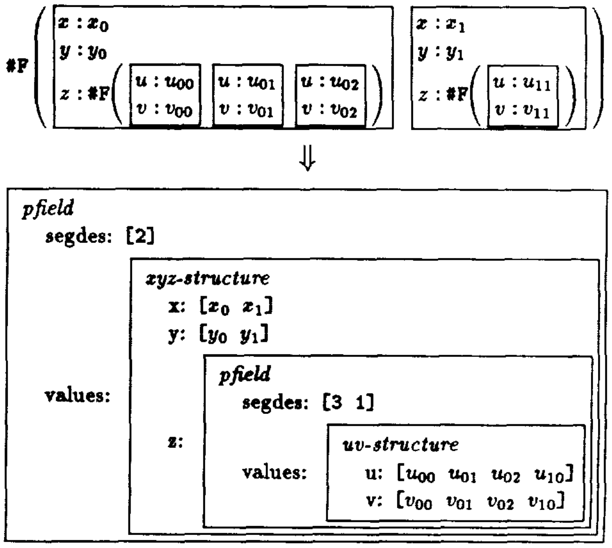


Figure 6: An example of how a nested field with structures is represented. In the example, the *xyz-struct* and the *uv-struct* are user-defined structures.

## 5.2 Operations

We now discuss the manipulations necessary to translate code from the subset of PARALATION LISP into SV-LISP. We break the discussion into four parts: compiling two versions of all code, one parallel and one serial; compiling the elwise form; compiling conditionals; and implementing the PARALATION LISP collection operations.

### Compiling Two Versions

When a function is defined in PARALATION LISP, it must work

[4] In the actual compiler the segment-descriptor contains several descriptions of the segmentation, each useful in different contexts. For the purposes of this paper, we assume the segment-descriptor only contains the lengths of each segment.

[5] Since the segment-descriptor is actually a structure, we can check if two fields are from the same paralation by seeing if the two segment-descriptors are `eql`.

both if called at top level (not within an elwise), and if called within an elwise. Consider the following example:

```
(defun plus-times (a b)
  (* a (+ a b)))
```

```
     Called at top level          Called within an elwise

(plus-times 5 2)              (elwise ((a #F(2 1 6))
 ⇒ 35                                  (b #F(3 5 2)))
                                 (plus-times a b))
                                ⇒ #F(10 6 48)
```

In the first case, the compiler uses a serial version of **plus-times** while in the second case the compiler uses a parallel version of the routine. The serial version uses the standard COMMON LISP + and * operations while the parallel version uses the pvector primitives p-+ and p-*. Figure 7 shows an example of the translation of a PARALATION LISP routine into the two SV-LISP routines.

```
(defun plus-times (a b c)
  (+ a (* b c)))
```

⇓

```
(defun s-plus-times (a b c)
  (+ a (* b c)))

(defun p-plus-times (a b c)
  (p-+ a (p-* b c)))
```

Figure 7: Compiling both a parallel and a serial version of a routine. The parallel version replaced all function calls with their parallel versions.

The compiler keeps two versions of every user-defined function and every function supplied by PARALATION LISP. To generate the parallel version of a new function, the compiler replaces each function call within the routine with its parallel version, and also executes some manipulations on the special forms. The most interesting of these manipulations is for the `if` special form and will be discussed in Section 5.2. The `let`, `let*` and progn special forms require no manipulations.

### Compiling Elwise Forms

The compiler applies several manipulations to translate an elwise form. First, it executes the same manipulations required when creating a parallel form of a function as discussed in the last sections. Second, it inserts code that copies all the free variables—variables that appear in the body but not in the binding list—across the elements of the elwise. Third, it inserts code that steps-down all the values bound in the binding list, and steps-up the result of the body.

We first discuss copying free variables. In PARALATION LISP, if a variable appears in the body of an elwise but not in the binding list, the variable is implicitly copied across the elements of the elwise. For example, in the form:

```
(let ((b 3))
  (elwise ((a #F(4 1 2)))
    (+ a b)))
 ⇒ #F(7 4 5)
```

the value of the variable b is implicitly copied across the three elements and added to each. When translating from PARALATION LISP to SV-LISP, the translator inserts code that execute this copy at run time. The particular code inserted depends on the type of value that needs to be copied. If the value is a scalar, the distribute pvector primitive (see Section 4.2) is inserted. Figure 8 illustrates an example of this manipulation. If the value is a structure of scalars, a distribute primitive is inserted for each slot of the structure. If the value is a field, a **distribute-segment** operation is inserted that creates a nested field with the original field in each element. The type of a variable can often be inferred at compile time so that the correct code can be inserted at compile time (in the above example b must be a scalar since we are adding it). If the type cannot be inferred at compile time, the compiler inserts code that executes a type dispatch at run time.

```
(elwise ((a A))
  (+ a b))
```

⇓

```
(simp-elwise ((a A)
              (b (distribute b (pfield-segdes A))))
  (+ a b))
```

Figure 8: An example of the code inserted for copying free variables. All free variables are removed by this manipulation. The **simp-elwise** *form is a version of* **elwise** that does not accept free variables.

We now discuss *stepping-down* and *stepping-up*. Stepping down and up are crucial to the implementation of operations on nested fields. Stepping-down consists of stripping off the top pfield from each value being bound in the elwise bindings, and setting a variable called the *current-segdes* to this value. So for a nested field, each time the field is passed inside another elwise another of its pfield structures is stripped off. Stepping-up is the inverse of stepping-down. When leaving an elwise stepping-up consists of tagging on a pfield structure to the result returned from the body of the elwise and restoring the value of the *current-segdes*. Figure 9 illustrates the code inserted by these manipulations.

```
(simp-elwise ((a A)
              (b B))
  (+ a b))
```

⇓

```
(let ((a (pfield-values A))
      (b (pfield-values B))
      (current-segdes (pfield-segdes A)))
  (make-pfield
    :segdes current-segdes
    :values (p-+ a b)))
```

Figure 9: An example of the stepping-down and stepping-up manipulations.

To see how stepping-down and stepping-up are used, we consider an example. Consider the following code:

```
(let ((field-of-fields
        #F(#F(7 4) #F(11) #F(8 1 17))))
  (elwise ((field field-of-fields))
    (elwise ((value field))
      (+ value value))))

⇒  #F(#F(14 8) #F(22) #F(16 2 34))
```

Based on the representation discussed in Section 5.1, the original field is represented as:

```
                  ┌─────────────────────────────────────┐
                  │ pfield                              │
                  │   segdes: [3]                       │
field-of-fields = │           ┌─────────────────────────┐│
                  │   values: │ pfield                  ││
                  │           │   segdes: [2 1 3]       ││
                  │           │   values: [7 4 11 8 1 17]││
                  │           └─────────────────────────┘│
                  └─────────────────────────────────────┘
```

When entering the outer elwise, the stepping-down code strips off the top pfield leaving:

```
        ┌─────────────────────────┐
        │ pfield                  │
field = │   segdes: [2 1 3]       │
        │   values: [7 4 11 8 1 17]│
        └─────────────────────────┘
```

And when entering the inner elwise, the next pfield is stripped off leaving:

value = [7 4 11 8 1 17]

Now when p-+ is applied to value, the result is:

[14 8 22 16 2 34]

When exiting the inner elwise the stepping-up code appends a pfield back on returning:

```
┌───────────────────────────┐
│ pfield                    │
│   segdes: [2 1 3]         │
│   values: [14 8 22 16 2 34]│
└───────────────────────────┘
```

And when exiting the outer elwise another pfield is appended returning:

```
┌─────────────────────────────────────┐
│ pfield                              │
│   segdes: [3]                       │
│           ┌─────────────────────────┐│
│   values: │ pfield                  ││
│           │   segdes: [2 1 3]       ││
│           │   values: 14 8 22 16 2 34││
│           └─────────────────────────┘│
└─────────────────────────────────────┘
```

Which is the representation of the desired result:

#F(#F(14 8) #F(22) #F(16 2 34))

In this example, the code that executes the addition runs in parallel over all elements therefore taking advantage of the parallelism within each subfield and also the parallelism among the subfields. This technique works regardless of the depth of the nesting and regardless of the complexity of the operations executed within the elwise.

One way of thinking about what is going on is that the compiler converts an elwise, which is a mapping of a function over many sets of data, into a new function over one larger set of data—the data sets all appended together. The effect of stripping off a pfield by the translated elwise is to remove a level of dividing boundaries and therefore effectively appending the data sets. So, in the example, inside the inner elwise there are no longer any dividing boundaries—all the original values are appended into one long vector.

### Compiling Conditionals

When creating a parallel version of a function, the functions in the body can be simply converted to calls to their parallel versions. On the other hand, the special forms do not obey the normal rules of function application and therefore cannot simply be replaced by a parallel function (for example, the if form only evaluates the second argument if the first evaluates to T). In this section we briefly discuss how the parallel version of the if special form is implemented by the compiler discussed in this paper. A more complete discussion can be found in [3].

The problem with the if special form is that some elements will take one branch while others will take the other branch. At run time, as long as there are some elements that want to take each branch, both branches must be executed. Since only a small number of elements might take one of the branches, it would be inefficient to execute the code in that branch over all elements (this assumes that it is cheaper to operate on shorter vectors, which is the case if multiple elements are placed on each physical processor). This can be particularly bad for nested conditionals. The compiler, therefore inserts code that packs the elements that take each branch into pvectors that only contain the elements which take that branch. After both branches are executed, the results are merged.

Figure 10 shows an example of the translation.

```
(if flag
    (func1 a)
    (func2 b))

        ⇓

(if (or-reduce flag)
    (if (or-reduce (p-not flag))
        (recursive-flag-merge flag
          (let ((a (recursive-pack a flag)))
            (func1 a))
          (let ((b (recursive-pack b (not flag))))
            (func2 b)))
        (func1 a))
    (func2 b))
```

Figure 10: Translating the parallel version of the if special form. If the flag is NIL in all of the segments, only func2 is executed. If the flag is T in all of the segments, only func1 is executed. If some flags are T and other NIL then the respective segments are packed before execution and merged after execution.

### Operations

There are several Paralation Lisp operations that need to be implemented. We only need to implement the unsegmented versions because we can use the same translator that generates parallel versions of the routines to generate parallel versions of the PARALATION LISP operations. The operations we have implemented include **length**, **elt**, **match**, **<-**, **expand**, **collect** and **collapse**. The **length** and **elt** are implemented with primitives length and extract. We now show the code needed to implement expand, omitting the others due to space limitations.

```
(defun expand (field)
  (let ((child-field (pfield-values field)))
    (make-pfield
      :values (values (pfield-values child-field))
      :segdes
      (make-segment-descriptor
        :lengths (vo::+-reduce
                  (segment-lengths
                    (pfield-segdes child-field))
                  (segment-head-flags
                    (pfield-segdes field)))))))
```

## 6 Summary

In this paper we have presented techniques for mapping the nested execution of parallel constructs into flat parallel models. Since actual parallel hardware is usually flat in nature, these techniques are useful to implementors of high level collection oriented languages. These languages express parallelism in a natural way that at first does not appear to be supportable by massively parallel machines. These techniques form a bridge from the convenience and power of nested collections and operations to the speed and simplicity of machines like the CM-2.

## References

[1] *APL2 Programming: Language Reference.* IBM, first edition, August 1984. Order Number SH20-9227-0.

[2] Guy Blelloch. Scans as primitive parallel operations. In *International Conference on Parallel Processing*, pages 355–362, IEEE Computer Society, 1987.

[3] Guy E. Blelloch. *Scan Primitives and Parallel Vector Models.* PhD thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, October 1988.

[4] C. A. R. Hoare. Quicksort. *Computer J.*, 5(1):10–15, 1962.

[5] Kenneth E. Iverson. *A Programming Language.* John Wiley & Sons, Inc, New York, 1962.

[6] Kenneth E. Iverson. A dictionary of APL. *APL Quote Quad*, 18(1):5–40, September 1987.

[7] Trenchard More. The nested rectangular array as a model of data. In *APL79 Conference Proceedings*, pages 55–73, ACM, 1979.

[8] Gary W. Sabot. *The Paralation Model: Architecture-Independent Parallel Programming.* The MIT Press, Cambridge, Massachusetts, 1988.

[9] J. T. Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL.* Springer-Verlag, New York, 1986.

[10] Guy L. Steele Jr., Scott E. Fahlman, Richard P. Gabriel, David A. Moon, and Daniel L. Weinreb. *Common Lisp: The Language.* Digital Press, Burlington, Massachusetts, 1984.

[11] Guy L. Steele Jr. and W. Daniel Hillis. Connection Machine LISP: fine-grained parallel symbolic processing. In *Proc. 1986 ACM Conference on Lisp and Functional Programming*, pages 279–297, ACM SIGPLAN/SIGACT/SIGART, Cambridge, Massachusetts, August 1986. Also available as Thinking Machines Technical Report 86.16.

[12] Thinking Machines Corporation. *The Essential *Lisp Manual.* Cambridge, Massachusetts, 1986.

# Implementation and Use of an Image Processing Algebra for Programming Massively Parallel Machines†

J.N. Wilson G.R. Fischer G.X. Ritter

Center for Computer Vision Research
Rm. E301 CSE Building
University of Florida
Gainesville, FL 32607

## ABSTRACT

Image Processing problems are often ideally suited to implementation on massively parallel machines. Providing a machine independent image processing language that can be readily targeted at massively parallel machines can be of great benefit in aiding researchers to use such machines. Such a language can free the user from having to learn the details of directly programming such a complex machine. We discuss the implementation and use of such a language, the AFATL (Air Force Armament Laboratory) image algebra, on a massively parallel machine. We introduce the problem of specifying image processing algorithms in a machine independent way, introduce the image algebra, provide an overview of how image algebra constructs are implemented in Connection Machine *lisp and provide examples of the use of image algebra for a variety of image processing operations. Finally, we discuss the generality, level of portability, and the efficiency of the existing implementation.

## INTRODUCTION

Massively parallel computers offer great potential benefits to end users. Sometimes, however, users who could benefit directly from the use of such massively parallel machines are reluctant to use them. In some cases the decision not to use a massively parallel machine is driven by the high cost of such architectures, but in other cases the primary motivation for not using massively parallel machine is a lack of understanding of the computing paradigm on which such architectures are based or a reluctance to tackle the problem of adapting current programs and programming techniques to a radically different architecture. To confuse the issue even further, different massively parallel machines have different architectures, constraints, host systems, and so forth, making the porting of programs prepared for massively parallel machines difficult.

One approach to solving this problem is to develop a general purpose high-level language that expresses massively parallel computations in a machine independent way. This solution suffers from the problem of being architecture class specific. Programs in such a language will not in general express algorithms in a way that easily permits them to be ported to different classes of architectures such as sequential or pipeline machines.

One can overcome this problem of architecture class dependence in a specific application domain by developing a high level application specific language that is architecture class independent. In this paper we discuss implementation and use of such a language, the AFATL (Air Force Armament Laboratory) image algebra [1,2,3], on a massively parallel machine. The architecture independence of this algebra is a result of its formal mathematical structure and method of development. Implementations of the image algebra have been developed for three kinds of processor architectures: sequential, vector parallel, and massively parallel. We discuss a massively parallel implementation in this paper.

In the next section, we introduce the image algebra. We follow that with an overview of a Common Lisp implementation of the image algebra on the CM2 Connection Machine. We then provide examples of the use of this image algebra implementation. We summarize with a discussion of the generality, level of portability, efficiency, and extensibility of the existing implementation.

## IMAGE ALGEBRA

The image algebra was designed to provide a mathematical system to support implementation, comparison, and analysis of image processing transformations. A number of architecture specific image processing notations have been developed in recent years [4], but programs developed in such languages are not easily ported to other architectures. Other image processing notations have been based on mathematical models of images [5], but most have not been demonstrated to be capable of describing certain classes of image processing transformations. The AFATL image algebra (henceforth referred to simply as image algebra) was developed with the intent of avoiding these pitfalls. The algebra was developed after carrying out a study of over 200 image processing algorithms. The operands and operations employed in these algorithms were analyzed to determine the fundamental components upon which they are built. The image algebra was then developed by extracting the relevant components discovered during that study.

The image algebra has been shown to be sufficient to express all image transformations over images with finitely many gray-values [6] as well as all image transformations described by finite program schemes [7]. Informally stated, these results mean that the image algebra can express any usable image transformation. Of greater interest, however, is the algebra's coherent set of operands and operations and the ease with which the algebra can be used to describe image

† This work supported in part by the U.S. Air Force and DARPA under contract F08635-84-C-0295, and was conducted using the computational resources of the Northeast Parallel Architectures Center (NPAC) at Syracuse University, which is funded by DARPA, under contract to Rome Air Development Center (RADC), Griffiss AFB, NY.

CH2649-2/89/0000/0587$01.00 © 1988 IEEE

transformations. In the rest of this section we describe the image algebra briefly, concentrating on those components most relevant to our discussion of massively parallel implementation of the algebra.

An *algebra* consists of a set of operands together with a set of finitary operations on those operands. We briefly describe the operands and operations comprising the image algebra in the following paragraphs. In the limited space provided, we are unable to give a complete presentation of the image algebra. The interested reader should refer to [2] for a more complete discussion.

The operands of the image algebra are values and values sets, coordinates and coordinate sets, images, and generalized templates. Informally speaking, a *value set* can be any set that might be used for image pixel values such as the reals ($\mathbf{R}$), integers ($\mathbf{Z}$), binary representations of subsets of the integers, and so forth. The *coordinate sets* of the image algebra are subsets of $\mathbf{R}^n$, that is, they are subsets of real n-space. In implementations of the image algebra, we restrict our attention to finite coordinate sets such as rectangular subsets of integral cartesian space $\mathbf{Z}^2$. We use symbols $\mathbf{X}$ and $\mathbf{Y}$ to represent coordinate sets, with elements $\mathbf{x}$ and $\mathbf{y}$ respectively. An *image* is simply the graph of a function from a coordinate set into a value set. The set of all real valued images over coordinate set $\mathbf{X}$ is denoted $\mathbf{R}^{\mathbf{X}}$.

The fundamental operations on images in $\mathbf{R}^{\mathbf{X}}$ are, for the most part, pixelwise induced operations on the reals. That is, functions $f:\mathbf{R}\rightarrow\mathbf{R}$ and $g:\mathbf{R}\times\mathbf{R}\rightarrow\mathbf{R}$ induce similar functions $\mathbf{R}^{\mathbf{X}} \rightarrow \mathbf{R}^{\mathbf{X}}$ and $\mathbf{R}^{\mathbf{X}}\times\mathbf{R}^{\mathbf{X}}\rightarrow\mathbf{R}^{\mathbf{X}}$, also denoted by $f$ and $g$, and defined by

$$f(\mathbf{a}) = \{(\mathbf{x},\mathbf{c}(\mathbf{x})) : \mathbf{c}(\mathbf{x}) = f(\mathbf{a}(\mathbf{x}))\}$$
$$g(\mathbf{a},\mathbf{b}) = \{(\mathbf{x},\mathbf{c}(\mathbf{x})):\mathbf{c}(\mathbf{x}) = g(\mathbf{a}(\mathbf{x}),\mathbf{b}(\mathbf{x})\}.$$

For example, $\sin(\mathbf{a}) = \{(\mathbf{x},\sin(\mathbf{a}(\mathbf{x}))): \mathbf{x} \in X \}$, and $\mathbf{a}*\mathbf{b} = \{(\mathbf{x},\mathbf{a}(\mathbf{x})*\mathbf{b}(\mathbf{x})):\mathbf{x}\in\mathbf{X}\}$.

In addition to these pixel-wise operations on images, several unary operations on images and image dot product serve to map real images into real values. The image sum operation, $\Sigma$, is defined on image $\mathbf{a}\in\mathbf{R}^{\mathbf{X}}$ as

$$\Sigma\mathbf{a} = \sum_{\mathbf{x}\in\mathbf{X}} \mathbf{a}(\mathbf{x}).$$

Besides these unary and binary operations on images, the Image Algebra supports what are called *generalized template* operations. Each of these operations takes as operands an image and a generalized template. The generalized templates formalize and extend the concept of *mask* or *template* entities used in neighborhood image processing algorithms.

Roughly speaking, if one wants to compute a generalized template operation on a real-valued image over coordinate set $\mathbf{X}$, giving as its result a real-valued image over coordinate set $\mathbf{Y}$, one uses a real-value template in the set $(\mathbf{R}^{\mathbf{X}})^{\mathbf{Y}}$ (the set of functions from coordinate locations in $\mathbf{Y}$ into real-valued images in $\mathbf{X}$. That is, if template $\mathbf{t}$ is an element of $(\mathbf{R}^{\mathbf{X}})^{\mathbf{Y}}$, then if $\mathbf{y}\in\mathbf{Y}$, $\mathbf{t}(\mathbf{y})\in\mathbf{R}^{\mathbf{X}}$, that is, $\mathbf{t}(\mathbf{y})$ is an image on $\mathbf{X}$. For convenience sake, we write $\mathbf{t}_{\mathbf{y}}$ to mean $\mathbf{t}(\mathbf{y})$. The image assigned to a result location is used to *weight* each of the values in the source image and then *gather* those values together to yield the result image value at that location. The weighting and gathering of values is specified by an operation taking an image and template as its operands and yielding an image result.

Although there are three fundamental image template operations in the image algebra, we consider only one of them, ***generalized convolution***, in this paper. The other image-template operations provided by the image algebra are ***multiplicative maximum*** and ***additive maximum***. Additive maximum generalizes the gray-scale morphology to non-rigid structuring elements [8], and multiplicative maximum provides a pseudo-linear operation with characteristics that are currently under exploration.

Given an image $\mathbf{a}$ on coordinate set $\mathbf{X}$, and a template $\mathbf{t} \in (\mathbf{R}^{\mathbf{X}})^{\mathbf{Y}}$, we define the generalized convolution of $\mathbf{a}$ with $\mathbf{t}$, written $\mathbf{a}\oplus\mathbf{t}$ as follows:

$\mathbf{a}\oplus\mathbf{t} \equiv \mathbf{c}$ where $\mathbf{c}\in\mathbf{R}^{\mathbf{Y}}$ and for all $\mathbf{y}$ in $\mathbf{Y}$, $\mathbf{c}(\mathbf{y}) = \Sigma(\mathbf{a}*\mathbf{t}_{\mathbf{y}})$.

In Figure 1, we show an image $\mathbf{a}$, that will be convolved with template $\mathbf{t}$ to yield a result image $\mathbf{c}$, that is,

$$\mathbf{c} = \mathbf{a}\oplus\mathbf{t}$$

Figure 2, shows how the generalized convolution operation is applied to determine the the value of $\mathbf{c}$ at the pixel location $\mathbf{y}_1$. First one obtains the image $\mathbf{t}_{\mathbf{y}_1}$, then one multiplies this image pointwise with the source image $\mathbf{a}$. The sum of the pixel values in this image are added together, the result providing the value of $\mathbf{c}(\mathbf{y}_1)$.

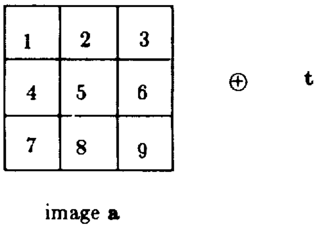


**Figure 1.**

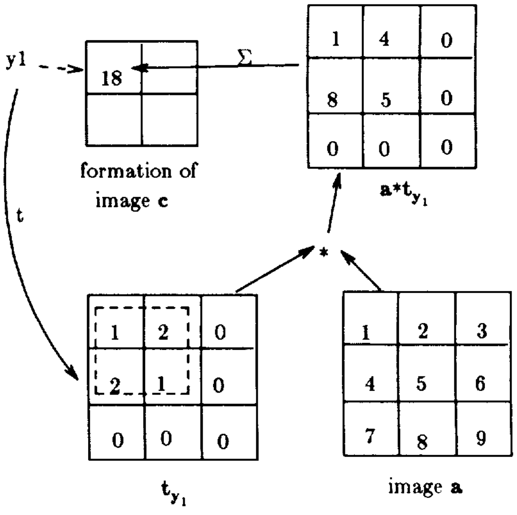


**Figure 2.**

Note that in using a template operations to compute the result value at a pixel location, such as $\mathbf{y}_1$ in Figure 2, one need only consider multiplication at the locations in the *support* of the image $\mathbf{t}_{\mathbf{y}_1}$, that is, the locations where $\mathbf{t}_{\mathbf{y}_1}$ is non-zero. The support of $\mathbf{t}_{\mathbf{y}_1}$, denoted $\mathcal{S}(\mathbf{t}_{\mathbf{y}_1})$, is highlighted in Figure 2. Thus, $\Sigma(\mathbf{a}*\mathbf{t}_{\mathbf{y}_1}) = \sum_{\mathbf{x}\in\mathcal{S}(\mathbf{t}_{\mathbf{y}_1})} \mathbf{a}(\mathbf{x})$. One can think of the template, then, as mapping each result image location into its source support. In this sense, then template operations of the image algebra are *backward driven*, since the result pixel location is first mapped to a support, then an operation involving the support and the image operand is performed to yield the result pixel value.

Many different formulations of neighborhood operations have been developed and implemented in computer systems. Such formulations may provide an operation similar to the image algebra image-template operation but restricted in some sense. One type of restriction is so prevalent we have given it a special name. A template **t** is said to be ***translation invariant*** if given **x**, **y**, and **z** satisfying **x**, $\mathbf{x}+\mathbf{z}\in\mathbf{X}$ and **y**, $\mathbf{y}+\mathbf{z}\in\mathbf{Y}$ we have $\mathbf{t}_\mathbf{y}(\mathbf{x}) = \mathbf{t}_{\mathbf{y}+\mathbf{z}}(\mathbf{x}+\mathbf{z})$, i.e. when translating the result location argument of the template, the template image yielded is identical (up to the same translation). Translation invariance is not required of image algebra templates. As shown in Figure 3, the example template **t** of Figures 1 and 2 may assign to point $\mathbf{y}_2$ an image completely different from that assigned to point $\mathbf{y}_1$.

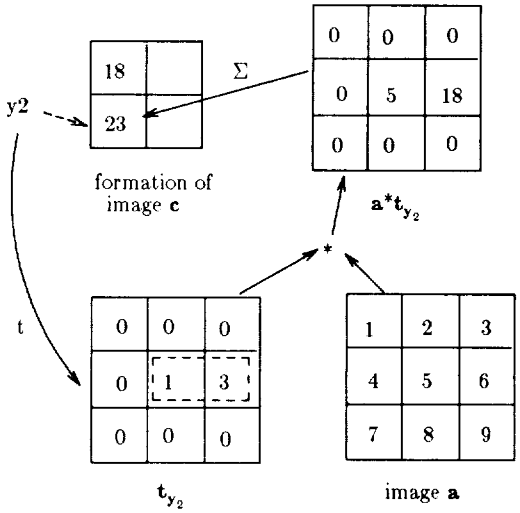


**Figure 3.**

One important generalization of templates is to provide for the definition of families of templates where each particular element can be distinguished by parameter values. If P is a set of parameter values, a real-valued *parameterized template* **t** with parameters in P is a function of the form

$$\mathbf{t}: \mathrm{P} \rightarrow (\mathbf{R}^{\mathrm{X}})^{\mathrm{Y}}.$$

The parameterized template maps its parameter into a normal template. The ability to define parameterized templates is of great benefit in describing complex image processing and vision algorithms. We discuss parameterized template definitions further in the next section.

The subalgebra of the image algebra providing only the operations of generalized convolution, image multiplication, and image sum is isomorphic to linear algebra [9]. Removal of the requirement for translation invariance provides the capability to express non-linear transformations such as warpings with image-template operations. The conceptual power of such template operations comes from the distinction drawn between the template operation, e.g. generalized convolution, and the assignment of images to result locations by the template.

## MASSIVELY PARALLEL IMPLEMENTATION

The image algebra described briefly in the preceding section, while not in any way limited to implementation on massively parallel processors, can easily exploit the power of such computer architectures. The obvious paradigm for parallel implementation of the image algebra is to equate coordinates in coordinate sets with processors in the machine architecture. Unary and binary image operations can then be implemented in a SIMD fashion. The implementation of image template operations, on the other hand, is somewhat more complicated.

Let us consider a model for implementing the expression $\mathbf{a}\oplus\mathbf{t}$. Our paradigm above indicates that each coordinate, whether in the source or result coordinate set, will be associated with some processor. As noted in the previous section, the template operations are backward driven, so we will consider the set of result processors as controlling the template operation. Consider the processor associated with result location **y**. This processor will need to compute the support of **t** at **y**, $\mathcal{S}(\mathbf{t}_\mathbf{y})$, then for each location **x** in $\mathcal{S}(\mathbf{t}_\mathbf{y})$ it must compute $\mathbf{a}(\mathbf{x})*\mathbf{t}_\mathbf{y}(\mathbf{x})$ and sum these values to determine the result value at **y**. If template **t** is translation invariant then such an operation is clearly SIMD. If **t** is more complex then SIMD implementation becomes more difficult.

We now discuss a language interface for providing the image algebra to users. Let us first note that the image algebra is not a programming language. In comparison to scalar arithmetic, one can consider the image algebra to be the arithmetic of images. We have implemented a subset of the image algebra as an extension to Common Lisp [10]. Our massively parallel implementation of the image algebra has been conducted on a CM2 Connection Machine using *lisp [11]. This provides us with several key benefits: we have the benefit of working in the rich development environment provided by Common Lisp; and all image algebra constructs described in this paper have been able to undergo parallel development in a uniprocessor implementation carried out in Kyoto Common Lisp on various Sun workstations. This exemplifies the architecture independence of the image algebra. Aside from execution speed, there is no way of distinguishing what underlying architecture the image algebra algorithmist is using.

Our implementation has undergone several stages and is continuing to evolve. In the current implementation a seamless extension of Common Lisp is provided by execution of an image algebra read function. The user interface provided by this function can distinguish between forms that employ image algebra constructs and execute appropriate code to implement those image algebra operations. The new data type `image` is provided by this interface. While images are represented in the CM2 implementation as *lisp *pvars*, the user of the image algebra need not be concerned with this detail. Common Lisp arithmetic forms are extended to have appropriate meanings when operating on images. Image algebra operations with no analogues in Common Lisp are provided via defined functions. It is interesting to note that many of these functions are already directly available in *lisp, indicative of the close relationship between the Connection Machines model of computation and the image algebra. Parameterized templates are defined with a form analogous to `defun` and are used in image-template operations. As implementation of unary and binary image operations is relatively straightforward, we will concentrate in the rest of this section on definition and use of templates.

As noted in the previous section, a parameterized template is a function mapping a parameter value into a template, which is a function from coordinates into images. One can, however, more compactly specify a template by considering only the support (non-zero portion) of the image it yields. This view leads to a significantly more efficient implementation and permits more compact specification of templates. Parameterized templates are declared much like Common Lisp functions with the special form **deftemplate**. The **deftemplate** form specifies the template name, its parameter names, the result location coordinates, and a template body which constructs the support of the template when given bindings for parameter values and result coordinate location. The special form **weight** is evaluated to yield a single pixel in the support. The forms that may appear in a template body under our current image algebra implementation are limited. Scalar arithmetic functions may be used, as well as predicate functions, **do**, **if**, **let**, **let***, **when**, and **weight**. The syntax for template definitions is given below:

```
deftemplate template ( {var}* )
    [ (coordinates {var}* ) ) ]
    body
```

The parameter names for the template are contained in the list following the template name, this is followed by an optional coordinate specification giving a sequence of names corresponding to the result location's coordinates, and the template body containing forms to assign weights to the support follows.

Templates are used in the context of image-template operations. The generalized convolution operation, ⊕, has the following form:

```
gcon image (template {parameter}*)
```

Note that the template argument is evaluated with its parameters, indicating that the specific template associated with the parameter values has been computed. The **weight** special form has the following form:

```
weight expression ( {expression}* )
```

and assigns the value of first expression to the source image support location addressed by the coordinates given in the list of expressions.

Perhaps a simple example will help illustrate the use of **deftemplate**. Consider a translation invariant template with no parameters whose support contains the result pixel location, its immediate right neighbor, and its immediate left neighbor, each with weight 1/3. Such a template can be defined as follows:

```
(deftemplate smooth ()
    (coordinates (y0 y1))
    (weight 1/3 (y0      y1))
    (weight 1/3 ((1+ y0) y1))
    (weight 1/3 ((1- y0) y1)))
```

Note that this template has an empty parameter list. Hence there is only one template in the family described by this template definition. Note also the coordinate specification naming the horizontal and vertical coordinates in the result image coordinate set. These pseudo-variables, **y0** and **y1**, will take on values corresponding to result image coordinates when the template is evaluated. In this simple example, the first **weight** form assigns weight 1/3 to the source coordinate point in the same location as the result coordinate. The second and third **weight** forms assign 1/3 to the source coordinate points corresponding to the nearest right and left neighbor locations to the first support location. This template might be used in a generalized convolution operation to horizontally smooth an image **a** as follows:

```
(gcon a (smooth))
```

The user interface presents image-template operations and templates in a fashion that is completely analogous to their image algebra definitions. The implementation of image-template operations on the CM2 is not, however, directly analogous to this interface. Let us look at how the generalized convolution of an image and template is implemented. Consider the evaluation of

```
(gcon image (template arg))
```

In our CM2 implementation, **template** will have been declared by evaluating a **deftemplate** form. Deftemplate is a macro yielding a **defun** of a like named function as its result. This function takes the following parameters:

(1) any parameters declared in the **deftemplate** specification,

(2) parameters specifying with image-template operation is to be performed, and

(3) an image.

Evaluation of the **gcon** form will dispatch the template function with the appropriate parameter values. In this case, the dispatched function is roughly equivalent to

```
(template arg 'gcon image)
```

The template function constructed as a result of evaluation of **deftemplate** computes the result image pvar in parallel on the CM2. The image-template operation determines what weighting and combining operations to perform as well as determining what value to use as a result upon finding an empty support. The evaluation of **weight** forms executes a weighting and combining operation. The result returned by the template function is the computed image pvar.

Several alternative implementations that more directly represent the semantics of the image algebra were considered but found to be too complex to efficiently implement on the Connection Machine. While the algebra specification of an image-template operation looks like a forward-directed operation, the template is implemented in a more backward-directed fashion, computing result locations by iterating over the support.

It is of interest to note that template bodies need not necessarily be transparently SIMD in nature. Consider the fact that **do** forms with initial and exit specifiers depending on the result location may be specified. This means that the number of elements in a template's support may vary from result location to result location. This has required the implementation of a general parallel do (***do**) structure. The final example in the section following uses precisely such a template.

## EXAMPLES OF USE OF IMAGE ALGEBRA ON THE CM

In this section we provide a few simple examples of the definition of templates in the Common Lisp image algebra notation and the use of these templates with the generalized convolution operation. It must be noted that the image algebra consists of much more than just a few convolution operations and complete algorithms for such applications as tracking and identification of objects can be implemented using the image algebra.

The first example we present is the smoothing template of the last section. The source template is as follows:

```
(deftemplate smooth ()
    (coordinates (y0 y1))
    (weight 1/3 (y0      y1))
    (weight 1/3 ((1+ y0) y1))
    (weight 1/3 ((1- y0) y1)))
```

The actual *lisp code generated for this template is listed here.

```
(*DEFUN SMOOTH (#:SMOOTH-SOURCE-IMAGE-13
            #:SMOOTH-SOURCE-WEIGHTING-OP-135
            #:SMOOTH-ACCUMULATION-OP-136
            #:SMOOTH-OFF-SOURCE-RETURN-VALUE-134)
 (*ALL
  (*LET
   ((#:SMOOTH-TARGET-IMAGE-132
        (!! #:SMOOTH-OFF-SOURCE-RETURN-VALUE-134))
    (X (SELF-ADDRESS-GRID!! (!! 0)))
    (Y (SELF-ADDRESS-GRID!! (!! 1))))

    (*SET #:SMOOTH-TARGET-IMAGE-132
      (*FUNCALL
        #:SMOOTH-ACCUMULATION-OP-136
        #:SMOOTH-TARGET-IMAGE-132
        (*FUNCALL
          #:SMOOTH-SOURCE-WEIGHTING-OP-135
          (!! 0.333)
          (PREF-GRID!!
            #:SMOOTH-SOURCE-IMAGE-133
            (ROUND!! (1-!! X)) (ROUND!! Y)
            :COLLISION-MODE :MANY-COLLISIONS
            :BORDER-PVAR
     (!! #:SMOOTH-OFF-SOURCE-RETURN-VALUE-134)))))
    (*SET #:SMOOTH-TARGET-IMAGE-132
      (*FUNCALL #:SMOOTH-ACCUMULATION-OP-136
        #:SMOOTH-TARGET-IMAGE-132
        (*FUNCALL
          #:SMOOTH-SOURCE-WEIGHTING-OP-135
          (!! 0.333)
          (PREF-GRID!!
            #:SMOOTH-SOURCE-IMAGE-133
            (ROUND!! X) (ROUND!! Y)
            :COLLISION-MODE :MANY-COLLISIONS
            :BORDER-PVAR
     (!! #:SMOOTH-OFF-SOURCE-RETURN-VALUE-134)))))
    (*SET #:SMOOTH-TARGET-IMAGE-132
      (*FUNCALL #:SMOOTH-ACCUMULATION-OP-136
        #:SMOOTH-TARGET-IMAGE-132
        (*FUNCALL
          #:SMOOTH-SOURCE-WEIGHTING-OP-135
          (!! 0.333)
          (PREF-GRID!!
            #:SMOOTH-SOURCE-IMAGE-133
            (ROUND!! (1+!! X)) (ROUND!! Y)
            :COLLISION-MODE :MANY-COLLISIONS
                 :BORDER-PVAR
     (!! #:SMOOTH-OFF-SOURCE-RETURN-VALUE-134)))))
    #:SMOOTH-TARGET-IMAGE-132)))
```

Figure 4 shows an image of an sr71 airplane, **sr71**. Figure 5 shows the result of evaluating the expression

```
(gcon sr71 (smooth))
```

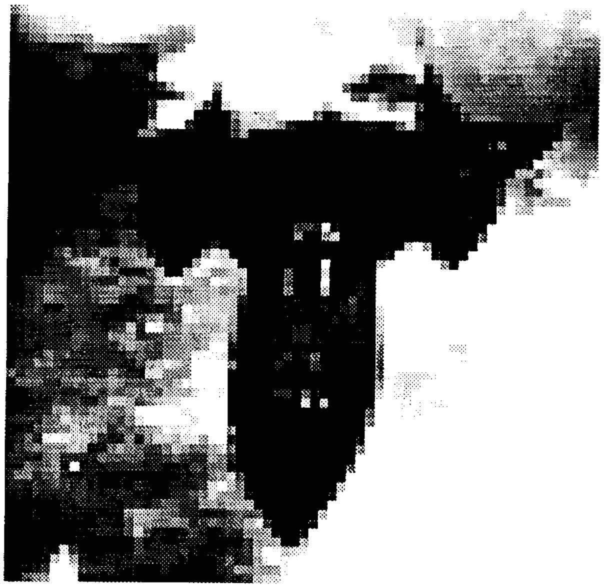


**Figure 4.**

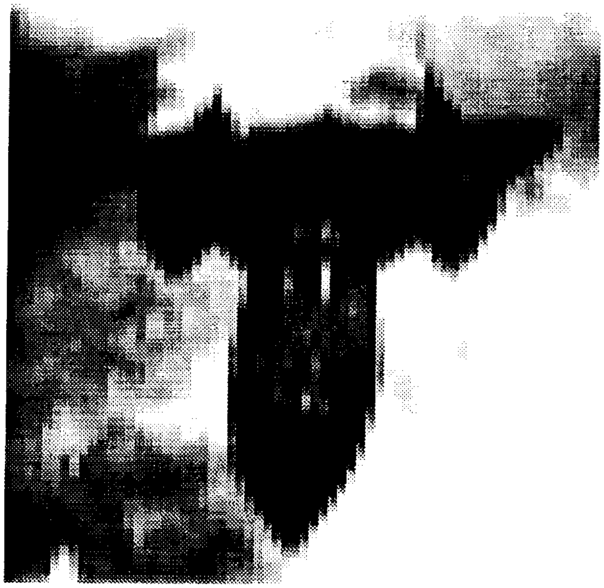


**Figure 5.**

The Second example demonstrates a translation variant template in which the support may be empty for some target pixel locations. The template we present can be used to cause all pixels within a given rectangular regions in an image to take on the value 0. The rectangular region to be set to zero is specified by template parameters.

ORIGINAL PAGE
BLACK AND WHITE PHOTOGRAPH

```
(deftemplate matte (x-low x-high y-low y-high)

  (coordinates (x y))
  (when (or (> x x-high)
            (< x x-low)
            (> y y-high)
            (< y y-low))
    (weight 1 (x y))))
```

Note that when evaluating the result of a generalized convolution operation if the template support is empty at some location **y**, then the result value at **y** will be 0. Any processor not executing a **weight** operation in evaluating a template will have empty support and hence be assigned this value. The matte template yields an empty support at each location inside the specified rectangle. Figure 6 shows the result of evaluating the expression

```
(gcon sr71 (matte 32 64 64 96))
```

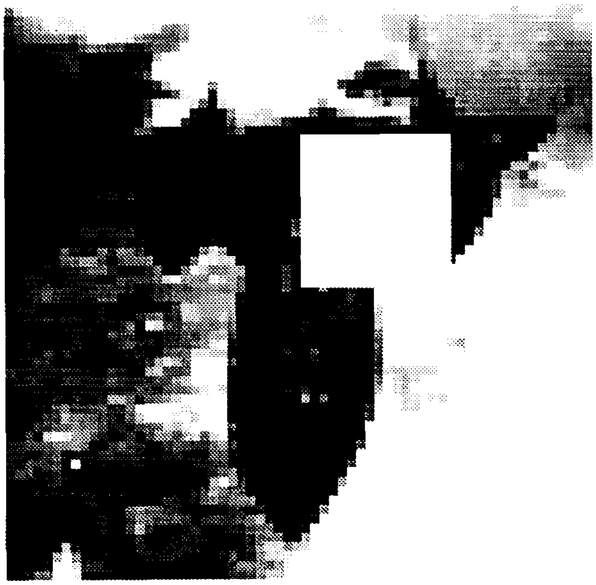


**Figure 6.**

The third example shows the definition of a template that can be used to achieve rotation of a source image. We demonstrate this technique with a simple nearest-neighbor rotation in which the nearest point to the ideal rotation source is chosen and placed in the target image. A more appropriate and physically accurate interpolation of the source gray-value could be described by suitable modification of the template definition. The template definition is as follows:

```
(deftemplate rotate (i j theta)
  (coordinates (y0 y1))
  (weight 1 ((+ (* (- y0 i) (cos theta))
                (* (- y1 j) (sin theta))
                i)
             (+ (- (* (- y0 i) (sin theta)))
                (* (- y1 j) (cos theta))
                j))))
```

The definition of the rotate template assigns to each result location a single support pixel which is the nearest neighbor to the inverse of the rotation specified. This, once again, brings up the backward-driven nature of the template definition. The template tells use where to find the source values associated with a result location. Other warpings, such as projective transformations can be specified with template operations in a similar manner.

Figure 7 shows the result of executing the following expression:

```
(gcon sr71 (rotate 64 64 0.436))
```

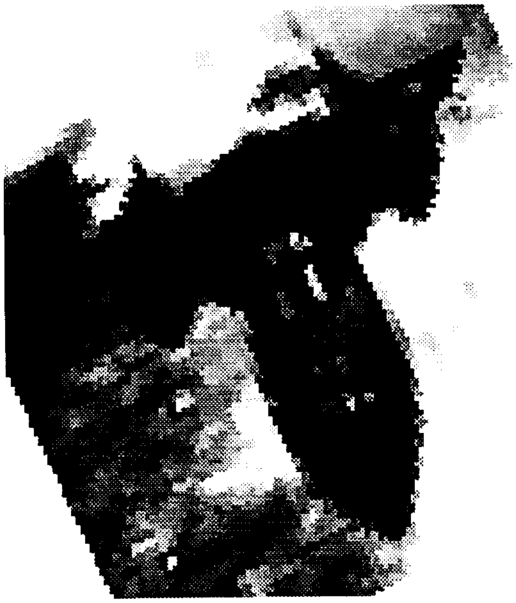


**Figure 7.**

Our final example demonstrates the versatility of templates by providing a translation variant shrinking template definition. The support at different points in this template have different positions, weights, and numbers of elements, as determined from the template arguments. Each result pixel **y** is assigned a support containing the location $\mathbf{y}^p$, where p is specified as a parameter to the template. The template definition is as follows:

```
(deftemplate power-shrink-y  (power)
  (coordinates (x y))
  (if (/= y 0)
      (do ((i (expt (- y 0.5) power)
              (1+ i)))
          ((> i (expt (+ y 0.5) power)))

        (weight 1 (x (expt y power))))))
```

The *lisp code generated for this template is shown here.

ORIGINAL PAGE
BLACK AND WHITE PHOTOGRAPH

```
(*DEFUN POWER-SHRINK-Y
 (#:POWER-SHRINK-Y-SOURCE-IMAGE-150
  #:POWER-SHRINK-Y-SOURCE-WEIGHTING-OP-152
  #:POWER-SHRINK-Y-ACCUMULATION-OP-153
  #:POWER-SHRINK-Y-OFF-SOURCE-RETURN-VALUE-151
  POWER)
 (*ALL
  (*LET
   ((#:POWER-SHRINK-Y-TARGET-IMAGE-149
(!! #:POWER-SHRINK-Y-OFF-SOURCE-RETURN-VALUE-151))
    (X (SELF-ADDRESS-GRID!! (!! 0)))
    (Y (SELF-ADDRESS-GRID!! (!! 1))))

   (IF!! (/=!! Y (!! 0))
    (*LET
     ((#:DO-PASSED-EXIT-P-155 T!!)
      (I (EXPT!! (-!! Y (!! 0.5)) (!! POWER))))

  (BLOCK NIL
   (TAGBODY
    #:DO-LOOP-TAG-BODY-154
    (*SET #:DO-PASSED-EXIT-P-155
     (AND!! #:DO-PASSED-EXIT-P-155
      (NOT!! (>!! I (EXPT!! (+!! Y (!! 0.5))
                            (!! POWER))))))
      (IF (NOT (*OR #:DO-PASSED-EXIT-P-155))
          (RETURN-FROM NIL))
      (*WHEN #:DO-PASSED-EXIT-P-155
       (*SET #:POWER-SHRINK-Y-TARGET-IMAGE-149
        (*FUNCALL
          #:POWER-SHRINK-Y-ACCUMULATION-OP-153
          #:POWER-SHRINK-Y-TARGET-IMAGE-149
          (*FUNCALL
      #:POWER-SHRINK-Y-SOURCE-WEIGHTING-OP-152
           (!! 1)
           (PREF-GRID!!
            #:POWER-SHRINK-Y-SOURCE-IMAGE-150
            (ROUND!! X)
            (ROUND!! I)
            :COLLISION-MODE :MANY-COLLISIONS
            :BORDER-PVAR
(!!
#:POWER-SHRINK-Y-OFF-SOURCE-RETURN-VALUE-151)))))
         (*SET I (1+!! I)))
     (GO #:DO-LOOP-TAG-BODY-154))))
   #:POWER-SHRINK-Y-TARGET-IMAGE-149)))
```

Note that this code has implemented the MIMD **do** of the template by iterating enough times to cover the range **(expt (- y 0.5) power)** to **(expt (+ y 0.5) power)** for any processor with address **y**. Figure 8 shows a simple line drawing of a skull, and Figure 9 contains the result of application of the following expression involving that image:

```
(gcon skull (power-shrink-y 1.1))
```

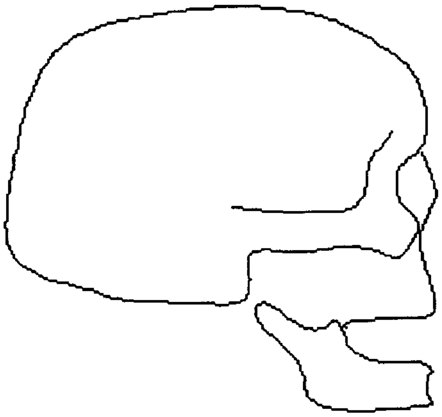


**Figure 8.**

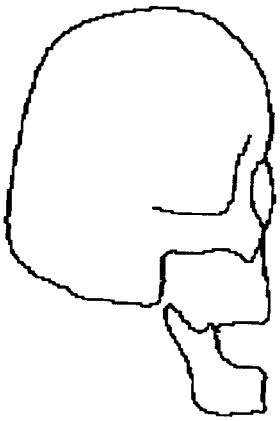


**Figure 9.**

## SUMMARY

As noted in the introduction, one can make massively parallel computers accessible to researchers in a specific application area by providing a domain specific user interface that will execute efficiently on such machines without requiring knowledge of the underlying architecture. In the case of the image algebra, this has been achieved by carefully choosing a set of operands and operations that, while capable of expressing all image processing transformations, is not dependent on the particular attributes of any special computer architecture or class of architectures. At the same time, the image algebra is clearly well suited to implementation on massively parallel machines such as the CM2.

The particular choice of Common Lisp for this implementation has provided great portability for user programs. The image algebra has been as easy to implement on uniprocessors as on massively parallel machines. The only differences discernible to the user are the dramatic difference in speed of execution of the developed algorithms and the different behaviors of systems when error conditions arise.

The current Connection Machine implementation, while relatively efficient, can be improved in several ways. Many of the constants appearing in template bodies could be assigned to pvars using `*let`. Templates could be implemented as functions that generate optimized versions of themselves when dispatched from particular image-template operations. A variety of data flow analysis techniques and peephole optimizations could be used to improve the code generated for templates. One example of where data flow analysis might pay off is in the substitution of front end control flow for distributed control and termination on execution of `do`'s with constant bounds. Another is in combining the weighting and combining operations into single *lisp operations, rather than sequences of operations.

Despite the rudimentary nature of the current implementation, the results we have seen are quite promising. Interpreted execution speed of image algebra code compares favorably with hand coded versions of similar algorithms and the implementation can be easily modified to add further functionality. Work continues on expanding the subset of the image algebra currently supported.

## REFERENCES

1. G.X. Ritter and J.N. Wilson, "The Image Algebra in a Nutshell," pp. 641-645 in *Proceedings of the First International Conference on Computer Vision*, IEEE Computer Society, London (June 1987).
2. G.X. Ritter, J.N. Wilson, and J.L. Davidson, "Image Algebra: an Overview," UF-CIS Technical Report TR-88-05, Dept. of Comp. and Info. Sci., Univ. of Florida, Gainesville, FL (May 1988).
3. G.X. Ritter and P.D. Gader, "Image Algebra Techniques for Parallel Image Processing," *Journal of Parallel and Distributed Computing* **4**(5) (March 1987), 7-44.
4. M. Duff and S. Levialdi, edit., *Languages and Architectures for Image Processing,* Academic Press, New York, NY (1981).
5. B. Lay, "MORPHOLOG: An Image Processing Software Package," pp. 463-468 in *IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, Miami Beach, FL (1985).
6. G.X. Ritter, M.A. Shrader-Frechette, and J.N. Wilson, "Image Algebra: A Rigorous and Translucent Way of Expressing All Image Processing Operations," in *Proc. of the 1987 SPIE Tech. Symp. Southeast on Optics, Elec.-Opt., and Sensors,* Orlando, FL (May 1987).
7. G.X. Ritter and J.N. Wilson, "Image Algebra: A Unified Approach to Image Processing," in *Proceedings of the SPIE Medical Imaging Conference*, Newport Beach, CA (February 1987).
8. J. Serra, *Image Analysis and Mathematical Morphology,* Academic Press, London (1982).
9. P. D. Gader, "Image Algebra Techniques for Parallel Computation of Discrete Fourier Transforms and General Linear Transforms," Ph.D. Dissertation, University of Florida, Gainesville, Fl. (1986).
10. G.L. Steele, *Common Lisp: The Language,* Digital Press, Maynard, MA (1984).
11. Thinking Machines Corporation, "*Lisp Reference Manual Version 4.0," unnumbered technical report (April 1986).

# MPP PASCAL

Tim Busse

Computer Sciences Corporation

## ABSTRACT

MPP Pascal is one of the high level languages available on the Massively Parallel Processor (MPP) at NASA's Goddard Space Flight Center. The MPP is a SIMD computer that obtains its computational power by employing 16,384 bit-serial processing elements to solve a single problem. MPP Pascal is a subset of the language Parallel Pascal, which extends the Pascal language by providing constructs for explicitly expressing parallelism in the form of array operations and functions. Access to several of the MPP's unique hardware features is available through language constructs that map directly onto the architecture of the machine. This paper presents the language MPP Pascal and discusses the implementation decisions pertinent to mapping parallel constructs onto the MPP. These issues are presented in order to provide insight into the design and development of future language systems for massively parallel computers.

## INTRODUCTION

The emergence of parallel computing has produced a variety of architecture designs capable of peak processing rates many times greater than that of the fastest sequential machines. Each new architecture requires new programming techniques to achieve the potential processing rates of that machine. However, to achieve peak processing rates, the parallelism of the problem must match the parallelism of the machine at the time of execution. Mapping a problem to a machine requires both suitable algorithms and suitable programming languages for expressing those algorithms. Without an appropriate algorithm, the computational power present through parallel processing is lost.

By design, high level programming languages mask the underlying hardware architecture to provide a machine independent method of representing algorithms. To support high level languages, compilers must be capable of manipulating instruction sequences to produce efficient code for parallel machines.

However, compiler technology does not offer the capability to transform programs written in a high level language into efficient executable code for parallel computation. Decomposing a problem so the inherent parallelism matches the machine is difficult even for experienced programmers. Current compiler technology cannot decompose serial code effectively. Even when an algorithm is expressed in a parallel language, compilers are not capable of efficiently partitioning data and mapping communications.

Attempts at creating parallel programming languages and environments have not succeeded in eliminating the limits of compiler technology. This inherent problem prevents the parallelism of the problem and the parallelism of the machine from being in balance at execution time.

To create the proper balance between a programming language and the Massively Parallel Processor (MPP), MPP Pascal was designed to take advantage of the hardware features of the MPP. This paper discusses the language MPP Pascal and issues pertinent to its implementation. A brief overview of the MPP is presented prior to discussing the language and implementation.

## THE MACHINE

The MPP is a single-instruction/multiple-data stream (SIMD) computer consisting of 16,384 simple, bit-serial processing elements (PE's). Each PE has 1,024 bits of local memory and the ability to communicate with its four nearest neighbors. The PE's can be logically thought of as a 128 by 128 array. The array of PE's is controlled by the array control unit, which is comprised of three separate processors that run asynchronously: the PE Control Unit (PECU), the

CH2649-2/89/0000/0595$01.00 © 1988 IEEE

Main Control Unit (MCU) and the Input Output Control Unit (IOCU) (Figure 1).

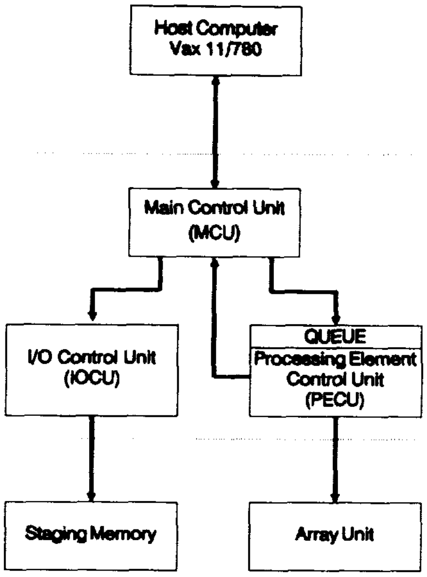


Figure 1. Architecture of the Array Control Unit

The PECU broadcasts instructions to the PE's, which operate in lockstep on data held in their local memories. Since the PE's do not have the ability to modify memory addresses issued by the PECU, each instruction in the array is performed on a bit-plane operand (16,384 bits). All arithmetic and relational operations are implemented as subprograms, and need to be built from basic single bit instructions.

Connected to the PECU through a set of control register queues is the Main Control Unit (MCU). The MCU coordinates all processors in the MPP, and invokes operations in the PECU by issuing requests through the register queues. The MCU was designed for high speed, 16-bit integer calculations, and is similar in architecture to a reduced instruction set computer (RISC). As the master controller, the MCU communicates with the other processors and initiates all processes. Moreover, all scalar operations are carried out in the MCU.

When the MCU requests an I/O operation, the IOCU responds by initiating a data transfer either between the array and staging memory, or the staging memory and the host Vax computer. The stager is a permutation device with a large internal memory. Its primary function is to rearrange data as it moves between the host memory and array, so the internal data representations are correctly formatted for the target memory. This process is commonly referred to as corner turning.

The MPP is connected to a host Vax computer, which provides the entire program development environment. Access to the MPP is controlled by programs running in the Vax, and all I/O operations during program execution are handled by the front end.

## THE LANGUAGE

MPP Pascal evolved from the language Parallel Pascal as it was designed by A.P. Reeves [1]. Parallel Pascal introduces extensions to standard Pascal for advanced architectures (i.e., vector and parallel computers) by providing language constructs for explicitly expressing parallelism in the form of array operations and functions.

Original efforts at NASA's Goddard Space Flight Center focused on implementing Parallel Pascal, as defined by Reeves, on the MPP. Various problems with the implementation of Parallel Pascal led to a machine specific version of the language, called MPP Pascal.

MPP Pascal is a subset of the language Parallel Pascal. The semantics of several Parallel Pascal constructs are redefined to match the parallelism of the machine. In addition, a new keyword was added, and several language constructs removed. The differences between the two languages are directly attributed to the architecture of the MPP and provide direct access to hardware features that were masked by Parallel Pascal.

The most significant modification to Parallel Pascal is the semantic definition of the keyword parallel. In Parallel Pascal, the keyword parallel can only be used in conjunction with data of type array, and is treated as a compiler directive informing it of the intended use of an array.

In MPP Pascal, the keyword parallel array refers to a primitive data item that matches the parallelism of the MPP exactly (i.e., a 128 by 128 array) (Table 1). A parallel array may have an associated type of integer, real or Boolean, and is manipulated by the same primitive operators as other Pascal primitive data types (+, -, *, /, div, mod, and, or, not, <, >, <=, >=, =, <>, :=) and primitive functions (abs, exp, ln, sqr, sqrt, trunc, round, odd, pred, succ, sin, cos, arctan). The operators act on parallel arrays on an element by element basis. For example, the addition of two parallel arrays result in a parallel array, where each element is the sum of the corresponding elements of the operands.

TABLE 1: Syntax for MPP Pascal Extensions

```
Parallel Arrays
   type <identifier> = parallel array [{<index type>,}
      0..127,0..127] of <element type>;

Stager Arrays
   type <identifier> = stager array [{<index type>,}
      0..127,0..127] of <element type>;

Where Statement
   where <mask expression> do <statement>
or
   where <mask expression> do <statement>
      otherwise <statement>
```

Given the semantic meaning of parallel array, it is not possible to access individual elements through indexing operations. Therefore, the where-do-otherwise construct was introduced to provide a mechanism to selectively process elements of a parallel array. Similar to the if-then-else construct, which causes conditional execution, the where-do-otherwise statement causes conditional assignment. All assignment statements in a where block selectively store data based on a logical bit-plane mask.

A syntactic construct that was added to the language MPP Pascal, which is not part of Parallel Pascal, is the keyword stager. Similar to the keyword parallel in Parallel Pascal, the keyword stager in MPP Pascal can only be associated with arrays, and is treated as a compiler directive indicating the memory location of the specified variable. The only valid functions for stager arrays are get, put, swap and transfer. These functions move data between the staging memory and array memory or between staging memory and disk files maintained by the host Vax computer. Stager arrays cannot not be part of any arithmetic expressions. The stager can be viewed as either temporary memory or as an I/O buffer.

Other constructs shared by both MPP Pascal and Parallel Pascal are array manipulation functions. In the case of MPP Pascal, these functions are only defined for parallel array data items. The reduction functions (sum, prod, max, min, all, any) operate on a parallel array and produce a scalar result of the appropriate type (Table 2). The transformation functions (shift, rotate) move data elements to neighboring locations.

TABLE 2: Predefined Functions for Parallel Arrays

| | | |
|---|---|---|
| Reduction | | |
| | sum | arithmetic sum |
| | prod | arithmetic product |
| | all | boolean and |
| | any | boolean or |
| | max | arithmetic maximum |
| | min | arithmetic minimum |
| Permutaion | | |
| | shift | end-off shift, zero fill |
| | rotate | end-around shift |

As mentioned previously, various data types and statement constructs were omitted from the initial implementation of MPP Pascal. These items were not needed to support the scientific applications that were being developed on the MPP, and could, therefore, be implemented at a later date. The pointer and set data types were omitted along with operations and operators that were specifically included for use with data of these types. In addition, the case statement and recursion are not supported.

## THE IMPLEMENTATION

The architecture of the machine suggests a decomposition of the executable image that maps well to the environment of the MPP. The language system is comprised of three integrated components: the primitive library, the I/O run-time library, and a compiler that generates code for the MCU. Each of these components corresponds to the three processors available in the MPP's array control unit (Figure 2).

The primitive library contains the set of subprograms required to support the arithmetic, relational and function operations for parallel array data. All parallel operations are implemented through these primitives. The subprograms associated with these parallel operations execute in the PECU. To perform the sequence of parallel opera-

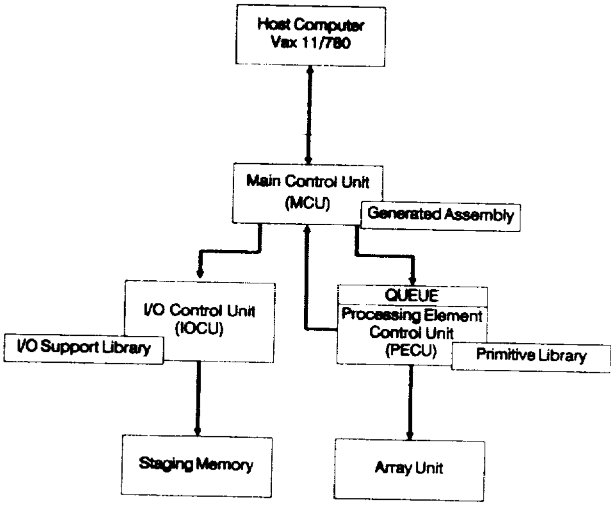


Figure 2. Architecture of MPP Pascal

tions specified by a program, a series of primitive subprogram calls are invoked. As mentioned previously, the MCU initiates PECU operations, and therefore, must be programmed to initiate array operations that correspond to the semantics of the user program.

To improve efficiency, parallel arrays of type integer are stored and manipulated based on their required bit length. In Pascal, the program can specify a subrange of acceptable values for integer variables. MPP Pascal uses this capability to reduce memory requirements in the array and reduce execution time. Given the bit-serial nature of the array, processing time is greatly reduced for shorter integer representations.

The I/O run-time support library provides the subprograms needed to perform I/O operations in the MPP. Subprograms are available that transfer data between the MPP and host Vax computer (get and put), as well as providing routines to move data between the array and stager memories (transfer and swap). All I/O is initiated by the MCU and responded to by the IOCU. The asynchronous processing capability of the three array control units allows for I/O, array and scalar operations to execute simultaneously. MPP Pascal utilizes this capability by initiating PECU and IOCU processes and continuing with MCU computations.

The final component required for executing a program on the MPP is the control program, which resides in the MCU. The MCU program is the only component that varies from program to program. A compiler transforms MPP Pascal programs into assembly language for the MCU in a two step process. The first step translates MPP Pascal into Parallel Pcode, and the second step transforms Parallel Pcode into MCU assembly language.

Parallel Pcode is an intermediate language for a hypothetical stack machine. Parallel Pcode is an extended version of Pcode, which is the intermediate form produced by the original Pascal compiler. In fact, the MPP Pascal compiler is a modified version of the original Pascal P4 compiler. The original compiler was modified to accept the parallel constructs of MPP Pascal and produce the new intermediate form. The compiler performs all syntax analysis.

The second step in the compilation process is code generation. The Parallel Pcode is processed and converted to assembly language. The MCU assembler representation of a program includes calls to the primitive library, I/O library, and scalar instructions. Experience showed that the speed of the MCU was insufficient to keep the array busy when short operands are involved in a sequence of parallel operations. The overhead associated with calculating address and executing the primitive call exceeded the time required to perform the operation. To eliminate much of the overhead required to calculate addresses for parallel operations, all address calculations are done at either compile time or at the beginning of an executable process.

This modification to the language Parallel Pascal led to the elimination of recursion and argument passing by value. The difference in execution time between code generated by the MPP Pascal and Parallel Pascal compilers on the MPP can be attributed mainly to the reduction in address calculations. The overhead associated with address calculations even affects the computational speed of complex operations, such as floating point arithmetic. Thus, recursion is not addresses are allocated statically by the code generator.

The differences in computation time between Parallel Pascal and MPP Pascal justify the changes made to the language. The potential power of the MPP is easier to harness with the modified language, since the architecture is not hidden by the language. Originally, much effort was put into writing a compiler that generates efficient code for the machine. However, the compiler itself could not compensate for the speed of the scalar MCU processor. Therefore, the language needed to be modified to help reduce the overhead associated with a high level language such as Parallel Pascal.

## CONCLUSION

The implementation of MPP Pascal utilizes the architecture of the machine to provide an efficient mapping from program syntax and semantics to program execution. Moreover, it combines the functionality of a high level language and the characteristics of the architecture to provide an efficient implementation environment for applications.

Other computer architectures would, in general, require a different implementation or language constructs than those needed for the MPP. MPP Pascal demonstrates the close association needed between hardware and a programming language for generating highly efficient executable code for a parallel computer. Until programming tools are capable of utilizing parallel hardware technology, language implementations similar to the one described for MPP Pascal will be prevalent in high speed scientific computing.

## REFERENCES

[1] A.P. Reeves. "Parallel Pascal: An Extended Pascal for Parallel Computers." Journal of Parallel and Distributed Computing, vol. 1, pp. 64-80, 1984.

# A FORMAL MODEL FOR SIMD COMPUTATION

M.D. Rice S.B. Seidman P.Y. Wang

Center for Parallel Computation
Department of Computer Science
George Mason University
Fairfax, Virginia 22030-4444

## ABSTRACT

A formal model for SIMD computation is presented that captures the essential operating features of current SIMD computers, yet which allows for extensions and variations of existing architectures. The fundamental components of the model are a host computer, a set of processsing elements $\mathcal{P}$, a set of control units $\mathcal{C}$, a set of input/output controllers $\mathcal{IO}$, and a set of external devices $\mathcal{D}$. Each component sends or receives data or instructions to/from other components. The communication patterns among the model components are described by six networks, each of which governs the communication between a single pair of components. The networks are represented as functions or collections of functions with formally specified mathematical properties that have natural interpretations in the context of SIMD computation. Using the functional approach, a set of four assumptions for SIMD computers is proposed, and consequences of these assumptions are explored.

Keywords: SIMD computation, computational model

## 1. INTRODUCTION

During the past forty years, the von Neumann model of computation has been used to describe a wide variety of sequential computers. The major components of this model are an arithmetic-logic unit, a control unit, memory, and an input/output system. The simplicity and generality of the von Neumann model is largely responsible for the creation of programming languages and systems software that can be transported between computers with different underlying architectures.

Parallel computers can be obtained by replacing the individual components of the von Neumann model by multiple units that may interrelate in different ways. For example, a parallel computer with multiple ALUs can have memory that is shared between all of the processors or is distributed among them. Input and output may be performed by only one distinguished processor or by all of the processors. Communication between the processors may be accomplished by explicit interprocessor channels, or through a bus or switching network.

Rather than attempting to extend the von Neumann model to parallel computers in general, these computers can be classfied by considering the way in which data and control flow in a parallel computation. From this perspective, the fundamental distinction is between single-instruction, multiple-data-stream (SIMD) computers and multiple-instruction, multiple-data-stream (MIMD) computers. In SIMD computers, the execution of a program can be regarded as the execution of the same instruction on a number of independent data streams. The Connection Machine System, the Distributed Array Processor, and the Massively Parallel Processor are examples of such SIMD machines. In MIMD computers, the processors typically execute mutually independent programs on independent data streams. The Intel iPSC hypercubes and INMOS Transputer networks are examples of MIMD computers.

In this paper, a formal model is proposed for SIMD computation. At a minimum, such a model must include a control unit to manage the computation, a collection of N processing elements, and an interconnection network that allows data to be routed between processors. In a seminal paper (Ref. 7), SIMD computers are modeled in just this way. Although Siegel uses his model to investigate important relationships between the organization of the processing elements and the structure of the interconnection network, the model does not easily handle parallel input and output, nor is it able to describe SIMD computers with more than one control unit.

The intent of the SIMD model is to capture the essential features of computation on currently available SIMD machines, while allowing for potential extensions and variations of current architectures. Using the model, it is possible to represent the fundamental concepts of SIMD (or *data parallel*) computation, including virtual processor topologies, processor selection (masking), and data routing between processors. Furthermore, the semantics of any high-level SIMD programming language can be expressed in terms of the model, which thus supports the development of machine-independent software.

## 2. BASIC ORGANIZATION OF THE MODEL

The SIMD computational model is defined formally as a collection of basic components and networks: $\mathcal{M} = <\mathcal{H},\mathcal{C},\mathcal{P},\mathcal{D},\mathcal{IO},\mathcal{N}>$. The basic components of the model

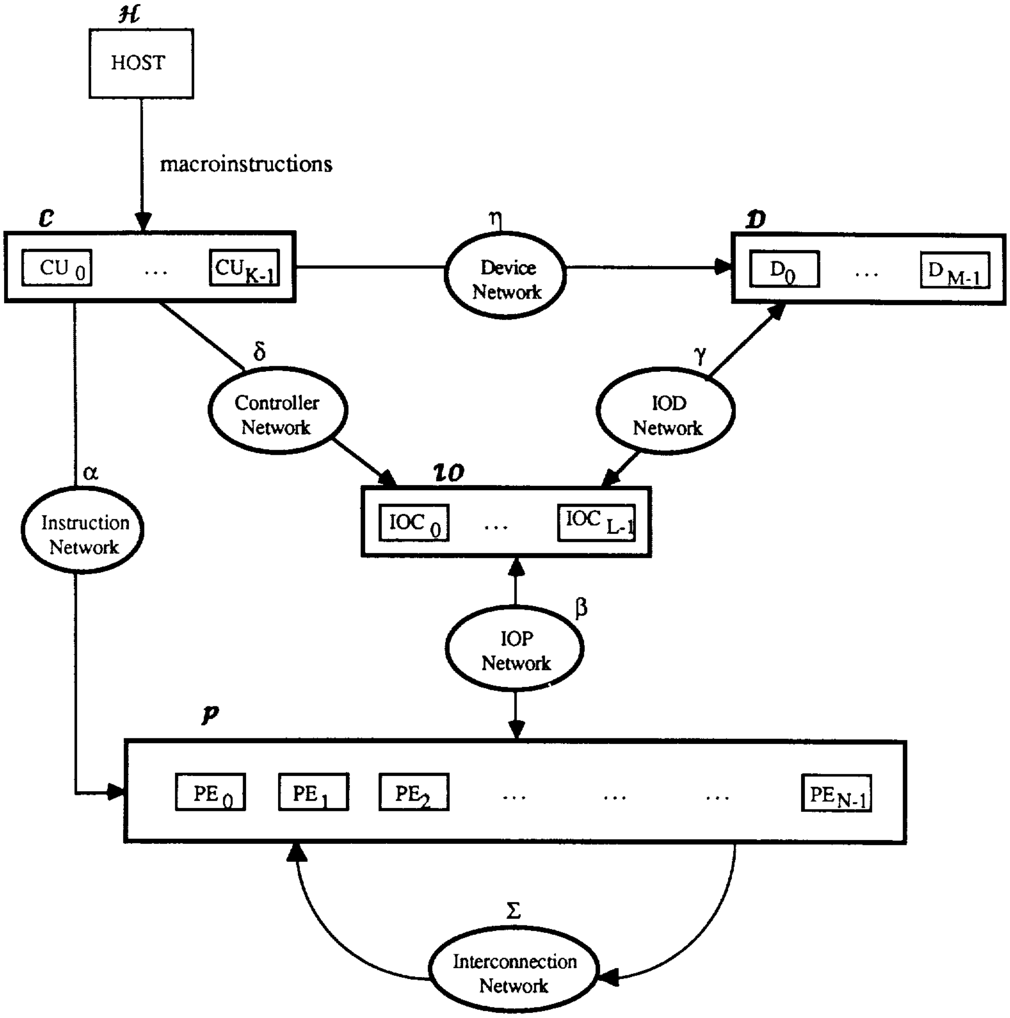


**Figure 1. Model of SIMD organization**

are the control units $\mathbf{C} = \{CU_0,...,CU_{K-1}\}$, the processing elements $\mathbf{P} = \{PE_0,...,PE_{N-1}\}$, the devices $\mathbf{D} = \{D_0,...,D_{M-1}\}$, and the input/output controllers $\mathbf{IO} = \{IOC_0,...,IOC_{L-1}\}$. A host computer $\mathbf{H}$ executes a high-level language program by sending corresponding macroinstructions to one or more control units in $\mathbf{C}$. Each control unit generates microinstructions and sends them to processing elements in $\mathbf{P}$, input/output controllers in $\mathbf{IO}$, or (input/output) devices in $\mathbf{D}$. In the model, the host computer is restricted to the storage and compilation of high-level programs. The input/output controllers are viewed as interfaces that control transfer of data (including buffering) between the processing elements and the devices. Figure 1 shows the basic organization of the model.

The communication patterns among the components of the model are described by a collection of six networks, collectively known as $\mathbf{N}$: the instruction network $\alpha$, the input/output processor (IOP) network $\beta$, the input/output device (IOD) network $\gamma$, the controller network $\delta$, the device network $\eta$, and the interconnection network $\Sigma$. For example, the interconnection network describes how data may be passed between the processing elements in $\mathbf{P}$. The instruction network similarly describes how microinstructions are sent from the control units in $\mathbf{C}$ to the processing elements in $\mathbf{P}$.

In the model, each of the networks $\alpha$, $\beta$, $\delta$, and $\eta$ is represented as a function, while the networks $\gamma$ and $\Sigma$ are represented as collections of functions. If a network is seen as a collection of functions, it will be useful to regard it as a relation. Choosing functions for $\alpha$, $\beta$, $\delta$, $\eta$, and $\Sigma$ corresponds to an instantiation of the model, which produces a particular SIMD machine. The choice of $\gamma$ depends on other considerations that will be discussed later. This functional view of networks was first presented by Siegel (Ref. 7). The use of functions to represent networks allows mathematical formalisms such as functional notation and commutative diagrams to be applied to the description and analysis of the SIMD model.

As a first illustration of the functional approach to networks, consider the instruction network. This network is a function $\alpha$ from the set $\mathbf{P}$ of processing elements to the set $\mathbf{C}$ of control units. Each $PE_i$ in $\mathbf{P}$ is then associated with the control unit $CU_j = \alpha(PE_i)$ in $\mathbf{C}$ which sends it instructions. For a given control unit $CU_j$ in $\mathbf{C}$, all processing elements in $\alpha^{-1}(CU_j)$ are assumed to receive the broadcast instruction at the same instant.

## 3. COMPONENTS AND NETWORKS

### 3.1 Basic Components

The utility of the model will depend in large part on the particular properties of the units that make up each component of the model. A control unit in $\mathbf{C}$ generates a sequence of microinstructions for each macroinstruction. These microinstructions can be instructions to the processors, input/output controllers, or devices. It is assumed that microinstructions are broadcast simultaneously by a control unit to its targets, and that the broadcast occurs concurrently with the interpretation of program instructions. In the model, it is possible for different instructions to be broadcast at a given time by different control units. That is, synchronization across control units is not assumed. Each control unit contains the logic necessary to determine the IOD network configuration $\gamma$ for any program instruction. The role of $\gamma$ will be discussed in more detail below.

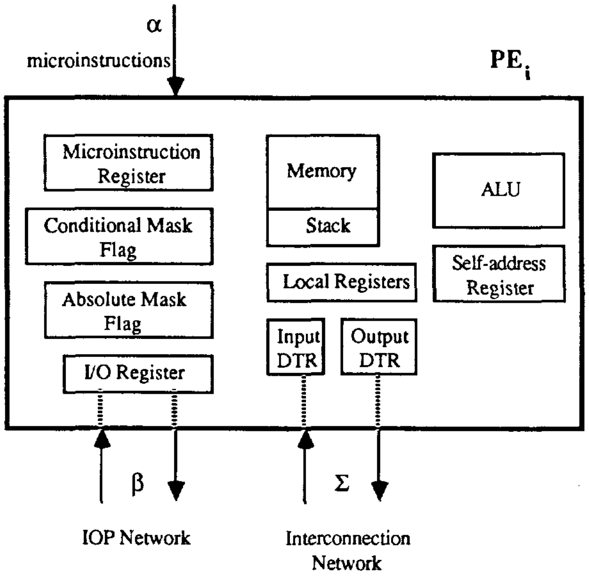


**Figure 2. Organization of a Processing Element**

The organization of a typical processing element in $\mathbf{P}$ is shown in Figure 2. Microinstructions are sent to the processor on the instruction network $\alpha$. The interconnection network $\Sigma$ and the data transfer registers *input DTR* and *output DTR* are used for interprocessor data transfer. If $\sigma \in \Sigma$, and $PE_i$ and $PE_j$ are elements of $\mathbf{P}$ with $PE_j = \sigma(PE_i)$, then data in the output DTR of $PE_i$ is transferred to the input DTR of $PE_j$ in one instruction cycle. We require that $\sigma(i_1) \neq \sigma(i_2)$ whenever $i_1 \neq i_2$, so that no data collisions can take place during a single instruction cycle.

Each element of $\mathbf{P}$ is given a unique address, which is contained in the self-address register. The *input/output registers* are used for transferring data between the PEs and the input/output controllers. The *local registers* have been depicted as a group since the issues of intermediate data storage and DTR to memory transfers will not be discussed here. The activities of the microinstruction register, the ALU, the DTRs, and the input/output register are assumed to be concurrent.

A processing element may ignore a microinstruction, depending on the status of the two mask flags. The *conditional mask flag* uses program data to control data-dependent operations, while the *absolute mask flag* uses the contents of the self-address register to control address-dependent operations.

The input/output devices in $\mathbf{D}$ represent any memory outside the processing elements that is used for data storage or retrieval. A device that receives an instruction from a control unit in $\mathbf{C}$ will execute that instruction. Device instructions will generally involve data transmission on the IOD network, but they may also specify such file operations as "open" or "close". The interpretation of device instructions will proceed concurrently with memory accesses, activity on the IOD network, and other processing activities.

Each input/output controller in $\mathbf{IO}$ contains the logic needed to perform two tasks: the interpretation of instructions received on the controller network $\delta$ and the distribution of data to and from the devices and processing elements. Interpretation of instructions and data transfer between devices and processing elements takes place concurrently.

The collection of functions denoted by $\gamma$ can be regarded as a relation that associates a set of devices in $\mathbf{D}$ with each input/output controller in $\mathbf{IO}$. We will see below that the form of the relation $\gamma$ is restricted by properties of the relation between the processing elements in $\mathbf{P}$ and the devices in $\mathbf{D}$. Each input/output controller must include sufficient logic to compute the relation $\gamma$ from a high-level description of the relation between processing elements and devices.

### 3.2 Networks

Four of the networks of the SIMD model can be specified as single functions: the instruction network $\alpha$, the input/output processing element (IOP) network $\beta$, the controller network $\delta$, and the device network $\eta$. Each of these corresponds to a natural partition of the elements making up one of the components of the model.

Recall that the instruction network is defined by the function $\alpha$: $\mathbf{P} \rightarrow \mathbf{C}$ where a processor $p \in \mathbf{P}$ is controlled by, and receives microinstructions from, the control unit $\alpha(p) \in \mathbf{C}$. If we assume that every processor is controlled by some control unit, then the set $\mathbf{P}$ of processors can be naturally partitioned as the union of the sets $\{\alpha^{-1}(c) : c \in \mathbf{C}\}$, where $\alpha^{-1}(c)$ is the set of processors that are controlled by control unit c.

The assumption that every processor is controlled by some control unit is equivalent to the assertion that the function $\alpha$ is a total function. If $\alpha$ is partial, only designated processing elements would receive microinstructions for further distribution to the remaining processing elements. In this case, the partition can only be applied to the subset *dom* $\alpha$ of $\mathbf{P}$, which consists of those processors that are controlled by control units. Here, *dom f* and *range f* denote the *domain* and *range* of a function f.

The IOP network is defined by a function $\beta$: $\mathbf{P} \rightarrow \mathbf{IO}$, where input and output for a processor $p \in \mathbf{P}$ is controlled by input/output controller $\beta(p) \in \mathbf{IO}$. If every processor is directly controlled by an input/output controller, then the processors in $\mathbf{P}$ can be partitioned as the union of the sets $\{\beta^{-1}(i) : i \in \mathbf{IO}\}$, where input/output controller i handles the data transfers involving the processing elements $\beta^{-1}(i)$.

The assumption that every processing element is controlled directly by an input/output controller is equivalent to assuming that $\beta$ is total. This may not be a reasonable assumption in general, since certain computers (e.g. GAPP systems) have only a particular subset of their processing elements directly connected to input/output units. In such machines, the interprocessor network $\Sigma$ is used to shift data between this subset and other processing elements.

The controller network is defined by a function $\delta$: $\mathbf{IO} \rightarrow \mathbf{C}$. As we have seen above, if $\delta$ is a total function, it partitions $\mathbf{IO}$ as the union of the sets $\{\delta^{-1}(c) : c \in \mathbf{C}\}$ where a control unit c controls each member of $\delta^{-1}(c)$ by broadcasting instructions to it.

The device network is similarly defined by a function $\eta$: $\mathbf{D} \rightarrow \mathbf{C}$. If $\eta$ is total, the sets $\{\eta^{-1}(c) : c \in \mathbf{C}\}$ partition $\mathbf{D}$, where once again a control unit c controls each member of $\eta^{-1}(c)$ by broadcasting instructions to it. This network represents a relationship between control units and devices that cannot be derived solely from topological considerations. $\eta$ may be regarded as representing an aspect of system software, rather than hardware.

To focus on networks that represent realizable SIMD computers, we will make the following assumptions:

(1) $\alpha: \mathbf{P} \rightarrow \mathbf{C}$ is a total function

(2) *range* $\alpha \cup$ *range* $\delta = \mathbf{C}$

(3) $\beta$ is a surjection

(4) $\beta^{-1}\delta^{-1} \supseteq \alpha^{-1}$

Each of these assumptions corresponds to an important feature of the SIMD computational model. As previously noted, Assumption 1 is equivalent to saying that each processor receives microinstructions from a control unit. Assumption 2 states that every control unit must be connected to at least one input/output controller or one processing element. Similarly, Assumption 3 states that every input/output controller handles at least one processing element. Finally, Assumption 4 is equivalent to saying that if a control unit sends microinstructions to a processing element, then some input/output controller that it handles must also be connected to that processing element.

The above assumptions are sufficiently powerful to imply interesting and not immediately obvious properties of machines that are represented by the computational model. For example, Assumption 4 states that for any control unit c, $\beta^{-1}(\delta^{-1}(c)) \supseteq \alpha^{-1}(c)$, which implies that *dom* $\beta \supseteq$ *dom* $\alpha$. Since Assumption 1 states that $\alpha$ is a total function, it follows also that $\beta$ is a total function.

Now suppose that p is a processing element controlled by some control unit (i.e. $p \in$ *dom* $\alpha$). Then $p \in \alpha^{-1}(\alpha(p))$, which is contained in $\beta^{-1}(\delta^{-1}(\alpha(p))$ by Assumption 4, so that $\delta(\beta(p)) = \alpha(p)$. Thus Assumptions 1 and 4 together yield $\alpha = \delta\beta$.

The idea underlying Assumption 4 can be restated in the following manner. If an input/output controller is connected to a processing element, then the control unit handling the input/output controller must also handle that processor. This can be formalized as (4') $\alpha^{-1} \supseteq \beta^{-1}\delta^{-1}$. Just as Assumption 4 and the totality of the function $\alpha$ imply that $\alpha = \delta\beta$, Assumption 4' and the totality of the function $\beta$ can be shown to imply that $\alpha = \delta\beta$. It is also true that Assumption 3 and $\alpha = \delta\beta$ imply that *range* $\alpha$ = *range* $\delta$. We have therefore obtained the following results:

Proposition 1: (a) If $\alpha$ is total, Assumption 4 implies $\alpha = \delta\beta$.
(b) If $\beta$ is total, Assumption 4' implies $\alpha = \delta\beta$.

Proposition 2: Assumptions 1-4 imply that $\alpha$ and $\delta$ are surjections.

Since the considerations underlying Assumptions 4 and 4' are so similar, it is natural to investigate the circumstances under which they constitute the same assumption. This turns out to be the case when the functions $\alpha$ and $\beta$ have the same domain, which happens when a processing element receives microinstructions from some control unit if and only if it is directly handled by some input/output controller. The proof of this result, along with other consequences of the assumptions, can be found in Ref. 5.

### 3.3 Parallel Input and Output

The discussion of the networks of the computational model has so far omitted the input/output device (IOD) network represented by $\gamma$. The IOD network associates input/output devices in $\mathbf{D}$ with input/output controllers in $\mathbf{IO}$. It mediates the transfer of data, by means of the controllers, between the devices in $\mathbf{D}$ and the processing elements in $\mathbf{P}$. The IOD network arises naturally out of the relationship between devices and processing elements that is determined by such high-level language input/output instructions as "read" or "write". Such instructions specify which devices are to be associated with which processing elements, and thus provide an input/output relation $\mathcal{R}$ between processing elements and devices. This input/output relation can be used to derive the IOD network $\gamma$, which is naturally represented as a relation between the input/output controllers and the input/output devices.

In particular, a high-level input/output instruction that states that certain processing elements should receive data from or write data to a given set of devices corresponds to a relation $\mathcal{R}$ contained in $\mathcal{P} \times \mathcal{D}$, where $(p,d) \in \mathcal{R}$ if device d transfers data to or from processor p. We can now define the IOD network $\gamma$ as a subset of $\mathcal{IO} \times \mathcal{D}$ : $\gamma = \{(i,d) : \exists\, p \in \mathcal{P}$ with $(p,d) \in \mathcal{R}$ and $\beta(p) = i\}$.

If $2^S$ denotes the set of all subsets of a set S, the relation $\mathcal{R}$ can be viewed as a function r: $\mathcal{P} \rightarrow 2^{\mathcal{D}}$, where the set r(p) contains d if and only if $(p,d) \in \mathcal{R}$. From this perspective, the definition of $\gamma$ can be expressed more simply as $\gamma(i) = r(\beta^{-1}(i))$, or still more simply as $\gamma = r\beta^{-1}$, where $\gamma(i)$ is the set of devices that transfer data using the input/output controller i.

These ideas are illustrated by the network configurations shown in Figure 3, which is derived from the following high-level language directives: "read data from device 2 into even processors with positive index, and write copies of the data from odd processors to device 0 and device 1". For this configuration, the IOC network is defined by $\beta(PE_{2i+1}) = IOC_1$ for $i \geq 0$, $\beta(PE_{2i}) = IOC_2$ for $i > 0$, and $\beta(PE_0) = IOC_0$. The IOD network is $\gamma(IOC_0) = \emptyset$, $\gamma(IOC_1) = \{D_0, D_1\}$, and $\gamma(IOC_2) = \{D_2\}$. $\gamma(i) = \emptyset$ is to be interpreted as "no data is sent to or received from any device through controller i".

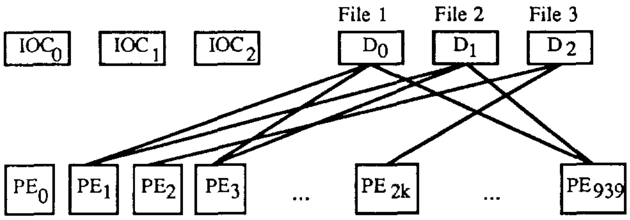


**Figure 3. A Sample IOD Network**

While the IOD network $\gamma$ governs the relationship between the input/output devices and the input/output controllers, the devices themselves are managed by the control units. This connection is under the control of the device network $\eta$. Thus there are complex interrelationships between $\gamma$ and the other networks of the model. A detailed discussion of the way in which the IOD network $\gamma$ relates to the other networks of the model is found in Ref. 5.

### 3.4 Interconnection Network

The interconnection network is represented by a family of functions $\Sigma = \{\sigma : \mathcal{P} \rightarrow \mathcal{P}\}$, where each function $\sigma$ represents a transfer of data (that can be performed in one instruction cycle) between the DTR registers on different processors. The model makes the following assumptions about the interconnection network: (i) each $\sigma \in \Sigma$ is assumed to be a bijection (a one-to-one, onto mapping) from $\mathcal{P}$ to $\mathcal{P}$, and (ii) using appropriate masks, the collection $\Sigma$ can generate all permutations on $\mathcal{P}$. The assumption that each member of $\Sigma$ is a one-to-one function avoids the need to store data in additional registers during the instruction cycle. Since $\mathcal{P}$ is finite, $\sigma$ being one-to-one implies that $\sigma$ is also onto.

Recall from Figure 2 that each processing element has absolute and conditional mask flags, as well as input and output DTR registers. Using these flags, each processor can be classified (at any particular moment) as either active or inactive. Each member function $\sigma \in \Sigma$ uses the following rule to determine the contents of a processor's input DTR. *If p is active,* $DTR'_{\sigma(p)}(input) \leftarrow DTR_p(output)$; *if p is inactive,* $DTR'_{\sigma(p)}(input) \leftarrow DTR_{\sigma(p)}(output)$. For processor j, $DTR'_j$ denotes the post-instruction contents of the register and $DTR_j$ denotes its pre-instruction contents. The symbol ← denotes assignment. The rule can be justified as follows: Before any interconnection function in $\Sigma$ is executed, each processor p performs an operation that copies its output DTR to its input DTR. Every processing element, whether it is active or inactive, is eligible to receive a value in its input DTR. If the processing element is active, it will send a value from its output DTR; otherwise, no value will be sent.

The importance of the assumption on permutation generation is that masked interconnection functions can be used to construct virtual processor topologies from the existing interconnection network. The provision of such topologies must include the corresponding parallel data routing capabilities. As an example, we will show how the cube functions associated with the 3-dimensional hypercube shown in Figure 4a can be used to simulate the shift operations (SHIFT_EAST, SHIFT_WEST, SHIFT_NORTH, and SHIFT_SOUTH) needed for the 2x4 grid shown in Figure 4b.

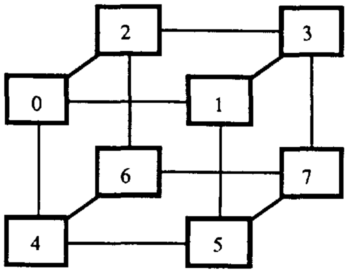


**Figure 4a. A Hypercube Network**

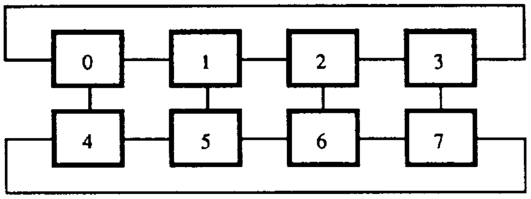


**Figure 4b. A 2 x 4 Grid**

Every application of an interconnection network function must be associated with a mask which indicates whether each processing element will be active or inactive. Such masks will be very space-intensive, since each requires N bits. A useful subset of masks (first introduced by Siegel in Ref. 7) requires only $2*\log_2 N$ bits for each mask. A mask in this subset is represented as a string of characters $C_0C_1...C_{k-1}$, where $k = \log_2 N$, and each $C_i$ is drawn from {0,1,X}. A processor is active if its binary representation matches the mask, where 0 matches 0, 1 matches 1, and X matches either 0 or 1. The application of a mask to a function will be denoted by appending [*mask*] to the function.

The grid shift operations can now be obtained in terms of the 3-dimensional cube functions as follows (composition is left to right):

SHIFT_EAST = $\text{cube}_0$[XXX] $\text{cube}_1$[XX0]
SHIFT_WEST = $\text{cube}_0$[XXX] $\text{cube}_1$[XX1]
SHIFT_NORTH = SHIFT_SOUTH = $\text{cube}_2$

If the Siegel subset of masks is enhanced by complementation, cube functions can be used to express data routing for a wide variety of topologies, including grids and toruses of all dimensions and binary trees. These results are established in Ref. 5. Siegel has shown in Ref. 6 that the cube functions do satisfy the permutation hypothesis.

## 4. SIGNIFICANCE OF THE MODEL

### 4.1 Instantiation

We will now show how the model can be instantiated to produce a typical SIMD computer, the Thinking Machines Corporation CM-2. This computer contains a large number (between 8,192 and 65,536) of processing elements, along with at least one VAX or Symbolics host computer. Program instructions are sent from one to four host computers to at most four *sequencers*, which in turn generate microinstructions and distribute them to the processors. The CM-2 is equipped with a high-speed input/output device called a *data vault*. The interface between the data vault and the processors is governed by eight input-output controllers.

To instantiate the model, the following choices are made:

Control Units: As many as four control units ($1 \leq K \leq 4$), identified with the CM sequencers

Processing Elements: N processing elements ($N = 2^i$, $i = 13, 14, 15$, or 16)

Input/Output Devices: the data vault ($M = 1$)

Input/Output Controllers: eight input-output controllers ($L=8$)

The first four networks of the instantiated model are defined as follows:

Instruction Network:
$P_i \rightarrow CU_{\lfloor i/16384 \rfloor}$ for $i = 0,...,N-1$

Input/Output Processing Network:
$P_i \rightarrow IOC_{\lfloor i/8192 \rfloor}$ for all i

Controller Network: $IOC_i \rightarrow CU_{\lfloor i/2 \rfloor}$

Device Network: $CU_i \rightarrow D_0$ for all i

The form of the input/output device network $\gamma$ will depend on the specific input or output high-level language instructions that are coded. Since the processing elements of the CM-2 are organized as a hypercube, the interconnection network $\Sigma$ corresponds to the cube functions that have been described above. All functions of this network are clearly bijections, and Siegel has shown in Ref. 6 that it can be used to generate all permutations.

The CM-2 instantiation satisfies the assumptions proposed above. Since the function $\alpha$ is a total function and a surjection, Assumptions 1 and 2 are satisfied. It is clear that $\beta$ is a surjection, as required by Assumption 3. Assumption 4 is straightforward to verify; it states that whenever a control unit is connected to a processing element, then it must also be connected to an input/output controller that handles that processing element.

### 4.2 Programming Language Development

The programming languages (FORTRAN 8X, C*, Parallel Pascal, and *LISP) that are currently available for SIMD computers owe their syntax and semantics to their familiar sequential ancestors: FORTRAN, C, Pascal, and LISP. While the semantics of each of the serial languages is based on a sequential model of computation, its parallel version contains extensions that handle data parallel instructions based on a specific underlying model of SIMD machine organization. Since each language contains different extensions, its utility for general data parallel programming could be correspondingly limited. For example, it is difficult to use some of these languages to express parallel input and output. Also, it is often an involved process to develop high-level code for arbitrary processor topologies. Just as the von Neumann sequential computational model fostered the development of transportable sequential programming languages, the model proposed here should do the same for data parallel programming languages.

The model has been used as the basis of the syntax and semantics of the data parallel programming language DAPL (Refs. 3,4), which is currently under development. It also forms the context for the formal semantics of the SIMD algorithmic language described in Ref. 9. DAPL is an object-based language which provides the programmer with a number of basic geometric structural types that can be combined into the data structures representing the virtual topologies needed for data parallel application development. The language also associates natural communication operations with the geometric structures, so that a programmer can use high-level language constructs to route data between virtual processors.

As a preliminary target machine for DAPL, a 1024 processor SIMD computer has been simulated by Klein and Rice (Refs. 1, 2). The simulation, which is based on the computational model presented in this paper, was specified in CSP and developed in *occam* on a network of INMOS Transputers. An instruction set that included parallel instructions for input/output, broadcasting, masking, routing, assignment, and arithmetic operations was implemented. An interactive menu system permits the user to specify and execute a single data parallel instruction or execute a program consisting of such instructions. This menu also allows the user to select the dimension of the hypercube interconnection network ($1 \leq \dim \leq 10$) and to perform basic input/output operations on sample data sets.

## 5. EXTENSIONS AND CONCLUSIONS

An important advantage of the SIMD model presented here is that it can be extended to future developments in SIMD machines. One particular example is the possibility that a parallel computer may be partitioned into several SIMD machines, which are controlled in a MIMD fashion. Such a MSIMD machine, which has been proposed by Siegel (Ref. 8), can be included straightforwardly in our model, since each control unit $CU_i \in \mathbf{C}$ controls a disjoint set of processors. For a second example, if shared memory is to be added to a future SIMD machine, the model could be extended to include the shared memory units $\mathcal{SM}=\{M_0, \ldots, M_{T-1}\}$, and the *shared memory network* $\mu: \mathbf{P} \rightarrow \mathcal{SM}$, which associates each processor with the shared memory unit that it can access.

The SIMD computational model presented in this paper adopts a functional approach. The model's components are sets representing the processing elements, control units, input/output devices, and input/output controllers of a modeled machine. These components are related by a number of networks, each of which is represented as one or more functions. Assumptions about the components of a SIMD computer can be translated into assumptions on the sets and functions of the model, and the implications linking these assumptions can be investigated.

The model is sufficiently complex to model all current SIMD computers, including such features as multiple control units and multiple input-output devices. It is easily extensible to future SIMD machines, including MSIMD computers and SIMD computers incorporating shared memory devices. A particular SIMD computer is modeled by choosing parameters that determine the size of the model's fundamental sets and by defining networks that express the relationships between those sets. Finally, the model has recently been used to support the design of a programming language for SIMD computers, and it has served as the target for the simulation of general SIMD machines. It is reasonable to hope that this computational model is a step toward one that will be as significant for SIMD computation as the von Neumann model has been for sequential computation.

### Acknowledgement

This research was partially supported by a grant from the Software Productivity Consortium and the Virginia Center for Innovative Technology (#SPC-88-001).

## REFERENCES

1. Klein, C. S., "Simulation of a SIMD Computer on a Transputer Network", George Mason University Department of Computer Science, Technical Report TR-4-88, 1988.

2. Rice, M. D. "User's Manual for Transputer-Based SIMD Simulator", George Mason University Department of Computer Science, Technical Report TR-6-88, 1988.

3. Rice, M. D., S. B. Seidman, and P. Y. Wang, "A High-Level Language for Data Parallel Computation", *Proceedings of CONPAR 88*, British Computer Society, 1988.

4. Wang, P. Y., S. B. Seidman, M. D. Rice, and T. E. Gerasch, "An Object-Method Programming Language for Data Parallel Computation", *Proceedings of the 22nd Hawaii International Conference on System Sciences*, January 1989, to appear.

5. Rice, M. D., S. B. Seidman, and P. Y. Wang, "A Functional Model for SIMD Computation", George Mason University Department of Computer Science, Technical Report TR-7-88, 1988.

6. Siegel, H. J., "Analysis Techniques for SIMD Machine Interconnection Networks and the Effects of Processor Masks", *IEEE Transactions on Computers*, C-26 (1977), pp. 153-16

7. Siegel, H. J., "A Model of SIMD Machines and a Comparison of Various Interconnection Networks", *IEEE Transactions on Computers*, C-28 (1979), pp. 907-917.

8. Siegel, H. J., L. J. Siegel, F. C. Kemmerer, P. T. Mueller, Jr., H. D. Smalley Jr., and S.D. Smith, "PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition", *IEEE Transactions on Computers*, C-30 (1981), pp. 934-947.

9. Rice, M. D., "An Algorithmic Language for Data Parallel Computation", preliminary manuscript.

# ABSTRACT DATA TYPES FOR SIMD HYPERCUBE MACHINES

Farokh B. Bastani and Dar-Ren Leu

Department of Computer Science
University of Houston—University Park
Houston, TX 77004

## ABSTRACT

While massively parallel processing promises a high performance implementation for various applications, it also exhibits the difficulty of programming in a modular way. Programs for conventional programming models can be structured as a hierarchy of clients and servers that can be individually implemented using fine grained parallel processors; however, the clients and servers in this approach are separate entities and, hence, create a von Neumann bottleneck that is unacceptable for fine grained parallel systems. In this paper, we present methods of merging distributed clients and servers to allow efficient parallel operations. In particular, we propose parallel primitives as well as normalized representation schemas for efficiently implementing various abstract data types. The mapping strategies to reduce communication cost for various abstract data types on the hypercube structure are also discussed.

**Keywords:** Abstract Data Types, Distributed Representation, Mapping Strategies, Multiple Entry Data Structures, Performance Analysis, SIMD Hypercube Machines.

## INTRODUCTION

Fine grained parallel supercomputers hold great promise for achieving potentially dramatic speed ups in computing power. To realize the full capabilities of these machines for various applications (robotics, expert systems, simulation, databases), ingenious and fundamentally new kinds of data parallel algorithms must be devised. These include memory based models for databases, parallel pattern matching and constraint resolution for expert systems, and direct modeling of physical systems for simulation purposes.

A major problem which impedes the widespread use of fine grained parallel architectures is the difficulty of programming these machines in a modular way. Conventional distributed and coarse grained parallel programming models do not scale up to fine grained parallel systems. Programs for the former systems can be structured as a hierarchy of clients and servers that can be individually implemented using fine grained parallel processors (Ref. 1), such as broadcasting sequential processors (BSP) for high performance abstract data type components, systolic arrays for functional components, and processor networks for interface components. However, the clients and servers in this approach are separate entities, thus creating a von Neumann bottleneck that is unacceptable for fine grained parallel systems.

In this paper we investigate efficient implementation of abstract data types on SIMD hypercube machines. To achieve high performance in parallel programs, the requests of various clients of an abstraction should be processed in parallel. One way to overcome the von Neumann bottleneck is to merge the clients with the servers of an abstraction. We first explore the essences of this implementation theoretically, and then realize it for various abstract data types. In particular, we propose parallel primitives as well as normalized representation schemas for efficiently implementing different abstract data types. Since the performance of an abstract data type component may vary considerably depending on the placement of its data elements on the physical processors, the mapping strategies to reduce communication cost are also discussed. These abstract components can serve as building blocks for implementing other parallel components, such as parsers, constraint resolvers, and pattern recognizers.

## A THEORETICAL EXPLORATION

In this section we investigate the essential elements in implementing a high performance abstract data type component on SIMD hypercube machines.

An abstract data type component is a collection of functions that implement some mathematical objects, such as lists, queues, and sets. Internally, an abstract data type, $A$, is a tuple

$$A = (D, F)$$

where $D$ is a data structure that can be expressed as a state machine, and $F$ is a collection of functions that operate upon $D$. The data structure, $D$, is also a tuple

$$D = (O, R)$$

in which $O$ denotes the collection of data elements for storing states of $D$ and $R$ denotes the relations among these elements. Thus, $O$ can be expressed by

PRECEDING PAGE BLANK NOT FILMED

CH2649-2/89/0000/0609$01.00 © 1988 IEEE

PAGE 608 INTENTIONALLY BLANK

$$O = \{o_i \mid 1 \leq i \leq n\}$$

where $n$ is the total number of data elements in $O$, and $R$ is a function defined on $o_i$, $1 \leq i \leq n$. For data intensive applications on SIMD machines, $R$ can be defined as a collection of relations $\{r_i \mid 1 \leq i \leq n\}$ such that, for element $o_i$, the associated relation $r_i$ denotes the absolute position of $o_i$ in $D$ or the relative relations between $o_i$ and other elements $o_j$, $j \neq i$. In this case, $D$ can be expressed as a collection of $o_i$ and $r_i$; i.e.,

$$D = \{(o_i, r_i) \mid 1 \leq i \leq n\}$$

where $n$ may be very large.

A data structure, $D$, may be represented in several ways on SIMD machines. To search for the best representation of a data structure, we first notice that an SIMD machine such as the Connection Machine (Refs. 2-3) may contain up to tens of thousands of processors, and at a given time can operate only on data elements of identical type. In order to fully exploit the underlying architecture of SIMD machines, a data structure must be represented in a distributed way by having one processor per data element. This representation is called a distributed representation which can also be defined by a mapping function $h$ from $D$ to the network of processing elements, say $\{PE_j\}$, i.e.,

$$h : (o_i, r_i) \longrightarrow PE_j \quad \text{for} \quad 1 \leq i \leq n \quad \text{and} \quad 1 \leq j \leq N$$

where $N$ is the total number of $PE$'s. Distributed representations can facilitate concurrent operation upon a large set of data and, hence, achieve the maximum degree of parallelism for the implementation of an abstract data type.

However, a distributed representation for $D$ by itself does not guarantee that requests from multiple clients can be processed simultaneously. To achieve this, the clients and servers of an abstraction must be merged so that a $PE$ which acts as a server for an abstraction can also act as a client of that abstraction. This is implemented via a kind of distributed representation called a normalized representation which is defined by the above mapping function with

$$address(PE_j) = f(o_i)$$

where $f$ is a one-to-one function. That is, the address of $PE$ which is allocated to $o_i$, by using $h$, can be computed simply from $o_i$ or even $i$. The ability to normalize the data structure is the key to the parallel processing of multiple client requests.

In addition to multiple entry data structures based on normalized representations, we have to provide parallel primitives for an abstraction so that multiple requests can be processed simultaneously. First, consider the computation pattern of a basic parallel primitive $f$. The computation pattern of $f$ is used to denote the effect of $f$ by showing the operation points and data flow patterns in the network of $PE$'s. The data flow patterns in a computation pattern are also called a communication pattern. Let $g$ denote the computation pattern of $f$. Then $g$ is defined as follows.

$$g \equiv \bigcup_{1 \leq i \leq N} g_i$$

where $N$ is the total number of $PE$'s, and $g_i$ is the computation pattern by tracing the effect of applying $f$ to $PE_i$.

A computation pattern $g$ is said to be regular if all of the $g_i$'s have the same pattern; otherwise, it is irregular. In terms of the number of phases in $g$ and the change in communication pattern across different phases, the useful parallel primitives on SIMD machines can be classified into five types (see Figure 1). They are

(A) Single phase, single $PE$ (local processing). This is the simplest type of primitives. A set of $PE$'s perform the same operation (i.e., $f_i$) upon their own local data elements. No communication is required for this type of primitives.

(B) Single phase, two $PE$'s. A set of data elements are moved or copied from original locations (a set of $PE$'s) to new locations (a new set of $PE$'s). The computation pattern here may be regular or irregular. The messages transferred can be regarded as requests from a set of clients to a set of servers. If required, the servers may return some results to the clients.

(C) Multiple phases, identical pattern (multiple $PE$'s). The computation pattern for this type of primitives consists of $k$ phases, where $k$ is determined statically or dynamically, and $k > 1$. The computation pattern is identical in each of the $k$ phases.

(D) Multiple phases, deterministic pattern (multiple $PE$'s). This type of primitives performs a series of operations in a deterministic fashion (usually, tree-like fashion) on a hypercube structure. Thus, the computation pattern for this type of primitives is regular, but changes from phase to phase. The set of active $PE$'s is not defined and thus we may have many different primitives for this type. Two simple examples are ***aggregate*** and ***broadcast***. ***Aggregate*** is used to collect some summary information from a set of $PE$'s, while ***broadcast*** is used to duplicate a data element to a set of $PE$'s.

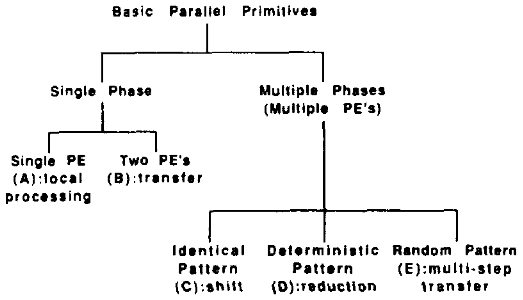

Basic Parallel Primitives
Single Phase
Multiple Phases (Multiple PE's)
Single PE (A):local processing
Two PE's (B):transfer
Identical Pattern (C):shift
Deterministic Pattern (D):reduction
Random Pattern (E):multi-step transfer

**Figure 1.** Classifications of basic parallel primitives.

(E) Multiple phases, random pattern (multiple $PE$'s). The computation patterns here are irregular. The addresses and the number of intermediate $PE$'s are determined dynamically. Since more than one message may pass through some $PE$'s, this type of primitives implements the concept of pipelining in transferring messages to destinations.

It should be noted that primitives of types B and E may make heavy use of random communication feature of the underlying architectures, and primitives of type D take advantage of the hypercube structure of $PE$'s. The above primitives form a basis for constructing higher levels of primitives to efficiently implement an abstract data type. A useful primitive may be a combination of several of the above operations.

## PRACTICAL IMPLEMENTATION

In this section we discuss practical methods to implement a high performance abstract data type component on SIMD hypercube machines. In terms of $r_i$, we first classify abstract data types on SIMD machines into the following three categories:

(1) Unrelated collections: For an unrelated collection, we have

$$r_i = \emptyset, \quad 1 \leq i \leq n.$$

This means that there is no relation among different data elements. Thus,

$$D = \{o_i \mid 1 \leq i \leq n\}.$$

Examples of unrelated collections are sets, bags, search tables, and symbol tables.

(2) Crystalline collections: Here, for component $A = (D, F)$, each element, $o_i$, has an absolute position in $D$. In this case, $r_i$ is defined by storing the logical indices of $o_i$ in $r_i$; i.e.,

$$r_i = f(i), \quad 1 \leq i \leq n.$$

Examples of crystalline collections are lists (with indices), vectors, matrices, etc.

(3) Amorphous collections: Here, for component $A = (D, F)$, the position of $o_i$ in $D$ is determined by the position of $o_i$ relative to other elements $o_j$, $j \neq i$. In this case, $r_i$ is defined by storing in $r_i$ the addresses of neighboring elements of $o_i$ in $D$; i.e.,

$$r_i = f(\&o_{j_1}, \cdots, \&o_{j_k}), \quad 1 \leq i \leq n$$

where $1 \leq j_1, \cdots, j_k \leq n$; $\&o_{j_m}$, $1 \leq m \leq k$, denotes the address of element $o_{j_m}$; and $o_{j_1}, \cdots, o_{j_k}$ are neighbors of $o_i$ in $D$. Examples of amorphous collections are lists, trees, graphs, and semantic networks that use pointer-based representation.

For an abstract data type $A = (D, F)$ with $D = \{(o_i, r_i)\}$, the simplest one of its distributed representations is to allocate a distinct $PE$ for each $(o_i, r_i)$ in any convenient way. This representation can be created efficiently and allows clients to access any point of its data structure directly; however, it may need a traversal of the whole data structure, which is time-consuming, to locate a specified element. A suitably normalized form of an abstraction allows its elements to be locatable efficiently. Two approaches are possible for the normalized representation of abstract data types in SIMD machines, and are described in the following.

(1) Based on vectors with indexed access. If the data elements of an abstraction are identified by unique sets of indices, then the locations of the data elements (i.e., the addresses of the $PE$'s allocated for those data elements) can be uniquely determined from these indices.

(2) Based on vectors with hashed access. If the data elements of an abstraction are identified by unique keys, then we can apply a parallel hash function on these keys to determine the distinct addresses of all data elements. This approach requires a predetermined hash function along with a collision resolution scheme.

Several type-independent primitives are useful for many abstract data types and are described below.

(1) $\alpha\langle op\rangle$ (Ref. 4): The symbol $\alpha$ denotes *ApplyToAll*; it applies the same *op* to all of the data elements at a time; it is a "single phase, single $PE$" operation.

(2) Copy: This primitive is used to copy data elements from one set of $PE$'s to another set of $PE$'s; it is a "single phase, two $PE$'s" operation.

(3) Multiple copies: This primitive will copy data elements from one location ($PE$) to multiple locations ($PE$'s) (e.g., *broadcast*); it is a "multiple phases, deterministic pattern" operation.

(4) $/\langle op\rangle$ (Ref. 4): The symbol / denotes *insert*; we also denote this by *reduce*$\langle op\rangle$; it reduces a set of elements using *op* and returns a scalar value; it is a "multiple phases, deterministic pattern" operation.

In the following subsections, we discuss the data representation and primitive operations for unrelated, crystalline, and amorphous collections, respectively.

### Unrelated Collections

An unrelated collection may be in either a random or a normalized representation. The random representation is created through the parallel evaluation of "predicates" upon a collection of data elements having a distributed representation. An unrelated collection will be in a normalized representation if it is created through the use of parallel functional operations such as the union, intersection, and difference operations for sets. The characteristics of unrelated collections are that their data elements are identified by distinct keys, and thus their normalized representation are based on vectors with hashed access.

Two useful primitives for unrelated collections are described in the following.

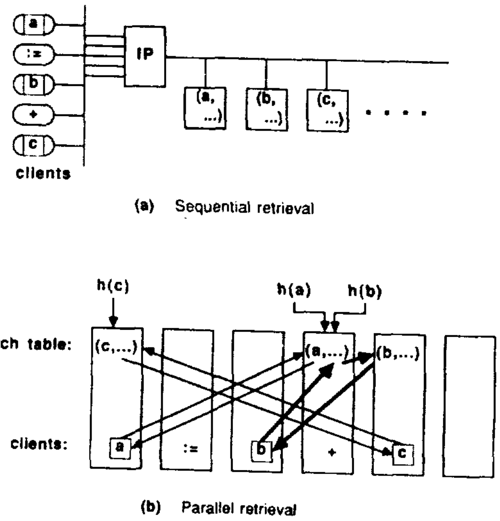
*Normalize*: It is used to transform an unrelated collection in a random form to the normalized form, and is a "multiple phases, random pattern" operation.

*Search*: It is used to search for a set of keys in parallel. It is also a "multiple phases, random pattern" operation.

Before mentioning any algorithm, we introduce the following notations:
$p$ denotes the address of the current $PE$,
$var$ is a variable in the host,
$var(p)$ is a local variable in $PE(p)$,
$vector[0..S-1](p)$ is a local vector of size $S$ in $PE(p)$,
":=" denotes a local assignment within a $PE$,
"←" indicates data movement from one $PE$ to another, and
"$\overset{\langle op \rangle}{\longleftarrow}$" indicates an inter-$PE$ data movement which has specified the operation $op$ to be performed by the destination $PE$.
The last two notations are implemented via four network primitives, namely, **send**, **receive**, *EmptyQueue*, and *ResetQueue*. The functionalities of these four primitives are explained below.

(1) **send**: It is used to assemble packets for each active $PE$, and then pump them into the interconnection network.
(2) **receive**: It enables a $PE$ to fetch the packet at the head of its input queue if that queue is not empty.
(3) *EmptyQueue*: It is used to check the existence of any received packet in the input queue for each $PE$.
(4) *ResetQueue*: It is used to discard all packets remaining in queues or in the network.

As an example, consider the parallel retrieval of attributes of a set of keys from a search table. In BSP, this must be done in several cycles. In each cycle an identifier is broadcast to the processors one of which returns the desired attribute (Figure 2a). To retrieve these in parallel we must first obtain a normalized representation of the search table, which is based on vectors with hashed access. The code for *parallel fetch* is as follows (see Figure 2b):



(a) Sequential retrieval

(b) Parallel retrieval

**Figure 2.** Retrieval of attributes from a search table.

```
Procedure ParallelFetch (client : Boolean;
     id, idc : KeyType; n : Natural;
     status : StatusType; var attr : AttributeType);
var  j, source : 0..n − 1;
      idp : KeyType;
      PacketType : {retrieve, attributes};
begin
  if client(p) then {
    j(p) := h(idc(p));
    send ("retrieve", idc(p), p) to PE(j(p)); }
  while NOT EmptyQueue(p) do
    if NOT EmptyQueue(p) then {
      receive (PacketType(p), idp(p), source(p));
      if PacketType(p) = "attributes" then
        { store the received attributes in attr }
      elsif PacketType(p) = "retrieve" then
        if status(p) = FREE then
          send ("attributes", "id does not exist")
                to PE(source(p));
        elsif id(p) = idp(p) then
          send ("attributes", attributes(p), p)
                to PE(source(p));
        else {
          j(p) := (p + 1) mod n;
          send ("retrieve", idp(p), source(p))
                to PE(j(p)); }}
end;
```

Here, $client(p)$ specifies whether $PE(p)$ has a key $idc(p)$ the attributes of which need be fetched; $n$ is the total number of processors. The search table is represented by two variables, $id$ denoting keys and $attributes$, at each $PE$. If $PE(p)$ has a key stored in it then its $status$ is set to $OCCUPIED$ in which case $id(p)$ contains the key and $attributes(p)$ contains the assiciated attributes; otherwise, its $status$ is set to $FREE$, indicating that $PE(p)$ does not (yet) have a key.

Each client uses a hash function $h$ to determine the address of $PE(j)$ to which it will transmit its request for $idc$ using the **send** primitive. When a $PE$ receives a packet, it first checks the type of that packet, *PacketType*. If *PacketType* is "*attributes*", then this $PE$ should store the fetched attributes in $attr$; otherwise, *PacketType* is "*retrieve*" and this packet is a request for some key. When a $PE$ receives a request for key $idp$, then (a) if its $status$ is $FREE$ then it sends the message "*id does not exist*" to the client; (b) if its $status$ is $OCCUPIED$ and $idp = id$ then it sends the associated attributes, *attributes*, to the client; otherwise, it forwards the request for $idp$ to the adjacent $PE$ which has address $((p+1)\ mod\ n)$. Each $PE$ repeats this process till its input queue is empty; this is indicated by a $TRUE$ value for *EmptyQueue*. It should be noted that the conditions expressed in a **while** statement are used only for determining the termination time of that **while** loop; they are not used for selecting the set of active $PE$'s to execute the instructions within the loop.

### Crystalline Collections

A crystalline collection may have one of two possible representations, namely, index-based and normalized representations. In the index-based representation of a crystalline collection, each data element has an index to indicate its relative position in the data structure, irrespective of where it may actually be allocated. The normalized representation of crystalline collections are based on vectors with indexed access. In the normalized representation, the data elements of an abstraction are sequentially allocated in contiguous locations so that the physical address of a data element can be directly computed from its logical indices. Crystalline collections have two distinguishing characteristics, namely, (1) the alignment of data elements of two instances of an abstraction for efficient binary operations, and (2) the stream processing of data.

The following primitives are useful for crystalline collections.

*Shift(data/operation)*: This is applicable for crystalline collections in a normalized form. It moves *data* or *operation* forward, backward, or along a ring. The notation *shift(operation)* means that the processor adjacent to the current one is activated. This primitive is a "single phase, two $PE$'s" operation.

*Align*: It will move a set of data elements from one set of $PE$'s to a new set of $PE$'s. One example is to transpose a matrix. This primitive is also a "single phase, two $PE$'s" operation.

*Prefix/postfix⟨op⟩*: This primitive will compute in parallel all prefixes or postfixes of vectors using the specified *op*. It is a "multiple phases, deterministic pattern" operation.

*Normalize*: This primitive is used to transform a crystalline collection from the index-based representation to the normalized representation, and is a "single phase, two $PE$'s" operation.

*Enumerate*: A crystalline collection may be created in a temporary representation which is neither an index-based nor a normalized form. In this temporary representation, the data elements are allocated sequentially, but discontiguously. *Enumerate* is used to transform a crystalline collection from this temporary representation to the index-based representation. This primitive is a "multiple phases, deterministic pattern" operation.

*Compact*: This primitive is used to transform a crystalline collection from the temporary representation (see *enumerate*) to the normalized representation in which all data elements are allocated sequentially and contiguously. It is a "multiple phases, identical pattern" operation.

As an example, consider the multiplication of two matrices, $C = A \times B$, where the dimensions of $A$, $B$, and $C$ are $L \times M$, $M \times N$, and $L \times N$, respectively. Let $n = \lceil log_2\ (max(L,N))\rceil + \lceil log_2\ M\rceil$, $m = \lceil log_2\ M\rceil$, and $M' = 2^m$. Thus the total number of processors required is $2^n$, Initially, the elements of matrix $A$ are stored in processors in row-major order, and matrix $B$ in column-major order. That is, the element $a_{ij}$ of $A$ is stored in the processor which has address $p = i \times M' + j$; and for the element $b_{jk}$ of $B$, $p = k \times M' + j$. After multiplication, element $c_{ik}$ of the resulting matrix will be stored in $C[k](i)$, the $k$th entry of vector $C$ in processor $i$ which has address $p = i \times M'$.

The idea for solving this problem is to *shift* $b_{jk}$ to appropriate processors so that the multiplication of $a_{ij}$ and $b_{jk}$ can be performed in parallel. Then the element $c_{ik}$ can be obtained by using the *reduce*⟨+⟩ primitive. The binary representation of $p$ is denoted by $p_{q-1:0}$, which can be split into $i = p_{q-1:m}$, and $j = p_{m-1:0}$. The program segment for the matrix multiplication is as follows

```
begin
  $L' := max(L,N)$;
  $L'' := min(L,N)$;
  $k(p) := p_{q-1:m}$;
  $k'(p) := k(p)$;
  if $0 \le k(p) < L'$ then
    for $t := 1$ to $L'$ do {
      if $0 \le k'(p) < L''$ then {
        $temp(p) := A(p) \times B(p)$;
        $reduce\langle + \rangle\ (0, m-1, temp)$;
        if $p_{m-1:0} = 0$ then
          $C[k(p)](p) := temp(p)$; }
      $k(p) := (k(p)+1)\ mod\ L'$;
      $shift\ (L', M', B)$;
      if $L > N$ then
        $k'(p) := k(p)$; }
end;
```

The *shift* primitive is defined by

```
Procedure $Shift\ (L', M' : Natural$;
    var $A : MatrixType$)
var $newp : 0..n-1$;
begin
  $newp(p) := [p + (L'-1)M']\ mod\ (L'M')$;
  $A(newp(p)) \leftarrow A(p)$;
end;
```

and *reduce*⟨+⟩ is defined by

```
Procedure $Reduce\langle + \rangle\ (s, e : Natural$;
    var $A : MatrixType$)
var $i : Natural$;
    $temp1 : MatrixType$;
begin
  for $i := s$ to $e$ do {
    $temp1(p^{(i)}) \leftarrow A(p),\ (p_i = 1)$;
    $A(p) := A(p) + temp1(p),\ (p_i = 0)$; }
end;
```

The complexity of this algorithm is $O(max(L,N) \times log(M))$. If the number of processors is sufficiently large, i.e., $q \geq \lceil log\ L\rceil + \lceil log\ M\rceil + \lceil log\ N\rceil$, a much more efficient algorithm ($O(log\ max(L,M,N))$ for matrix multiplication can be devised.

### Amorphous Collections

An amorphous collection may be in either a pointer-based or a normalized representation. In the pointer-based representation, each node of a graph associated with the pointers to other nodes is allocated a $PE$. The normalized representation for amorphous collections are based on indexed vectors or hash tables depending upon how their nodes are identified (unique ordinal numbers or unique names). Two normalized forms for graphs are possible. The first one is to allocate $PE$ for each node in a graph in a way that the physical address of the $PE$ for each node can be computed from the id or name of that node; it is still in a pointer-based form. The second normalized form is generated from the adjacency matrix of a graph; that is, we assign a distinct $PE$ to each element of an adjacency matrix according to the logical indices of elements. A graph in the second normalized form can be regarded as a crystalline collection, and no explicit pointers are required.

Amorphous collections are characterized by their communication-oriented algorithms. For graph components (including trees), communication is usually restricted to those pairs of nodes having a pointer between them. A node in a list component, however, may communicate with different nodes in the same list; thus, parallel primitives for list components are different from those for other amorphous collections. Some useful primitives for amorphous collections are described below.

*ReversePtr*: It is used to reverse the direction of pointers in a graph, and is a "single phase, two $PE$'s" operation.

*AdvPtr*: It is used to advance the temporary pointers in a list to point from one node to another node in the same list. It is a "single phase, two $PE$'s" operation.

*Send**: This primitive will move data from one node to another via zero or more intermediate nodes for a set of nodes in a graph. It is a "multiple phases, random pattern" operation.

*Convert*: This primitive is used to transform a graph in pointer-based representation to the normalized form using adjacency matrix. It is a "single phase, two $PE$'s" operation.

For list components in pointer-based representation, we need some higher level primitives which use more than one type of basic primitives discussed before. They are:

*Reduce*⟨*op*⟩: This primitive reduces a list in a binary tree fashion and returns a scalar value.

*Prefix/postfix*⟨*op*⟩: This primitive computes in parallel all initial prefixes/postfixes of a list using the specified *op*.

*Normalize*: This will convert a list in pointer-based representation to a normalized representation. Naturally, a list must be enumerated first before being normalized.

As an example, consider the problem of comparing two lists, $l1$ and $l2$, to determine whether $l1 < l2$, $l1 = l2$, or $l1 > l2$. (The comparison is done in alphanumeric order.) These two lists are kept in pointer-based representation throughout the computation. This example is used to demonstrate the simultaneous processing of more than one list component. Let *list* denote the variable storing the elements of lists, and *next* denote the pointer used within lists. Suppose $l1$ and $l2$ are the id's of the $PE$'s at the head of two lists, respectively. Initially, all the elements of these two lists are stored in distinct $PE$'s. The first step here is to connect the corresponding elements in these two lists by pointers called *rival*. The alphanumeric comparison (denoted by $\otimes$) is then performed by each $PE$ after obtaining the data from its rival. The final comparison of the two lists is then computed and the result is stored at the head $PE$ of each list. The program segment for this comparison is described in the following.

```
begin
  MatchLists (l1, l2, next, rival);
  if $rival(p) \neq NIL$ then
    $tmplist(rival(p)) \leftarrow list(p)$;
  $ResultList(p) := list(p) \otimes tmplist(p)$;
  ReduceList⟨$\otimes$⟩ (next, ResultList);
end;
```

Here, *MatchLists* matches up the corresponding elements of two lists, *ResultList* stores the result of the comparison of the corresponding elements, and *ReduceList* computes the final result of the comparison of these two lists. Before defining *MatchLists* and *ReduceList*, we first introduce three basic primitives, namely, *ReversePtr*, *AdvPtr*, and *send**. Let $A$ denote the pointers in a list, and $B$ be a variable of the same type. Primitive *ReversePtr* is defined as follows.

```
Procedure ReversePtr (A : PtrType; var B : PtrType);
begin
  if $A(p) \neq NIL$ then
    $B(A(p)) \leftarrow p$;
end;
```

Thus, *ReversePtr* is used to establish a set of pointers in $B$, which are the reverse of those in $A$. For a list in pointer-based representation, it is frequently useful to have a temporary pointer in each node to point to another node (not the next one) in the same list. Let $A$ be this temporary pointer. Initially, $A$ in each node points to the next node. Then, at each step, $A$ is updated by the value of $A$ in the node to which the current node points using $A$. *AdvPtr* primitive is used to advance $A$ for this purpose, and is defined by

```
Procedure AdvPtr (var A : PtrType);
var tmp : PtrType;
begin
  tmp(p) := NIL;
  ReversePtr (A,tmp);
  if $tmp(p) \neq NIL$ then
    $A(tmp(p)) \leftarrow A(p)$;
end;
```

Using *ReversePtr* and *AdvPtr* primitives, primitive *ReduceList* is defined below.

```
Procedure ReduceList⟨op⟩ (next : PtrType;
    var list : ListType);
var jump, rjump : PtrType;
begin
  jump(p) := next(p);
  while $jump(p) \neq NIL$ do {
    ReversePtr(jump, rjump);
    if $rjump(p) \neq NIL$ then
      $list(rjump(p)) \xleftarrow{\langle op \rangle} list(p)$;
    AdvPtr (jump); }
end;
```

Here, *jump* denotes the temporary pointers to be advanced, and using *jump*, *list* is reduced to a scalar value in a binary tree fashion. The final result after applying *op* to *list* is stored at the head of *list*.

Here, primitive *send** is used to copy the id of the current node, via a mailstop, to the destination node. The address of the destination node is stored in the mailstop, and the address of mailstop is known to the source node. Through the use of *send**, we can establish the pointers in the destination nodes to point to the source nodes. The procedure for *send** is defined as follows.

```
Procedure send* (mailstop, destination : PtrType;
    var A : PtrType);
var received : Boolean;
    letter : PtrType;
begin
  received(p) := FALSE;
  if $mailstop(p) \neq NIL$ then {
    $letter(mailstop(p)) \leftarrow p$;
    $received(mailstop(p)) \leftarrow TRUE$; }
  if received(p) then
    if $destination(p) \neq NIL$ then
      $A(destination(p)) \leftarrow letter(p)$;
end;
```

In terms of *send** and *AdvPtr*, we can define the parallel operation *MatchLists.* Let *mailstop* in each node of lists be used to store the address of the *PE* which holds the address of the *rival PE* of that node, and *jump* be temporary pointers to be advanced. Initially, we let the *rival* at the head of both lists point to each other. Then we set up the *rival* in parallel for other nodes in the lists through the use of *jump.* The procedure for *MatchLists* is defined in the following.

```
Procedure MatchLists (l1,l2 : 0..n - 1; next : PtrType;
    var rival : PtrType);
var mailstop, jump : PtrType;
begin
  rival(p) := NIL;
  mailstop(p) := NIL;
  rival(p) := l2, (p = l1);
  rival(p) := l1, (p = l2);
  jump(p) := next(p);
  while $jump(p) \neq NIL$ do {
    if $(jump(p) \neq NIL)$ and $(rival(p) \neq NIL)$ then
      $mailstop(jump(p)) \leftarrow rival(p)$;
    if $mailstop(p) \neq NIL$ then
      send* (mailstop, jump, rival);
    AdvPtr (jump); }
end;
```

It should be noted that the *jump* of a node will be *NIL* if it can not point to a node in the same list. Since *next* is *NIL* for the last node in a list initially, *AdvPtr* will handle *jump* properly.

## MAPPING STRATEGIES

Since abstract data types are represented in a distributed way on SIMD hypercube machines, the interactions among data elements play an important role in determining their performance, that is, the performance of an abstract data type may vary considerably depending on the mapping of data elements on to the physical processors. In this section we discuss the mapping strategies to reduce communication cost for various abstract data types. The rich communication structure of a hypercube interconnection network makes this mapping feasible.

The major factor in selecting a mapping for an abstraction is the communication pattern embedded in that abstraction; it illustrates the characteristics of both its normalized representation and the associated parallel primitives. The possible communication patterns can be classified into four categories, namely, neighbor communication, group communication, dynamic communication, and random communication. By neighbor communication, we mean that communication occurs only in the neighboring processors of a logical structure such as a graph. That is, only the pairs of processors which represent an edge may allow communication to occur between them. The communication pattern for graphs in pointer-based representation usually belongs to this type. For some abstract data types such as list and matrix, communication may occur within a group of data elements in a fixed binary tree pattern. Examples are the *broadcast* and *reduce⟨op⟩* operations for a matrix component. The above two categories of communication patterns can be considered as fixed communication. By dynamic communication, we mean the communication pattern which is regular, but subject to change after each step. One example is the communication pattern for the comparison of two lists in pointer-based representation. Random communication pattern means that no regular communication pattern is available. Unrelated collections usually have a random communication pattern.

If the communication pattern for an abstraction is regular (i.e., a neighbor, group, or dynamic communication), its data elements should be arranged in a way that the pair of communicating processors are just one hop apart. This will reduce the communication cost to a minimum. However, the mapping of a general graph to a hypercube structure is $NP$-complete (Refs. 5-6). Only a highly regular and sparse graph may have a suitable mapping in a hypercube if this graph is in a pointer-based form.

Another important factor in selecting a mapping is the locality of data elements within an abstraction. This factor will dominate when there is no regular communication pattern available, such as in a set, or when a suitable mapping for a regular communication pattern is unavailable, such as in an arbitrary graph. For this kind of abstract data types, it is better to place data elements in a smaller cube in a way that all the relevant (i.e., communicating) data elements are close to one another. For example, the physical processors corresponding to a mapping are constrained to lie within a small hypercube when the number of elements in a data abstraction is small. It is well known that in an $r$-dimensional hypercube, the farthest distance between any two nodes is $r$ and the average distance is only $r/2$. The mapping based on the locality also benefits the overall memory utilization and conserves hardware resources.

For detailed mapping strategies of various abstract data types on hypercube interconnection structure, readers please refer to (Ref. 7).

## SUMMARY AND CONCLUSIONS

It is difficult to write parallel programs in a modular way on fine grained parallel systems. Conventional distributed and coarse grained programming models exhibit the bottleneck of processing client requests sequentially. In this paper we investigate the parallel implementation of abstract data types on SIMD hypercube machines.

To achieve high performance in parallel programs, the requests of various clients of an abstraction should be processed in parallel. Three major factors are important in implementing an abstract data type on SIMD machines, namely, a suitably normalized representation, parallel primitives, and mapping. The normalized representation of an abstraction, which is the key to the parallel processing of multiple client requests, is based on multiple entry data structures such as indexed vectors and hash tables. Parallel primitives, which are constructed to facilitate parallel operations on an abstraction, are based on the characteristics of underlying architectures. Parallel primitives associated with the normalized representation usually make extensive use of communication for exchanging information among $PE$'s, and thus a mapping strategy must be selected to reduce the communication overhead. Two major factors in selecting a mapping are the communication pattern and locality of data elements. A suitable mapping of the normalized representation onto the underlying network of $PE$'s can greatly improve the performance of an abstract data type.

In summary, an abstract data type on SIMD machines must be in a normalized representation so that its distributed clients and servers can be merged to achieve high performance parallel programs. Parallel primitives must also be provided for facilitating the parallel processing of client requests. To improve the performance, a suitable mapping for the normalized representation of an abstraction should be used to reduce the communication cost.

## REFERENCES

1. F. B. Bastani, W. Hilal, S. S. Iyengar, "Efficient Abstract Data Type Components for Distributed and Parallel Systems," *Computer*, Oct. 1987, pp. 33-44.
2. W. D. Hillis, *The Connection Machine*, MIT Press, Cambridge, Mass., 1985.
3. W. D. Hillis, G. L. Steele, jr., "Data Parallel Algorithms," *Comm. of ACM*, Vol. 29, No. 12, Dec. 1986, pp. 1170-1183.
4. J. Backus, "Can Programming Be Liberated From the Von Neumann Style? A Functional Style and Its Algebra of Programs," *Comm. of ACM*, Vol. 21, No. 8, Aug. 1978, pp. 613-641.
5. G. Cybenko, K. N. Venkataraman, D. W. Krumme, "Hypercube Embedding is $NP$-Complete," Proceeding of the first SIAM Conference on Hypercube Multiprocessors, Oak Ridge National Laboratory, 1986, pp. 148-160.
6. A. Wagner, D. G. Corneil, "Embedding Trees in a Hypercube is $NP$-complete," Technical Report 197/87, Dept. of Computer Science, University of Toronto, 1987.
7. D. R. Leu, F. B. Bastani, "Parallel Implementation of Abstract Data Types on SIMD Hypercube Machines," Technical Report, Dept. of Computer Science, University of Houston, 1988.

# AN OPTIMALLY PORTABLE SIMD PROGRAMMING LANGUAGE

Russ Tuck
Computer Science Department, Duke University, and
Computer Science Department, University of North Carolina at Chapel Hill *

## ABSTRACT

Existing programming languages for SIMD (Single-Instruction Multiple-Data) parallel computers make implicit architectural assumptions. These limit each language to architectures satisfying its assumptions. This paper presents a theoretical foundation for developing much more portable languages for SIMD computers. It also describes work in progress on the design and implementation of such a language.

An *optimally portable* programming language for a set of architectures is one which allows each program to specify the subset of those architectures on which it must be able to run, and which then allows the program to exploit exactly those architectural features available on *all* of the target architectures. The features available on an architecture are defined to be those the architecture can implement with a constant-bounded number of operations. This definition ensures reasonable execution efficiency, and identifies architectural differences which are relevant to algorithm selection.

An optimally portable programming language for SIMD computers, called Porta-SIMD (porta-simm'd), is being developed to demonstrate these ideas. Based on C++, it currently runs on the Connection Machine and Pixel-Planes 4.

Keywords: Portable, SIMD Parallel, Programming Language, Porta-SIMD, Taxonomy, Pixel-Planes, Connection Machine, C++.

## INTRODUCTION

Portable high-level languages for von Neumann computers are major accomplishments in computer science. These languages have radically improved the quality, cost, reliability, and availability of software. However, the greater architectural diversity of SIMD (Single-Instruction Multiple-Data) computers has so far kept them from fully benefiting from such languages. Each existing SIMD language contains architectural assumptions which make it suitable for programming only a certain subset of SIMD machines.

Optimal portability is a new concept which can guide the development of much more portable SIMD programming languages. It is based on the recognition that some differences among SIMD architectures significantly influence algorithm selection. These should not be completely hidden from the programmer.

The programmer makes an algorithm's architectural assumptions explicit by expressing the algorithm as a program for a particular set of architectures. These architectural assumptions precisely define the program's portability. The programmer may then take full advantage of all architectural features common to all members of that set, and no more. Selecting a small set of very similar architectures limits a program's portability, but allows it to take full advantage of specialized features the members share. Selecting a large diverse set of architectures produces a program that is very portable, but may not take full advantage of some of the architectures. This selectable tradeoff between breadth and power provides optimal portability.

This is entirely consistent with Chandy and Misra's (Ref. 8) ideas on algorithm portability. They advocate developing algorithms that are progressively more tightly bound to particular architectures, until an algorithm is specialized sufficiently to provide the desired performance. They provide a language-independent notation for expressing algorithms during development, which must be translated into a language for a particular architecture before execution. With an optimally portable language, this would not have to be a different language for each target architecture. Avoiding the necessity of learning and remembering details of a different language for each architecture is a significant time and cost savings.

In practice, an optimally portable language for a set of architectures needs both a definition and a taxonomy of that set. These provide a precise way to specify the architectures on which a program must run. They also contribute to improved understanding of the architectures, and their algorithms and languages. Both a definition and a taxonomy of SIMD architectures are given in the section "A SIMD Taxonomy for Programmers."

Existing SIMD programming languages are not optimally portable. They are built on a variety of inflexible architectural assumptions, including specific processor interconnection networks and the presence or absence of features like local addressing of memory. The section titled "Existing SIMD Languages" surveys these languages.

I am currently working on the design and implementation of a new optimally portable language for SIMD computers: Porta-SIMD (pronounced porta-simm'd). Its overall structure is modeled on the proposed SIMD taxonomy for programmers, allowing it to present to the programmer an appropriate programming model for any subset of SIMD architectures. It is intended to demonstrate the feasibility of designing, implementing, and using optimally portable languages. The ongoing design and implementation of Porta-SIMD are discussed in the section "An Optimally Portable Language."

## OPTIMAL PORTABILITY

*Optimal portability* is best defined in terms of a few supporting definitions. An *abstract architecture* is the set of fundamental data types and operations provided by a computer, without regard to how the data and operations are represented. It does not include implementation details such as the the amount of memory present in a machine, or the number of processors in a parallel machine. Except where explicitly stated otherwise, I will use *architecture* as a synonym for abstract architecture.

The members of a set of architectures are *equivalent* if and only if their intersection is identical to their union. The *union* of a set of architectures is an architecture containing all data types and operations contained in any member of the set. The *intersection* of a set $S$ of architectures is an architecture constructed as follows:

1. Let architecture $u$ be the union of $S$. To each member $A_i$ of $S$ add each data type and operation in $u$ which $A_i$ can simulate with a constant number of its own data elements and operations.

2. Take the intersection of the sets of data types and operations of all members of $S$, as augmented by the previous step, to create the intersection architecture.

The intersection of a set of architectures will also be called the *shared architecture* of the set. These definitions imply that any member of a set of equivalent architectures can simulate the operation of any other member, and the number of native operations they execute will be within a constant factor of each other.

*This work was supported by the Pixel-Planes Project, Henry Fuchs and John Poulton, P.I.s, and its grants: National Science Foundation grant #MIP-8601552, Defense Advanced Research Projects Agency order #6090, Office of Naval Research contract #N0014-86-K-0680; and by the GRIP Project, Frederick Brooks, P.I., under National Institutes of Health grant #RR 02170. Access to a Connection Machine was provided by the Advanced Computing Research Facility at Argonne National Laboratories, under grants NSF-ASC-8808327 and DOE-W-31-109-ENG-38. Author's phone and electronic address: *(919) 962-1755* or *(919) 684-5110*; *tuck@cs.unc.edu* or rt@cs.duke.edu.

A particular computer may be considered to implement only a single set of equivalent architectures. This set must be the set of architectures equivalent to the architecture defined by the computer's lowest-level publically documented programming interface. For most sequential computers, that interface is assembly language. For some SIMD computers it is a library.

A program is *portable* across a set $S$ of architectures if and only if it can be compiled and correctly executed on the shared architecture of $S$. Such a program can therefore be compiled and correctly executed on every member of $S$. The architecture on which a program is intended to run is called the program's *target architecture*. A program is said to *use* a data type or operation if and only if it contains a direct or indirect reference to a language feature that provides a capability equivalent to that data type or operation.

A programming language $L$ is *optimally portable* for a set $S$ of architectures if and only if all of the following are true:

- $L$ requires each program $p$ to specify some architecture $A_p \in S$ as its target architecture. (A default target architecture may be implicitly specified in the absence of an explicit specification.)
- $L$ does not allow $p$ to use any data type or operation not in $A_p$.
- $L$ allows $p$ to use any data type or operation in $A_p$.

This definition implies that $p$ is portable across any set $S_1 \subseteq S$ such that $A_p$ is the shared architecture of $S_1$, including the maximal such set, $S_p$. Therefore, $p$ cannot be portable across a larger set of architectures without giving up the use of one or more data types or operations. In addition, $p$ cannot use additional data types or operations without adding to $A_p$. This would potentially reduce $p$'s portability by removing architectures from $S_p$.

A few points in the definition of optimal portability deserve discussion. It is difficult, perhaps impossible, to find a simple set of rules to accurately and impartially determine the programmer-visible architecture of every computer. Computer systems have many layers of architecture, and features are sometimes implemented in the "wrong" layer conceptually to improve performance. However, identifying such features is a matter of judgement which is not easily reduced to simple rules. Great care has been taken in constructing the definitions above, but they are not perfect.

It is important to construct a good test for whether an abstract architecture can usefully simulate some data type or operation. Any Turing-equivalent machine may simulate any architecture, but not always with useful performance. The constant-bounded criterion above for operations and data ensures reasonable performance and fits well with intuitive notions of equivalent architectures. It also makes equivalence transitive. (Suppose architecture $A_x$ can simulate architecture $A_y$ in $op(A_x, A_y)$ operations, and equivalence is denoted by "=". Then $A_i = A_j$ and $A_j = A_k$ implies $op(A_i, A_j) \leq op(A_i, A_j) op(A_j, A_k)$, which implies $A_i = A_k$ because $op(A_i, A_j)$ and $op(A_j, A_k)$ are constants.) Logarithmic and polynomial bounds do not have this important property.

In some cases, a single machine may be reasonably described by two or more quite different abstract architectures. As long as they are equivalent, they are equally valid descriptions. For example, a bit-serial SIMD machine may be described as having operations on bits, on multi-bit integers, or on floating-point numbers. Operations on the multi-bit data types can be simulated by a constant number of bit-serial operations. The constant (which may be over 1000) depends on the nature and size (in bits) of the simulated data type, but does not depend on the values stored in data elements of that type. The architectures are equivalent. This is consistent with the common practice of building implementations of a single architecture with varying execution speeds.

Another example is a SIMD machine with a 2-dimensional grid interconnection network which allows communication in parallel between pairs of adjacent PEs (Processing Elements), using its lowest-level publically documented programing interface.. With an additional layer of software to do automatic routing, it might also be described as providing communication between arbitrary pairs of PEs. The number of operations required to simulate arbitrary communication with this network depends heavily on the dynamically chosen communication pattern. A lower bound for the worst case is the diameter of the network, which is at least the square root of the number of PEs. Since a SIMD architecture does not specify a maximum number of PEs, this is not a constant bound. Therefore, the two descriptions are not equivalent, and only the first is part of a valid abstract architecture for this machine.

However, if the automatic routing software were hidden beneath the lowest-level publically documented programming interface, the architecture would be considered by the above definitions to provide communication between arbitrary pairs of PEs.

There are several reasons to define a machine's architecture by its lowest-level publically documented programming interface, rather than by its hardware. A programmer has no access to the hardware except through this interface. Hardware documentation is not always publicly available; it is often less complete and precise than the programming interface, largely because programming interfaces must be well documented in order for important software to be developed. Machine builders are free to implement a single architecture with different hardware designs, transparently to the programmer. These identically programmed machines should be considered to have the same architecture (from a programmer's perspective).

It is difficult to define precisely which data types and operations a program uses. The important feature of the definition of *use* above is that usage is defined with respect to the source code, not the compiled object code. This prevents the compiler from making features not available in the target architecture available to the program by generating code to simulate them with arbitrary numbers of data elements and operations. (Of course, a compiler generating code for an architecture equivalent to $A_p$ may generate a constant number of data elements and operations to simulate data types and operations of $A_p$.)

Prohibiting compilers from simulating data types and operations not present in $A_p$ ensures portability with useful performance, not just theoretical portability. This does not restrict the function of programs, since $p$ may simulate such data types and operations itself. The implementers of $L$ may even provide, as a convenience to programmers, a package written in $L$ to do this simulation.

# A SIMD TAXONOMY FOR PROGRAMMERS

A programming language is optimally portable only for a specific set of architectures. Therefore, any optimally portable SIMD programming language will require a definition of SIMD architectures.

## Definition of SIMD Architectures

An architecture $A$ is a SIMD architecture if and only if all of the following are true:

- $A$ has a host computer which handles ordinary scalar computations and flow control, and which broadcasts instructions, one at a time, to all PEs (Processing Elements).
- $A$ has $n > 1$ identical PEs which all execute, simultaneously, each instruction broadcast by the host.
- Each PE is able to evaluate basic arithmetic and logical expressions.

I believe every useful SIMD architecture also has the following properties:

1. Each PE is able, in response to broadcast instructions, to independently choose whether to ignore instructions to modify its memory. (PEs executing all instructions are *enabled*, while those ignoring instructions to modify memory are *disabled*. PEs can be considered to have an *enable-bit* which is 1 only in enabled PEs.)
2. Each PE is able to compute its unique PE number $0 \leq p < n - 1$, given sufficient time.
3. Each PE has its own private memory.

Property 1 can be simulated with a constant number of ordinary arithmetic and logical operations. Architectures that do not have this property are therefore equivalent to those that do, and can be considered to have it. This property takes many different but equivalent forms in various machines, with it being possible to ignore different subsets of an instruction set.

Property 2 certainly holds for all architectures which have a connected communication graph, and which allow any single PE to be distinguished in any way. It also holds for all architectures with parallel input, since the data being read can be the PE numbers. Property 2 holds if an architecture can load into each PE a different element of a set of distinct values, by any means, since this set can be the PE numbers. If there is a SIMD architecture which does not have this property, I do not think it is very interesting because the PEs cannot be given unique predetermined data on which to operate. That is the whole purpose of a SIMD architecture.

The only claimed exception to property 3, that I am aware of, is an alternative set of architectures where PEs access a global memory space through a network of some kind (e.g., (Ref. 20, pp. 326-327)). I believe that any such architecture is equivalent to a local-memory architecture in which the PEs are connected to each other by the same network that connects the PEs to the global memory.

Specifically, the BSP (Burroughs Scientific Processor) (Ref. 20, pp. 326-327, 410-422) is the only non-local memory architecture I know of. It is equivalent to a large subset of the CM (Connection Machine) architecture (Refs. 18, 10, 1). (Both architectures are discussed briefly in a later section.) The BSP can simulate the CM simply by assigning a distinct portion of global memory to each PE for private use, and accessing memory assigned to other PEs only to simulate communication. Similarly, the CM can simulate the BSP by using its communication primitives to access memory, treating all the private memory as a single global memory space. Both simulations take constant time, so the BSP's global memory and arbitrary PE to memory interconnection network is equivalent to the CM's local memory and a subset of its communication primitives. The only difference between the architectures is that the CM has somewhat more powerful mechanisms for resolving simultaneous accesses to a single memory location.

If any of these properties is not true of all SIMD architectures, then the taxonomy below is considered to have an additional dimension for each such property. Because all architectures currently classified by this taxonomy have the same coordinates along these dimensions, those coordinates will not be mentioned further.

## Taxonomy of SIMD Architectures

An optimally portable SIMD programming language must recognize and handle the full diversity of SIMD architectures that exist within this definition. A taxonomy of SIMD architectures will be crucial to this task. Although many architectural differences can be almost completely hidden by a high-level language, others fundamentally influence the programmer's algorithm selection. To be most useful for portable language design, the taxonomy should exclude the former and focus on the latter. The differences that do not influence algorithm selection can be uniformly hidden from the programmer by language abstraction. However, an optimally portable language must make the remaining differences visible to the programmer, in the form of language features which exploit the target architecture.

Previous SIMD taxonomies have been constructed with different goals, and consider some architectural features which need not be visible to a programmer. Examples include work by Hwang and Briggs (Ref. 20, chapters 5-6), and a tutorial by Seitz (Ref. 32). Fountain (Ref. 13) and Gerritsen (Ref. 16) compare certain SIMD implementations at a level appropriate for system designers and architects, rather than programmers. An extended abstract by Jamieson (Ref. 21) considers matching algorithms with all kinds of parallel architectures, not just SIMD. Karp (Ref. 22) presents a taxonomy restricted to "those aspects that affect coding style," but considers only MIMD (Multiple-Instruction Multiple-Data) architectures. These taxonomies not suited for designing an optimally portable SIMD language.

Beginning with the most important, the architectural differences that can significantly influence algorithm selection include:

Topology — the labeling and adjacencies of the PEs;

Communication — whether each PE can read/write data to/from (0) no other PE, (1) a globally-selected adjacent PE, (2) a globally-selected location in a locally-selected adjacent PE, or (3) a locally-selected location in a locally-selected adjacent PE;

Collision Resolution — whether multiple writes to the same location under communication types (2) and (3) are resolved by (0) serializing the accesses, or (1) combining them by applying an arithmetic or logical operation;

Local Addressing — whether local PEs' memories can be addressed (0) only by a single globally computed address, or (1) also by addresses computed locally at each PE;

Global Logical-Or/Multiple-Response Resolver — whether the host can determine in a constant number of operations (0) neither of the following, (1) if any PE has a non-zero value in a certain field of memory (global logical-or), or (2) the identity of at least one PE having a non-zero value in a certain field of memory, if such a PE exists (multiple-response resolver);

Parallel I/O (Input/Output) — whether it is (0) impossible or (1) possible for all PEs to transfer data to and from a mass storage subsystem in parallel;

PE to Host I/O — whether the host can obtain data from (0) no PE, (1) only a subset of PEs, or (2) any selected PE.

These architectural differences define a discrete 7-dimensional space. A SIMD architecture can be characterized by a 7-tuple giving its location in this space. All the dimensions except the first, topology, have a finite set of values enumerated in their descriptions above. As new SIMD architectures are developed, it may be necessary to add new dimensions to this taxonomy to accomodate newly invented architectural features.

Topology and communication are very closely related. Without inter-PE communication, all topologies are equivalent. However, a SIMD architecture without inter-PE communication may still use a particular topology. The 2D topology of Pixel-Planes (discussed below) is a good example. The $(x, y)$ labeling and adjacency of PEs are necessary to evaluate bilinear expressions, and to map computed values from PEs to pixels.

In both communication and local addressing, local selection subsumes global selection, since it is trivial to make the same local selection at all PEs.

Communication type (3) provides local addressing as a side effect. It would be conceptually cleaner to eliminate this communication option and allow it to be simulated by communication type (2) and local addressing. This was not done because the simulation takes operations proportional to the maximum number of access to any one PE, and because communication type (3) is a single operation of the CM and BSP. However, both these machines essentially perform the same simulation in hardware or microcode. This is an example of an operation moved down a layer in the architecture for performance reasons. It exposes a limitation of the methods used here to delineate programmer-visible architectures.

Global logical-or has several equivalent variants. These include the similar "global logical-and", and the related special case "all enables off", which is the inverse of global logical-or applied to the bit which determines whether local memory is write-protected.

This taxonomy has not yet been extended to include two architectural features. The first is cut-through routing of data between PEs. Cut-through routing allows some PEs to send data to non-adjacent PEs, provided the intervening PEs do not send data. The Princeton Engine (Ref. 9) and the ASP (Associative String Processor) (Ref. 23), both 1D architectures, use this.

The second feature is performing parallel-prefix as a single operation. The CM provides this capability, though the microcode must simulate it in a number of operations logarithmic in the number of PEs involved. (This can be proven, since each PE can only combine two values in a single operation.) This is another example of an operation moved down a layer in the architecture for performance reasons.

This taxonomy of SIMD architectures specifically excludes a variety of differences which may be very important to computer architects, but which need not influence algorithm selection. Among these are word length, memory structure and size, special hardware for floating-point operations, and details of scalar and parallel machine instructions. These are all routinely hidden by the abstractions of ordinary high-level languages, and handled by compilers. Of course, the hiding is sometimes imperfect, and it is possible to write non-portable programs which depend on word length, byte order, or other machine-specific details. However, a few simple coding rules are generally sufficient to avoid these problems. Neither the problems nor the solutions differ fundamentally between sequential and SIMD-parallel architectures. SIMD languages should be able to hide these architectural differences as well as, but not necessarily better than, sequential languages.

Figure 1 represents as a tree the space of SIMD architectures defined by the proposed taxonomy. The labels on the left identify the dimension of space represented by each level of branching. The label at each interior tree node identifies the location of the subtree rooted at that node along one dimension of architectural space. Leaf nodes represent selected published SIMD architectures. Subtrees containing no selected architectures are not shown. The

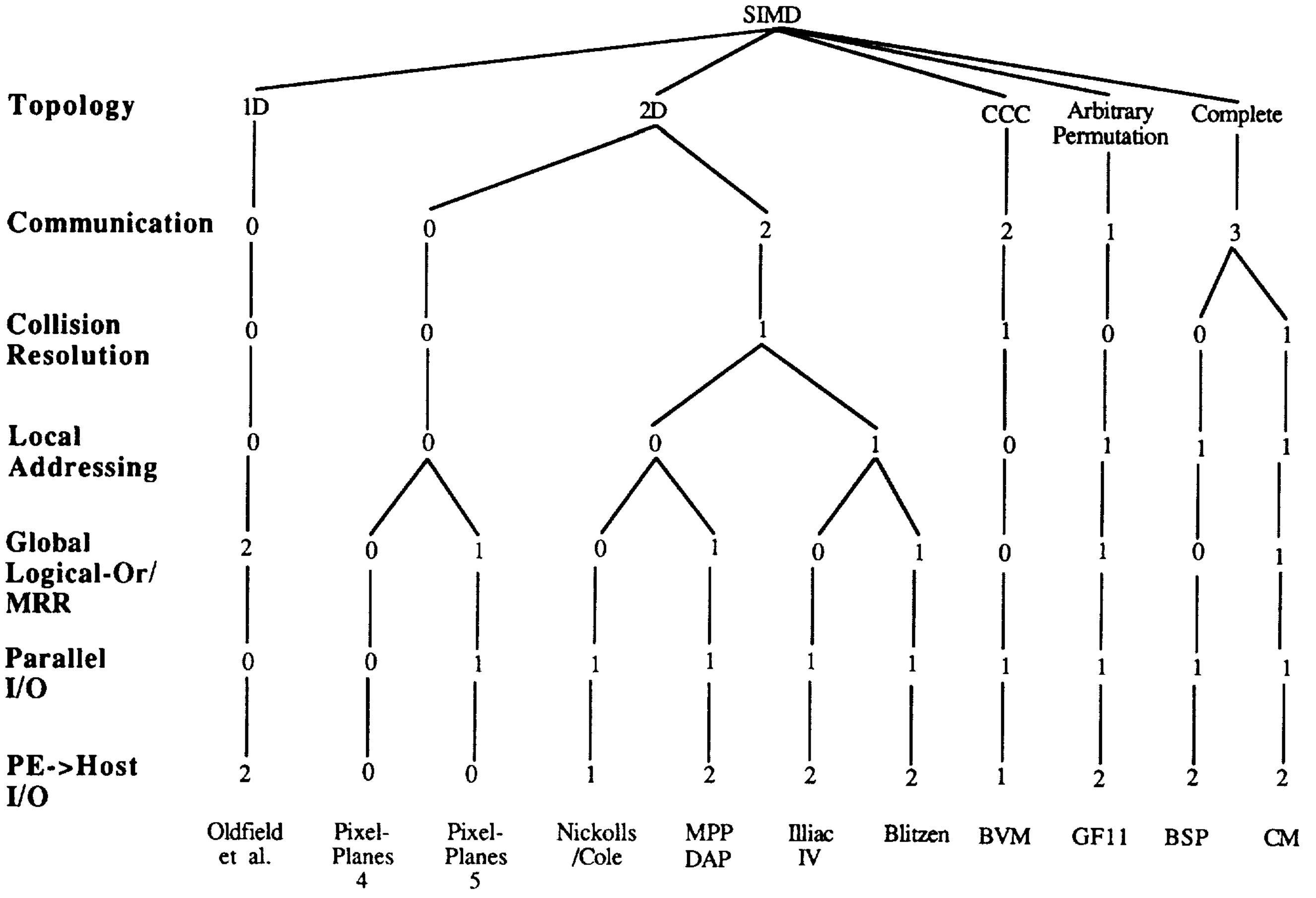


Figure 1. SIMD Architectures

space available is not sufficient for the entire set of SIMD architectures, so I have included as representative a variety as possible. Additional references are always welcome.

This taxonomy has the desirable characteristic that it is easy to determine that certain architectures are subsets of others. This is useful because programs for a particular architecture are portable to all supersets of that architecture. The enumerated dimensions all obey a strict subset ordering. Therefore, one architecture is a subset of another if they have the same topology and if each of the remaining elements of the first 5-tuple is no greater than the corresponding element of the second 5-tuple. For example, the MPP (2D, 2, 1, 0, 1, 1, 2) is a subset of BLITZEN (2D, 2, 1, 1, 1, 1, 2), but not of Pixel-Planes 4 (2D, 0, 0, 0, 0, 0, 0).

In a few special cases, an architecture may fail this criterion and yet be a subset of another. Examples include the following:

- For topologies with a constant number of neighbors per PE, local and global selection of neighbors for communication are equivalent. Collision resolution by serialization or combination are also equivalent for these topologies. Of the topologies discussed below, 1D , 2D, and CCC have a constant number of neighbors per PE, but Hypercube, Arbitrary Permutation, and Complete do not.
- Communication type (3) effectively provides local addressing type (1).
- Global logical-or effectively provides arbitrary PE to host I/O (2).
- An architecture which has parallel I/O to a random access storage device which the host can also manipulate, but does not have PE to host I/O, can simulate arbitrary PE to host I/O. A second architecture differing from the first only in having PE to host I/O and lacking parallel I/O is therefore a subset of the first.

In each case, the result is that adjacent points in architectural space are related by the equivalence rather than the subset relation.

## Survey of SIMD Architectures

Most of the remainder of this section surveys the SIMD architectures appearing in figure 1. It shows how they fit within the space of the proposed taxonomy, giving evidence that the taxonomy is reasonably complete. For simplicity, each architecture is described as if it were the equivalent canonical architecture defined by its location in architectural space. The proofs of equivalence are generally not difficult, but will not be presented here. The architectures will be treated in order from left to right across the tree of figure 1. Each heading includes the coordinates of the architecture it describes.

A tremendous variety of topologies is possible for SIMD machines. In practice, though, a few simple topologies are used by most SIMD architectures. The simplest, 1D (1-dimensional), is a property of SIMD architectures. Although it will not be mentioned in their descriptions, all the other topologies contain it in addition to their advertised features. A 1D topology simply labels each of $n$ PEs with a unique integer $0 < x \leq n$. PE $x$ has two neighbors, $x - 1$ and $x + 1$. Boundary conditions can be defined so PEs 0 and $n - 1$ are neighbors (forming a ring), or so their missing neighbors (PEs $-1$ and $n$) always provide null values (forming a line segment). Since these architectures are equivalent, they will not be distinguished.

The most common topology is 2D, which labels each PE with an ordered pair $(x, y)$ such that $0 < x < X$, $0 < y < Y$, and $n = XY$. Each PE has four or eight neighbors, differing by plus or minus one in one or both dimensions. Boundary conditions can be defined to provide wrap-around (forming a torus), or null boundary values (forming a rectangular sheet). The architectures using all the topologies allowed by these choices are equivalent, so they will not be distinguished. The remaining topologies will be discussed as necessary with the architectures using them. These include Cube-Connected Cycles, Arbitrary, and Complete graphs.

**Oldfield/Williams/Wiseman/Brûlé (1D, 0, 0, 0, 2, 0, 2)**—J. V. Oldfield, R. D. Williams, N. E. Wiseman, and M. R. Brûlé propose a CAM (Content Addressable Memory) with sufficient processing power at each row to qualify as a SIMD architecture (Ref. 26). (Simulation of arithmetic operations and the enable-bit is rather laborious, but possible with a constant number of operations.) There is no communication between PEs, but the 1D topology provides row addresses. There is no local addressing or parallel I/O.

**Pixel-Planes 4 (2D, 0, 0, 0, 0, 0, 0)**—Pixel-Planes 4 (Refs. 15, 14, 12) is designed for high-performance interactive graphics applications. It has a simple 2D topology. There is no communication between PEs, but the PE coordinates $(x, y)$ are used to compute bilinear expressions of the form $ax + by + c$ at each PE (for scalar floating-point values $a$, $b$, and $c$). Although there is special hardware to evaluate these expressions quickly, they can be computed in constant time without it. These expressions can be used to display polygons and spheres very quickly. There is no local addressing, global logical-or, parallel I/O, or PE to host I/O. However, images can be displayed on a video monitor, with each PE providing the data for one pixel of the image.

Video display of data in most architectures is done by parallel output to a frame buffer. The fact that data can be seen, but not otherwise externally accessed due to the absence of I/O, is a minor anomaly of Pixel-Planes 4. Because it cannot influence algorithm selection, there is no need to recognize it in the taxonomy.

**Pixel-Planes 5 (2D, 0, 0, 0, 1, 1, 0)**—Pixel-Planes 5 (Refs. 17, 12) is designed to provide greater speed and flexibility in order to interactively display more complex and realistic images. With regard to the taxonomy, it differs architecturally from Pixel-Planes 4 only in providing global logical-or and parallel I/O.

However, it has hardware support for biquadratic expressions in $x$ and $y$, in addition to bilinear expressions. It also has a MIMD host. Both of these differences provide significant constant-bounded speedups. In addition, multiple sets of PEs can be combined in a single system. A program may choose to treat them as separate machines controlled by different processes in the host, or as a single large machine controlled by a single logical process. This is similar to the partitioning allowed by the Connection Machine.

**Nickolls/Cole (2D, 2, 1, 0, 0, 1, 1)**—P. M. Nickolls and T. W. Cole (Ref. 25) present a fault-tolerant 2D processor array for image synthesis. It has a 2D topology, with globally selected neighbor communication. It does not provide local memory addressing or global logical-or. It also provides parallel I/O and allows the host to obtain data from certain PEs at the edge of the PE array.

The distinguishing feature of this machine is not visible architectually. It is a programmable interconnection network that allows defective PEs and network connections to be configured out of the machine by deleting rows or columns containing the defective hardware.

**MPP (2D, 2, 1, 0, 1, 1, 2)**—The MPP (Massively Parallel Processor) (Ref. 29) has a 2D topology and allows each PE to communicate with a locally chosen neighbor. There is only global memory addressing. Global logical-or and parallel I/O are provided, and the host can obtain data from any PE.

**DAP (2D, 2, 1, 0, 1, 1, 2)**—The Active Memory Technology DAP (Distributed Array Processor) (Ref. 27) — formerly the ICL DAP — architecture appears identical to that of the MPP, at the level under discussion. (However, I have not been able to verify support for global logical-or.)

**Illiac IV (2D, 2, 1, 1, 0, 1, 2)**—The Illiac IV (Ref. 19) is an early SIMD architecture. Its 2D topology provides communication between each PE and its immediate neighbors, with local neighbor selection. The PEs have local addressing of their memories. Global logical-or is not provided. There is support for parallel I/O, and PE to host I/O from any PE.

**BLITZEN (2D, 2, 1, 1, 1, 1, 2)**—BLITZEN (Refs. 6, 11, 7) builds on many ideas from the MPP. Its architecture differs primarily in providing local addressing of PE memory. The architecture is almost identical, at this level, to that of the Illiac IV, differing only in supporting global logical-or.

**BVM (CCC, 2, 1, 0, 0, 1, 1)**—The BVM (Boolean Vector Machine) (Ref. 38) arranges PEs in a CCC (Cube-Connected Cycles) network (Ref. 30). Each PE can communicate with its choice of its three neighbor PEs. Only global memory addressing is provided. Global logical-or is not provided. Parallel I/O is supported, and the host can read data directly from a single distinguished PE.

**GF11 (Arbitrary Permutation, 1, 0, 1, 1, 1, 2)**—The GF11 (designed to achieve 11 GFLOPS) (Refs. 5, 4) can provide multiple arbitrary permutations for inter-PE communication. Each permutation is defined by a directed graph which specifies the PE from which each PE receives data, with exactly one PE receiving data from each PE. A particular permutation is globally selected for each communication operation between PEs.

Local addressing, global logical-or, parallel I/O, and arbitrary PE to host I/O are all supported.

**BSP (Complete, 3, 0, 1, 0, 1, 2)**—The BSP (Burroughs Scientific Processor) (Ref. 20, pp. 326-327, 410-422) architecture provides a complete interconnection graph, and allows each PE to determine locally with which neighbor to communicate, and which memory location to use. Since the complete graph makes neighbors of every pair of PEs, this provides completely arbitrary locally controlled inter-PE communication. Collision resolution is by serialization.

Local addressing, parallel I/O, and arbitrary PE to host I/O are all supported. Global logical-or is not.

As discussed above, although the BSP's memory is physically global, its architecture is fully equivalent to the description just given.

**CM (Complete, 3, 1, 1, 1, 1, 2)**—The Thinking Machines CM (Connection Machine) (Refs. 18, 10, 1) architecture provides a complete interconnection graph, and allows each PE to determine locally with which neighbor to communicate, and which memory location to use. Since the complete graph makes neighbors of every pair of PEs, this provides completely arbitrary locally controlled inter-PE communication. Collision resolution can be by serialization or combination.

Local addressing, global logical-or, parallel I/O, and arbitrary PE to host I/O are all supported.

There is a discrepancy between the CM's architecture, which provides a complete graph connecting PEs, and its hardware, which provides a hypercube (also known as a binary $n$-cube). This is a result of its system software and the definitions given earlier in this paper. As previously discussed, those definitions require a machine's architecture to be equivalent to the lowest-level publically documented programming interface. For the CM, that interface is currently Paris (Parallel Instruction Set) (Ref. 1). Paris's operations provide the communication system described above, but they are currently implemented by a physical hypercube with routing hardware. Paris operations can take time proportional to the number of PEs, so the architecture and hardware are not equivalent.

## Evaluating The Taxonomy

It is probably not possible to prove that a taxonomy of SIMD architectures is complete, in the sense of adequately classifying all possible architectures that will ever be imagined. A more reasonable test of such a taxonomy is twofold:

- Does it adequately classify each SIMD architecture in the literature?
- Does it adequately classify every SIMD architecture which could be formed by taking different combinations of features from SIMD architectures in the literature?

The previous paragraphs have begun the work of showing that the proposed taxonomy satisfies the first of these criteria.

The nature of the proposed taxonomy makes the second criterion trivial to establish, once the first has been established. The taxonomy defines a multidimensional orthogonal space without holes, with a one-to-one and onto relation between dimensions and architectural features. This ensures that any combination of features corresponds to a single defined point in the architectural space.

## EXISTING SIMD LANGUAGES

The research reported in this paper is primarily concerned with procedural languages, with a level of abstraction similar to C, C++, Pascal, or Fortran. Languages of this type both allow and require the programmer to express an algorithm unambiguously. Except for eliminating obviously redundant operations arising from the way an operation is expressed, the compiler for such a language is not involved in algorithm selection.

Some other families of languages allow the programmer to express the computation in a less algorthmic form, leaving the language implementation more latitude in choosing an exact algorithm. Some claim that the relative algorithm independence of the program allows greater portability among diverse parallel architectures. This is most often claimed with regard to modest parallelism on MIMD (multiple-instruction multiple-data) architectures. However, the way the problem is stated by the programmer can have a perhaps subtle but nevertheless profound effect on the algorithm ultimately used. In my opinion, this effect often ties such programs to a particular architecture as effectively as a procedural program expressing the same algorithm. I am not aware of any work on the use of non-procedural languages to programm SIMD architectures. Non-procedural languages will not be discussed further.

### Survey of SIMD Languages

A careful search of the literature has found no SIMD programming languages satisfying the definition of optimal portability. Most existing languages for SIMD computers include implicit architectural assumptions. These limit them to some subset of the architectural space defined in the previous section. Some languages are not portable at all. To my knowledge, only one language, Fortran 8x, has been implemented on more than one SIMD machine. However, none is a complete implementation, and it is not clear how similar the subsets are. In the brief survey of SIMD languages below, languages other than Fortran 8x are grouped by machines. Very low-level languages are not considered, leaving no languages to discuss for some machines.

**Illiac IV Languages**—Three main languages were developed for the Illiac IV: GLYPNIR (Algol-like), CFD (Fortran-based), and IVTRAN (Fortran-based). (Ref. 19) All require the programmer to use and understand low-level hardware features and limitations. They are not true high-level languages. A more portable Pascal-based language called Actus (Ref. 28) was also developed. Actus is limited by its assumption of 2D grid communication.

**MPP Language**—The MPP's implementation of Parallel Pascal also fails to insulate programmers from hardware details, contrary to the language definition. Even as defined, Parallel Pascal is suitable only for architectures with a 2-dimensional rectangular inter-PE communication network. (Ref. 29)

**CM Languages**—Likewise, C* and Connection Machine Lisp, two admirably well-designed high-level languages for the CM, assume the presence of the CM's powerful, expensive, and almost unique support of arbitrary inter-PE communication. (Refs. 10, 31, 33)

**BVM Language**—BVL-0 (Boolean Vector Language 0) (Refs. 36, 37) is a C-like language for the BVM. It was designed to be the only language for the BVM, so it includes some very low-level machine-specific features. It assumes the presence of a CCC network, and does not provide for features not present in the BVM, like local addressing. Although it could be adapted for use on other architectures with a constant number of adjacent PEs, programs written to use the BVM's CCC network would have to be rewritten.

**BSP Language**—The BSP Fortran Vectorizer (Ref. 20, pp. 417-422) combines some automatic vectorization of ordinary Fortran with some vector-oriented language extensions. Some of these extensions assume the presence of the BSP's arbitrary communication.

**Fortran 8x**—A language consisting of Fortran 77 with some VAX extensions and some proposed Fortran 8x array extensions and a few machine-specific features was proposed in 1984 (Ref. 24), but not implemented (Ref. 3). More recently, a subset of Fortran 77, with proposed Fortran 8x array extensions (including some "removed extensions"), has been implemented for the CM (Ref. 3). FORTRAN-PLUS for the DAP 500 is an implementation of Fortran 77, minus I/O facilities, plus some proposed Fortran 8x array extensions (Refs. 27, 2). It is not yet clear how compatible these implementations are.

The proposed Fortran 8x standard (Ref. 35) is the most portable language yet implemented for SIMD architectures. Although it is not optimally portable, its "removed extensions" are a step in that direction because they can be implemented on those architectures that support them efficiently. They include vector-valued array subscripts, which require arbitrary communication. Still, Fortran 8x requires communication and uses 2D grid communication heavily, so it cannot be implemented on all SIMD architectures.

### Existing Languages Fail

Each of these languages contains embedded assumptions about the architecture or architectures on which programs will run, violating the first part of the definition of optimal portability. The discussion of each language commented on these assumptions. Every language discussed allowed the use of one or more features not present in all architectures, and most failed to allow the use of some feature present in some architecture. Therefore, they all failed to satisfy the second or third part of the definition of optimal portability.

## AN OPTIMALLY PORTABLE LANGUAGE

A programming model is a complete description of the visible features and behavior of a computer system, as seen by a program. One reason existing SIMD languages are not optimally portable is each one provides only a single programming model, reflecting a fixed set of architectural features and assumptions. The second programming model provided by Fortran 8x's "removed extensions" is a small step away from this problem, but Fortran 8x still embodies many architectural assumptions.

An optimally portable SIMD language must support a family of programming models corresponding to the architectures defined by a taxonomy like the one proposed above. Each model is specified by the coordinates of its point in architectural space. Thus, each model embodies the architectural requirements of the algorithms expressed in that model.

Porta-SIMD is a new language which will provide these programming models. Its design and prototype implementation are being carried out to demonstrate the feasibility and power of optimally portable SIMD languages. It is not intended to be the only or ultimate such language, but to stimulate the development and use of optimally portable languages. For this reason, some compromises have been made in aesthetic details of the language, and in performance, in order to proceed in a timely manner with limited resources.

These considerations contributed to the choice of C++ (Ref. 34) as the base language for Porta-SIMD. There was no need nor time to invent new syntax and semantics for the scalar and sequential sections of SIMD programs, and much to be gained by using a language with which programmers were already familiar. SIMD parallel datatypes and operations can be expressed as classes and overloaded operators in C++, extending the language cleanly without modifying the compiler. This would not have been true with Fortran, C, or Pascal.

Porta-SIMD defines a set of classes, one per data type, for each programming model, and a model for each point in the architectural space defined by the taxonomy proposed above. The models are derived (using C++ inheritance) from the base model, which implements the "least common denominator" SIMD architecture (1D, 0, 0, 0, 0, 0, 0). C++'s coming multiple inheritance will be used to derive an arbitrary model from the base model and an additional model for each architectural dimension along which the arbitrary model has features above the base model. This will prevent the implementation effort from exploding combinatorially with the size of architectural space.

```
/* Define programming model: (2D,0,0,0,0,0,0) */
#include <simd_int_2d.h>
simd_mach_2d mach;

/* square accepts the upper left and lower right
 * corners of a square.  Returns 1 in each PE
 * inside the square, 0 in each PE outside.
 */
simd_int_2d square(int x1, int y1, int x2, int y2)
{
  simd_int_2d inside(mach, 1);
  simd_int_2d x(mach, 16), y(mach, 16);
  inside = 1;
  x.coord_x();
  y.coord_y();
  inside &= (x > x1);
  inside &= (y > y1);
  inside &= (x < x2);
  inside &= (y < y2);
  return(inside);
}

main()
{
  display(-square(2,6,24,57));
}
```

Figure 2. Example Porta-SIMD program.

Parallel expressions are evaluated at each active PE according to the normal C++ rules.

A parallel language needs parallel control structures, as well as parallel data types. It is sufficient to extend the semantics of the `if` statement to allow a parallel value in the test expression. An element of this value is used by each PE to to determine whether to execute the body of the `if` or the `else` clause following the test. Unfortunately, C++ does not provide a means to extend the semantics of control structures, like it does for data types. This semantic extension could be accomplished by a conceptually simple Porta-SIMD to C++ pre-processor which replaced parallel `if` statements with small blocks of code to enable and disable PEs appropriately. Unfortunately, writing such a pre-processor (or deriving one by modifying a C++ compiler) is a difficult and time-consuming task in practice. For now, a few macros are used to express parallel `if` statements, instead. For example, if p is a parallel variable,

```
if (p)
          a;
else
          b;
```

is instead written as

```
IF (p)
          a;
ELSE
          b;
ENDIF
```

A more detailed language description is beyond the scope of this paper. A sample program is shown in figure 2.

Choosing to implement Porta-SIMD primarily as C++ classes has both welcome and unwelcome consequences. The primary benefit is avoiding the need to write a compiler. The amount of work this saves cannot be overemphasized. Another benefit is that the Porta-SIMD prototype is itself very easy to port: C++ is widely available, and the prototype has been written in a coding style which carefully separates machine-independent from machine-dependent code. The primary disadvantage is that the evaluation of parallel expressions proceeds operator by operator, without any overview of the expression. This is because the code implementing each parallel operator has no way to know anything about its place in the expression. The result is that extraneous temporary values and redundant copies are sometimes necessary, reducing execution efficiency. Although this would probably be unacceptable in a production-quality language implementation, it is acceptably small for the current purposes. It is certainly possible to write an optimizing compiler for Porta-SIMD, but this is well beyond the scope of the current research.

Initial development was done on Pixel-Planes 4, a 256K PE machine in regular use at UNC. The base model (1D, 0, 0, 0, 0, 0, 0) was ported to a 16K PE CM-2 in five days, including the time required to learn Paris. This was done in the ACRF (Advanced Computing Research Facility) at Argonne National Labs. The Pixel-Planes 4 model (2D, 0, 0, 0, 0, 0, 0) is now running on both Pixel-Planes 4 and the CM. Integers of all sizes are supported. However, floating point types have been deferred while effort focuses on the central architectural and language design issues. Other models are in various stages of development. A port to the Pixel-Planes 5 simulator is planned for the near future. No performance tuning or detailed measurements have been attempted, but this early prototype obviously provides lots of room for improvement. A few brave early users are already providing valuable and encouraging feedback.

## CONCLUSIONS

The extraordinary architectural diversity of SIMD computers is too important to algorithm selection to completely hide from programmers. Optimal portability is a new concept for managing this architectural diversity. It provides specific criteria for identifying the architectural features a programmer needs to see. It allows the programmer to precisely specify the portability of each program. This lets the programmer judge the proper tradeoff between acheiving broad portability and taking full advantage of a particular architecture. Existing languages usurp this decision with predetermined architectural assumptions.

Porta-SIMD is being implemented to demonstrate the power and feasibility of optimally portable languages. It takes advantage of C++ classes and operator overloading to reduce the implementation effort. Although only a few programming models have been implemented so far, Porta-SIMD is already running on Pixel-Planes 4 and a CM-2. This is probably the first language to be implemented identically on more than one SIMD computer.

Although optimal portability has been applied here to SIMD architectures, it is potentially valuable for any diverse but related class of architectures.

## ACKNOWLEDGEMENTS

Profs. Frederick Brooks and Henry Fuchs have given me valuable advice, support, and encouragement. Greg Turk has been a valuable sounding board for ideas and problems. He and Brice Tebbs were willing to use Porta-SIMD very early, and have given me valuable feedback. It is a privilege to be part of the Pixel-Planes team. I appreciate the people and facilities of the ACRF, especially Rick Stevens, who helped me get started on their CM.

## REFERENCES

1. *Connection Machine Parallel Instruction Set (Paris): The C Interface (Version 4.0).* Thinking Machines Corporation, Cambridge, MA, 1987.

2. *DAP 500 Introduction to FORTRAN-PLUS Programming.* Active Memory Technology Limited, Reading, UK, 1987.

3. Eugene Albert, Kathleen Knobe, Joan D. Lukas, and Guy L. Steele, Jr. Compiling Fortran 8x array features for the Connection Machine computer system. *SIGPLAN Notices*, 23(9):42–56, September 1988. (Proceedings of the ACM/SIGPLAN PPEALS 1988).

4. John Beetem, Monty Denneau, and Don Weingarten. GF11. *Journal of Statistical Physics*, 43(5/6), June 1986.

5. John Beetem, Monty Denneau, and Don Weingarten. The GF11 supercomputer. In *IEEE Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 108–115, June 1985.

6. Donald W. Blevins, Edward W. Davis, and John H. Reif. *Processing Element and Custom Chip Architecture for the BLITZEN Massively Parallel Processor.* September 1987.

7. Donald W. Blevins and R. A. Heaton. The BLITZEN PE array chip feature set. In *Second Symposium on the Frontiers of Massively Parallel Computation*, October 1988.

8. K. Mani Chandy and Jayadev Misra. Architecture independent programming. In *Third International Conference on Supercomputing, Vol. 3*, pages 345–351, International Supercomputing Institute, Inc., 1988.

9. D Chin, J Passe, F Bernard, H Taylor, and S. Knight. The Princeton Engine: a real-time video system simulator. *ICCE*, May 1988.

10. Thinking Machines Corporation. *Connection Machine Model CM-2 Technical Summary.* Technical Report HA87-4, Thinking Machines Corporation, April 1987.

11. Edward W. Davis and John H. Reif. Architecture and operation of the BLITZEN processing element. In *Third International Conference on Supercomputing, Vol. 3*, pages 128–137, International Supercomputing Institute, Inc., 1988.

12. John Eyles, John Austin, Henry Fuchs, Trey Greer, and John Poulton. Pixel-Planes 4: a summary. In *Proceedings of Eurographics '87 Second Workshop on Graphics Hardware*, 1987.

13. T. J. Fountain. A survey of bit-serial array processor circuits. In M. J. B Duff, editor, *Computing Structures for Image Processing*, chapter 1, Academic Press, Inc., Orlando, FL, 1983.

14. Henry Fuchs, Jack Goldfeather, Jeff P. Hultquist, Susan Spach, John D. Austin, Frederick P. Brooks, Jr., John G. Eyles, and John Poulton. Fast spheres, shadows, textures, transparencies, and image enhancements in Pixel-Planes. *Computer Graphics*, 19(3):111–120, July 1985. (Proceedings of SIGGRAPH '85).

15. Henry Fuchs and John Poulton. Pixel-Planes: a VLSI-oriented design for a raster graphics engine. *VLSI Design*, 2(3):20–28, 1981.

16. F. A. Gerritsen. A comparison of the CLIP4, DAP, and MPP processor-array implementations. In M. J. B Duff, editor, *Computing Structures for Image Processing*, chapter 2, Academic Press, Inc., Orlando, FL, 1983.

17. Jack Goldfeather, Jeff P. Hultquist, and Henry Fuchs. Fast constructive solid geometry display in the Pixel-Powers graphics system. *Computer Graphics*, 20(4):107–116, July 1986. (Proceedings of SIGGRAPH '86).

18. W. Daniel Hillis. *The Connection Machine. MIT Press Series in Artificial Intelligence*, The MIT Press, Cambridge, MA, 1985.

19. R. Michael Hord. *The Illiac IV: The First Supercomputer.* Computer Science Press, Inc., Rockville, MD, 1982.

20. Kai Hwang and Fayé A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill Book Company, New York, 1984.

21. Leah H. Jamieson. Features of parallel algorithms. In *Second International Conference on Supercomputing, Vol. 1*, pages 476–478, International Supercomputing Institute, Inc., 1987.

22. Alan H. Karp. Programming for parallelism. *Computer*, 43–56, May 1987.

23. A. Krikelis and R. M. Lea. Low-level vision tasks using parallel string architectures. In *Parallel Processing for Computer Vision and Display*, January 1988.

24. *A FORTRAN Compiler for the Massively Parallel Processor.* Massachusetts Computer Associates, Inc., February 1984. CADD-8402-2101.

25. P. M. Nickolls and T. W. Cole. A fault-tolerant 2-d processor array for image analysis. In *Parallel Processing for Computer Vision and Display*, January 1988.

26. J. V. Oldfield, R. D. Williams, N. E. Wiseman, and M. R. Brûlé. Content-addressable memories for quadtree-based images. In *Proceedings of Eurographics '88 Third Workshop on Graphics Hardware*, 1988.

27. D. Parkinson, D. J. Hunt, and K. S. MacQueen. The AMT DAP 500. In *Spring COMPCON 88: digest of papers*, pages 196–199, The Computer Society of the IEEE, IEEE Computer Society Press, February 1988.

28. R. H. Perrott. A language for array and vector processors. *ACM Transactions on Programming Languages and Systems*, 1(2):177–195, October 1975.

29. J. L. Potter, editor. *The Massively Parallel Processor. MIT Press Series in Scientific Computation*, The MIT Press, Cambridge, MA, 1985.

30. Franco P. Preparata and Jean Vuillemin. The Cube-Connected Cycles: a versatile network for parallel computation. *Communications of the ACM*, 24(5):300–309, May 1981.

31. John R. Rose and Guy L. Steele Jr. *C*: An Extended C Language for Data Parallel Programming.* Technical Report PL87-5, Thinking Machines Corporation, April 1987.

32. Charles L. Seitz. Concurrent VLSI architectures. *IEEE Transactions on Computers*, C-33(12):1247–1265, December 1984.

33. Guy L. Steele Jr. and W. Daniel Hillis. *Connection Machine Lisp: Fine-Grained Parallel Symbolic Processing.* Technical Report 86.16, Thinking Machines Corporation, May 1986.

34. Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Publishing Company, Reading, MA, 1986.

35. Accredited Standards Committee X3 - Information Processing Systems Technical Committee X3J3 - Fortran. *X3.9-198x: Draft Proposed Revised American National Standard Programming Language Fortran (Version 104).* American National Standards Institute, April 1987.

36. Sherry J. Tomboulian, Mary Mace, and Robert A. Wagner. Language report and description for BVL-0. April 1985. Unpublished paper.

37. Russell R. Tuck, III. *Issues in the Design of an Optimizing Code Generator for BVL-0.* Master's thesis, Duke University, Durham, NC, 1987.

38. Robert A. Wagner. The Boolean Vector Machine (BVM). In *IEEE 1983 Conference Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 59–66, 1983.

# A GENERIC FINE-GRAINED PARALLEL C

L. Hamet J. Dorband

NASA/Goddard Space Flight Center/635
Greenbelt, MD 20771

## ABSTRACT

With the present availability of parallel processors of vastly different architectures, there is a need for a common language interface to multiple types of machines. Our parallel C compiler, currently under development, is intended to be such a language. This language is based on the belief that an algorithm designed around fine-grained parallelism can be mapped relatively easily to different parallel architectures, since a large percentage of the parallelism has been identified. The compiler generates a FORTH-like machine-independent intermediate code. A machine-dependent translator will reside on each machine to generate the appropriate executable code, taking advantage of the particular architectures. The goal for this project is to allow a user to run the same program on such machines as the Massively Parallel Processor, the CRAY, the Connection Machine, and the CYBER 205, as well as serial machines such as VAXes, Macintoshes and Sun workstations.

Keywords: Fine-Grained Parallelism, Portability, Operator Overloading, Massively Parallel, SIMD, MPP, C, Data Parallel.

## INTRODUCTION

As the variety and availability of parallel machines increases, the need for a portable parallel compiler becomes critical. To be effective, however, this compiler must be able to take full advantage of each machine's unique architecture. Two concepts are necessary to achieve these goals: modularity or layering and fine-grained parallelism.

A modular compiler design allows machine-dependent characteristics to be separated from the machine-independent (generic) characteristics. The object code generator (or P-code translator) is the lone machine-dependent piece of the compiler; consequently, parsing may be done once, with the output submitted to various versions of the machine-dependent layer, one version for each unique machine.

Fine-grained parallelism assigns the task of extracting parallelism within code to the programmer. The user-defined parallelism may be mapped to any architecture, since it may be easily assembled into a serial implementation or a parallel implementation of any desired degree. A parallel description of a program is much easier and straightforward to assemble into a serial description than a serial description into a parallel one.

It may seem as though it is an unreasonable task to expect a skilled programmer, let alone an unskilled one, to extract parallelism from an algorithm when it is recognized that this is such a difficult task for a compiler. The members of the MPP Working Group have shown that both skilled and unskilled programmers alike may easily extract fine-grained parallelism. In reality, extracting fine-grained parallelism is no more difficult than recognizing what code must be repeated within a loop in serial code.

## PROGRAMMING MODEL

The C language implementation supported here is based on a model of computation where there is one serial processor (the control unit) and many independent SIMD parallel processors (ALUs controlled by the control unit). Serial data is stored in the control unit memory (S) and parallel data elements are stored in the memory (P) of the parallel SIMD processors. The same operation is performed simultaneously on parallel data elements. The only exception to this occurs if a processor is masked out of the operation. Parallel control structures using this mask capability provide a means of restricting operations performed within a parallel processor to only those operations that apply to the data in that processor.

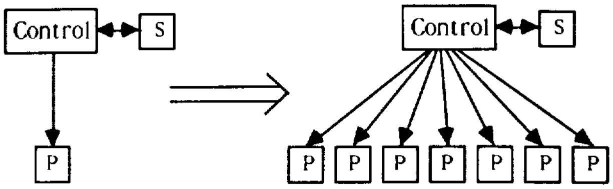


Figure 1. Logical View of SIMD Processing Model

Programming such a model (Figure 1) can be viewed as programming a single controller that has two memories, P and S. If data from S is used with data from S, the result remains in S. If data from P is used with data in P, the result remains in P. However, if data from S is used with data in P, the result must remain in P. Totally independent operations need not be concerned with the fact that there are many P memories. This model is an oversimplification, since it does not account for inter-processor interactions.

Most operations performed by a SIMD processor do not involve data from different processors, but the need for inter-processor interaction does arise. Inter-processor communication and data reduction operations facilitate the ability for parallel data elements to interact . Data reduction operations produce a single result from data in many parallel processor memories (P) and store the result in S.

Consider more closely Figure 1, containing two views of SIMD processing: a simplified model and a complete model. The simplified model consists of a control unit and two memories, S and P. The data in S is considered serial data and data in P is considered parallel data. The complete model differs in that it has multiple P memories. If the complete model is run with all but one processor masked out, it will give the same result as the simplified model (with the exception of inter-processor communications). The results from the simplified model should be the same as a serial processor, where S and P make up the memory of the serial processor.

In the simplified model, the control structure should act the same whether the condition is based on results in P or results in S. The complete model has multiple P memories; the data in each should be manipulated only by those instructions that are pertinent to it. This means that some processors must be turned off based on conditions computed in them. These conditions result in a determination that the corresponding conditionally executable code is not pertinent. Actually, only code that effects the user detectable state is masked. This includes assignment statements and conditional expressions (?:).

The preceding has several subtle implications. 1) Code within control structures, where the conditional result is in P, must be executed as long as the condition is true for at least one processor. 2) More subtly, however, if the condition is not true for any processor, then the code must not be executed. The subtlety is that although no processors' memory (P) will be modified by the code within the parallel control structure, data in S might be modified by executing this code. However, this violates the above constraints of the simplified model, and consequently must be prevented from occurring.

## LANGUAGE DESCRIPTION

The popularity and flexibility of C made it the natural choice as the language to be implemented in such a manner. Parallelism is achieved through operator and control structure overloading (to be further explained). This preserves the Kernighan and Ritchie[1] look of C, yet allows a wide range of levels of parallelism to be implemented, depending on the targeted machine. The parallelization of C is based on experience learned in the development of MPP Parallel Forth[2]. The only syntactical addition to the language is the storage class PARALLEL.

The language has been altered slightly to accommodate the parallelism. Due to the different architectures of the machines using this compiler, the storage class REGISTER has been eliminated. On the other hand, for the sake of bit serial processors, the ability to specify number of bits in a declaration has been expanded to all variables, not only to fields within a structure or union. Depending on the machine architecture, however, the programmer may get more precision than requested, but never less. All cases of precision increases will be consistent and documented for each version of the compiler.

## COMPILER DESIGN

The C compiler is divided into four components: the scanner/ parser, intermediate code generator, intermediate code translator, and virtual machine. The scanner/parser and intermediate code generator are machine-independent; the translator and virtual machine must be rewritten for each machine type.

The scanner/parser is an SLR(1) parser, written without the use of the UNIX™ utilities yacc™ and lex™, due to Macintosh memory partitioning limitations. (The Macintosh II is the first machine for which a version of the compiler is being written.)

The intermediate code generator generates postfix P-code. This style was chosen because of its speed and minimal size. Furthermore, because it is English-based, it is not difficult to read. Since the intermediate code generator is really only a postfix converter, this module remains machine-independent.

The translator converts the P-code to a FORTH-like "assembly". FORTH, a stack-oriented language, was chosen because of its speed and register simplification. Furthermore, based on prior experience with the Massively Parallel Processor, FORTH has been demonstrated to be a logical and efficient language to run as a virtual machine for SIMD architectures; each processing element memory is treated as a stack.

The virtual machine is a simple FORTH engine, actually coded in C, which executes the "object module" output from the translator.

Although we have implemented both the translator and virtual machine as machine-dependent modules, the translator could be generalized so that it would be machine-independent, requiring only recompilation with a modified include file. However, we opted against this, avoiding as much unnecessary overhead (speed loss) as possible.

## OVERLOADED OPERATORS

There are no operators added to the parallel C; all existing operators are overloaded. The version of each operator routine called by the translator is determined by the types of the operands. Unary operators are trivial; there is a version for each parallel and serial type. For binary operations, if both operands are serial or both parallel, the result corresponds. Standard C conversion rules still apply to both serial and parallel (always to the greater precision, signed to unsigned, and integer to floating point.) Added to these rules, however, is serial to parallel conversion when these two types of operands are operated on together. A serial to parallel conversion is equivalent to a broadcast of the serial value.

The bit shift operators (<< and >>) take on interesting results when done in parallel. For these operators, where the left operand is the value and the right operand is the number of bits by which to shift, the unusual case occurs when the number of bits by which to shift is parallel. If the operand is a variable, different processors may contain different values. The operation is implemented with a parallel mask, where, after each bit shift, the processors which have completed the required number of shifts are masked out, until all have completed.

Parallel logical operators (&& and ||) are implemented with parallel versions of the if-else structure (See Overloaded Control Structures). Parallel addressing operators (* and &) are undefined, as parallel pointers are not implemented in the current implementation of the compiler.

### Parallel Pointers

Although parallel pointers are not implemented in this version of the compiler, serial pointers to parallel variables are legal. They must be declared in two parts. The parallel data object must be declared as a type, then the pointer variable is declared as a pointer to that type, in a separate declaration. To illustrate,

```
typedef parallel int A;
A *ptr;
```

is legal, whereas the declaration

```
parallel int *ptr;
```

would be recognized as a parallel pointer declaration, and flagged as an error.

### Parallel Assignment Operators

Assignment operators (=, +=, \=, &=, etc.) do not observe the standard conversion rules, because the resultant type must be the type of the left operand — the one receiving the final value. Serial—serial and parallel—parallel left-right operand pairs are trivial; no serial—parallel conversion is necessary. Parallel—serial requires a standard serial to parallel conversion. Serial—parallel, however, yields interesting results.

Up to this point, no operations involve the data in different parallel processors. When parallel data is assigned to a serial variable, a data reduction operation must be performed. This involves data in all the parallel processors. A simple assignment (=) of a parallel to a serial is implemented as a bitwise cumulative OR over all values of the parallel operand, with the serial operand being set to the resulting value.

Each complex assignment operator is treated uniquely. The addition-assignment (+=) is implemented with a cumulative sum added to the serial operand; the subtraction-assignment (-=) subtracts the cumulative sum from the serial operand. Multiplication-assignment (*=) and division-assignment (/=) are treated comparably, with a cumulative product.

Bitwise AND-,OR-,and XOR-assignments (&=, |=, and ^=) are implemented as expected: a cumulative AND/OR/XOR is done over all values of the parallel operand, with the serial operand being set to the resulting value.

Modulus-assignment (%=) and shift-assignment (>>=, <<=) are undefined for the serial—parallel case.

## OVERLOADED CONTROL STRUCTURES

All control structures in the compiler apply to both serial and parallel conditions. Each structure is executed in parallel if the test expression evaluates to parallel.

For SIMD machines, all structures must use a parallel mask, to mask out processors which have failed the test condition. A bit in the mask is set or cleared based on the value of the test expression in the corresponding processor.

Parallel control structures consist of the same structures as serial control structures: if-else,while, for, and switch. For code to be executed in a parallel control structure, at least one parallel processor must require it.

In a parallel 'for' loop, either the initialization or incrementation expressions (or both) may be serial, as long as the test expression is parallel.

An example of a parallel 'while' loop is the C code:

```
parallel int a,b;
long c = 0;
        .
        .
        .
while (a >5) {
  b *= a;
  a—;
  c++;
}
```

resulting in the statements inside the loop being executed for each processor where that element of the array a is greater than five. Since c is a serial variable, it will be incremented each time the loop is executed, thus counting the maximum times the loop is executed for any processor.

## INTER-PROCESSOR COMMUNICATION

Inter-processor communication allows data from different processors to interact. This is performed by adding an integer value (n) to a pointer to a parallel value (i.e. *(para+n) ). The data (*para) in processor m+n modulo the number of processors will be moved to processor m. However,this does not in any way imply the time it takes to perform the move. It is totally architecture dependent.

## COMPILER CONSTRUCTION

Whereas the scanner/parser is standard SLR(1) and the code generator is a straightforward postfix converter, the interesting design issues pertain to the translator. The most notable points are the symbol table setup and serial and parallel memory allocation.

The symbol table is "a linked list of linked lists". All variables are chained in a list in the order encountered in the code. Each variable is, in turn, the beginning of a definition chain. Since C allows loosely formatted type definitions, the only consistent method to create a definition, for type checking purposes, is to chain the "pieces" of the type definition. A piece can be a base type (e.g., int, char, float) or a modifier (e.g., pointer, array dimension, parallel). Type checking is done by walking the chain.

Memory allocation is handled with four constants, defined by the virtual machine. These are LP, GP, PLP, and PGP — local pointer, global pointer, parallel local pointer, and parallel global pointer, containing the starting address of serial local variables, serial global variables, parallel local variables, and parallel global variables, respectively. The translator keeps track of the last space allocated in terms of offsets for each of those constants. Variables are thereafter referred to by address in the object code generated by the translator. Because the translator is FORTH-like, the virtual machine is composed of FORTH "words", functions executed when named. The virtual machine defines LP+, GP+, PLP+, and PGP+, to add these constants. Consequently, addresses appear as an offset, followed by one of those words. (Remember that FORTH uses postfix format.) The lone requirement of the virtual machine is that a block of memory be explicitly allocated before manipulated. Therefore, allocation statements may appear throughout the generated object code. Two more FORTH words are defined by the virtual machine — ALLOC and PALLOC, for serial and parallel memory allocation.

Parallel variables are actually allocated both parallel and serial memory. A serial longword (four bytes) is allocated to contain two word-length values: parallel starting address and size. Hence, parallel variables are referenced just as serial variables. The information in the serial longword is used at execution time to locate the parallel variable.

## POSSIBLE USES

This compiler will have versions on both serial and parallel machines. (On serial machines, parallel structures and operations are implemented serially.) Furthermore, a program need only be retranslated, as opposed to completely recompiled, to be run on a new machine. Consequently, a natural use of this type of compiler is to use a serial machine as a simulator for a parallel machine. This would offload much of the traffic on more costly parallel machines during parallel code debugging.

Another possible use for this type of compiler is in conjunction with the newest computational strategy: network computing. Each node of the network would have its own version of the machine-dependent portion of the compiler. An entire program would be run through the machine-independent section of the compiler, then each piece of the program would be translated by the machine-dependent portion corresponding to the machine on which that section is to be run.

## IMPLEMENTATIONS AND FUTURE PLANS

The first complete version of this compiler is currently being implemented in Macintosh Programmer's Workshop (MPW) C on an Apple Macintosh II workstation, to execute serially. Since the code itself is written in C, using only the simplest library routines (to ensure portability), it will be trivial to port the same code to other serial workstations. The next target is the Sun.

For parallel machines, a new version of the translator and virtual machine must be written. The first type of parallel architecture for which a version will be written is an array processor, such as the Massively Parallel Processor or other commercially available SIMD processor. After this implementation there are plans for a vector processor version, such as for a CRAY.

## SUMMARY

By isolating the machine-dependent and machine-independent pieces of a compiler, we have created a compiler which reflects a high degree of portability: the same code may be run on very different machines (architecturally) with only partial recompilation. Because the programmer extracts the parallelism, the degree to which this parallelism is utilized becomes a completely machine-dependent issue. Consequently, each architecture is utilized to its fullest, without any code modification.

## REFERENCES

1. Kernighan, B. W., and Ritchie, M. D., *The C Programming Language, Prentice-Hall, Inc., Englewood Cliffs, NJ.*

2. Dorband, J. E., *MPP Parallel Forth,* Frontiers of Massively Parallel Scientific Computation, NASA CP-2478, September 1986, pp. 275-283.

# MPP IMPLEMENTATION
# OF ABSTRACT DATA PARALLEL ARCHITECTURES
# FOR DECLARATIVE PROGRAMMING LANGUAGES

John T. O'Donnell*
Department of Computing Science
University of Glasgow
Glasgow G12 8QQ, Scotland

## ABSTRACT

Declarative programming languages require complex data structure operations that are often inefficient on sequential machines and MIMD multiprocessors. It is sometimes possible to implement declarative data structures efficiently on a data parallel architecture. This method is illustrated by the implementation of a new data parallel algorithm for functional arrays on the Massively Parallel Processor (MPP). Functional arrays cannot be implemented efficiently on conventional architectures without severely restricting the way a program may access the array. Conventional unrestricted algorithms typically require $O(\log n)$ time to update an array and also to access an element. The MPP algorithm performs these operations in a constant time of about 100 microseconds.

Keywords: declarative language, functional language, data parallel architecture, functional array, massively parallel processor.

## INTRODUCTION

Declarative programming languages (including pure functional and pure logic languages) have several semantic properties that seem to make them well suited for parallel evaluation. Considerable progress has been made in detecting potential parallelism in declarative programs and exploiting that parallelism on multiprocessors. This work is surveyed in Ref. 6.

Declarative languages have another form of potential parallelism that is not well suited for MIMD multiprocessors. This is the parallel execution of the numerous primitive operations that are needed in order to perform complex data structure operations.

One reason that declarative languages are usually slower than conventional imperative languages is their heavy use of complex data structure operations. Examples include operations on abstract data types, combinator graphs and functional arrays. In sequential von Neumann machines, a large amount of execution time is spent exchanging data between the central processor and the memory in order to execute some of these operations.

It is often possible to avoid the penalty associated with declarative data structure operations by exploiting their inherent parallelism. A natural way to do this is to define an ***abstract data parallel architecture*** which supports the necessary data structure operations. It is then possible to implement the abstract architecture directly in hardware or to emulate it using an existing SIMD machine.

This paper illustrates the approach by presenting a fast implementation of functional arrays, a typical and very important declarative data structure. Functional arrays are difficult to implement efficiently on sequential and MIMD architectures without restricting the way they are used. The next two sections define functional arrays and show why the basic operations on them are slow when implemented on a sequential machine. The following section then describes an abstract data parallel architecture that implements the functional array operations efficiently. This abstract architecture has been emulated on the NASA Massively Parallel Processor (MPP), and the remaining sections discuss the emulation and assess the performance results.

There are several interesting relationships among functional arrays, parallelism and nondeterminism. Hudak discusses these in Ref. 3. For more discussion of architectures for functional programming, see Refs. 9 and 10. Steele and Hillis describe a complete programming language that exploits data structure parallelism in Ref. 8.

## FUNCTIONAL ARRAYS

Declarative programming languages do not allow side effects. Therefore it is impossible to modify existing data structures; instead of changing an old object the programmer must construct a new one which is similar to the old object except for some specified differences. This paradigm

* Present address: Computer Science Department, Indiana University, Bloomington IN 47401 USA.

CH2649-2/89/0000/0629$01.00 © 1988 IEEE

is used throughout declarative programming, and arrays for functional languages provide a typical example.

Imperative languages provide two primitives for operating on arrays: indexing and assignment. The indexing operation takes an array and index value and returns the array element with that index. The assignment operation takes an array, index and new value, and replaces the old value of the indexed element with the new value. The assignment is disallowed in a functional language because the old aggregate value of the array (the set of all index–value pairs) no longer exists.

Functional arrays replace the assignment operation with an *update* operation which constructs a new array without changing the old one. The indexing operation remains the same. Any algorithm using arrays can be expressed with indexing and update just as easily as with indexing and assignment. However, functional arrays are more general and some algorithms are easier to implement using them. For example, the lexical environment for a programming language interpreter can be represented as a functional array using variable names as indices.

The *incremental update* takes $n$ operations to replace all the values of an array containing $n$ elements. In contrast, a similar operation called the *monolithic update* can change all the array elements simultaneously (Ref. 11). In this paper we consider only the incremental update operation.

We define an array $A$ to be a function which takes an integer index and returns the value of the indexed element. The element may have an arbitrary type $\alpha$, although all the elements of the array must have the same type. The type of $A$ is thus

$$A : \mathit{int} \rightarrow \alpha$$

We may fetch the $i$th element of $A$ by applying $A$ to $i$. This paper uses the standard notation for function application. For example, $f\ x$ is the result of applying $f$ to $x$, while $f\ x\ y\ z$ is the result of applying $f$ to three (curried) arguments $x$, $y$ and $z$. Therefore $A\ i$ applies the array $A$ to the index $i$, looking up the $i$th element. This is equivalent to the expression $A[i]$ in conventional programming languages.

In general an array $A$ will not contain a value for every index. If $A$ is undefined at index $i$ then $A\ i$ is $\perp$ (the bottom element in the value domain, representing an undefined value). There is a constant empty array called **nil** which is unbound for all indices. Thus for any index $i$, $\mathbf{nil}\ i = \perp$. The **nil** array serves as the basis from which all other arrays must be constructed.

The *update* function takes three arguments: an old array $A$ whose elements are of type $\alpha$, an index $i$ and a value $x$ of type $\alpha$. It returns a new array $A'$ whose elements are also of type $\alpha$. The type of *update* is

$$\mathit{update}\ :\ (\mathit{int} \rightarrow \alpha) \rightarrow \mathit{int} \rightarrow \alpha \rightarrow (\mathit{int} \rightarrow \alpha)$$

The new array is identical to the old one except that the element with index $i$ has the value $x$. The *update* function is defined by two equations. For an arbitrary array A,

$$(\mathit{update}\ A\ i\ x)\ i = x$$
$$(\mathit{update}\ A\ i\ x)\ j = A\ j, \quad \text{for } j \neq i$$

In other words, since an application of *update* produces a new array, we define *update* by specifying what happens when that new array is applied to an index. If the new array is applied to $i$ then the result must be $x$; otherwise the result is the same value that the original array would give.

The *lookup* function takes an array $A$ whose elements have type $\alpha$ and an index $i$, and it returns the value of the $i$th element. The type of *lookup* is

$$\mathit{lookup}\ :\ (\mathit{int} \rightarrow \alpha) \rightarrow \mathit{int} \rightarrow \alpha$$

Since *lookup* simply applies the array (which is itself a function) to the index, there is no need for an explicit *lookup* function in the programming language. However, we will be considering the implementation of the language rather than programs written in it, so it is useful to have an explicitly named *lookup* subroutine. Its value is defined by

$$\mathit{lookup}\ A\ i\ \ =\ \ A\ i$$

## SEQUENTIAL IMPLEMENTATION

It is useful to consider the difference between functional arrays and conventional arrays, and then examine why it is difficult to implement functional arrays efficiently on sequential machines.

An imperative language treats an array as a block of memory, and the values stored in the memory words may change over time. Consider the following program which prints -25. By the time the **print** statement is executed the original value of 1002 that was stored in $A[2]$ is gone forever.

```
A[0] := 1000
A[1] := 1001
A[2] := 1002
    ⋮
A[100] := 1100
A[2] := -25
print A[2]
```

When this algorithm is written in a functional style, the program constructs a sequence of distinct arrays $A_i$ for $i \leq 0 \leq 100$, and each of these values is immutable. (Both the imperative and functional programs would normally be written using loops rather than a long sequence of statements; the functional equivalent of a **for** loop is tail recursion.)

$$
\begin{aligned}
A_0 &= \textit{update}\ \mathbf{nil}\ 0\ 1000\\
A_1 &= \textit{update}\ A_0\ 1\ 1001\\
A_2 &= \textit{update}\ A_1\ 2\ 1002\\
&\ \vdots\\
A_{100} &= \textit{update}\ A_{99}\ 100\ \ 1100\\
A_{101} &= \textit{update}\ A_{100}\ 2\ \text{-}25
\end{aligned}
$$

At this point, *lookup* $A_{101}$ 2 evaluates to -25, but $A_{100}$ still exists (as do all the other arrays) and *lookup* $A_{100}$ 2 still evaluates to 1002.

It is possible to use imperative arrays in an almost-functional language using I-structures (Ref. 1). Several dataflow languages do that. However, we will be considering pure functional arrays.

The most obvious way to implement *lookup* $A\ i\ x$ is to make a new copy of $A$, storing $x$ into the $i$th element instead of the value of $A\ i$. This allows *lookup* always to execute as fast as fetching a value from an imperative array. The problem is that *update* would take both time and space proportional to the size of the array. Since an array must be constructed one element at a time by a sequence of updates, the cost would certainly be prohibitive.

Another approach, which is equally obvious and almost as bad, is to build a linked list of "exceptions" to an array. Thus *update* $A\ i\ x$ would allocate a new word that contains $i$ and $x$ and points to the representation of $A$. This makes *update* take a small constant amount of time and space, as it should, but the average *lookup* time is now proportional to the array size.

Various representations are possible which build balanced tree structures to represent a functional array. These representations typically cost $O(\log n)$ time for *lookup* and *update*. The space complexity of *update* is also $O(\log n)$. Although far better than either of the first two algorithms, this still means that functional array operations are very expensive.

A better alternative is to analyze the functional program. If it is possible to prove that after evaluating *update* $A\ i\ x$ the original array $A$ will never be needed again, then the compiler can implement the update with a destructive assignment. Sophisticated automatic compile time analysis algorithms have been developed to do that (Ref. 4). This approach works very well in many cases. In particular, when an imperative program is rewritten in a functional language without a complete reworking of the algorithm, the functional arrays will be used as if they were imperative arrays, and only one version of each array (the result of the most recent update to it) needs to be retained. Analysis algorithms can usually tell the compiler to use imperative arrays in such cases.

Although compile time analysis is extremely useful, it does have two drawbacks. First, the analyzer may not understand the program well enough to determine that imperative array operations would be safe. The second problem is more fundamental: some programs use functional arrays in a truly functional style where imperative array operations would be incorrect. No compile time analysis tool will be able to help in that situation. One way out would be to teach functional programmers not to exploit the full power of functional arrays unless they are willing to tolerate extremely slow execution. But this is surely inferior to the best approach: using data structure parallelism to implement all functional array operations in a small constant amount of time, with no restriction whatever on the way the arrays are used. The next section shows how to do that.

## ABSTRACT ARCHITECTURE

This section describes an abstract data parallel architecture for functional arrays. This architecture implements both *lookup* and *update* in a small constant amount of time, independent of the size of the array. An *update* always requires a constant amount of space.

It is essential for an efficient implementation of functional arrays to share as many array elements as possible. Therefore if the array $A$ contains $n$ elements, including $A\ i$, then $A' =$ *update* $A\ i\ x$ must allocate only one new word of data in order to hold the value of $A'\ i$. The other $n-1$ elements of $A$ and $A'$ are shared. Without sharing, too much time and memory would be spent copying data needlessly. However, the sharing must be done in a manner that does not slow down the *lookup* operation. In particular, the *lookup* algorithm should not traverse a list of updated array elements.

The abstract architecture shares common array elements without slowing down access to any individual element. The basic idea is that every array element is stored with some additional information that tells it exactly which arrays contain it. The *lookup* operation then associatively asks every array element to decide on its own whether it belongs to the particular array being searched. These decisions are all made in parallel. Then *lookup* can quickly find the indexed element using another associative search. During an *update* many words in the array must change their representation, but all of those changes can be made in parallel.

It is important to distinguish between the value of an array element $x = A\ i$ and the memory word $w$ that represents it. We will call $x$ an ***array element*** and $w$ will be called an ***array word*** with index $i$. The algorithms must share array words when possible, but it is meaningless to talk about sharing array elements. The ***containment set*** of an array element $x$ is the set of arrays that contain the word representing that element, and it is written $cs\ x$. For example, $cs\,(A\ i)$ is the set of arrays whose representation contains the word that stores the value of $A\ i$.

Suppose there is an array $A$ and the program then modifies the array data structures by creating a new array $A' =$

*update* $A\ i\ x$. It is useful to consider the data structure representation both before and after the *update* is executed.

A new word is needed to store the value of $A'\ i$. (We ignore an optimization which avoids creating a new word if it happens to be the case that $A'\ i\ =\ A\ i$.) The new word represents a completely new array element, and it cannot be shared with any other array, so

$$cs\ (A'\ i) = \{A'\}.$$

However, every other word of $A'$ is shared with the corresponding word of $A$. For any $j$ in the domain of the arrays, where $j \neq i$, let $C_j$ be the value of $cs(A\,j)$ before the *update* is executed. Then after the *update* is executed,

$$cs\ (A'\ j) = cs\ (A\ j) = \{A'\} \cup C_j.$$

Thus the word that represents $A\ j$ also represents $A'\ j$, and its containment set includes $A$, $A'$, and all other arrays that had already been in the containment set of $A$.

The key problem in the data parallel algorithm is finding a good way to represent the containment set of every word. This representation must be able to specify any set of arrays that might contain the word using a constant amount of space. Furthermore, an *update* operation may change the containment sets of many words, and all those words must be able to recalculate their containment set in parallel using a constant amount of time.

An *array reference* is an integer value that represents a pointer to an array. In effect, the arrays are all numbered, and references to arrays always use those numbers. The array reference of **nil** is -1, and *update* generates an array reference for the new array that it creates. If $a$ is the array reference of an array $A$, then we define a function *aref* by

$$A\ =\ aref\,a.$$

The algorithm represents each containment set with a containment interval. A *containment interval* is a pair of integers $L$ and $U$, where $L$ gives the lower bound of the interval and $U$ gives the upper bound of the interval. A containment interval represents the set of all arrays whose array reference number lies between $L$ and $U$. If $w$ is a word with bounds $L$ and $U$ then

$$cs\ w = \{aref\,L,\ \ aref\,L+1,\ \ \ldots,\ aref\,U\}$$

An *update* operation generally must recalculate the array reference numbers of some of the arrays, and many of the containment intervals must also be changed. An array does not have a constant array reference number. A pointer to an array must be represented by an integer variable which holds the current value of the array reference number, but that value may change over time as various *update* operations are performed. The *update* algorithm below shows how these values are recomputed, and the *lookup* algorithm shows how they are used. The following section gives a detailed example illustrating containment sets, containment intervals and array reference numbers.

A *processing element* is a hardware unit containing the memory, arithmetic and logic needed for a word. The architecture provides several basic operations (these are typical of SIMD architectures).

- **for each cell**

  This executes a sequence of statements in parallel in all the processing elements. The notation "*cell.field*" refers to a local field within a processing element.

- select-available

  This operation finds one processing element that is not currently in use and sets its *select* flag.

- select-minimum (*field*)

  This considers the value of *field* in every processing element whose *select* flag is currently set. It determines which of those processing elements has the smallest value of *field* and leaves it unchanged. All other processing elements clear their *select* flag.

- fetch()

  This returns the contents of the selected processing element. (If several are selected, the logical *or* of their contents is returned.)

- store (*field* = *value*, ...)

  The values are stored into the corresponding fields in every processing element whose *select* flag is set.

The data parallel *update* and *lookup* algorithms are described below. Each processing element contains the following fields:

- *type* specifies whether the word is empty or contains an array element.
- $T$ gives the element type if the word contains an array element.
- $K$ holds the index of the array element.
- $V$ holds the value of the array element.
- $L$ gives the lower bound of the containment interval.
- $U$ gives the upper bound of the containment interval.
- *select* is a flag used during associative searches to mark

the processing elements that contain an array element whose containment interval includes the index argument to the *lookup* function.

- ***temp*** is a temporary integer used by *lookup* to resolve which word to use if several words have the right index and containment interval.

The data parallel algorithm for *update* first recalculates the containment interval in every processing element that holds an array word. Then it allocates an available word of memory, and stores the new array element in it. Finally it returns a reference to the array that has just been created. The argument $a$ is the array reference number of the array that is being updated.

```
update a i x
    for each cell
        if $cell.L > a$ then $cell.L := cell.L + 1$
        if $cell.U \geq a$ then $cell.U := cell.U + 1$
        if $cell.T = aref$ and $cell.V > a$
            then $cell.V := cell.V + 1$
    select-available()
    store $(L = a + 1,\ U = a + 1,\ K = i,\ V = x)$
    return $(aref\, a + 1)$
```

It is important to note that every array reference number that is greater than $a$ must be incremented, regardless of where it is stored. If there are any of these in the control processor's memory it must update them one by one. Normally there will be only a few registers to check, because most of the array references will be in the heap which is checked in parallel.

The ***lookup*** algorithm begins by finding all the words with the specified index value that are contained in the array being accessed. Since it is possible for several words with the same index to be in the containment interval, *lookup* then finds the most recent one by selecting the matching word with the smallest value of $U - L$. Finally it fetches the array element from that word and returns the result.

```
lookup a i
    for each cell
        $cell.select := (cell.type = element)$
                        and $cell.L \leq a \leq cell.U$
                        and $cell.K = i$
        $cell.temp := (cell.U - cell.L)$
    select-minimum $(temp)$
    if select-exists
        then return (fetch())
        else return undefined
```

## EXAMPLE OF EXECUTION

It is useful to work through how the algorithm builds a set of functional arrays. Initially there is only one empty array called **nil**, whose array reference number is -1. The only way to construct any other array is to update **nil**. After executing

$$A \quad = \quad update\ \mathbf{nil}\ 3\ a$$

there is one array word with index 3 and value $a$. The containment set of this word is $\{A\}$. A reference to $A$ is represented by 0, and we write this as $A \approx 0$. Both $L$ and $U$ are 0, and the containment interval is written as $0 \times 0$.

|  |
|---|
| [3]=a |
| $\{A\}$ |
| $0 \times 0$ |

$$A \approx 0$$

Now the program executes

$$B \quad = \quad update\ A\ 4\ b$$

which creates a new word and adjusts the containment interval of the previous word. Since $A \approx 0$ the upper bound of the word with interval $0 \times 0$ is incremented, but the lower bound is not. This produces the following representation.

|  |  |
|---|---|
| [3]=a | [4]=b |
| $\{A, B\}$ | $\{B\}$ |
| $0 \times 1$ | $1 \times 1$ |

$$A \approx 0 \quad B \approx 1$$

Another array can be constructed similarly by

$$C \quad = \quad update\ B\ 5\ c$$

|  |  |  |
|---|---|---|
| [3]=a | [4]=b | [5]=c |
| $\{A, B, C\}$ | $\{B, C\}$ | $\{C\}$ |
| $0 \times 2$ | $1 \times 2$ | $2 \times 2$ |

$$A \approx 0 \quad B \approx 1 \quad C \approx 2$$

So far each array has been created by updating the previous array, leading to a simple pattern of containment intervals. The situation becomes more complex when a new array is created by updating some earlier array, rather than the last one. For example,

$$D \quad = \quad update\ B\ 4\ d$$

produces the following data structure representation:

|  |  |  |  |
|---|---|---|---|
| [3]=a | [4]=b | [5]=c | [4]=d |
| $\{A, B, C, D\}$ | $\{B, C, D\}$ | $\{C\}$ | $\{D\}$ |
| $0 \times 3$ | $1 \times 3$ | $3 \times 3$ | $2 \times 2$ |

$$A \approx 0 \quad B \approx 1 \quad C \approx 3 \quad D \approx 2$$

The array reference number of $C$ has been changed from 2 to 3, and the new array $D$ now has 2 as its reference number.

Suppose that we now evaluate ***lookup*** $D$ 4. Since $D \approx 2$, the word that holds [5]=c is not selected (its containment interval is $3 \times 3$, which does not include 2). The word that holds [3]=a is not selected because its index doesn't match.

Now there are two selected words, holding [4]=b and [4]=d. The most recently created of these words is the one holding the correct value, and *lookup* finds it by choosing the selected element with the narrowest containment interval. The word holding [4]=b has interval $1 \times 3$, so the word holding [4]=d with interval $2 \times 2$ is chosen instead. Therefore *lookup* $D$ 4 evaluates to "d" as it should. However, if we evaluate *lookup* $B$ 4 the result is "b".

## MPP IMPLEMENTATION

The Massively Parallel Processor (MPP) is a large scale SIMD machine (Ref. 7). This section describes how the data parallel functional array algorithms were implemented on the MPP, and the next section discusses their performance.

The MPP system consists of several major components. A ***host processor*** provides a file system and network communications. The ***main control unit*** is a sequential computer that can perform computations on its own as well as controlling the operation of the array unit. The *array unit* contains 16,384 processing elements. Each one has 1024 bits of memory as well as a small bit-serial arithmetic and logical unit. (The MPP also has a staging memory which was not used for this work.)

Most of the functional array system runs in the main control unit and the array unit; the host processor only performs I/O support. The functional program object code executes in the main control unit. (Currently there is not a complete functional language implementation for the main control unit; those algorithms were written by hand in assembly language.) When the user program needs to access a functional array it calls a lookup or update subroutine in the main control unit. These subroutines then communicate with the array unit, passing it the input parameters and receiving the result. The array unit executes the parallel parts of the algorithm. The portions of *lookup* and *update* marked **for each cell** are executed in the array unit, and the rest is executed in the main control unit. The main control unit and array unit work together to implement the "select-minimum" operation.

The array unit has subroutines for bit serial arithmetic. The *lookup* and *update* subroutines in the array control unit call these lowest level subroutines in sequence. Conditional operations work by disabling those processing elements where the condition is false. For example, when the array unit executes

$$\textbf{if } cell.L > a \textbf{ then } cell.L := cell.L + 1$$

it first calls an arithmetic comparison subroutine that masks off all processing elements where $cell.L \le a$. Then it calls an increment subroutine which performs $cell.L := cell.L + 1$ only in the appropriate processing elements.

## MPP PERFORMANCE RESULTS

The MPP implementation of functional arrays is written in the main controller assembly language MCL and the processing element assembly language PRL. All integer representations were 16 bits. The MPP performance monitor (a highly accurate clock which produces very repeatable results) gave the following timings for the *lookup* and *update* operations:

- *lookup* $A$ $i$ takes 105.5 microseconds for all values of $A$ and $i$.
- *update* $A$ $i$ $x$ takes 114.9 microseconds for all values of $A$, $i$ and $x$.

The corresponding times for imperative array lookup and assignment operations are typically on the order of one microsecond, so the functional array operations are still much slower. A number of points must be considered in order to evaluate these results. First we will compare the MPP functional array system with other pure functional array implementations; then we will compare it with imperative arrays.

As discussed in an earlier section, the other algorithms that support functional arrays either execute slowly or work only for some programs. The algorithm given in this paper is the fastest general implementation of pure functional arrays. The most important attribute of the data parallel algorithm is that *lookup* and *update* always take constant time and space. This doesn't just give a constant factor speedup; it reduces the time complexity and space complexity of the algorithm, compared with other general implementations of functional arrays.

There are three main reasons why the MPP implementation of functional arrays is slower than standard imperative arrays.

1. Most of the slowdown is caused by the fact that all the arithmetic and data movement operations on the MPP use bit-serial algorithms, while conventional machines perform arithmetic on a word of data in one clock cycle. Since the MPP algorithms were using 16-bit wide data, this accounts for a factor of 16 loss in performance.

2. The second most important cause of the slowdown is that each *update* and *lookup* operation require several arithmetic operations. In particular, it is necessary to compare the $L$ and $U$ fields with a broadcast value on both *lookup* and *update*; *update* must also spend time incrementing $L$ and $U$ fields in some of the cells; *update* must also increment some of the array reference numbers; *lookup* needs to find the minimum value of a set of cell fields in order to find the most recent element with the specified index. Conventional array accessing

also requires some arithmetic operations, but not as many.

3. The third and least significant cause of the slowdown is that the MPP algorithms required a number of instructions that constitute pure overhead. These include the instructions needed to communicate between the Main Control Unit and the Processing Element Control Unit, along with the subroutine calls and register initializations that take place in both control units. Sequential machines implement array accesses without any corresponding overhead.

Clearly it would be possible to remove each of these sources of overhead through special-purpose hardware. In particular, a dedicated functional array machine could be built that contained several word arithmetic units in each processing element. However, it would probably be better to build a general-purpose programmable data parallel architecture with sufficient arithmetic and logical units in each processing element. This would give almost as good performance for functional array operations, and it would also be able to execute a wide range of other data structure algorithms.

The general functional array algorithm given here could be used in a system that also tries to use destructive array updates whenever program analysis proves that to be safe. This might be the ideal solution, because it would reduce the overhead due to array operations, and yet it would still allow programmers to use arrays in a completely general manner without worrying too much about efficiency.

## COMMUNICATION

The algorithms for ***lookup*** and ***update*** given above do not require any communication among the processing elements. They rely almost entirely on arithmetic and logic operations that are executed independently, in parallel, in all the processing elements. The only communication takes place between the control processor and the array unit, and the MPP hardware supports all the necessary capabilities. The ***lookup*** algorithm uses the MPP's ability to calculate the logical *or* of a bit in every processing element using a purely combinational logic network, requiring only a few microseconds per bit.

Many declarative data structure operations are not so simple; they require communications among processing elements in addition to communications between the control processor and the array unit. A good example of this is the addition of storage management to the functional array system.

The algorithms given above allocate a word of memory for each *update* operation, but they never deallocate a word. Whenever an array becomes inaccessible, some of the individual array words may also become inaccessible. When this happens the system should (eventually) reclaim that storage so it can be reused. The easiest way to reclaim storage for the data parallel functional array algorithm is through garbage collection. Another strategy similar to reference counting is also possible, and has several advantages, but its implementation is considerably more complex.

The garbage collection algorithm for functional arrays is similar in style to the ***lookup*** and ***update*** algorithms. It spends most of its time performing parallel arithmetic within the processing element array. However, the garbage collection algorithm can require a lot of execution time because it has to traverse references from all accessible data words. It would be better to reclaim storage automatically, in constant time, as soon as it becomes inaccessible. Such algorithms require a large amount of communication among the processing elements.

A central issue in designing data parallel algorithms is matching the hardware's interconnection network to the requirements of the algorithm. Machines have been built or proposed with nearest-neighbor connections (the MPP), a combinational tree network (Apsa), and a hypercube (Connection Machine).

Many declarative data parallel algorithms, including reference counting for functional arrays, can be implemented with a combinational tree-structured interconnection network. When this kind of network is able to support an algorithm without communication bottlenecks, it has very significant advantages:

- The tree network can easily be scaled up to very large numbers of processing elements.
- Since the network is combinational, and does not require messages to pass through a number of latched nodes, its latency is extremely low. This leads to fast communication.

The Apsa system (Ref. 5) uses a combinational tree network for declarative data parallel algorithms, including functional arrays.

Many algorithms, of course, require too much communication to work well in a tree network. Sorting and Fast Fourier Transform are typical examples. More powerful interconnection networks, such as the hypercube or cube connected cycles, are necessary for these applications. The Connection Machine (Ref. 2) supports these algorithms well.

There is a tradeoff between the power of an interconnection network and its latency and scalability. For example, a hypercube does not scale up to large numbers of processing elements as well as a tree does. Furthermore, a multistage network with latches in each stage is considerably slower than a purely combinational network.

Because of these tradeoffs, we should not simply use the most flexible data parallel architecture for all algorithms. In some important cases (such as functional arrays) the performance can be improved by using a much cheaper architecture. Therefore it is very desirable to determine the kinds of network needed for a wide variety of algorithms, giving better insight into these tradeoffs for specific problems.

It would be helpful for costly machines with rich interconnection networks to support simple communication algorithms efficiently. For example, a multistage hypercube machine executes some algorithms more slowly than a combinational tree machine. Yet with a very small increase in cost, the hypercube machine could also support combinational tree operations. We need to know more about data parallel algorithms in order to make such decisions correctly.

## CONCLUSION

Declarative programming languages rely on several complex data structure operations that execute slowly on von Neumann architectures. One way to improve the performance of declarative languages is to use fine grain parallelism to implement the data structure operations directly in the memory. Although this technique does not exploit large grain parallelism in the user's program, it can improve the speed of all programs that use the declarative data structures. Furthermore, it should be possible to combine some coarser grain parallelism with the data structure parallelism.

This paper has shown how data structure parallelism can improve the performance of functional array operations. Since declarative (including functional) languages do not allow side effects, array assignment operations usually require a large amount of copying. By incorporating simple arithmetic and logic capabilities in each word of memory, all the functional array operations can be executed in a constant amount of time and space. These operations are still slower than array accesses in imperative languages running on conventional hardware, but most of this slowdown is caused by the bit-serial architecture of the data parallel machine. The performance could be improved considerably with the use of special purpose hardware. However, a general purpose data parallel architecture (such as the MPP or the Connection Machine) is very useful for prototyping the declarative data structure algorithms, and the flexibility of a general purpose machine makes it possible to exploit several other forms of parallelism in the same program.

## REFERENCES

1. Arvind, R. S. Nikhil and K. K. Pingali. "I-Structures: Data Structures for Computing", *Graph Reduction*, Lecture Notes in Computer Science 279, Springer-Verlag, 1986, pp. 336–369.

2. W. Daniel Hillis, *The Connection Machine*, Cambridge: The MIT Press, 1985.

3. P. Hudak, "Arrays, Non-Determinism, Side-Effects and Parallelism: A Functional Perspective", *Graph Reduction*, Lecture Notes in Computer Science 279, Springer-Verlag, 1986, pp. 312–327.

4. P. Hudak and A. Bloss, "The Aggregate Update Problem in Functional Programming Systems", *12th ACM Symposium on Principles of Programming Languages*, 1985, pp. 300–314.

5. J. T. O'Donnell, "Parallel VLSI Architecture Emulation and the Organization of APSA/MPP", *Frontiers of Massively Parallel Scientific Computation*, Proceedings of the first symposium (September 1986), NASA Conference Publication 2478, 1987, pp. 75–84.

6. S. L. Peyton Jones, *The Implementation of Functional Programming Languages*, Englewood Cliffs NJ: Prentice-Hall International, 1987.

7. J. L. Potter (ed.), *The Massively Parallel Processor*, Cambridge: The MIT Press, 1985.

8. G. L. Steele Jr. and W. D. Hillis, "Connection Machine LISP: Fine-Grained Parallel Symbolic Processing", *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pp. 279–297.

9. P. C. Treleaven, "Computer Architecture for Functional Programming", *Functional Programming and its Applications* (ed. J Darlington), Cambridge University Press, 1982, pp. 281–306.

10. S. R. Vegdahl, "A Survey of Proposed Architectures for the Execution of Functional Languages", *IEEE Transactions on Computers*, Vol. C-33, No. 12, December 1984.

11. P. Wadler, "A New Array Operation", *Graph Reduction*, Lecture Notes in Computer Science 279, Springer-Verlag, 1986

# SECTION IV: LANGUAGES

## Part 2: Poster Presentations

# PERFORMANCE ANALYSIS OF INTERCONNECTION NETWORKS FOR MASSIVELY PARALLEL MULTICOMPUTERS

Hassan Z. Abdalla and Scott F. Midkiff

Bradley Department of Electrical Engineering
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

## ABSTRACT

This paper presents a performance analysis model and describes a network analysis program that can be used to estimate end-to-end delay and maximum system throughput for massively parallel multicomputer networks. The model explicitly accounts for both message processing and transmission delays. The model is general enough to allow analysis of message and virtual cut-through switching, point-to-point and multiple bus topologies, unidirectional and bidirectional links, and varying traffic locality assumptions. The model and program allow the analytical evaluation of network and node design alternatives, as illustrated by an application example.

Keywords: Interconnection Networks, Multicomputer Design, Performance Analysis, Queueing Models.

## INTRODUCTION

Advances in VLSI technology have made multicomputer networks a promising approach to parallel processing. Multicomputers contain multiple processing nodes that operate concurrently and share information by passing messages over links that connect the nodes. Each node contains a functional processor (FP) that executes computational tasks and a communications processor (CP) that handles message transfers. Since the network is rarely completely connected, messages may pass through one or more intermediate nodes from source to destination. In a massively parallel multicomputer, this may lead to significant communications delay since messages may need to pass through many intermediate nodes and traffic on individual links may become quite high. Therefore, the communication delay must be carefully considered in the design of large multicomputer networks.

The space of interconnection networks can be represented by the cartesian product of four sets of design features: operation mode, control strategy, switching methodology, and network topology (Ref. 1). The operation mode of multicomputer networks, at least at the macro level, is asynchronous and does not directly affect the performance analysis. Multicomputers exhibit a distributed, rather than centralized, control strategy. Traditionally multicomputers have employed message switching, although virtual cut-through switching (Ref. 2) also holds promise. A variety of topologies have been implemented or proposed including point-to-point topologies such as the mesh and hypercube, and multiple bus topologies such as the spanning bus hypercube.

This paper describes a performance analysis model and a PC-based network analysis program that allow an accurate and efficient analysis of large multicomputer network designs. The analytic model can be used to determine end-to-end message transfer delays and the maximum system throughput.

Traditional performance analysis models for computer networks have focused on local and wide-area networks. Because these networks exhibit long transmission times due to long messages and relatively low transfer rates, the performance models ignore communication processing delay in favor of transmission delays. Recent research has focused on performance modeling and analysis of point-to-point multicomputer networks (Ref. 3). Multicomputer networks are characterized by high link transfer rates and relatively short messages. Therefore, message processing time is a significant and often dominating component of delay and cannot be ignored. The performance model proposed in Ref. 3 is limited to point-to-point networks with bidirectional physical links and message switching. The performance model described in this paper generalizes the model of Ref. 3 and can model both point-to-point and multiple bus topologies and both packet and virtual cut-through switching. The model may also be used to analyze networks with bidirectional or unidirectional links.

## PERFORMANCE ANALYSIS MODEL

### Assumptions

The performance analysis model makes the following assumptions.

1. Composite traffic arrival at each node is Poisson distributed.
2. Message lengths, and hence transmission times, are exponentially distributed.
3. Network topologies are symmetric.
4. Nodes are identical, at least up to the links and CP.
5. The message generation and destination distributions are identical at all nodes.
6. Node message queues have infinite storage capacity.

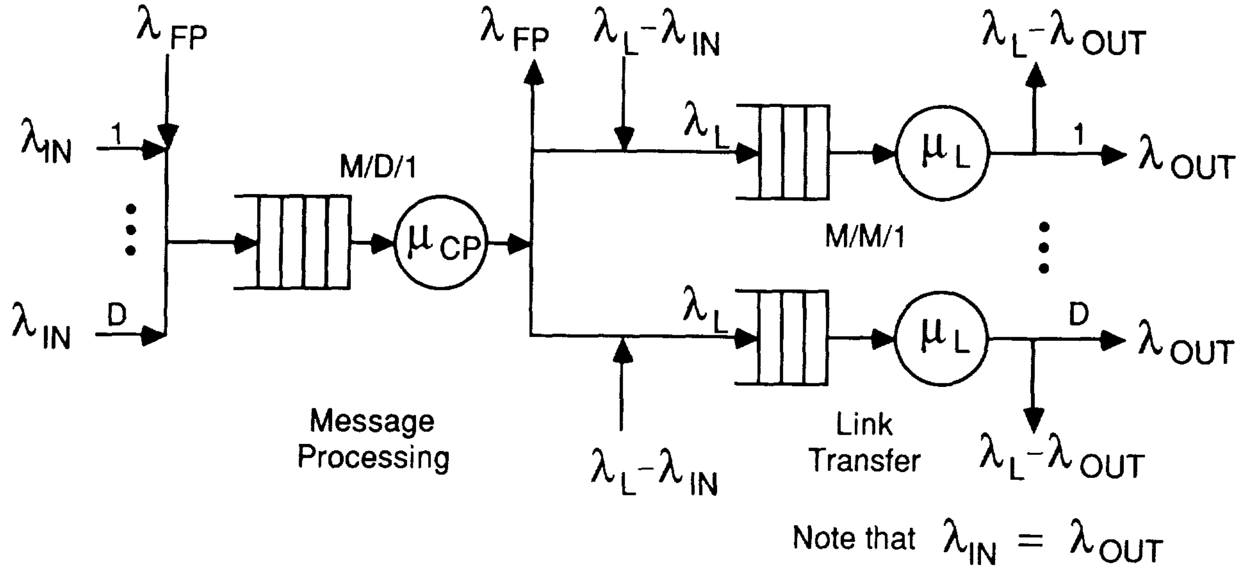


Figure 1. Queueing model for a single node

Given assumptions 3, 4, and 5, traffic conditions are the same at all links and nodes. Assumptions 1, 2, 5, and 6 allow the analysis to be tractable and are reasonable for large, well-designed multicomputer networks with varying message sizes. Assumptions 3 and 4 are true for many, but certainly not all, popular topologies.

### Node Model

Nodes consist of an FP, a CP, and a communication controller (CC) or port for each link. Arriving messages are received at a CC and placed in memory for processing by the CP. Messages from links and the local FP form a queue at the CP. The CP processes each message and passes it on to the local FP or to a CC. The CC transfers outgoing messages to the next node when the link is free. Figure 1 shows a queueing model for a single node adapted from Ref. 3. Message processing time at the CP is fixed and arrivals are Poisson, so the CP stage is modeled as an M/D/1 queue with arrival rate $\lambda_{CP}$ given by $\lambda_{CP} = \lambda_{FP} + D\lambda_L$. $\lambda_{FP}$ is the mean message generation rate by the FP, $\lambda_L$ is the mean message arrival rate on one link, and $D$ is the number of links.

Let $1/\mu_{CP}$ be the CP processing time. Then $T_{CP}$, the wait and service time at the first stage, is

$$T_{CP} = \frac{1}{\mu_{CP}} + \frac{\lambda_{CP}}{2\mu_{CP}(\mu_{CP} - \lambda_{CP})} \tag{1}$$

The departure distribution from the M/D/1 queue, and hence the arrival distribution at each link queue, is very close to Poisson (Ref. 3). With this approximation and exponential message transfer times, the link stage is modeled as an M/M/1 queue with service rate $\mu_L$ determined by the mean message length and link bandwidth. For a balanced network, the arrival rate at each of the M/M/1 queues is the same and the total delay through the second stage, $T_L$, is

$$T_L = \frac{1}{\mu_L - \lambda_L} \tag{2}$$

The total delay at a single node is $T_N = T_{CP} + T_L$.

### End-to-End Message Delay

The end-to-end delay experienced by a message from its source to its destination is determined by delays at each node, $T_{CP}$ and $T_L$, and the distance or number of hops travelled. The expected number of hops, $N_h$, depends on topology and the distribution of source-destination pairs. Common destination assumptions are uniform, where all nodes in the network are equally likely to be the destination, and sphere of locality, where the destination is within fixed distance $d$ from the source with probability $\phi$ and is at a distance greater than $d$ with probability $1 - \phi$ (Ref. 4).

For example, consider a $W^D$ torus topology with uniform traffic distribution. The $W^D$ torus is a $D$ dimensional nearest neighbor mesh of width $W$ with end-around connections (Ref. 5). Each node is connected to its neighbors by point-to-point links. The mean internode distance for the torus with $W$ even is (Ref. 4)

$$N_h = D \times (\frac{W}{4}) \times (\frac{N}{N-1}) \simeq \frac{DW}{4} \tag{3}$$

The factor $N/(N-1)$ occurs since nodes do not send messages to themselves. Similar results exist for $W$ odd.

In message switching, a message must be completely received at each intermediate node before it can be forwarded. If a message takes $N_h$ hops from source to destination, it encounters $N_h + 1$ message processing delays and $N_h$ link delays. Using Equations 1 and 2, the expected end-to-end delay for a message switched network is

$$T_{MS} = (N_h + 1)[\frac{1}{\mu_{CP}} + \frac{\lambda_{CP}}{2\mu_{CP}(\mu_{CP} - \lambda_{CP})}] + \frac{N_h}{\mu_L - \lambda_L} \quad (4)$$

Cut-through switching (Ref. 2) is similar to message switching except that messages do not have to be received completely before being transmitted out of the node. After the header is received, the outgoing link can be selected, and if the link is free, message transmission may start immediately. If the selected link is busy, the operation follows that of message switching. The probability of finding a free link is the probability of an idle M/M/1 queue, $1 - \rho$, where $\rho = \lambda_L/\mu_L$ is the link utilization factor. Out of $N_h - 1$ intermediate nodes, the expected number of cuts, $N_c$, is $N_c = (N_h - 1)(1 - \rho)$.

When a cut is made, the node service time, conditioned on the wait time at the link being zero, less the header transfer time is saved. Let $\alpha$ be the ratio of header length to total average message length, including the header. Using $N_c$ from above, the end-to-end delay for cut-through switching is

$$T_{CT} = T_{MS} - (N_h - 1)(1 - \rho)[T_{CP} + \frac{(1 - \alpha)}{\mu_L}] \quad (5)$$

Values for $\lambda_L$ and $\lambda_{CP}$ needed to compute $T_{MS}$ and $T_{CT}$ are determined for a given network from $N_h$ and $\lambda_{FP}$. $\lambda_{FP}$ is a function of the application executing on the multicomputer. If $N$ is the number of nodes and $N_L$ is the number of links in the network, then the traffic on each link, $\lambda_L$, is

$$\lambda_L = \frac{N_h N \lambda_{FP}}{N_L} = \gamma \lambda_{FP} \quad (6)$$

The CP traffic, $\lambda_{CP}$, is

$$\lambda_{CP} = (1 + N_h)\lambda_{FP} = \beta \lambda_{FP} \quad (7)$$

For a $W^D$ torus with uniform traffic, $\lambda_L = (W\lambda_{FP})/4$ and $\lambda_{CP} = (1 + DW/4)\lambda_{FP}$.

### Network Analysis Tool

To assess design alternatives, the "Network Analyzer" program, based on the performance model, was developed to serve as a computer-aided design and analysis tool. The program accepts a specification of a network and message distribution and computes end-to-end delay times that can be plotted as a function of $\lambda_{FP}$. Parameters can be changed easily to perform a *what-if* analysis. The program is written in PASCAL for the IBM PC.

The program allows seven parameters to be specified:

1. network topology,
2. CP processing time,
3. link bandwidth,
4. mean message length,
5. header length,
6. switching technique, and
7. traffic pattern.

Analysis parameters are automatically calculated for the general $W^D$ torus and $W^D$ spanning bus hypercube topologies. Note that many common networks, e.g. the two-dimensional mesh with wrap-around and the binary hypercube, are forms of these two topologies. Other topologies can be analyzed by specifying $N_h$, $\gamma$, and $\beta$.

## APPLICATION EXAMPLE

To illustrate the effectiveness of the model and the Network Analyzer as a design and analysis tool, we consider a hypothetical multicomputer network design problem. Assume a multicomputer system with the following characteristics:

- network size of $N = 1024$ or 4096 nodes,
- average message length of 512 bytes,
- uniform traffic distribution,
- message switching,
- $1/\mu_{CP} = 0.1$ milliseconds, and
- $\lambda_{FP}$ not to exceed 1000 messages/second.

We consider using a $W^D$ torus versus a $W^D$ spanning bus hypercube (Ref. 5). The delay in each network is given by Equation 4. The average number of hops, $N_h$, is given by Equation 3 for the torus and by Equation 8 for the spanning bus hypercube.

$$N_h = D(\frac{W - 1}{W}) \quad (8)$$

The design objective is to select the most economical network design, without compromising network performance, for cost function $C = BW \times N_{cn} \times N_L$. $BW$ is the link bandwidth, $N_{cn}$ is the number of connections per node, and $N_L$ is the total number of links in the network.

The cost of a $N = W^D$ node spanning bus hypercube is

$$C_{SB} = BW \times D \times DW^{D-1} \quad (9)$$

and the cost of a $N = W^D$ node torus network is

$$C_T = BW \times 2D \times DW^D = 2WC_{SB} \quad (10)$$

As indicated in Equation 10, for a given bandwidth $BW$, network width $W$, and dimension $D$, the spanning bus hypercube offers a $2W$ cost advantage over the torus. For $W = 4$, a spanning bus hypercube with $BW = 40$ megabits/second has half the cost of a torus with $BW = 10$ megabits/second for the torus. Using $W = 4$, $D = 5$ produces a network with $N = 1024$ nodes and $D = 6$ produces a network with $N = 4096$ nodes.

Figure 2 shows message delays for the spanning bus hypercube and torus for $N = 1024$ nodes and $N = 4096$ nodes. The performance of the spanning bus network is clearly superior. Even in the 4096 node spanning bus hypercube, low delays can be expected in the spanning bus network with $\lambda_{FP}$ as high as almost 1600 messages/seconds. This can be compared to the 4096 node torus which begins to saturate at around 1300 messages/second. For $W = 2$ , costs are equal and the spanning bus hypercube is still superior, although its performance is less than for $W = 4$ due to increased internode distances.

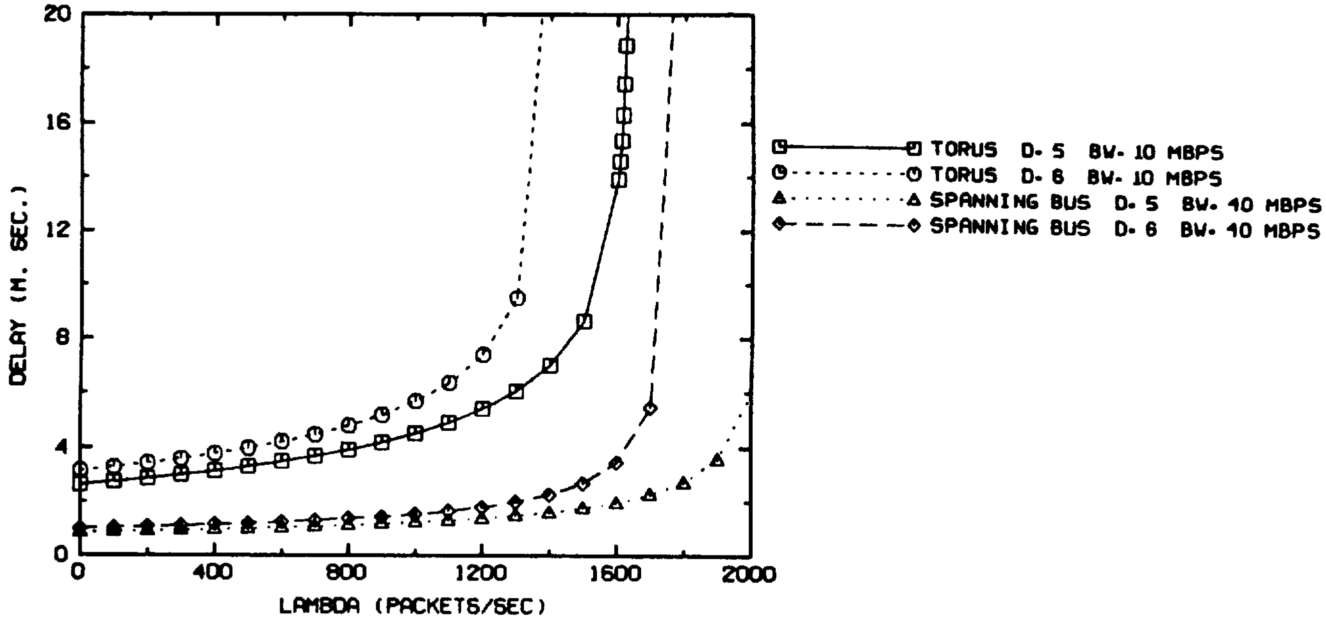


Figure 2. Performance of torus and spanning bus networks

## CONCLUSIONS

Communication delay must be carefully considered in the design of multicomputer networks since it may contribute significantly to the overall system performance. General performance analysis models are needed to evaluate different network design alternatives. This paper presented a general network model for a wide class of multicomputer networks that explicitly considers both message processing and transmission delays. The model can be used to analyze networks using either message or virtual cut-through switching, bidirectional or unidirectional links, and point-to-point or multiple bus topologies. The Network Analyzer program uses the model to analyze alternative designs under different work load assumptions.

## ACKNOWLEDGEMENTS

This work was supported in part by a "Digital Faculty Program/Incentives for Excellence" grant from the Digital Equipment Corporation.

## REFERENCES

1. T. Feng, "A Survey of Interconnection Networks," *Computer*, Vol. 14, No. 12, pp. 12-27, Dec. 1981.

2. P. Kermani and L. Kleinrock, "Virtual Cut-through: A New Computer Communication Switching Technique," *Computer Networks*, Vol. 3, pp. 267-286, Sept. 1979.

3. D. A. Protopapas and J. N. Denenberg, "A New Model for Performance Analysis of Large Scale Multicomputer Networks," *Proc. 6th Phoenix Conf. on Computers and Communications*, 1987, pp. 451-456.

4. D. A. Reed and D. C. Grunwald, "The Performance of Multicomputer Interconnection Networks," *Computer*, Vol. 20, No. 6, pp. 63-73, June 1987.

5. L. D. Wittie, "Communication Structures for Large Networks of Microcomputers," *IEEE Trans. on Computers*, Vol. C-30, No. 4, pp. 264-273, April 1981.

# PARALLEL HASHED KEY ACCESS ON THE CONNECTION MACHINE

Paul B. Anderson
Planning Research Corporation
McLean, VA 22102

## ABSTRACT

This paper discusses retrieval of a number of keys in parallel from a single hash table. The algorithm is a data parallel (Ref. 1) adaptation of a familiar sequential method. Changes were made to suit the single instruction stream, multiple data stream Connection Machine (Ref. 2). Running time is proportional to the maximum run of collisions rather than the average number. The algorithm was developed as a portion of a large X-ray crystallography application for the Connection Machine (Ref. 3). The generality of the scheme may make it useful for other massively parallel applications.

Keywords: Hashing, Searching, Parallel Algorithms, Connection Machine, SIMD, MIMD.

## INTRODUCTION

The algorithm is based on three assumptions:

1. The algorithm must be suitable for efficient implementation on the Connection Machine.
2. The hash table contains no duplicate keys.
3. The algorithm is to make use of the maximum number of processors otherwise needed for the application, but no more.

These assumptions appear to be reasonable for other uses of the algorithm.

## THE PARALLEL ALGORITHM

The basic concept behind the implementation is the sequential hash table algorithm as described in Ref. 4 using a fixed size table and linear probing. Insertion in the parallel version follows the algorithm given below. The keys are denoted by $K$ and the parallel hash function by $h$. One key is stored in each active processor and the table has one entry in each processor, active or not.

```
I1. [Hash] Evaluate hash function on all keys $K$ and save indices in parallel variable $i$.
I2. [Store key] Store keys $K$ in hash table at indices $i$. Note that for collisions, only one of the colliding keys stored in a given table entry (processor) will survive.
I3. [Check storage] Retrieve stored keys from table indices $i$ and compare in parallel with keys $K$. Disable all processors with keys that match the retrieved keys. These keys have been successfully stored in the table.
I4. [Termination?] If no active processors remain, the algorithm terminates. Otherwise continue on to step I5.
I5. [Next probe point] Increment indices $i$. For those processors with $i$ values beyond the table size, set $i$ to $0$. Repeat from step I2.
```

The running time of this algorithm is clearly dependent on the maximum number of times steps I2 through I5 are executed. That is, the running time is proportional to the maximum run of collisions. This is in contrast to the normal sequential hash algorithm where overall performance depends on the average collision run length.

Retrieval from the parallel hash table follows a similar algorithm.

## IMPROVEMENTS

The basic scheme of parallel hashing can be improved in several ways. Some of these ways have been investigated in Connection Machine implementations. The methods are:

- rehashing collisions,
- use of a larger table,
- primary and secondary tables, and
- chaining.

The first variation is a parallel analog to the rehash method described in (Ref. 4, page 521). In this method, collisions are resolved using a secondary hash function which determines, for each key, a sequence of secondary hash locations. These locations are examined for an insertion point or, during retrieval, for a search key. The method does improve performance of the algorithm but has a basic limitation in the sense that it is dependent on the quality of the hash key and on the actual data. As in the non-parallel case, the method comes close to approximating a perfectly uniform hash distribution.

A larger hash table can greatly improve performance by lowering the load factor. However, this is frequently not possible in the non-parallel case because the table is sized to fit the maximum allowable space. On the Connection Machine, the clear choice for the table size is one element per processor. Since it is difficult to change the number of processors available with current (Spring 1988) software, this technique is not generally feasible.

A variation on increasing the table size called twin hash tables has been implemented. In the implementation, two parallel variables are allocated per processor, one as the primary hash table and the other as a secondary hash table. Initial hashing uses the primary table. Collisions in the primary table are stored in the secondary table. Tertiary and higher collisions are also stored in the secondary table. This results in good performance with little increase in the size and complexity of the basic algorithm. Keys are spread out among the two tables resulting in a lower overall load factor. Experience with the algorithm suggests that much of the benefit expected from using a table twice as large is obtained in this implementation.

The final variation involves chaining (Ref. 4, page 513). Chaining is a popular method of organizing hash tables. In typical implementations a short, fixed length table is used as a vector of list heads. Keys are hashed to the list heads and collisions are chained together. This results in effectively dividing the linear search time by the size of the vector of list heads. There are problems implementing chaining using the current Connection Machine *Lisp software (Ref. 5) due to the need to allocate memory uniformly across all processors. The primary and secondary table method can be seen as a hybrid between fixed table hashing techniques and chaining techniques.

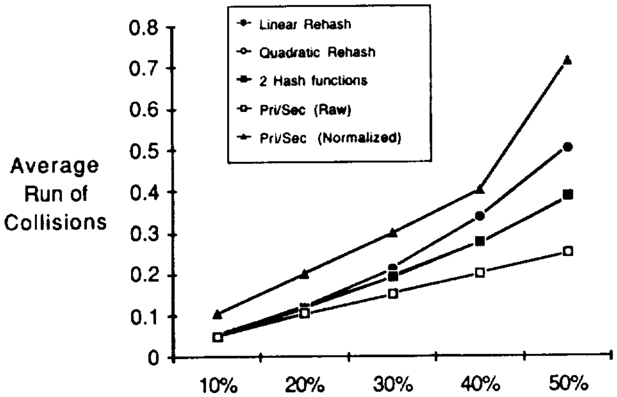


Figure 1. Average Collisions Versus Load Factor.

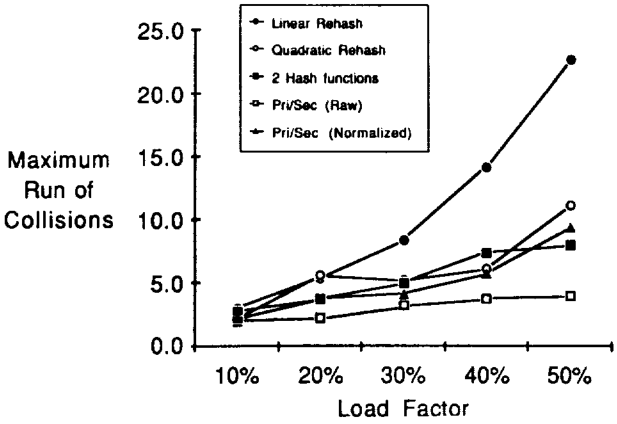


Figure 2. Maximum Collision Run Versus Load Factor.

## PERFORMANCE

The algorithm has been analysed using a simulation technique which permits comparison of algorithm performance with uniform hashing under multiple conditions. The results of these simulations are shown in Figures 1 and 2. In these figures, the twin table method is labeled as Pri-Sec and appears both in raw form and with normalization. Normalization is needed to compensate for the fact that the twin table method uses twice the space of the other method and hence should be compared to tables of twice the size or equivalently with half the load factor.

Figure 1 shows the average number of collisions, the usual performance measure of hash tables. In terms of this metric, the twin table method appears better in raw form but when normalized, does not appear to offer any advantages. In Figure 2, the use of primary and secondary hash tables is shown to be better than or roughly equal to the best of the other methods. Although not shown in the figures, the twin table method deteriorates at higher normalized load factors because the secondary table fills long before the primary table. This is an obvious property of the algorithm and may preclude its use in some situations.

Operationally, we have measured the algorithm in practical use and have concluded that the twin table method is superior to both the single table and rehash methods. The twin table method is simple in its parallel implementation and maps well onto the Connection Machine architecture.

## GENERALITY

The algorithm described is a dependent on the Connection Machine's SIMD architecture. However, it is possible to modify the algorithm to execute on MIMD architectures with private processor memories. Retrieval uses the sequential algorithm essentially unchanged but insertion requires some processor coordination to eliminate race conditions.

## REFERENCES

1. Hillis, W. D. & Steele, G. L., "Data Parallel Algorithms". *Communications of the ACM 29, 12* (Dec 1986), 1170-1183.

2. Hillis, W. D., *The Connection Machine*. MIT Press, Cambridge, Mass., 1985.

3. Anderson, P. B. & Flippen-Anderson, J. L., "A Crystallographic Application for the Connection Machine". In Proceedings of the Third International Conference on Supercomputing, Boston, May 1988.

4. Knuth, D. E., *The Art of Computer Programming*. Volume 3, *Sorting and Searching*. Addison-Wesley, Reading Mass., 1973, page 518.

5. Thinking Machines Corporation, *The Essential *LISP Manual*. Thinking Machines Corporation, Cambridge, Mass., 1986.

# Object-Oriented Prototypes of Parallel Architectures for the Performance Evaluation of Algorithms

Herb Barad
Electrical Engineering Dept.
Tulane University
New Orleans, LA 70118-5674
email: barad@bourbon.tulane.edu

## Abstract

This paper describes a methodology for constructing prototypes of parallel architectures. The prototypes model the functionality of the components of the architecture to any specified level of detail. The prototypes are programmable, thus benchmarks of algorithms can be executed. The prototypes are also flexible, thus allowing the "fine-tuning" of the architecture to enhance the performance.

In particular, this paper covers a specific working case study. A working prototype of a pyramid architecture has been constructed to 7 levels (5461 processors and 53564 interconnections) and is used to evaluate the mapping of a wide range of computer vision algorithms (i.e. low level image processing through high level symbolic computations) on such an architecture. The results indicate that a pyramid architecture is not only useful for low level image processing tasks, but that segmentation and even higher level symbolic computations can be performed efficiently. Results are presented for a pyramid implementation of a split-and-merge segmentation algorithm.

## 1. Introduction

The design of parallel architectures often requires a prototype or test bed in which to develop and simulate algorithms. A test bed also provides an environment for collecting performance benchmarks.

The method of constructing the SCOOP[1] architecture uses an object-oriented methodology and allows for the modeling of the architecture at various levels of detail: architecture level, processor level, processor component level, gate level, and so on. The methods for constructing the prototype provide great flexibility. A prototype built using these methods is easily reconfigured and allows for the construction of other architectures of interest.

The prototype constructed for this research is configured as a pyramid that is modeled at the processor level. By changing as little as a single method the prototype can represent a SCOOP architecture of another topology. By changing the description of the processors (or ports), the prototype can represent architectures constructed from different components. Thus, the SCOOP pyramid is a useful tool for researchers who wish to prototype, benchmark, and design algorithms for their proposed architectures.

## 2. Motivation

There are problems when attempting to integrate different levels of vision tasks on the same architecture. Many architectures are suited for a particular class of tasks. Massively parallel two-dimensional mesh-connected arrays are well suited for low level local operations, but their SIMD nature does not efficiently support more complex algorithms. MIMD mesh-connected arrays still suffer from lack of global communication and global control.

Figure 1 emphasizes another difficulty in attempting to integrate various tasks into a system. Suppose there tasks A and B are to be processed. Further, suppose that two separate architectures are used—each one is designed to efficiently execute task A and task B, respectively. If the communication between processes A and B (i.e. between subarchitectures A and B) takes longer than the processing of tasks A and B, then the overall process is now I/O-bound. That is, an inter-architecture bottleneck is formed as data is pushed from one architecture to the other. The effects of this bottleneck become even more severe when we consider that the overall process might be an iterative process between tasks A and B.

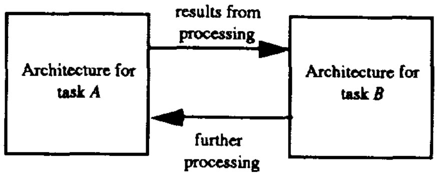


Figure 1. Separate architectures for separate tasks.

A prime motivation of this work is the desire to integrate all levels of the vision process onto the same architecture to avoid such a bottleneck that would result in reevaluation of the data. If higher-level analysis showed that further lower-level processing is needed (goal directed processing), then switching back and forth between separate specialized architectures results in a bottleneck. Therefore, it becomes important to be able to efficiently perform different tasks at different levels of abstraction on a unified architecture that would allow a smooth transition between these levels.

Another motivation is to develop a methodology to construct prototypes of proposed architectures. It is often difficult to build the full-sized architecture to be studied and often a scaled-down hardware prototype is not a suitable representation of the final architecture. Software prototypes have several advantages over scaled-down hardware prototypes:

1. The time needed to design, build, and debug the prototype is considerably less in software than in hardware.

[1] The acronym SCOOP stands for "Southern California Object-Oriented Prototype."

CH2649-2/89/0000/0647$01.00 © 1988 IEEE

2. The software prototype can better assist the software developer of algorithms for the prototype since the software prototype can represent an architecture of a larger size.
3. Software prototypes are much cheaper to build.
4. Software prototypes can be easily modified to reflect architectures of other topologies, or even architectures composed of radically different types of hardware (e.g. optical).

Of course, the major drawback of the software prototype is that the actual time to execute an algorithm is much longer than the hardware prototype. Several microseconds of processing in hardware may require minutes of simulation. Thus, the software prototype is ideal for designing algorithms, analyzing the architecture, and performing benchmarks, but extremely poor for processing data in a production environment.

## 3. Structure

Figure 2 shows the class hierarchy of the structure of the processors modeled. This pyramid is constructed so that processors at higher levels are more powerful than the processors at the lower levels, thus suggesting the class structure.

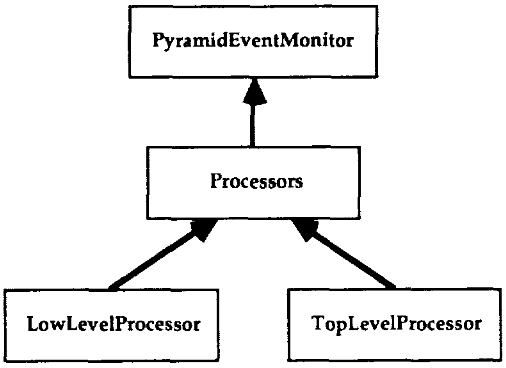


Figure 2. Class hierarchy of processors

The construction of the pyramid involves the creation of thousands of Smalltalk objects. There are objects to represent each of the processors, the ports between the processors, and objects to represent the architecture as a whole. There are also objects to represent the different values and parameters during the simulation: pixel values, results from operations, Boolean flags, and so on. The process of building the pyramid begins with the sending of the message `createPyramid:` to the driving simulation object class (a subclass of **Pyramid** class). The argument of the message is the number of levels for the pyramid. The method starts out by instantiating a new instance of a **Pyramid** (a bare skeleton) and then starts a long process of creating instances of **Processors**, **TopLevelProcessor**, **LowLevelProcessor**, and **UnidirectionalPort**. Each of these will be connected properly to its neighboring processors through the ports.

The actual simulation starts by creating an instance of the class representing the simulation. For example, an instance of class **Convolution** is created for the simulation of a convolution of the image by a kernel. The instance of the simulation sends a message to the classes **Processor**, **LowLevelProcessor**, **TopLevelProcessor**, and **UnidirectionalPort** to create instances of each of these. These objects are then interconnected in a fashion that represents the topology of the architecture. The pyramid is created by connecting each level of processors together and then interconnecting the levels. An interconnection is created by assigning a common instance of class **UnidirectionalPort** to the processor's instance variables that represent the correct port connection to its neighbor. For example, the processor-(5, 5, 5)[2] is connected to its northern neighboring processor-(4, 5, 5) by assigning an instance of **UnidirectionalPort** to processor-(5, 5, 5) instance variable toNorth and also assigns the very same **UnidirectionalPort** to processor-(4, 5, 5) instance variable fromSouth. This is continued until the entire architecture is constructed and interconnected. To repeat, the prototype is built by creating an instance of each element of the architecture and then by interconnecting those elements to form the topology of that architecture.

## 4. Results

Figure 3 represents object table usage as a function of real-clock time (not simulated time) for a 7 level pyramid simulation of a particular image segmentation algorithm called the Ohlander-Price-Reddy method [1].

In addition to object table usage, a prototype requires memory for each of the objects themselves. Figure 4 represents memory usage as a function of time for a 7 level pyramid OPR simulation.

The pyramid is constructed (i.e. all processors and ports in place) by 4.5 minutes. This is the most intensive use of objects during the preparation for the simulation. In fact, it is the most intensive use of objects throughout the entire simulation. The consumption of object table space increases rapidly and almost linearly during the pyramid construction stage. This happens during the execution of the method `createPyramid:` when sent to the class **Pyramid**. The 5,461 processors and 53,564 ports consume almost 150,000 objects during the construction. These results are expected because a large number of similar objects are created during the construction phase. Both Figure 3 and Figure 4 represent the construction phase and the simulation startup phase. The simulation continues with approximately constant object table usage and memory usage for the next 600 minutes until completion of the segmentation.

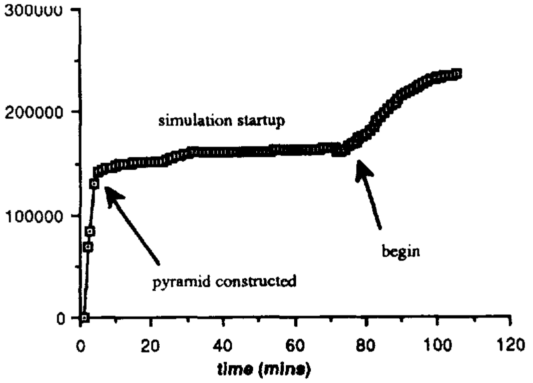


Figure 3. Object table usage in 7 level SCOOP pyramid

For the following times, N is the size of the bottom level, p is the number of data bands, R is the number of regions, G is the number of gray levels, and k is the size of the neighborhood used in merging. The following times are for 1 iteration.

[2] Processor-(x,y,l) refers to the processor at location x,y on level l.

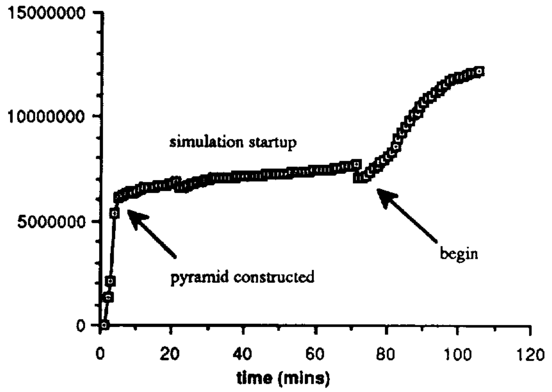


Figure 4. Memory usage for a 7 level SCOOP pyramid

The communication for accumulating the bin counts takes O(N) steps. The time to send the histograms up takes O(RpG) steps. The histogram analysis requires O(RpG) steps (serially). Broadcasting the threshold rule takes O(logN) steps. Table 1 lists the times needed (order of magnitude) for the different subtasks of a single iteration of the OPR segmentation algorithm using serial, 2-D mesh-connected network[3], and pyramid architectures.

| | histogram | threshold | analysis | merge |
|---|---|---|---|---|
| serial | $RpN^2$ | $N^2$ | RpG | $k^2N^2$ |
| 2-D MCN | RpN | N | RpG | $k^2$ |
| pyramid | N+RpG | log N | RpG | $k^2$ |

Table 1. Performance for OPR iteration

There is quite a speed up for certain tasks in the pyramid (more noticeable as N becomes large). One takes advantage of using a powerful processor as the top processor in the SCOOP pyramid. When the histogram is loaded into this processor for analysis, it can analyze the histogram at a very fast speed so that the resulting threshold will be obtained very quickly. The histogram is still the dominant task in the OPR algorithm and the SCOOP Pyramid manages to speed it up quite a bit. Note that all the results are for just one OPR iteration. The number of iterations is dependent on the data, and the effects of the speed-up is compounded if there are many iterations (i.e. as the complexity of the data increases).

Figure 5 shows an example of a segmented image using the Ohlander-Price-Reddy segmentation algorithm on an image. The results are good considering the input data was 64x64 pixels, 3 data bands, each 6-bits deep. This image is typical of the results obtained using the prototype.

## 5. Conclusion

In summary, a working software prototype of a pyramid architecture has been constructed. An object-oriented methodology is used for constructing the model of the architecture. The SCOOP pyramid is then used as a test bed to perform simulations of a wide range of computer vision tasks. The motivation behind this is to explore the potential of the pyramid architecture as a single, unified architecture to perform a wide range of vision tasks. The software prototype offers great flexibility. The structure or the size of the architecture can be changed with a small effort. The methods of building prototypes used in this work can assist future work in architecture studies. Prototypes of other architectures, including optical architectures, can be built and studied. These SCOOP models can serve as a set of test beds for performing benchmarks on different algorithms.

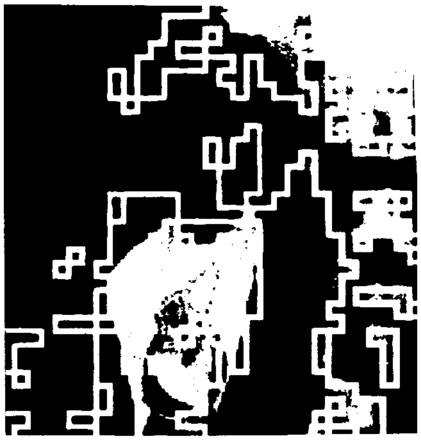


Figure 5. Segmentation results on 64x64 image

## References

[1] Ohlander, R., Price, K. and Reddy, D. R., "Picture Segmentation using a Recursive Region Splitting Method," *Computer Graphics and Image Processing*, no. 3, pp. 313—333, 1978.

[2] Cantoni, V. and Levialdi, S., *PAPIA: A Case History*, in Parallel Computer Vision, Academic Press, Inc., Uhr (ed.), pp. 3—13, San Diego, 1987.

[3] Schaefer, D.H., Ho, P., Boyd, J., and Vallejos, C., *The GAM Pyramid*, in Parallel Computer Vision, Academic Press, Inc., Uhr (ed.), pp. 15—42, San Diego, 1987.

[4] Tanimoto, S.L., Ligocki, T.J., and Ling, R., *A Prototype Pyramid Machine for Hierarchical Cellular Logic*, in Parallel Computer Vision, Academic Press, Inc., Uhr (ed.), pp. 43—83, San Diego, 1987.

[5] Weems, C., Lawton, D., Levitan, S., Riseman, E., Hanson, A., and Callahan, M., "Iconic and Symbolic Processing Using a Content Addressable Array Parallel Processor," *Conference on Computer Vision and Pattern Recognition*, pp. 598—607, IEEE Computer Society, IEEE Computer Society Press, San Francisco, June, 1985.

[6] Levitan, S.P., Weems, C.C., Hanson, A.R., and Riseman, E.M., *The UMass Image Understanding Architecture*, in Parallel Computer Vision, Academic Press, Inc., Uhr (ed.), pp. 215—248, San Diego, 1987.

[7] Pope, S.T., Goldberg, A., Krasner, G., and Bay, D., *The Smalltalk-80™ Programming System: Reference Guide and Release Notes*, ParcPlace Systems, Palo Alto, CA, revision 2.2c, 1987, DE Version -Release 1.

[8] Law, A.M. and Kelton, W.D., *Simulation Modeling and Analysis*, McGraw-Hill Book Company, McGraw-Hill series in industrial engineering and management science, San Francisco, 1982.

[9] Goldberg, A. and Robson, D., *Smalltalk-80: The Language and its Implementation*, Addison-Weseley, Menlo Park, CA, 1983.

[10] Barad, H.S., "The SCOOP Pyramid: An Object-Oriented Prototype of a Pyramid Architecture for Computer Vision," Ph.D. Thesis, University of Southern California, Los Angeles, December, 1987.

[11] Uhr, L., *Highly Parallel, Hierarchical, Recognition Cone Perceptual Structures*, in Parallel Computer Vision, Academic Press, Inc., Uhr, pp. 249—292, San Diego, 1987.

[3] The 2-D Mesh-Connected Network (MCN) histogram computation leaves a distributed histogram along the first column of processors. An additional N steps are required to have them "funnel-off" to a single processor for analysis.

# CHARACTERIZING THE ADVANTAGES OF MASSIVELY PARALLEL COMPUTING

R. M. Hord, T.A. Kraay, E.P. McMahon

MRJ, Inc.

## ABSTRACT

In August 1986, the Advanced Development Center of Perkin-Elmer, operated by its subsidiary, MRJ, Inc., took delivery of the first commercial Connection Machine™ (CM) computer produced by Thinking Machines Corporation of Cambridge, MA. A second 16K machine was delivered in March 1987 and was upgraded to a CM-2 in January 1988. A Data Vault and Frame Buffer have also been added. Although this paper talks of our experience with the CM, we should note that our company and staff have experience with a wide range of advanced computers including Cray, the Massively Parallel Processor at GSFC, the Martin Marietta GAPP (Geometric Arithmetic Parallel Processor), the DAP from Active Memory Technology, Butterfly (BBN), FPS array processors, and the Illiac IV. Hence, our opinions are meant to apply to massively parallel processing in general.

The intent at MRJ has been to develop applications on the CM for various Government defense and space customers by first performing research and feasibility studies, then implementing operational applications. We were aware before our commitment that there was an attitude in the computer science community that only a relatively small percentage of computer applications were suitable for implementation on a massively parallel SIMD architecture, problems that are sequential in nature or parallel to a low degree. It has been observed over our two plus years of experience that the applications relevant to our chosen business areas don't exhibit this characteristic. The diversity of our applications is reported elsewhere at this conference, and we can comment that virtually all of the applications we've seriously examined do lend themselves to massively parallel approaches. The CM was selected by MRJ instead of alternative parallel processors primarily because massively parallel processing offered the *possibility* of obtaining significant breakthroughs in various *algorithmic* solutions to problems.

The technical literature of parallel processing is replete with reports of efforts to obtain a speedup factor of k through the use of k processors. For some applications, particularly those involving no interprocessor communications, this k speedup factor is obtained in a straightforward way. Examples include processing k pixels, searching k documents, tracing k rays, and computing k parametric cases in parallel.

On a massively parallel, suitably connected computer, in addition to the quantitative difference in processor count from coarse grain machines, there is also a qualitative difference. That qualitative difference occurs because k, the number of processors, starts to approximate N, the characteristic size of the problems addressed.

One of us (Kraay) has devised new algorithms that allow the CM to achieve or closely approximate speedup factors of k(=N). A simple example is the MAX function, which is 0(N) on sequential machines but 0(1) on the CM. Another example is the FFT; instead of 0(N log N) for a serial machine, the CM algorithm is 0(log N). A third example is matrix inversion. By using N processors for a matrix of size N x N, the $0(N^3)$ sequential algorithm becomes $0(N^2)$; or if $N^2$ processors can be employed, inversion becomes an 0(N) activity. Execution times have been achieved for these algorithms that are better than published times for supercomputers costing significantly more than the CM.

It is not just sufficient to achieve the best order implementation; the scaling constant must be minimized also. This constant is generally strongly dependent on the efficiency of data communication which, in turn, is dependent on the specific hardware and the creative use made of it in an implementation.

These algorithms and others are at the heart of solutions for difficult large problems. The order advantage made possible by massively parallel processing and CM connectivity makes the commercial success of these machines and algorithms inevitable as ever larger problems are being pursued in science and engineering; more and more computational problems are real-world three-dimensional instead of one- and two-dimensional abstractions. Three-dimensional applications grow as the cube of a linear dimension S, so as S doubles, the number of data items increases eightfold. Having an algorithm, whether matrix solving, FFT, or some other, characterized by a lower order measure, gives rise to a growing advantage with problem size. Hence, these improved algorithms can result in orders of magnitude advantage over essentially sequential serial supercomputers.

Another consideration makes fine-grain data level parallelism more attractive than coarse-grain parallelism, and that is the potential for growth. We believe orders of magnitude in performance can be gained by straightforward methods such as extensions of the dimension of the hypercube structure, increasing the memory and power of each processor, and faster clock speeds. Exotic and expensive technologies are not yet necessary to achieve enormous increases in performance. Furthermore, the same algorithms we're developing for the CM-2 are directly extensible to these machines of the future.

The cost of programming and training cannot be ignored in contract applications. Our focus is on applications rather than in direct research and necessity has been the mother of our invention of new algorithms. The achievements gained by algorithm development under our corporate internal R&D programs have been applied in later contract work, and all our developers have access to superior algorithms just as would conventional scientific programmers have access to a scientific subroutine package. The unique creativity needed to produce new parallel algorithms is not generally required for development work. Most of our work has been in development and we have demonstrated that productivity on a massively parallel computer is about the same as productivity in conventional scientific programming; the same creativity, program design capability, expertise in the appropriate fields, and discipline are required. There is, of course, a learning curve, and there are those who acquire the skill of using massive parallelism better than others, but we find analogous differences with FORTRAN or C development programmers.

Our experience has been that developing programs on the CM is significantly easier than efforts reportedly involved on MIMD coarse-grain parallel processors with the attendant result that there is a competitive advantage in developing the application as well as in executing the application. MRJ has a mixed strategy with regard to programming languages and approaches. For primitive operations such as FFT and sort, routines that will be executed many times and used by many programs, coding by hand from scratch in PARIS is standard. Less computationally intensive programs cause us to study our choice: PARIS code runs faster, C* makes more cost effective use of the hardware because the VAX supports multiple development workstations, and *LISP provides a richer software development environment. Factors affecting the choice include project schedule in relation to hardware availability, need for access to the Data Vault and Frame Buffer, and available staff skills. For our kind of work it is very important that the application implementer be knowledgeable about the application itself as well as about how the coding is accomplished.

In summary, we have made a corporate commitment to and are meeting our business goals in pursuing the application of massively parallel processing.

John Cocke received the 1988 ACM Turing Award. In his prize lecture, "The Search for Performance in Scientific Processors," he said, "Of the three contributors [algorithm, computer, machine organization] algorithm improvements are the most important. An idea that changes an algorithm from $N^2$ to NlogN operations . . . is considerably more spectacular than an improvement in machine organization, where only a constant factor of run time is achieved."

We believe he is right. We also believe our algorithms that Cocke would admire have been made possible by the advent of the CM.

# A SYSTEMATIC APPROACH FOR DESIGNING PIPELINED DATA PARALLEL ALGORITHMS

Chung-Ta King
Department of Computer and Information Science, New Jersey Institute of Technology,
Newark, NJ 07102

Wen-Hwa Chou, Lionel M. Ni
Department of Computer Science, Michigan State University,
East Lansing, MI 48824

## ABSTRACT

Pipelined data parallel computations achieve a high degree of parallelism by partitioning the data set among the processors and operating on the data blocks in a pipelined fashion. The resulting algorithms are very suitable for execution on multicomputers. A systematic procedure for designing pipelined data parallel algorithms is introduced. This procedure starts with a nested-loop program, manipulates the dependencies between the loops, and groups related loops to obtain pipelined and data parallel operations. Using this procedure, it is possible to parallelize a nested loop automatically.

Keywords: Parallel processing, pipelining, algorithm design, data parallel algorithms, program restructuring

## INTRODUCTION

A data parallel computation divides the set of data among multiple processors. Parallelism is achieved by simultaneously operating upon large sets of data, rather than using multiple threads of control. The resultant algorithms are termed *data parallel algorithms* (Ref. 2). In terms of programming, programs with loops to handle static and regular data structures are suitable for data parallel computation.

*Pipelined data parallel algorithms* are a subset of data parallel algorithms, which, in addition to data parallelism, exploit processor-level pipelining among the multiple processors (Ref. 3,4). Pipelined data parallel algorithms are targeted at medium- to large-grain parallel computers, especially multicomputers (Ref. 1).

Multicomputers with a few hundred processors are common. Thus, they hold the promising potential for massive parallelism. However, the need for explicit data exchanges between processors to communicate has become the major system bottleneck. Pipelined data parallel algorithms reduce the effect of communication overhead by regulating data flows in the system so that data can be processed and transmitted between processors in a pipelined fashion.

### An Example

A pipelined data parallel algorithm for matrix multiplication is shown in Figure 1 (Ref. 4). Assume that the underlying multiprocessor supports an interprocessor connection of a mesh. Matrices $A$ and $B$ are partitioned along columns and rows. The host first loads submatrices of $A$ into the corresponding processors in the mesh (see Figure 1). Then, submatrices of $B$ are piped into the mesh from the host. During each iteration, the following operations are performed in the processors:

(1) receive a submatrix of $B$ from the north;
(2) send the submatrix to the south;
(3) perform the submatrix multiplication, $C_{ik}^{(j)} \leftarrow A_{ij} \times B_{jk}$;
(4) receive a $C$ submatrix from the west;
(5) add the received $C$ submatrix to the result in (3);
(6) send the resultant $C$ submatrix to the east.

Through processor-level pipelining, it can be seen that the activities in the processors are data-driven and highly overlapped. In addition, the communications between processors are accomplished by regular and local data flows. These properties make the resultant pipelined data parallel algorithm very efficient.

Pipelined data parallelism is a very general concept for algorithm design. To apply this concept to general applications, it is necessary to have a systematic procedure of designing this kind of algorithms. Major theme of this paper is thus such a systematic procedure. The basic technique used is the grouping of loop instances.

### Organization of the Paper

A general procedure for designing pipelined data parallel algorithms is first outlined in Section 2. Then, the grouping problem is defined in Section 3, followed by various grouping techniques in Section 4. Our conclusion is given in Section 5.

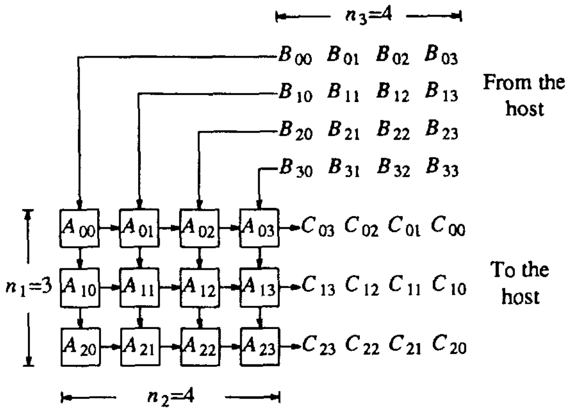

$n_3$=4
$B_{00}$ $B_{01}$ $B_{02}$ $B_{03}$
$B_{10}$ $B_{11}$ $B_{12}$ $B_{13}$
$B_{20}$ $B_{21}$ $B_{22}$ $B_{23}$
$B_{30}$ $B_{31}$ $B_{32}$ $B_{33}$
From the host
$A_{00}$ $A_{01}$ $A_{02}$ $A_{03}$
$C_{03}$ $C_{02}$ $C_{01}$ $C_{00}$
$n_1$=3
$A_{10}$ $A_{11}$ $A_{12}$ $A_{13}$
$C_{13}$ $C_{12}$ $C_{11}$ $C_{10}$
To the host
$A_{20}$ $A_{21}$ $A_{22}$ $A_{23}$
$C_{23}$ $C_{22}$ $C_{21}$ $C_{20}$
$n_2$=4

**Figure 1.** Data flows in pipelined matrix multiplication

CH2649-2/89/0000/0653$01.00 © 1988 IEEE

## A SYSTEMATIC DESIGN PROCEDURE

In this section a systematic design procedure is outlined. Due to the similarity of pipelined data parallelism and systolic arrays, initial stages of our procedure are identical to those used in synthesizing systolic arrays (Ref. 7). However, additional stages are required to control the granularity of the algorithm in order to balance the communication overhead on a multicomputer.

A nested-loop program is ***shift-invariant*** if data dependencies between loops do not change with the loop indices. The typical loop for matrix multiplication is shift-invariant:

```
for $i \leftarrow 0$ to 3 do
    for $j \leftarrow 0$ to 3 do
        for $k \leftarrow 0$ to 3 do
            $c_{ij} \leftarrow c_{ij} + a_{ik}b_{kj}$;
```

Using the above program as an example, the design procedure is outlined as follows:

### (1) Transform all variables into pipelined variables

The restructuring involves eliminating broadcast effects and enforcing the single-assignment rule (Ref. 6). After restructuring, the matrix multiplication loop looks like the following, with necessary initialization omitted:

```
for $i \leftarrow 0$ to 3 do
    for $j \leftarrow 0$ to 3 do
        for $k \leftarrow 0$ to 3 do
            $a_{ijk} \leftarrow a_{i\,j-1,k}$ ;
            $b_{ijk} \leftarrow b_{i-1,j,k}$ ;
            $c_{ijk} \leftarrow c_{i,j,k-1} + a_{ijk}b_{ijk}$ ;
```

### (2) Derive the computational structure

If the nested-loop program has $n$ levels, then the computational structure, $Q$, of the program is a directed graph on an $n$-dimensional Cartesian space. Each vertex in $Q$ represents one loop instance and has a coordinate $(c_0, \ldots, c_{n-1})$ if the corresponding loop instance has a loop index $(c_0, \ldots, c_{n-1})$.

There is an arc (*dependence vectors*) from one vertex $v_i$ to the other $v_j$ if the loop corresponding to $v_j$ references a variable which is generated in the loop corresponding to $v_i$. Note that, for a shift-invariant nested loop, all vertices in the corresponding computational structure have the same set of dependence vectors. It follows that such a computational structure $Q$ can be defined by the two-tuple $(V, D)$, where $V$ denotes the set of all vertices in $Q$ and $D$ is the set of dependence vectors.

The computational structure corresponding to the restructured matrix multiplication program is shown in Figure 2.

### (3) Group or project vertices in $Q$

In conventional systolic array synthesis techniques, a space-time projection is sought at this step to produce systolic effects (Ref. 7). All vertices in $Q$ are projected along a particular direction which represents the time-axis, i.e., the progress of time (Ref. 6). However, projection is just a special case of *grouping*. In grouping adjacent vertices in $Q$ are merged together to form larger vertices. By controlling the size of the groups, we can control the granularity of the algorithm.

In Figure 2, the grouping along $j$ with size 4 (i.e., a projection) and along $i$ and $k$ with size 2 respectively will result in a *contracted structure* as shown in Figure 3. As will be shown later, for many computational structures, certain grouping schemes will introduce extra dependencies between the groups. This implies extra communication in the resultant algorithm. Details of the grouping will be discussed in Section 3 and 4.

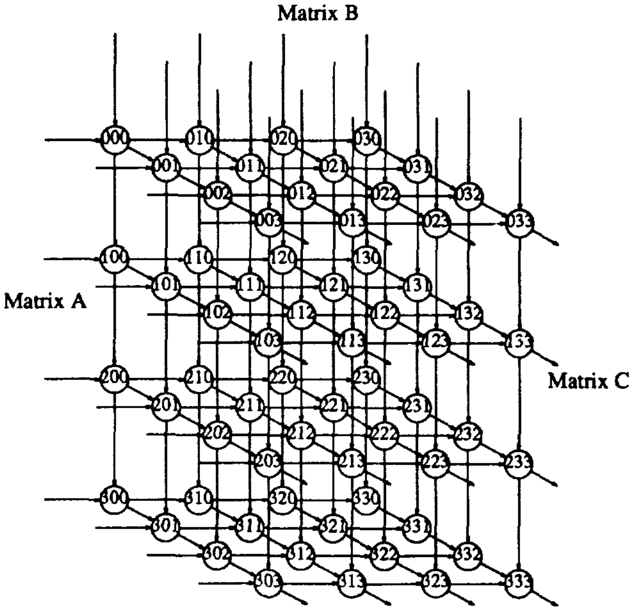


**Figure 2.** The computational structure of matrix multiplication

### (4) Obtain node programs

In Figure 3, if one group is allocated to one processor in a multicomputer, then the processor at location $(i', k')$, where $0 \le i', k' \le 1$, will execute the following program:

```
$l \leftarrow 0$ to 3 by 2 do
    /* communicate with neighboring processors */
    for $j \leftarrow l$ to $l$+1 do
        for $i \leftarrow 2i'$ to $2(i'+1)$ do
            for $k \leftarrow 2k'$ to $2(k'+1)$ do
```

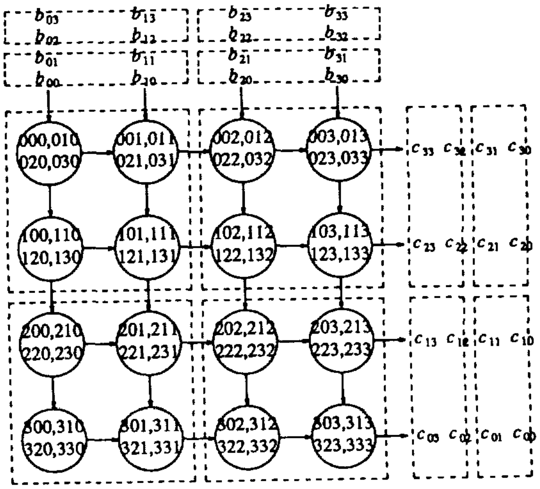


**Figure 3.** The contracted structure of matrix multiplication

```
$a_{ijk} \leftarrow a_{i\,j-1,k}$ ;
$b_{ijk} \leftarrow b_{i-1\,j\,k}$ ;
$c_{ijk} \leftarrow c_{i\,j\,k-1} + a_{ijk}b_{ijk}$ ;
```

In practice, the exact values of ***design parameters***, such as the size of groups, are unknown at this step. Thus, the constants introduced in the above program should be replaced by variables which correspond to the design parameters.

### (5) Determine design parameters

The design parameters are determined by taking into account input data sizes and machine characteristics. Optimal parameters are obtained by balancing the computation with the communication. An accurate estimation will require an accurate analytic model such as the one introduced in (Ref. 4).

## THE GROUPING PROBLEM

Let $Z$, $I$, and $I^+$, denote the set of integers, non-negative integers, and positive integers, respectively. The input to the grouping problem is a computational structure $Q(V, D)$ obtained from a shift-invariant nested-loop program.

Only *acyclic* computational structures are considered. In other words, for any subset of dependence vectors, $\mathbf{d}_0, \ldots, \mathbf{d}_{l-1} \in D$, if there exist $a_0, \ldots, a_{l-1} \in I$, such that

$$a_0\mathbf{d}_0 + a_1\mathbf{d}_1 + \cdots + a_{l-1}\mathbf{d}_{l-1} = 0$$

then $a_0 = \ldots = a_{l-1} = 0$.

**<Definition 1>**

The *grouping* $G_{\mathbf{d},r}(Q)$ of a computation structure $Q$ along a direction $\mathbf{d}$ of size $r$ is to partition all vertices in $Q$ into disjoint subsets, $P_0, \ldots, P_{k-1}$, such that

(1) $|P_0| = \ldots = |P_{k-1}| = r$

(2) For each subset $P_i$, $0 \le i \le k-1$, there exists an ordering, $(\mathrm{v}_0, \ldots, \mathrm{v}_{r-1})$, for all vertices in $P_i$ such that $\mathrm{v}_{j+1} - \mathrm{v}_j = \mathbf{d}$, $0 \le j \le r-2$.

Each subset $P_i$ is called a *group* of $G_{\mathbf{d},r}(Q)$. The first vertex, $\mathrm{v}_0$, in the ordering above is called the *base vertex* of the group.

□

Note that a grouping might not divide all vertices in a computational structure evenly. In this case, we can add dummy vertices at the boundary to make it even, or include those extra vertices into boundary groups. A group $P_i$ is *dependent* on another group $P_j$ along $\mathbf{d}$ if there are vertices $\mathrm{v}_i \in P_i$ and $\mathrm{v}_j \in P_j$ such that $\mathrm{v}_i - \mathrm{v}_j = \mathbf{d}$.

**<Definition 2>**

The *contracted structure*, $Q'$, with respect to the grouping $G_{\mathbf{d},r}(Q)$ is a directed graph, where

(1) Each vertex in $Q'$ corresponds to one group in $G_{\mathbf{d},r}(Q)$;

(2) If $P_i$ and $P_j$ are the groups in $Q$ corresponding to the vertices $\mathrm{v}_i$ and $\mathrm{v}_j$ in $Q'$, respectively, then there is an arc from $\mathrm{v}_i$ to $\mathrm{v}_j$, if $P_j$ depends on $P_i$.

$G_{\mathbf{d},r}(Q)$ and, thus, $Q'(V', D')$ are *dependence-preserving* if $Q'$ is an acyclic computational structure with $|D'| \le |D|$.

□

A dependence-preserving grouping will not introduce extra dependencies among the resultant groups. This is particularly important in multicomputers with limited connectivity and non-negligible communication overhead. The grouping problem consider in this paper can now be stated as follows:

**<Grouping problem>**

Given a computational structure $Q$, determine the groupings of $Q$ which will result in dependence-preserving contracted structures.

□

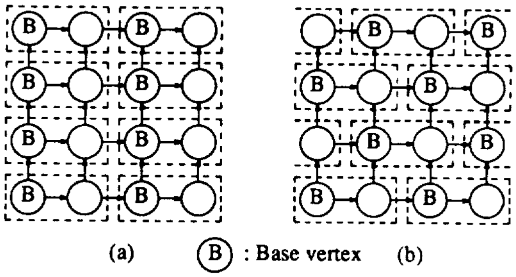


**Figure 4.** Grouping with two dependence vectors

## PROPERTIES OF GROUPING

In the following discussion, we will concentrate on computational structures in two-dimensional spaces. For higher dimensional spaces, the results obtained here can be extended.

Consider computational structures with one or two dependence vectors. There exist dependence-preserving contracted structures if the grouping is along the dependence vectors (see Figure 4(a)). This is because the dependence vectors are independent in acyclic computational structures. Thus, the resultant groups will not interfere with each other and introduce extra dependencies. Nevertheless, if the base vertices are not properly chosen, then extra dependencies will still be created (see Figure 4(b)).

Suppose the grouping is along a direction different from the dependence vectors of the computational structure. Then, this situation is equivalent to a computational structure, $Q(V, D)$, with three dependence vectors, $D = \{\mathbf{d}_0, \mathbf{d}_1, \mathbf{d}_2\}$. Since $Q$ is an acyclic two-dimensional structure, we can assume that

$$a_2\mathbf{d}_2 = a_0\mathbf{d}_0 + a_1\mathbf{d}_1 \qquad (1)$$

where $a_0$, $a_1$, and $a_2$ are the smallest positive integers to satisfy Equation 1. A typical example of such a computational structure is shown in Figure 5(a), where $D = \{[1,0], [1,1], [0,1]\}$ and $[1,1] = [1,0] + [0,1]$.

Consider the grouping along $\mathbf{d}_2$ first. In Figure 5(a), a possible grouping along $\mathbf{d}_2 = [1,1]$ is depicted in dashed boxes. Figure 5(b) shows the corresponding contracted structure with the group ids indicated in the circles. It can be seen that an extra dependence vector, $[0,-1]$, is introduced. Thus, the contracted structure is not dependence-preserving. In fact, we have the following theorem:

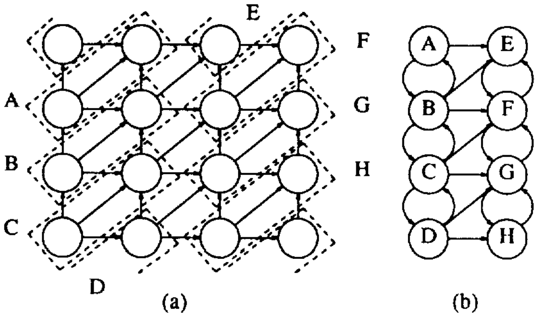


**Figure 5.** Grouping which is non-dependence-preserving

**<Theorem 1>**

Let $Q(V,D)$ be a computational structure. If there exist three dependence vectors, $\mathbf{d}_0$, $\mathbf{d}_1$, $\mathbf{d}_2 \in D$, where $\mathbf{d}_0$, $\mathbf{d}_1$, and $\mathbf{d}_2$ satisfy Equation 1, then the grouping along $\mathbf{d}_2$ with size $r > a_2$ is not dependence-preserving.

□

The proof is given in Ref. 4 and is omitted here. In Figure 5, since $a_2=1$, it is impossible to find a dependence-preserving grouping along [1,1].

Theorem 1 states the conditions when a grouping is not dependence-preserving. A more constructive way is to study under what conditions a dependence-preserving structure will be generated. Given an acyclic computational structure $Q(V, D)$ with the dependence set $D=\{\mathbf{d}_0, \ldots, \mathbf{d}_{m-1}\}$, where $m \geq 3$, we can always find (from linear algebra) two vectors, say, $\mathbf{d}_0$ and $\mathbf{d}_1$, such that

$$c_i \mathbf{d}_i = a_i \mathbf{d}_0 + b_i \mathbf{d}_1 \qquad (2)$$

where $a_i, b_i, c_i \in I^+$, $2 \leq i < m$. Define

$$a_{\max} = \underset{2 \leq i < m}{\mathrm{Max}} \left\{ \left\lceil \frac{a_i}{c_i} \right\rceil \right\} \qquad b_{\max} = \underset{2 \leq i < m}{\mathrm{Max}} \left\{ \left\lceil \frac{b_i}{c_i} \right\rceil \right\}$$

The "range of influence" of a vertex v in $Q$ can be defined as the vertex set $X(\mathbf{v})$, where

$$X(\mathbf{v}) = \{\mathbf{w} \mid \mathbf{w} \in V,\ \mathbf{w}=\mathbf{v}+x a_{\max}\mathbf{d}_0+y b_{\max}\mathbf{d}_1,\ 0 \leq x, y \leq 1\}$$

Then, any vertex $\mathbf{u} \in V$ which is dependent on v is in $X(\mathbf{v})$.

**<Theorem 2>**

Let $Q(V, D)$ be a computational structure with $m$ dependence vectors, where $m \geq 3$, and the dependence vectors satisfy Equation 2. Then, the grouping along $\mathbf{d}_0$ with size $r_0 \geq a_{\max}$ and along $\mathbf{d}_1$ with size $r_1 \geq b_{\max}$ is dependence-preserving.

□

Again, the proof can be found in Ref. 4. The implications of Theorem 2 are: (1) as long as the group size is large enough, there always exist dependence-preserving groupings along $\mathbf{d}_0$ and $\mathbf{d}_1$, and (2) the resultant contracted structure is a *universal planner array* (see Figure 6). A universal planner array is the most general systolic arrays in two-dimensional spaces (Ref. 6). It follows that, in the final implementation, each processor only has to communicate with at most three other processors. Note also that $\mathbf{d}_0$ and $\mathbf{d}_1$ can be any vectors which are not necessary in $D$ as long as Equation 2 is satisfied.

## CONCLUSION

We have presented in this paper a systematic procedure for designing pipelined data parallel algorithms from shift-invariant nested loops. This procedure concentrates on grouping loops in the original program so as to reduce the number of communicating processors, control the granularity, and increase the degree of pipelining.

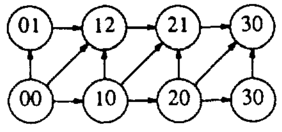


**Figure 6.** A universal plannar array

Results presented in this paper are preliminary. A mathematic foundation is needed to abstract the concept of grouping so that results presented here can be applied to higher dimensional spaces. Also, the relationship between projection and grouping needs to be further probed, which may in turn assist in designing more efficient systolic arrays. Ultimately, we could expect results obtain in this research will contribute to intelligent compilers for multicomputers, which parallelize programs (semi-)automatically.

## REFERENCE

1. W.L. Athas, C.L. Seitz, "Multicomputers: Message-Passing Concurrent Computers," *IEEE Computers*, Aug. 1988, pp. 9-24.
2. W.D. Hillis, G.J. Steele, Jr., "Data Parallel Algorithms," *Comm. ACM*, Dec. 1986, pp. 1170-1183.
3. C.T. King, "Pipelined Data Parallel Algorithms — Concept, Design, and Modeling," *Ph.D. Dissertation*, Department of Computer Science, Michigan State University, 1988.
4. King, C.T., Chou, W.H., Ni, L.M., "Pipelined Data Parallel Algorithms — Concept and Modeling," *Proc. of the 1988 ACM Int'l Conf. on Supercomputing*, July, 1988.
5. H.T. Kung, C.E. Leiserson, "Systolic Arrays (for VLSI)," *Sparse Matrix Proc.*, 1978, pp. 32-63.
6. W.L. Miranker, A. Winkler, "Spacetime Representations of Computational Structures," *Computing*, 1984, pp. 93-114.
7. D.I. Moldovan, J.A.B. Fortes, "Partitioning and Mapping Algorithms into Fixed-Size Systolic Array," *IEEE Trans. on Computers*, Jan. 1986, pp. 1-12.

# CMS: An Integrated Simulation Environment

J. Leslie Walker and Abbas Birjandi

College of Computer Science, Northeastern University, Boston Massachusetts

## Abstract

The Connection Machine is a massively parallel architecture designed for general computation. This paper describes a simulation environment which allows one to execute programs written for a connection machine. The CMS simulation environment provides several tools for empirical analysis of these parallel algorithms. First, it provides performance statistics on the simulated connection machine as it runs different user programs. These statistics compiled from the simulated execution of a set of algorithms provide a measure by which one may arrive at the optimal solution from that set. Secondly, The behavior of the simulated environment is governed by configuration parameters which may be altered to find the optimal connection machine configuration in which to run an algorithm. The CMS simulation environment runs on the Sun Workstation and uses the windowing and mouse interfaces provided by the SunView$^{TM}$ integrated application environment. Different windows allow the user to view different parts of the simulated connection machine at the same time. This paper also includes a discussion of the internals of the simulator and the reasoning behind the methods used. By examining these details one can learn more about the hardware implementation of the connection machine.

## Overview

This paper provides an introduction to CMS, a simulator for a connection machine [Hillis 86]. The goal of the simulation is to provide an environment in which to run connection machine algorithms and gather detailed statistics about their execution. The simulator runs in an integrated environment which provides windowing and support for a mouse. The simulator software and integrated display simplifies the presentation of the connection machine hardware enough to make it understandable to the novice user while keeping it accurate enough to represent the connection machine hardware.

The connection machine is a massively parallel computer architecture. The granularity of the processors which compose a connection machine is very small-one bit. The architecture works most efficiently on large, uniform problems with very small granularity. Some examples are database query searches and kernel convolutions.

The algorithms that run on a connection machine need not be vectorized to run efficiently. In fact, the processors may be considered to be arranged with complete adjacency, meaning that every processor is directly adjacent to every other processor. This means that trees, graphs, and other geometries may be implemented on the connection machine as well as vectors.

## Connection Machine Architecture

### Interfacing

The connection machine is not equipped to perform system tasks such as device control and data acquisition. For this reason it must interface to a host computer which provides these types of services. The host has full access to each processor's memory and also provides the instruction stream which the individual processors will execute. The host usually will provide an initial state for the connection machine by depositing the appropriate data into the memory of each of the processors. The host may also retrieve results from the processors' memories during execution or afterwards.

### Cells

Processors in the Connection Machine are referred to as *processing cells*. Each processing cell is able to independently execute the basic building block operations of the whole machine. These processing cells are identical and each contains it's own memory, registers, and ALU. Each of these components as well as all of the

Abbas Birjandi is Assistant Professor of Computer Science at Northeastern University in Boston and Les Walker is an employee at Charles River Analytics in Cambridge Massachussetts.

CH2649-2/89/0000/0657$01.00 © 1988 IEEE

corresponding data paths are uniformaly one bit wide. The single-bit registers in a cell are referred to as *cell flags*. *General Purpose* cell flags may be set, cleared, or read by the cell without side effects. These are used to store a single bit of information. *Special* cell flags have side effects to being set, cleared, or read. These flags are the means by which a cell communicates with other cells and the routers.

The connection machine is a SIMD (Single-Instruction Multiple-Data) multiple processor architecture. The instruction is broadcast to each cell via the *global instruction bus*.

### The Network

**Local Areas** The processing cells in the connection machine are grouped into uniform sets which we shall call *local areas*. The cells in a local area share several communication structures:

- **daisy chain** The output of each cell's ALU is linked with a special purpose register on another cell in such a way that a circular chain is formed. Data may be passed through this chain in serial fashion.
- **NEWS** Similarly to the daisy chain, the output of each cell's ALU is linked with a special purpose register in four other cells so that the cells form a square with each cell able to communicate directly with cells in four directions.
- **router** A special purpose register of each cell in a local area is linked with the router for that local area. The routers form a network that establishes the complete adjacency of the architecture.

In this simulation cells are refered to in relation to thier local area. Each cell has a number in it's local area and that local area has a router number. So, each cell is refered to as:

```
<cell,router>
```

**Network Topology** It is obvious that with groups of cells being served by routers, all cells are not actually directly adjacent. Complete adjacency is established in the architecture by the fact that each cell can communicate indirectly with any other cell in the connection machine ***without cells being a part of the communication process except at the terminal points.*** In other words cells only see messages that they send or are sent to them – a cell will never recieve a message that it must pass on. The task of transmitting messages is handled by the network. The network is composed of routers and data paths. Routers are described below. The data paths are bi-directional links between routers.

The routers are connected to each other by bi-directional data paths which allow routers to transmit messages to each other in both directions. The adjacencies made by these data paths form a boolean n-cube in which no router is further than $\lg n$ data paths from any other router. Messages are addressed to a particular cell within the local area of a particular router. The cell is addressed by a cell number (*cell address*), while the router is referred to by it's location on the n-cube relative to the router that is sending the message (*router address*). The router address contains all the information necessary to route the message at any point in it's transmission.

Each router has $\lg n$ data paths which establish it's adjacency with $\lg n$ other routers on the network. These data paths may be enumerated so that path $i$ on any given router is known as path $i$ to the router on the other end. In this way if two routers both transmit a message along path $i$ they exchange messages with each other. The numbers that are given to the paths are called the path's *dimension* and all the routers in the network transmit on only one dimension at a time. When all of the routers transmit on one dimension it is referred to as a *dimension cycle*. The network executes dimension cycles for each dimension consecutively beginning with the first:

$$1\ 2\ \ldots\ \lg n$$

Once the network has executed a dimension cycle for all $\lg n$ dimensions it starts over again at the first dimension. $\lg n$ dimension cycles make up one *petite cycle* in which dimension cycles are executed for all of the dimensions:

$$\left.\begin{array}{c} 1 \\ 2 \\ \vdots \\ \lg n \end{array}\right\} = \textit{petite cycle}$$

**Routers** The routers are composed mainly of buffers and termination points for data paths. Since the data paths are of limited capacity, the router saves it's messages in buffers. Messages have priorities and addresses that regulate their transmission. Addresses tell which data path a message needs to travel on and priorities are used to resolve the collision that occurs when more than one buffered message needs to travel on a single message path at the same time.

## The Simulator

We have constructed a piece of software that simulates the connection machine described above. This software simulates the hardware from the host machine interface down to the level of each bit of the connection machine. Cells, routers, and data paths are all visually represented.

The simulator runs as a separate process that recieves instructions and memory requests from a user application. The user application starts the simulator and establishes several lines of communication with it. These communication lines are analogous to the interface with the host machine and the user application is analogous to the process running on that host machine. The communication lines allow instructions to be sent to the simulator and allow access to the cell memory one byte at a time. The disk DMA path is not supported.

The network is composed of 256 routers each having 16 local processing cells. Each cell has 2k bits of local memory.

## Operating Environment

CMS is implemented under Unix$^{TM}$ using the SUNView$^{TM}$ integrated application environment. All displays that are called *control panels* are implemented using the SUNView$^{TM}$ panel objects and SUNView$^{TM}$ canvases. It is sufficient for the user to familiarize themselves with the operation of these objects to use most of the mouse commands in the simulator.

## User Interface

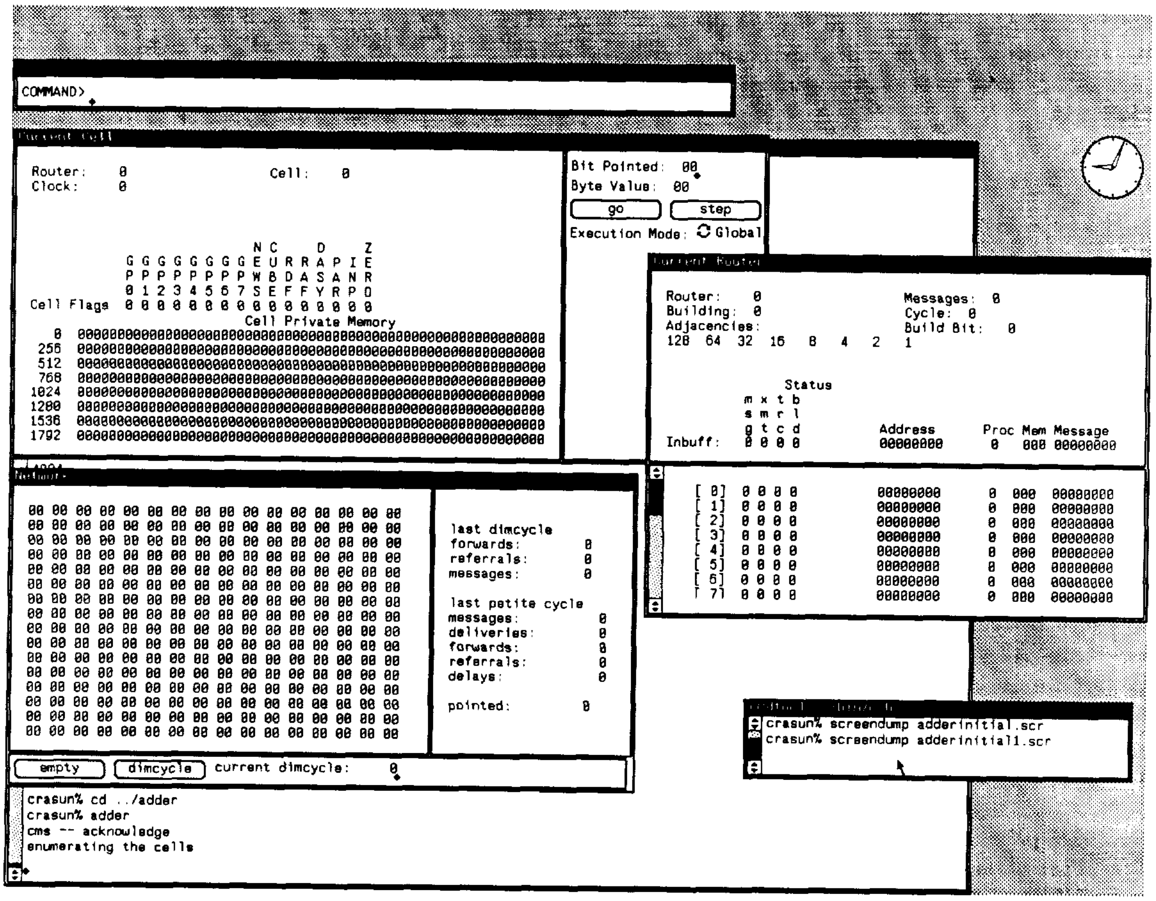
The simulator provides an interface through SUNView$^{TM}$ which behaves much like a traditional machine level debugger.

### Windows

The simulator visually represents the components of the simulated connection machine through several windows. These windows may be overlapped in the SUNView$^{TM}$ environment as shown in Figure 2. There are separate windows for:

- Cells
- Routers
- Complete Network
- Global Instruction Bus

These windows contain control panels which provide access to the commands via the mouse. The data in the windows is updated in real-time. For example, when a



```
COMMAND>
```

```
Current Cell

Router:    0                     Cell:    0
Clock:     0

                              N C     D     Z
            G G G G G G G G E U R R A P I E
            P P P P P P P P W B D A S A N R
            0 1 2 3 4 5 6 7 S E F F Y R P O
Cell Flags  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                       Cell Private Memory
   0  0000000000000000000000000000000000000000000000000000000000000000
 256  0000000000000000000000000000000000000000000000000000000000000000
 512  0000000000000000000000000000000000000000000000000000000000000000
 768  0000000000000000000000000000000000000000000000000000000000000000
1024  0000000000000000000000000000000000000000000000000000000000000000
1280  0000000000000000000000000000000000000000000000000000000000000000
1536  0000000000000000000000000000000000000000000000000000000000000000
1792  0000000000000000000000000000000000000000000000000000000000000000
```

```
Bit Pointed:  00
Byte Value:  00
```

```
Execution Mode: Global
```

```
Current Router

Router:     0                          Messages:  0
Building:   0                          Cycle:  0
Adjacencies:                           Build Bit:   0
128  64  32  16   8   4   2   1

              Status
           m x t b
           s m r l
           g t c d                Address        Proc Mem Message
Inbuff:    0 0 0 0                00000000        0   000 00000000

   [ 0]    0 0 0 0                00000000        0   000 00000000
   [ 1]    0 0 0 0                00000000        0   000 00000000
   [ 2]    0 0 0 0                00000000        0   000 00000000
   [ 3]    0 0 0 0                00000000        0   000 00000000
   [ 4]    0 0 0 0                00000000        0   000 00000000
   [ 5]    0 0 0 0                00000000        0   000 00000000
   [ 6]    0 0 0 0                00000000        0   000 00000000
   [ 7]    0 0 0 0                00000000        0   000 00000000
```

```
Network

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     last dimcycle
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     forwards:        0
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     referrals:       0
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     messages:        0
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     last petite cycle
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     messages:          0
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     deliveries:        0
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     forwards:          0
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     referrals:         0
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     delays:            0
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     pointed:         0
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
current dimcycle:     0
```

```
crasun% screendump adderinitial.scr
crasun% screendump adderinitial1.scr
```

```
crasun% cd ../adder
crasun% adder
cms -- acknowledge
enumerating the cells
```

Figure 2: A sample CMS screen

memory request comes through from the host to store a value in memory and the cell which owns that memory is shown, then the window which shows that cell will be updated with the new memory value.

### Commands

The simulator commands are meant to allow it to fuction similarly to a traditional machine language debugger. There are commands to alter the contents of cell memory and cell flags, execute instructions, and even alter the behavior of the machine. All commands may be entered as text in a command line format, and some are accessable via pushbuttons and control panels. There is also an indirect command file capability for automated execution.

The area of the windows which displays the data is active in the sense that clicking the mouse over certain areas of the display will cause commands to be executed. Some examples:

- Clicking the mouse over a cell flag will cause the flag to change in value to a '0' or '1'.
- Clicking the mouse over one of the adjacencies in the router window will cause the window to switch from displaying the old router to display the router adjacent to it on the dimension that was selected.

## Application Interface

In the actual connection machine hardware, a conventional computer is required to host the connection machine hardware. The host provides a front end for the connection machine's specialized computing power. It is essential for the simulator to have this design also so that the programmer has a feel for the abstraction provided by the interface between the connection machine and the host. The connection between the host and the machine is called the microcontroller.

The microcontroller functions as both an interface to the connection machine hardware, and a low-level abstraction of the connection machine's functionality. In general one *macro-instruction* is translated by the microcontroller into several connection machine instructions called ***nano-instructions***. Because one macro-instruction can produce many nano-instructions the microcontroller must buffer the nano-instructions in a FIFO buffer.

The simulator provides a microcontroller as well. This microcontroller passes nano-instructions which are produced by the application through a pipe to the simulator which will read them in one at a time and execute them. The simulator in turn responds through another pipe for each instruction, thus providing the necessary syncronization. The microcontroller is divided into 2 modules: the application microcontroller (used by the host) and the simulator microcontroller (used by the simulator).

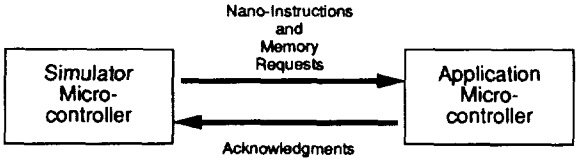

Nano-Instructions and Memory Requests
Simulator Micro-controller
Application Micro-controller
Acknowledgments

Figure 3: Diagram of relationship between microcontrollers.

### The Application Microcontroller

The application microcontroller provides access to the connection machine from the user application program. The goal in it's design was for the application to appear to be talking to an actual hardware device whose memory was attached to the bus. However, to be realistic the microcontroller must also provide macro-instructions and translate these into nano-instructions for the machine to execute. The decision was made to do this on the application side so that the user is able to experiment with his own high-level abstractions of the connection machine operations.

### The Simulator Microcontroller

The microcontroller on the simulator is not under the same burden of having to appear to be talking to hardware. It is implemented as a separate module of the simulator software. There is no direct communication between the programmer and the simulator microcontroller, except via the application microcontroller.

## References

[Hillis 86] Daniel W. Hillis. *The Connection Machine*. M.I.T. Press, Cambridge, Mass.

[Sun 86] Sun Microsystems. *SunView Programmers Guide*. Sun Microsystems, Inc., Mountain View, Ca.

[Think 86-14] *Introduction to Data Level Parallelism,* Thinking Machines Corp., Technical Report Series, TR86-14, April 1986.

[Think 87-4] *Connection Machine Model CM-2 Technical Summary,* Thinking Machines Corp., Technical Report Series, HA87-4, April 1987.

# ADDITIONAL PAPERS

# SIGNAL PROCESSING WITH NODAL NETWORKS ON A SIMD MASSIVELY PARALLEL PROCESSOR

William I. Lundgren

GE Aerospace Advanced Technology Laboratories
ATL Building, Moorestown, NJ 08057

## Abstract

The goal of the work reported in this paper is to develop methods for using a large scale SIMD parallel processor such as the Connection Machine for signal processing. The primary focus of the work reported in this paper is to develop nodal network methodologies that can be used to effectively implement algorithms that range from signal processing to discrete logic systems. One benefit of nodal network methodologies is their inherent SIMD nature. As a result their implementations are closely related to the architecture of SIMD parallel processors like the Connection Machine. Also, the homogeneity provided by such systems allows the uniform integration of signal processing and discrete logic systems. As a first step we have implemented two versions of an algorithm to track formants in speech. The first implementation uses data parallel coding techniques. The second implementation uses a nodal network. The algorithm contains logic that, at every frequency/time point in a spectrogram, chooses between several filters to find the filter that best matches linear energy structure at that point. The choice of filter at each point is determined on the basis of information in adjacent points. The nodal network implementation of the algorithm uses only two node types, a fuzzy AND and a fuzzy OR (henceforth referred to as AND nodes and OR nodes respectively). The connections between nodes can be either non-inverting or inverting. The inverting effectively produces a NOT. The algorithm relies on the parameters associated with each node and connectivity between the nodes to simulate the original algorithm. The result is a nodal network "programmed" to identify formants in a spectrogram. The two implementations are comparable in performance and speed of execution. The conclusion is to continue the investigation of this type of nodal network.

## 1.0 Introduction

The long term goal of this work is to develop nodal network techniques for large vocabulary speaker independent speech recognition. In speech recognition, the biggest problems are that small changes in the acoustic signal can change the word recognized, and conversely, acoustically different speech signals may be recognized as the same word. Systems that use data reduction to decrease the size of the input vector smooth the signal and remove some distinctive features of the speech. Large input vectors are necessary if the input is to contain enough information to make fine distinctions between acoustic inputs. Further, the classifier must be able to selectively focus on small or large amounts of acoustic information as required. This provides the capability to build a classifier that is a complex mapping between the acoustic input and the words recognized. Unfortunately these two characteristics, a large input vector and flexible use of the acoustic input, make the task of training very difficult.

Nodal networks provide the capacity and flexibility required. The network described below was designed to provide a means for dealing with large input vectors, complex classifiers and, eventually, automatic training. A high level diagram of the sought after system with training is shown in Figure 1. The system is divided into

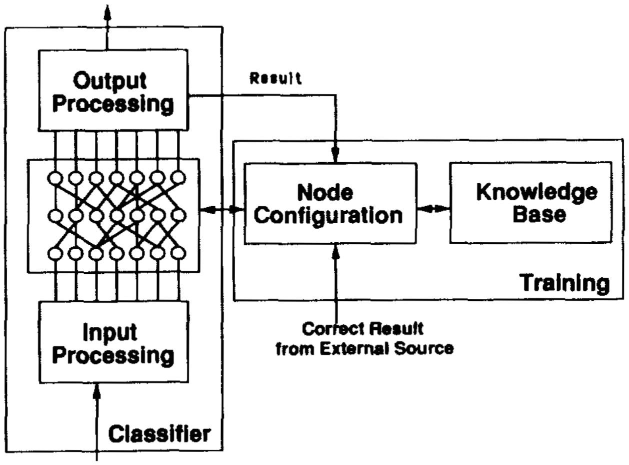


Figure 1. High Level Diagram of Future System.

a classifier and a trainer. The classifier consists of a large network of nodes, an input preprocessor and a post processor. The automatic training consists of a node configuration module and a knowledge base. Note that rather than applying the knowledge during the classification process the knowledge is used off line to train the network. The heart of the problem is to decide how the network configuration module should choose a network configuration to address a particular recognition task. For even the very simple network topology described below using a modest set of nodes (10 levels, 1000 nodes at each level) leads to a huge set of potential solutions. It is easy to establish a lower bound of 10 to the power of 60,000. The system described below does three things to help the node configuration module select a meaningful solution. It uses:

1) a knowledge base to guide the training of the network from the top down.
2) well established node configurations that can be used for specific types of tasks.
3) consistencies in the acoustic input to drive the training from the bottom up.

The knowledge makes it possible to train the network in spite of a small amount of training data compared to the size of the solution space.

This paper focuses on the development of nodal network configurations for processing data. After the configuration of nodes for various tasks are adequately understood, the effort will be shifted to the automatic training of the nodal networks. The approach to training proposed in this paper differs from that proposed by others (Refs. 1, 2]. Some basic ideas for the training of the nodal network are presented in the appendix of a technical report by Dave Graff and the author (Ref 3). The methodology discussed there is manual training. That methodology was developed with the intent of eventually automating the procedure and introducing a knowledge base to produce an automatic training capability.

## 2.0 The Nodal Network

The nodal network is designed to provide the following two features:

1. Hierarchical processing of the data. The goal is to provide layers of information, each slightly more abstracted from the data. The hierarchical nature of the system provides information at each level on which future training can be based. Hence, during the course of training an instance of the nodal networks, one would have an increasingly complete set of fundamental nodes. As the set of nodes becomes more complete the training will become easier.

2. Uniform integration of "symbolic" and numerical processing. Each node in the system can be thought of as a numerical value and symbol pair. For example, the input nodes described below are a Power Spectral Density (PSD) of the speech signal. The node can be labeled "Spectral Power at Δf * n hz." Δf is the frequency resolution of the PSD and n is the number of the PSD nodes. Higher in the network the node values approximate a likelihood. For example, one of the nodes described below will be "band of spectral power at Δf * n hz and angle k".

The intent is to develop a knowledge base and a set of nodes that are directly related. This will insure that the node configuration module will be able to use the knowledge base to assist in the configuration of the nodal network. Further, the hierarchical nature of the network will provide knowledge that is increasingly abstracted from the acoustic data and more directly related to the words. The homogeneous representation of knowledge at every level of the system makes feedback between any levels in the system possible. It also provides for homogeneous access to all levels of information when making a decision.

### 2.2 Description of the Nodal Network

A nodal network can be fully described by specifying the node layout, the interconnectivity of the nodes and the definition of how the nodes process their inputs. An example of the node layout and interconnectivity is shown in Figure 2. There are multiple levels of nodes (10 or 20 levels is reasonable) and in the current system there are two inputs to each node. Past implementations by the author have used four or five inputs to each node. Any network using two or more inputs can be implemented with a networks using only two inputs to each node. At this point it is not clear that there is any advantage to using a larger number of inputs per node. Feedback is a potentially powerful device not used in the current network. Figure 2 shows one example of feedback. The role of feedback and stability of networks with feedback will be the subject of future work.

The nodes are simple in behavior and similar to some of the other nodes used in the neural network community (Ref s.1, 2). The inputs are each transformed with a linear equation. The slope and the intercept used are unique for each input of each node in the network. The result is then thresholded to maintain the value

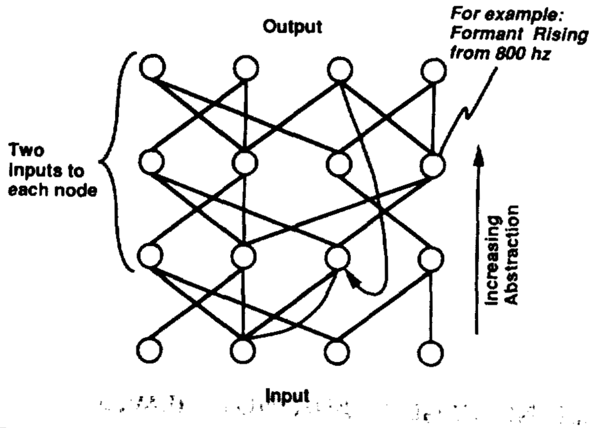


Figure 2. Node Layout and Interconnectivity.

between 1.0 and 0.0. The inputs can be "inverted", i.e. after transformation and thresholding the result can be subtracted from 1.0 to produce a NOT of the input. The two inputs to a node are either summed and divided by 2 (to insure the input remains between 1.0 and 0.0) or they are multiplied. The summation produces a node that behaves like an ORing of the inputs. The multiplication produces a node that behaves like an ANDing of the inputs. Two examples of nodes are shown in Figure 3.

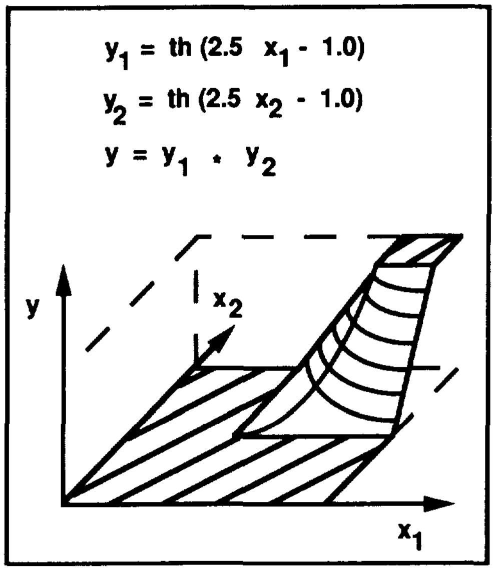

$y_1 = \mathrm{th}\,(2.5\ x_1 - 1.0)$
$y_2 = \mathrm{th}\,(2.5\ x_2 - 1.0)$
$y = y_1 * y_2$
y
$x_2$
$x_1$

Figure 3a. AND Node.

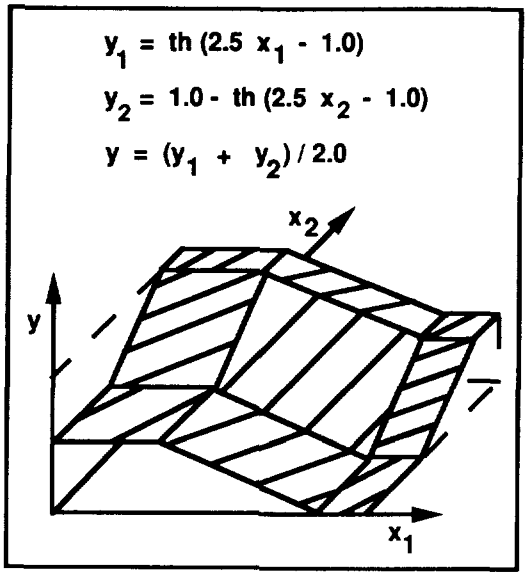

$y_1 = \mathrm{th}\,(2.5\ x_1 - 1.0)$
$y_2 = 1.0 - \mathrm{th}\,(2.5\ x_2 - 1.0)$
$y = (y_1 + y_2) / 2.0$
$x_2$
y
$x_1$

Figure 3b. OR Node.

3.0 Approach

The approach is centered on the use of a relatively simple application as a tool to explore programming of nodal networks. The application chosen for this study was recognition of formant peaks in a time-frequency-power representation of a speech signal. Two implementations of an algorithm to identify formant peaks were coded for comparison. The first used standard coding techniques with one processor assigned to each element in the time-frequency plane. The second implementation used a similar processing string, but was coded with a network of fuzzy AND and OR nodes. Each node in the network was assigned to a processor.

This approach was chosen because it illustrates: 1) The feasibility of using a nodal network to perform a signal processing task and 2) the design of a basic nodal network to perform some functions required by a signal processing task.

4.0 The Application

4.1 The Input Data

Formants are bands of energy in the spectral representation of speech. They are a result of the shape of the vocal tract and the associated resonances at the time the sound was produced. for this study, the time waveform of speech data was passed through an anti-aliasing filter with a cutoff at 4000 hz and was then sampled at a 10 kHz sampling rate. The analysis frames were hand aligned to the pitch periods in the time waveform (see Figure 4). This removed much of the temporal variation in the magnitude of the formants. (One of the next tasks will be to develop a spectral analysis network that automatically centers analysis frames on the pitch periods.) The analysis windows were hamming windows 128 samples long. The FFT produced a spectrum of 64 values ranging from DC to 4922 hz in 78.1 hz increments. An example of the formants used in this study are shown in Figure 5 (next page). The spectrogram shown is of an adult male speaker. The pitch period for this speaker averages about 10 msec.

The formants extend for a finite duration in the time direction and move up or down in frequency over time. The objective of the algorithms coded for this effort is to track those formants as they start, stop and move through time.

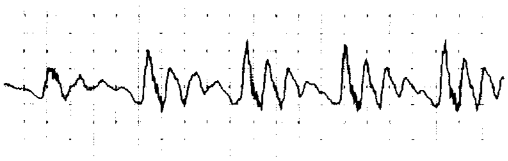


Figure 4. Segement of Speech Data. Each repetition of similar data is a pitch period.

4.2 Processing String - Conventional Algorithm

1) The first step is to convolve the image with 19 elongated kernels oriented at angles between -45 degrees and +45 degrees (0 degrees is aligned with the time axis). The idea of using elongated filters at different angles was first suggested by Dr. John Meckley (Ref 4). At each time-frequency point the kernel most closely aligned to the energy structure in that region is selected. The kernel is selected by choosing the convolution result that changes the most as the kernel is shifted along the frequency axis. If two or more results are the same size the angle nearest to 0 degrees is chosen. The value of that convolution becomes the new value at that pixel. The angle of orientation of the largest convolution result is also recorded. The result of this matched filtering is an enhancement of the formant peak energy relative to the other energy in the signal.

2) The new image is then rescaled at each point by thresholding with the average energy in nearby locations.

3) The data is then spread along the angle chosen in step one: a) The data is multiplied by 1/3 and added to the data samples which are two data points away along the chosen angle (both backward and forward along the angle). b) The data is then multiplied by 2/3 and added to the data in the adjoining points along the chosen angle.

This process, spreading the likelihood of being a peak along the angles selected in step one, would be equivalent to low pass filtering if all angles were aligned.

4) The final step is to again threshold the result of step four to produce a binary output image.

The implementation on the Connection Machine was accomplished by assigning one data point to each processor and simultaneously working on 128 spectrums of 64 data points each.

The rescaled data and the final results are shown in Figure 6.

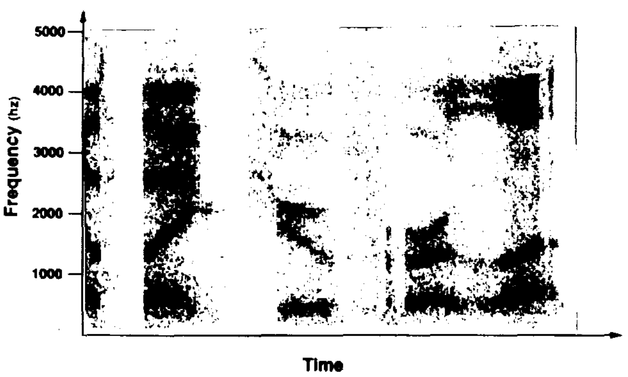


Figure 5. Spectrogram of Adult Male. Elongated dark areas are formants.

4.3 Processing String - Nodal Network Algorithm

The processing steps used in the nodal network are similar to those used in the conventional algorithm. The primary difference is a change in their sequencing.

1) The first step is low pass filtering (LPF) of the data along the time axis. This smooths out some of the irregularity in the spectrum that results from noise added to the formant structure.

2) A local average LPF spectrum is calculated based on 8 data points. Again this is along the frequency axis. This result is maintained independently of the LPF spectrum from step 1.

3) The LPF spectrum is then set to zero if it is less than the average in its area, to one if it is 1/8 or more bigger and otherwise scaled linearly between those two values. This rescales the whole spectrum relative to the average in the area around it.

4) The data is then reduced further by determining if each point is a peak in the modified energy along a given time slice.

5) The final step is to determine if there is a continuity of peaks along each of the possible angles.

The results at Steps 3, 4 and 5 are shown in Figure 7. For completeness, all node definitions are given in Table 1. The first three layers of the nodal network are diagrammed in Figure 8.

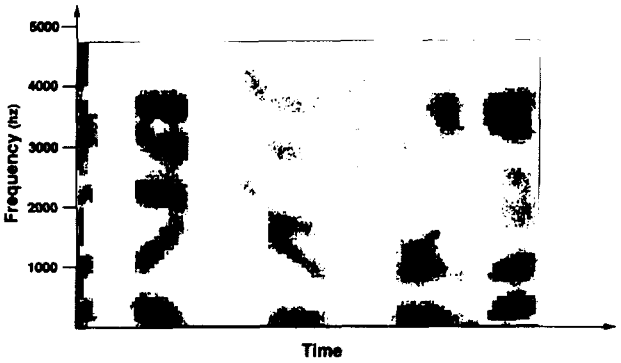


Figure 6a. Conventional Implementation - Rescaled Data.

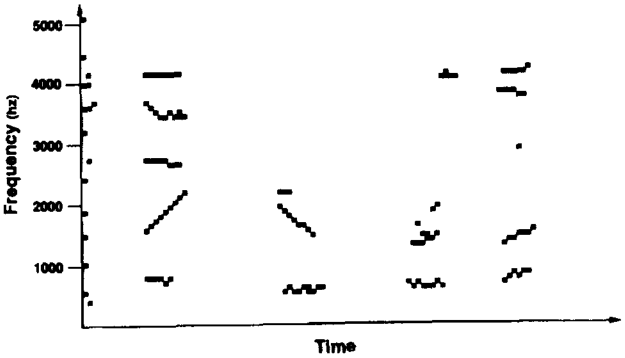


Figure 6b. Conventional Implementation - Finale Result.

ORIGINAL PAGE
BLACK AND WHITE PHOTOGRAPH

Explanation of table:

Each node Is repeated for every element (64 0f them) in the spectrum. The first array Index is the offset along the frequency axis. The second index is the offset along the time axis. Note that all time offsets are less than or equal to zero. This is because the "recent" history of a node is kept but future values are unavailable. If the time offset is 0 the second index is omitted. The name to the left of the equal sign is the name of the node. The two (or one) names to the right of the equal sign are the names of the input nodes. If a pair of numbers enclosed in braces follows the node name that Input is transformed using the first number as the slope and the second the intercept. If there Is no pair of numbers the slope is by default 1.0 and the intercept 0.0. If a NOT follows the node name (or the braces) the input is inverted.

New input data is inserted every time iteration.

```
    input[0]      INPUT
```

Low pass filter (LPF) the input along the frequency axis.

```
    lpf_1[0]     = input[0]              OR     input[1]
    lpf_2[0]     = lpf_1[0]              OR     lpf_1[-1]
    lpf_3[0]     = lpf_2[0]              OR     lpf_2[1]
    lpf_4[0]     = lpf_3[0]              OR     lpf_3[-1]
```

Average the 8 LPF along the frequency version.

```
    Alpf_1[0]    = lpf_4[0]              OR     lpf_4[1]
    Alpf_2[0]    = Alpf_1[0]             OR     Alpf_1[2]
    Alpf(0)      = Alpf_2[0]             OR     Alpf_2[-4]
```

Rescale the data to remove low amplitude energy.

```
    scl1[0]      = lpf_4[-3]             OR     Alpf[0] NOT
    scl[0]       = scl1[0] {16. -8.}     SCALE
```

Identify local maximum along the frequency version.

```
    pk1[0]       = scl[0]                OR     scl[-1] NOT
    pk2[0]       = scl[0]                OR     scl[1] NOT
    pk[0]        = pk1[0](50. -25.}      AND    pk2[0](50. -25.}
```

Identify Continuities along time version of length 2.

```
    C_00[0]      = pk[0](2. 0.}          OR     pk[0][-1] {2. 0.}
    C_01[0]      = pk[0](2. 0.}          OR     pk[1][-1] {2. 0.}
    M_01[0]      = pk[0](2. 0.}          OR     pk[-1][-1] {2. 0.}
```

Form Continuities of length 4 at various angles along time dimension.

```
    C_0000[0]  = C_00[0]             OR     C_00[0] [-2]
    C_0001[0]  = C_00[0]             OR     C_01[0][-2]
    C_0011[0]  = C_00[0]             OR     C_00[1][-2]
    C_0111[0]  = C_01[0]             OR     C_00[1][-2]
    C_0012[0]  = C_00[0]             OR     C_01[1][-2]
    C_0112[0]  = C_01[0]             OR     C_01[1][-2]
    C_0122[0]  = C_01[0]             OR     C_00[2][-2]
    C_0123[0]  = C_01[0]             OR     C_01[2][-2]
    M_0001[0]  = C_00[0]             OR     M_01[0][-2]
    M_0011[0]  = C_00[0]             OR     C_00[-1][-2]
    M_0111[0]  = M_01[0]             OR     C_00[-1][-2]
    M_0012[0]  = C_00[0]             OR     M_01[-1][-2]
    M_0112[0]  = M_01[0]             OR     M_01[-1][-2]
    M_0122[0]  = M_01[0]             OR     C_00[-2][-2]
    M_0123[0]  = M_01[0]             OR     M_01[-2][-2]
```

Determine if one of the continuities of length 4 is large. Input threshold, {8. -5.}, requires three out of the four input peaks be large. There are actually only 15 continuities. The input to the first node is multiplied by 4 (rather than the 8 used in the other nodes) so that its contribution will be the same as the other 14 nodes. The final scaling requires that 1 of 16 continuities be large.

```
    C_t11[0]    = C_0000[0] {4. -2.5}    SCALE
    C_t12[0]    = C_0001[0] {8. -5.}     OR     M_0001[0] {8. -5.}
    C_t13[0]    = C_0011[0] {8. -5.}     OR     M_0011[0] {8. -5.}
    C_t14[0]    = C_0111[0] {8. -5.}     OR      M_0111[0] {8. -5.}
    C_t15[0]    = C_0012[0] {8. -5.}     OR     M_0012[0] {8. -5.}
    C_t16[0]    = C_0122[0] {8. -5.}     OR     M_0122[0] {8. -5.}
    C_t17[0]    = C_0123[0] {8. -5.}     OR     M_0123[0] {8. -5.}
    C_t18[0]    = C_0112[0] {8. -5.}     OR     M_0112[0] {8. -5.}
    C_t21[0]    = C_t11[0]               OR     C_t12[0]
    C_t22[0]    = C_t13[0]               OR     C_t14[0]
    C_t23[0]    = C_t15[0]               OR     C_t16[0]
    C_t24[0]    = C_t17[0]               OR     C_t17[0]
    C_t31[0]    = C_t21[0]               OR     C_t22[0]
    C_t32[0]    = C_t23[0]               OR     C_t24[0]
    C_t41[0]    = C_t31[0]               OR     C_t32[0]
    final[0]    = C_t41[0] {32. -2.}     SCALE
```

Table 1. A Listing of All the Nodes

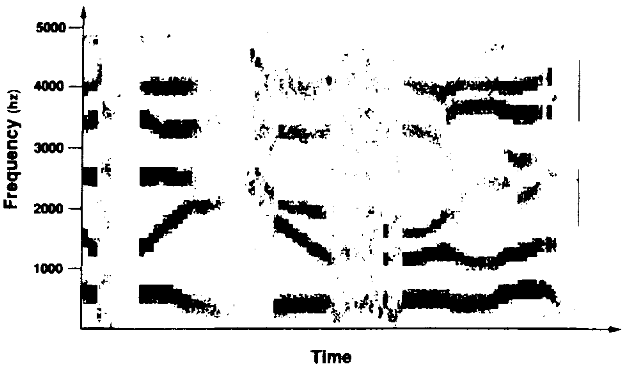


Figure 7a. Nodal Implementation - Resclaed Data.

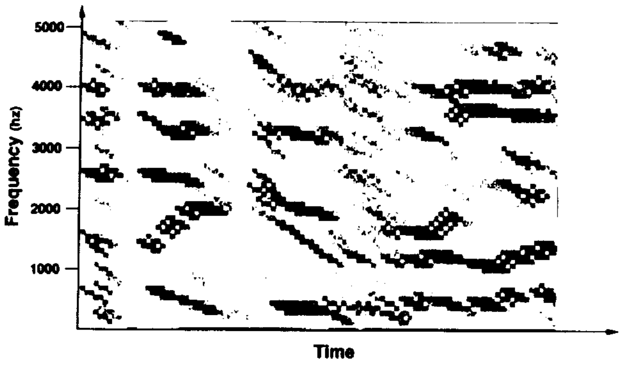


Figure 7c. Nodal Implementation - Continuity at -45°

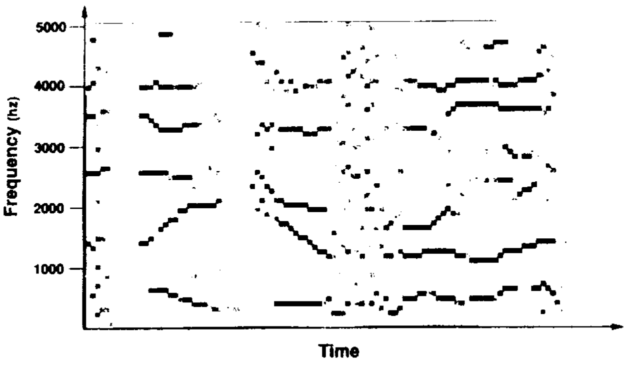


Figure 7b. Nodal Implementation - Formants Identified.

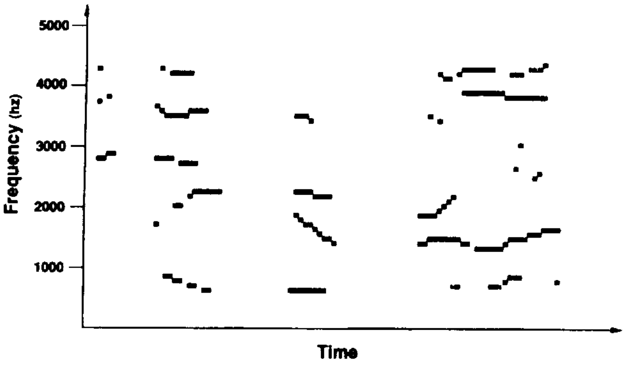


Figure 7d. Nodal Implementation - Final Result

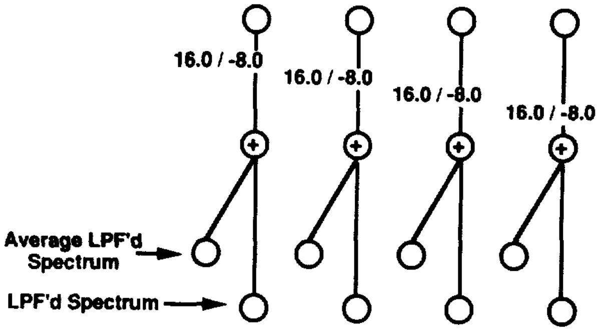


Figure 8c. Nodes to Enhance Spectral Peaks

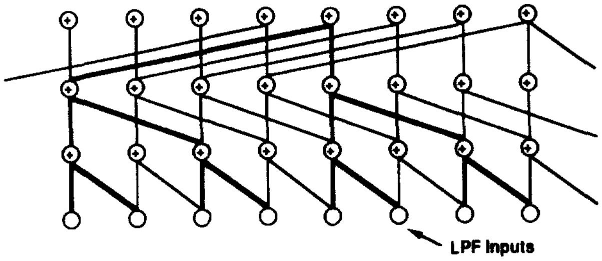


Figure 8b. Nodes to Average LPF'd Data

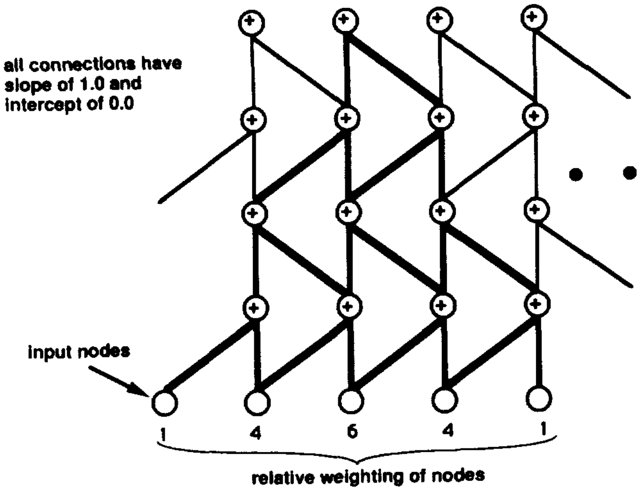

all connections have slope of 1.0 and intercept of 0.0
input nodes
1
4
6
4
1
relative weighting of nodes

Figure 8a. Nodes to Low Pass Filter Spectrum

The nodes shown in Figure 8a are low pass filter (LPF) the inputs. The connections shown by the heavy lines indicate the paths that lead to a single output node. Notice that the input nodes contribute through multiple paths to the output nodes. The numbers at the bottom indicate the relative contribution of each of the nodes. Notice that the scaling of inputs is consistant because at every level all values are divided by 2.

The nodes in Figure 8b calculate the average of the LPF. The three levels together produce outputs that are the sum of 8 LPF nodes. Notice that the input nodes contribute only through a single path to the output nodes. The division by two at each of the three levels accumlates to a division by 8. The result is an exact average of the 8 input connected by the heavy lines.

The nodes in Figure 8c perform a rescaling by summing the LPF value (lpf) and the inverted average (Alpf). The intermediate result (scl1) is:

$$\text{scl1} = (\text{lpf} + (1 - \text{Alpf})) / 2 \qquad (1)$$

or reducing

$$\text{scl1} = 1/2 + \text{lpf}/2 - \text{Alpf}/2 \qquad (2)$$

The next node multiplies this result by 8 and then subtracts 4 to give the output z. Inserting and reducing gives:

$$z = 4 * (\text{lpf} - \text{Alpf}) \qquad (3)$$

Notice that the result is 0 if Alpf is greater than lpf. It is equal to one if lpf is greater than Alpf by more than 1/4.

The implementation of this algorithm on the Connection Machine is accomplished by assigning one processor to each data point in the network. The spectrums, each with 64 data points, are pipelined through the system. In affect the system is being used in a "compute level" parallelism fashion.

5.0 Results

The figures in the previous section illustrate that the algorithms do work for the example given. Note that the single spectrogram evaluation is very insufficient but, the goal of this effort was to illustrate the use of nodal networks not to produce a "fieldable" algorithm. Both algorithms perform similarly on this spectrogram.

The speed of execution of the two implementations indicates that there is not a strong difference in the efficiency of the two algorithms. The ratio of execution time is approximately 4:1 (nodal network to conventional). It must be pointed out that performance was not a focus of this work. The conventional implementation was mildly massaged to improve the performance. No such massaging was done for the nodal network algorithm. Further, the nodal network algorithm was implemented in floating point and the conventional algorithm in 8-bit integer. The nodal network implementation can be easily implemented in fixed point and the data communications can be dramatically improved without much effort. It is very likely that the execution time of the two systems would then be comparable.

## 6.0 Conclusions

The first conclusion suggest the continuation of this work. Nodal networks can provide a method for parallel processing one dimensional signal data in a massively parallel SIMD processor if the processor has a reasonably efficient system for random interprocessor communications. There are certainly a lot of questions left unanswered about the long term system. Still the experience is encouraging. The software that supports "programming" of the nodal network is improving and, in turn, making the exploration of these systems much easier.

## 7.0 Acknowledgements

I want to thank my wife Amy and my children Jeremy and Autumn for their support during the work leading up to this paper and during its preparation.

## *8.0 References*

1. Minsky, M. and Papert, S. (1969). Perceptrons, An Introduction to Computational Geometry. MIT Press, Cambridge, Mass.

2. Fukushima, K. and Miyake, S. (1982). Neocognitrom: A New Algorithm for Pattern Recognition Tolerant of Deformations and Shifts in Positions. Pattern Pecognition, Vol. 15, No. 6, pp 455-469.

3. Graff, D. and Lundgren, W. (1987). Speaker-Independent Connected Speech. RADC Tech. Rep. RADC-TR-87-58.

4. Meckley, J. GE Areospace Advanced Technology Laboratories. Personal communication.

# USEFULNESS OF THE MASSIVELY PARALLEL PROCESSOR FOR STUDY OF ELECTRONIC PROPERTIES OF ATOMIC AND CONDENSED MATTER SYSTEMS

N. Sahoo    T. P. Das
Department of Physics
State University of New York at Albany
Albany, New York, 12222

S. N. Ray
Software Corporation of America
6703 Spenard Road, Bowie, Maryland, 20715

## ABSTRACT

We have been investigating the extent of increase possible in the efficiency of computations of electronic structures and properties of atomic, molecular and condensed matter systems when one uses the Massively Parallel Processor (MPP). The usefulness of MPP in improving the speed of evaluation of two-center electronic integrals and formation of Fock matrix elements necessary in electronic structure investigations of molecular and solid state systems is analyzed. The possibility of reduction in the computational time for calculation of the large numbers of matrix elements for electron-electron interactions needed in many-body perturbation theory for atomic systems by using a parallel algorithm for numerical integrations is also discussed.

Keywords: Electronic Structure, Hartree-Fock Theory, Many-Body Theory, Atoms, Molecules and Solids, Parallel Computation.

## SECTION I. INTRODUCTION

We have been engaged in an examinatiion of the possibility of enhancing the speed and scope of electronic structure investigations in atoms, molecules and condensed matter systems through the use of the Massively Parallel Processor (MPP) at the NASA-Goddard Space Flight Center in Greenbelt, Maryland. In this connection, we have been investigationg three different procedures for calculation of electronic structures and properties. The first procedure that we have been trying to adapt to the MPP is the semi-empirical procedure referred to in the literature as Self-Consistent Charge Extended Hückel (SCCEH) procedure (Ref. 1) which has been applied extensively in the literature for the study of the properties of large molecular systems, including hemoglobin derivatives, using serial computers. The second procedure uses the first-principles Hartree-Fock approach (Ref. 2) using Gaussian basis sets to study the electronic structures of molecular and solid state systems accurately in the one-electron approximation. The third procedure we have been examining for adaptation to the MPP is the many-body pertur-bation theory (Ref. 3) which has been employed within the framework of both non-relativistic and relativistic theories for electronic structures and hyperfine properties of a large number of atoms. It is expected that the analysis of these three procedures will provide a broad-based test of the potential advantages of the MPP (Ref. 4) for electronic structure investigations. While our investigations are currently in their early stages, we believe that the results of our efforts so far may be useful to other investigators interested in using MPP for similar calculations.

Section II will present our work on the Hartree-Fock Gaussian procedure for molecular and solid state systems and Section III on the many-body perturbation theory for atomic systems. Space does not permit us to describe our project concerned with the adaptation of the SCCEH procedure to the MPP. It will be presented in a future publication. Section IV summarizes our conclusions.

## SECTION II. ADAPTATION OF THE HARTREE-FOCK PROCEDURE INVOLVING GAUSSIAN BASIS TO MPP

The Hartree-Fock method(Ref. 2) involves a first-principles approach for evaluation of electronic energy levels and wave-functions in the one-electron approximation. The Hamiltonian of a molecule or a cluster of atoms used to simulate a solid state system (Ref. 5), which contains n electrons and N nuclei is given by

$$\mathcal{H} = -\frac{1}{2}\sum_{i=1}^{n}\nabla_i^2 - \sum_{i=1}^{n}\sum_{I=1}^{N}\frac{Z_I}{|\vec{r}_i-\vec{R}_I|} + \sum_{i>j}\frac{1}{|\vec{r}_i-\vec{r}_j|} + \sum_{I>J}\frac{Z_I Z_J}{|\vec{R}_I-\vec{R}_J|} \quad (1)$$

where atomic units have been used, the $Z_I$ referring to the charges on the nuclei at positions $\vec{R}_I$, $\vec{r}_i$ being the position vector of the ith electron. The problem of determining the electronic structure reduces to solving the many-electron Schrödinger equation $\mathcal{H}\Psi=E\Psi$, where in the Hartree-Fock approximation $\Psi$ is a determinantal function built out of LCAO-MO (Linear Combination of Atomic Orbitals-

CH2649-2/89/0000/0671$01.00 © 1988 IEEE

Molecular Orbitals) wave-functions of the form $\psi_\mu = \sum_i C_{\mu i}\chi_i$, $\chi_i$ being atomic basis functions. In the Gaussian basis set approach the $\chi_i$ have the form:

$$\chi_i^I(r_i^I) = N_i^I x_{ai}^I y_{bi}^I z_{ci}^I e^{-\zeta_i^I r_i^{I2}} \quad (2)$$

where the symbol I refers to the nucleus on which the orbital $\chi_i$ is based, $\vec{r}_i^I$ referring to the position vector of the ith electron with respect to the nucleus I. The solution of the many-electron Schrodinger equation by a variational procedure entails solution of the linear equations:

$$\sum_i C_{\mu i}(h_{ij}^{HF} - \mathcal{E}_\mu S_{ij}) = 0 \quad (3)$$

where $S_{ij}$ and $h_{ij}^{HF}$ are the elements of the overlap and Fock matrices and are given for the choice of real orbitals by

$$S_{ij} = \int \chi_i \chi_j d^3r, \quad (4)$$

$$h_{ij}^{HF} = \int \chi_i(1)(-\frac{\nabla_1^2}{2})\chi_j(1)d^3r_1 + \sum_J \chi_i(1)(\frac{-Z_J}{r_{1J}})\chi_j(1)d^3r_1$$
$$+\sum_\mu \sum_k \sum_l C_{\mu k}C_{\mu l}[2\int \chi_i(1)\chi_j(1)\frac{1}{r_{12}}\chi_k(2)\chi_l(2)d^3r_1 d^3r_2$$
$$-\int \chi_i(1)\chi_k(1)\frac{1}{r_{12}}\chi_j(2)\chi_l(2)d^3r_1 d^3r_2] \quad (5)$$

The non-trivial solutions of the Equations 3 are obtained as usual by solving the secular equation $\mathrm{Det}|h_{ij}^{HF} - \mathcal{E}_\mu S_{ij}| = 0$ to determine the MO energy levels $\mathcal{E}_\mu$, these bei-ng used in Equations 3 together with the normalization conditions on the MO, $\psi_i$ to determine the corresponding $C_{\mu i}$. An iterative procedure involving successive evaluation the $h_{ij}^{HF}$ using the $C_{\mu i}$ obtained from Equations 3 is necessary to obtain a self-consistent solution.

An analysis of the computational steps involved in the Hartree-Fock procedure leads one to expect that it can be effectively implemented on the MPP with significant enhancement in speed by using suitable algorithms for parallelization of the following steps:

(a) Evaluation of the multi-center one-electron and two-electron integrals in Equation 5 and the overlap integrals $S_{ij}$ in Equation 4.

(b) Formation of the Fock matrix $h_{ij}^{HF}$ using the calculated one and two-electron integrals and the $C_{\mu i}$ from the preceding step in the iteration.

(c) Solution of the Equations 3.

Considering the step (a), it is clear that the integrals involved in Equations 4 and 5 are independent of each other, making them good candidates for parallelization. However one has to develop suitable algor-ithms to map the computational problem to the architecture of the MPP. As a first step in this dirction we have utilized the MPP to evaluate the overlap matrix S and benchmarked the speed with respect to the serial computer UNIVAC 1100/91.

An important consideration in the parallelization of the procedure to evaluate the overlap matrix S is that the overlap integrals between two Gaussian orbitals of any angular momentum (such as s,p,d) on different centers can all be expressed in terms of overlap integrals between Gaussian s orbitals (with $a_i=o=b_i=c_i$) in Equation 2 on these centers, these integrals being expressible in the form (Ref. 2):

$$\langle s_A|s_B\rangle = K N_A N_B (\pi/\gamma_{AB})^{3/2} \quad (6)$$

where $K = e^{-\{(\zeta_A\zeta_B/\gamma_{AB})|\vec{AB}|^2\}}$ and $\gamma_{AB} = \zeta_A + \zeta_B$ (7)

and $|\vec{AB}|^2$ is the square of the distance between nuclei A and B. Typical overlap integrals $\langle z_A|s_B\rangle$ and $\langle z_A|z_B\rangle$, where $z_A$ and $z_B$ represent Gaussian $p_z$ orbitals on A and B with $a_i=0$, $b_i=0$ and $c_i = 1$ in Equation 2, are given by (Ref. 2):

$$\langle z_A|s_B\rangle = \frac{-\zeta_B}{\gamma_{AB}}|(\vec{AB})_z|\langle s_A|s_B\rangle|$$

and $$\langle z_A|z_B\rangle = (\frac{1}{2\gamma_{AB}} - \frac{\zeta_A\zeta_B}{\gamma_{AB}^2}|(\vec{AB})_z|^2\langle s_A|s_B\rangle \quad (8)$$

Thus the only overlap integrals that need to be calculated are the $\langle s_A|s_B\rangle$ between various centers in the molecule or cluster under study. It should be noted that some of the overlap integrals in Equations 6 and 8 can be one center integrals with K =1 for the $\langle s_A|s_B\rangle$ terms since $|\vec{AB}|^2$ vanishes. For orbitals of different symmetry such as s and $p_z$, Equations 8 show that the one-center integrals vanish. The evaluation of the $\langle s_A|s_B\rangle$ integrals using the MPP involves the following steps.

(a) Determination of the distance matrix components $(\vec{AB})_{x,y,z}$ from which $|(\vec{AB})|^2$ can be calculated.

(b) Determination of the $\zeta_A\zeta_B$ matrix.

(c) Determination of the $\gamma_{AB}$ matrix (Equation 7).

(d) Determination of the $N_A N_B$ matrix involving the products of the normalization factors occuring in the orbital expressions (Equation 2).

(e) Determination of the K matrix (Equation 7).

Since the MPP can do arithmetic operations involving 128x128 parallel arrays (Ref. 4), the Gaussian functions are divided into blocks of 128 Gaussians and all the parallel arrays or matrices involved in the steps (a)-(e) have dimension 128x128. As an example, we will describe the algorithm for the evaluation of the distance matrix for 128 points. For this purpose, one first constructs three 128x128 parallel arrays with each column containing the x,y or z coordinate of a single point, e.g.

$$[C]_{ni} = x_i, \quad [D]_{ni} = y_i, \quad [E]_{ni} = z_i \quad (9)$$

for n=1 to 128 and i=1 to 128. For instance:

$$[C] = \begin{bmatrix} x_1 & x_2 & \cdots & x_{128} \\ x_1 & x_2 & \cdots & x_{128} \\ \vdots & \vdots & & \vdots \\ x_1 & x_2 & \cdots & x_{128} \end{bmatrix} \quad (10)$$

with 128 rows and 128 columns. One then calculates the transpose of matrices [C], [D] and [E] from which

$$[X]=[C]-[C]^T,\ [Y]=[D]-[D]^T,\ Z=[E]-[E]^T \quad (11)$$

are obtained. One can easily see that

$$[X]_{ij}=x_i-x_j,\ [Y]_{ij}=y_i-y_j \text{ and } [Z]_{ij}=z_i-z_j \quad (12)$$

Therefore the distance matrix for different centers can be obtained from

$$[Y]_{ij}=\sqrt{[X]_{ij}*[X]_{ij}+[Y]_{ij}*[Y]_{ij}+[Z]_{ij}*[Z]_{ij}} \quad (13)$$

In contrast to serial computation where the arithmetic operations are carried over one data item, like $x_i-x_j$, here we are evaluating the difference between x-coordinates of all the 128 centers simultaneously This can lead to considerable saving in computational time. A similar procedure is followed in carrying out the steps (b) to (d). Once the matrices in (a), (b) and (c) have been evaluated, the matrix (e) can be obtained directly using Equation 7. The parallel pascal programs for these steps have been written and tested on the MPP. It is straightforward to calculate the overlap integrals $\langle s_A|s_B\rangle$ using Equation 6. If overlap integrals involving other orbitals besides s are needed, one makes use of relations like those in Equation 8, the quantities involved in these relations being already available on the processors of the MPP.

In actual molecular calculations, the atomic basis sets on different centers are expressed as linear combinations of Gaussian functions $G_{pi}^I$ (same form as in Equation 2) in the forms:

$$\chi_i^A=\sum_p d_{pi}G_{pi}^A,\ \chi_j^B=\sum_q d_{qj}G_{qj}^B \quad . \quad (14)$$

The overlap integral between $\chi_i^A$ and $\chi_j^B$ can then be written as

$$\langle\chi_i^A|\chi_j^B\rangle=\sum_p\sum_q d_{pi}d_{qj}\langle G_{pi}^A|G_{qj}^B\rangle \quad (15)$$

Equation 15 can also be evaluated on the MPP by computing $\langle G_{pi}^A|G_{qj}^B\rangle$, the overlap integrals between the primitives, and then multiplying it with the $d_{pi}d_{qj}$ matrices and performing the sum indicated in Equation 15.

Our benchmarking procedure using 128 atomic basis functions of the type $\chi_i^A$ in Equation 14, with each basis function consisting of a combination of three Gaussians, has shown that these one electron-two center integrals can be computed 15 times faster in MPP compared to UNIVAC 1100/91.

The main features of the algorithms developed for the overlap matrix elements can be directly applied to evaluate the one-electron integrals (referred to as kinetic energy integrals) in the first term in the Fock-matrix element expression in Equation 5. This family of integrals involves no more than two centers as in the case of the overlap integrals and similar expressions are available (Ref. 2) for them. The nuclear attraction integrals in the second terms of the Fock-matrix (Equation 5) involve three-center integrals in addition to one and two centers which occurred for the overlap matrix. The electronelectron Coulomb and exchange interaction integrals represented by the third and fourth terms of the Fock-matrix expression (Equation 5) can involve both three and four-center integrals. These integrals are much more numerous than the overlap and kinetic energy integrals. However, analytic expressions analogus to those for the overlap integrals, but somewhat more complicated, are available for them. In principle then, the nuclear attraction and electron-electron coulomb and exchange integrals can be adapted to the MPP by procedures similar to those used for the overlap integrals. We are currently investigating suitable algorithms for computing these integrals on the MPP.

Once the one- and two-electron integrals in the Fock matrix expression in Equation 5 are obtained, the Fock-matrix elements $h_{ij}^{HF}$ can be evaluated. As far as the one-electron contributions representing the first and second terms in $h_{ij}$ are concerned, nothing more is needed. For the two-electron terms however, Equation 5 shows that matrix multiplications have to be carried out between the density matrix (with elements $C_{\mu k}C_{\mu l}$ ) and the electron-electron interaction matrix (involving elements <ij|kl> for coulomb and <ik|jl> for exchange interactions, kl being running indices for fixed i and j depending on the Fock-matrix element being evaluated, followed by a summation over the occupied states $\mu$. We have developed algorithms for carrying out these operations on the MPP which involve terms like SUM {[<ij kl>] *[(COLBOARD $C_\mu$ )]*[(COLBOARD $C_\mu$ )]$^T$} where SUM (Ref. 4) represents the summation of all the elements in the parallel array resulting from the operations within the curly brackets. The COLBOARD operation (Ref 4) involves propogating a column vector $C_\mu$ over all the columns of the MPP producing a matrix with identical columns. Each processor in the MPP is assigned an element of the matrix [<ij kl>] corresponding to a particular choice of kl. The products between the matrices marked by asterisks in the curly brackets are element by element products and not matrix products. The entire procedure has to be repeated for all the occupied states $\mu$ and a summation of the contributions for different $\mu$ has to be carried out to get each Fock matrix element $h_{ij}^{HF}$. In principle, for a problem involving upto 128 basis functions, this procedure can lead to a reduction in the number of floating point operations by a factor of about $(128)^2$. From our experience with the evaluation of overlap integrals, the saving in terms of computational time is expected to be significantly smaller than the $(128)^2$ factor because of various machine-related factors like the I/O operations between the MPP and its host and the relatively lower speed of floating point operations on individual processors of the MPP. The algorithms for construction of matrix elements of $h_{ij}^{HF}$ are expected to be also applicable to the evaluation of expectation values over molecular orbitals of operators associated with specific electronic properties, for instance hyperfine properties of most interest to us (Ref. 5).

After the formation of the Fock-matrix and overlap matrix S, the next process is to solve the linear equations in Equation 3. This can also in principle be carried out on the MPP and we are exploring available algorithms for this purpose (Ref. 6) to use for both Hartree-Fock and semiempirical SCCEH procedures.

## SECTION III. ADAPTATION OF THE MANY-BODY PERTURBATION PROCEDURE FOR ATOMIC SYSTEMS TO MPP.

We have also been exploring the capability of the MPP for enhancing the scope and speed of accurate many-body perturbation theoretic investigations (Ref. 3) of electronic properties of atomic systems. The most time-consuming aspect of these investigations is that of computing electron-electron interaction integrals of the form:

$$I_4=\iint\psi_{R1}^*(1)\psi_{R2}^*(2)\frac{1}{r_{12}}\psi_{R3}(1)\psi_{R4}(2)d^3r_1d^3r_2 \qquad (16)$$

involving four excited state wave-funtions $\psi_{R1},\psi_{R2},\psi_{R3}$ and $\psi_{R4}$ of the system which are unoccupied. Since one has often to deal with as many as twenty-five excited states of different symmetry, the number of such integrals is a multiple of $(25)^4$ in many instances about ten times this number and it is often impracticable to carry the accuracy of the theory to a level where all these integrals are needed. These integrals have so far been carried out on serial computers using standard numerical integration procedures because the wave-functions $\psi_R$ used are in numerical form. The integration procedure most commonly employed is the Guass-Laguerre method (Ref. 7), where an integral of the form:

$$I=\int_0^\infty g(x)dx \qquad (17)$$

is expressed as

$$I=\sum_{i=1}^{\infty}(\omega_i e^{x_i}g(x_i) \qquad (18)$$

the $x_i$ being positions of the zeros of Laguerre polynomials and $w_i$ being weight factors. This summation form was found amenable for adaptation to MPP and we have developed an algorithm for calculating 128 such integrals I simultaneously on the MPP. In this algorithm, one constructs three parallel arrays (Ref. 4) involving $x_i$, $w_i$ and $g(x_i)$. For N=128 in Equation 18, three multiplications and seven addition of floating point numbers are required to evaluate the 128 integrals I. These replace the 49,152 multiplications and 16,384 additions that would be required for serial computations. With this large reduction in number of floating point operations, the evaluation of the large numbers of integrals of the form in Equation 16 is expected to become more practicable. Details of our algorithm and application to specific problems will be given in a future publication.

## SECTION IV. CONCLUSION

The possibility of adapting a number of current computational procedures in the investigation of electronic structures of atomic, molecular and condensed matter systems has been examined. We have developed a number of algorithms for carrying out the computational steps involved in these procedures, which are rather time-consuming when one uses serial processors. These algorithms show that using the massive parallelism of the MPP, it is possible to reduce the numbers of serial arithmetic operations by very substantial factors. However, our benchmarking in a few cases that we have been able to completely study so far indicate that only a fraction of the expected saving in computational time is being attained. While this is already quite useful, future improvements in the speeds of the individual processors of the MPP and in the data transfer to and from it, should make this kind of machine a very valuable tool in electronic structure investigations.

## ACKNOWLEDGEMENT

We are grateful to Mr. E. Seiler, Dr. M. Halem and Mr. J.R. Fischer for valuable help. This research was partially supported by NASA Grant #NAG 5836 and Office of Vice President for Research, SUNY Albany.

## REFERENCES

1. P.S. Han, M.F. Rettig and T.P. Das, Theo. Chim. Acta. 16, 1 (1970).

2. R. Daudel et al, "Quantum Chemistry", (John Wiley, New York, 1983).

3. Mina Vajed-Samii et al, Phys. Rev. A20, 1787 (1979).

4. "MPP User's Guide", and "MPP Pascal Programmer's Guide", (NASA Goddard Space Flight Center, Greenbelt, Maryland).

5. N. Sahoo, K.C. Mishra, and T.P. Das, Phys. Rev. Lett. 55, 1506 (1985).

6. J.J. Modi et al, in "Parallel Computing '83", ed.M. Feilmeir et al, (North Holland, Amsterdam, 1984), page 191.

7. "Handbook of Mathematical Functions", Ed. by M. Abramowitz and I.A. Stegun, (Dover, New York, 1970), page 923.

C-8

# Tools for Managing Massively Parallel Systems

K.M. Nichols
Systems Technologies Department
Apple Computer, Inc.
Cupertino, CA 95014
nichols@apple.com

## ABSTRACT

Tools for managing massively parallel systems are discussed in the context of existing tools and models for concurrent systems. Based on a survey of existing tools for parallel systems, creation of a "consensus tool", integrating the best features of these with additional functionality for behavioral study and performance evaluation, is proposed. Flow graph models for concurrent systems are shown to be in widespread use and flexible enough to be used as a basis for creating models for this tool.

PARET (Parallel Architecture Research and Evaluation Tool), an environment that uses interacting, multiple flow graphs to form a system model, is discussed in some depth as an existing tool with many of the desirable features of the target consensus tool. PARET, which has been used for medium-size systems, has features for use in modeling larger scale systems. The modeling of a Linda machine and its implementation using PARET is shown as an example of system modeling using a progression of interacting flow graphs.

Keywords: Multiprocessor Tools, Performance Evaluation, Multiprocessor Models, Visualization Tools, Discrete-Event Simulation

## INTRODUCTION

Now that large-scale parallel machines are in use, the time is past due for tools to assist system architects in performance and behavioral evaluation of such systems. Creating tools involves anticipating the questions of interest about massively parallel systems. The fact that we are still discovering the nature of the significant questions that will need to be answered complicates this process. This work partially focuses on the construction of tools and partially focuses on the discovery process, particularly for larger systems. Ultimately, some experimentation with tools will be required, and some wrong paths will be taken. The goal of this work is to synthesize the best aspects of previous approaches to tools for medium-size parallel systems with new ideas for handling more complex systems into a tool for the future.

Performance evaluation programs for parallel computers of any size are still in infancy, although a number of efforts are appearing. One of the striking things about these is the frequent emphasis on a visual component. These include Poker [1, 2], PIE [3], tools for the B-HIVE project [4], PROFILE [5], the Software Oscilloscope [6], and PARET[7]. The wide variety of approaches taken is indicative of the diversity of data available in a parallel machine and the range of requirements of architects of different parts of the system. It is time to move toward a "consensus tool", that is, one which represents the intersection of the previous major efforts and provides a platform for enhancements. The major lesson of these tools and of research work on parallel performance evaluation [8, 9, 10], is that performance cannot be represented solely by a number in the parallel computing world.

The ubiquitousness of certain features and the growing realization that performance characterization is a complex and application-dependent task points to the possibility of creating a consensus tool containing a toolbox of frequently used performance measurement functions. The consensus tool must also be customizable, so it can grow with the body of applications. A good starting point for such a tool is to use discrete-event simulation of models, followed by partial or full use of post-mortem machine trace information to drive simulations. A longer range possibility is to use the tool as a front-end to actual machine control and observation.

Parallel systems can be viewed at several layers of abstraction including: the user program(s) running, the operating system coordinating the running user program(s), and the topology of the physical interconnect. These real systems must be modeled in a way that expresses only the requisite degree of complexity, hiding and grouping details that are not of immediate interest. A model represents a real system in the tool's environment, facilitating user observation and interaction with as many or as few of these abstract representations as desired. Nonessential layers should be either idealized (e.g. zero communications time) or observed only by their effects on other layers (e.g. communications delays affect user programs). Furthermore, the environment should permit a model to evolve from a first pass of gross detail to a more sophisticated version of finer detail.

The principle advantages of PARET in the context of this paper is that the underlying model on which it is based is extensible to parallel systems of arbitrary size and complexity. and that the model used in PARET is intuitive and is similar to many other modeling approaches. Modeling approaches will be discussed in the next section. In the third section, the major questions that should be answered are examined, i.e. what should be the functionality of an effective tool? The fourth section shows an example of evolving a system model, and is followed by Conclusions.

## MODELING LARGE, CONCURRENT SYSTEMS

A model represents the actual system in the tool's environment in a way that is easily comprehensible to the user. An abstract method of modeling parallel systems is clearly required to represent massively parallel systems. Preferably, the modeling methodology should be continuously extensible from smaller to larger parallel systems. Further, a modeling environment should have the same general characteristics regardless of which subsystem is under consideration or the level of detail represented. As stated in the Introduction, a model must be flexible enough to represent all aspects of a parallel system at varying levels of detail. Thus, a system may be modeled at a crude level of detail early in the research and design processes, with progressive refinements permissible.

In order to characterize successful models, previous work in the field is reviewed. This work is roughly divided into four areas: user software representations, control and operating system

CH2649-2/89/0000/0675$01.00 © 1988 IEEE

models, topology of the physical interconnection, and communication structures and other architectural issues. Flow graphs are shown to appear as a persistent theme in these models, providing an intuitive, widely used method of modeling parallel systems and a prime candidate for a modeling the basic components of a system. What remains is the problem of tying the flow graph components together into a system model and creating a tool environment that adequately presents the model to the user.

There are two distinct approaches to creating parallel user programs. The first (transparent) hides the parallelism from the user and the second (explicit) gives the user total control over parallelism. Transparent parallelism occurs when sequential programs are transformed by intelligent compilers into parallel programs (e.g., Alliant FX/Fortran). In explicit parallelism, programmers designate which parts of the program may be executed concurrently and when and how to exchange data between processes (e.g. Cosmic Cube C). Various intermediate approaches allow users to annotate where parallelism should occur, as in Poker [2]. Explicit control usually results in optimized program execution times, while transparent parallelism reduces programmer development time. For medium-sized systems, transparent approaches can produce efficient code. This has been shown by various approaches to "dusty deck" transformation, and by the Apply system that has been used at Carnegie Mellon [11]. Both explicit and transparent approaches frequently make use of directed flow graphs (sometimes in the form of directed acyclic graphs, DAGs, as a special :ase) to represent the program. Explicit approaches often allow the user to enter or view a program as a flow graph of connected software components. Parallelism-extracting compilers transparently form these graphs and use them to make automatic program improvements. Some of the more explicit approaches, such as Poker [2] and large-grain dataflow (LGDF) [12, 13], have users enter programs graphically, at least in part. PARET [7] uses a general flow graph model that may be used to represent any graph. Typically, each node of the graph represents a portion of user code, from a few lines to an entire process.

Representation of operating systems and control structures is less prevalent in the literature than work with user programs. An operating system is itself composed of programs, with the major difference that the programs are more frequently responding to and creating events external to their own program code, so the user program representation methodologies should be applicable. In addition, Srini and Shriver show how *extended dataflow graphs* [14] (EDFGs) can be used to model the control structures of reconfigurable, concurrent systems. Work that has been more strongly identified with control structure analysis has been done by using variations on Petri Nets to develop performance-oriented models that represent the control structure of concurrent systems [15, 16]. Petri Nets may be thought of as flow graphs with a more restrictive set of formal rules.

Interconnection networks have been traditionally represented by directed flow graphs, with Poker, PARET, and the B-HIVE project tools[4], all examples of tools that incorporate such models. Unfortunately, interconnection graphs with a only a moderately large number of nodes and arcs, for example a 6-cube, are not easy to observe in a two-dimensional view. To represent large-scale or massively parallel interconnects, additional visual techniques are clearly required to present the graph in a coherent fashion, but hierarchical modeling techniques that provide users with a conceptual framework for focusing on portions of the interconnect are also needed. This approach was taken in a recent work on generalizing families of architectures for massively parallel computers [17] shows. The interconnection topologies of these architectures rely on the concept of hierarchically structured, interacting directed graphs. Interacting flow graphs have been successfully implemented in PARET, along with some hierarchical features that will be discussed later.

A number of other architectural features of varying levels of abstraction have been successfully modeled with flow graph techniques. The previously-mentioned EDFGs have been used to model the architecture of a Cray computer. The Poker environment uses visual programming of message passing to eliminate writing explicit message-passing code and so that communication structures are represented by flow graphs, a method that has proven easier to use than explicit port designation [2]. Among the more abstract concepts represented by flow graphs is the mapping of processes to an architecture [4].

The above survey of previous work shows that, as a modeling environment, flow graphs are: intuitive, evidenced by their ubiquitousness; extensible, by decreasing the complexity of the component represented by each node; and flexible, when an arbitrary number of flow graphs can be made to interact. The strength of basing a tool for massively parallel systems on flow graph models is that it inherits these advantages and can build on previous work if the methodologies cited above can be implemented under the new environment in a fairly straightforward fashion.

Although a tool that allows for general flow graph representations can apply previously used modeling techniques, a consensus tool should also unite all these disparate modeling techniques. In this paper, the consensus tool is presented as a prismoid, with a number of *facets*. Each facet can represent one level of abstraction, one subsystem, or one modeling technique for the target parallel system. It is possible to look into any individual facet, to rotate the prismoid, observing one facet at a time, or even to view a number of facets at a time. The model chosen to represent a parallel system must support a multifaceted view of the system and a way to integrate all the facets. To form a multifaceted view, flow graphs must be made to interact, preferably in a hierarchical fashion.

PARET is an example of a tool that is based on interacting flow graphs whose complexity and functionality is left to the user. This makes it a good point of departure for design of a consensus tool. A PARET model consists of one or more directed flow graphs, that interact during simulation either via explicit connections (*ethers*), and according to rules specified by the modeler, or by sharing resources. The model comprises objects: graphs are composed of *nodes* interconnected by *arcs*, either of which may contain *buffers* storing *tokens*, the PARET unit of data and control. (See Figure 1.) Nodes are executed by *elements* operating concurrently, and nodes from different graphs can execute on the same element. Buffers from different flow graphs can be connected by *ethers* that provide a path for tokens to follow between graphs. All subsystems are modeled under this general paradigm, but the descriptions and functionality of the objects differ. By permitting interaction of an arbitrary number of flow graphs to form a model, models can be developed in a hierarchical fashion.

In a PARET model, a user program may be represented by one flow graph (perhaps compiler output), and the interprocessor communications topology by another. These flow graphs are viewed one at a time during a PARET session, implementing a separation of system abstractions. By decreasing the functionality represented by each node of a graph and increasing the complexity of the graph, a model can be made progressively more complex. Thus, PARET permits abstract layers to be represented with a unified model within a uniform user environment where models may have a wide range of complexity.

The features of PARET, both advantages and shortcomings, will be examined and discussed as a basis for the consensus tool. In the next section, the visual features of PARET will be presented. An example of modeling a proposed architecture with increasing refinement will be shown in Section 4, illustrating the power and feasibility of this approach. Additional features that are lacking in PARET and appear to be necessary for the study of massively parallel systems will also be discussed.

## ANSWERING THE QUESTIONS

A tool for massively parallel systems will have to provide means for answering the important questions about architectures, applications, and their interaction. In this section, performance studies and existing tools are surveyed for the questions they have asked, and for the approaches that have been taken to provide these anwswers. The approach taken by PARET is described in some depth, so that the example of the following section can be followed.

Parallel programmers have long been concerned with the effect of changing the number of processors assigned to a target program. More recent work has revealed the importance of the effect of changing the number of processors on how one thinks about the problem or how much larger a problem can be accommodated as an architecture is scaled upwards [18]. A tool should provide the means for comparing the execution of a given application running on different numbers of processing elements. In addition, for a scalable algorithm, it should be possible to fix the number of processors and increase the problem size, until it is no longer possible to execute the problem.

Comparative studies of interconnection networks normally focus on the latency of message communications, and graph metrics like network diameter. For scalable architectures, important measures include the increase in latencies and the change in traffic levels on the network links or through communications processors with increasing network size.

From the architectural point of view, studies comparing a number of architectures executing the same program or a suite of programs [8] show the most promise for providing evaluations. Another important function is to observe the effects of incremental architectural changes on system performance [9, 14].

In general, supplying execution times, or estimated execution times, is not sufficient. The amount of processor idle time and the amount of time spent on overhead functions provide useful measures of how well the operating system matches the architecture and application [6, 10].

The above work requires many approaches. Comparative architecture studies clearly benefit from an environment where program models and architectural models are easily interchanged and incrementally alterable. Some of the statistics required are simple values, like execution time, average idle time, or the number of times a resource was used over a fixed time. In some cases, the questions require observation of a statistic over time, e.g. to detect "hot spots" for some resource. Information can concern usage levels, presence or absence of activity, state of a component or number of times a component has been in a certain state. The state of a certain process or of a single processor, or of a region of the software or architecture that may be considered to be a cluster of atomic actions, may be examined. Surveying current tools reveals a number of methods used to make this data available to the user.

Some tools show the flow of data and control in an architecture, while others are geared toward observation of resource usage. The former is usually accomplished by animation of a flow graph model, and the latter has been accomplished by providing graphical monitoring devices of resources or model components. To differentiate these two approaches, depictior of a portion of a model provides a view of the system and is referred to as a facet, to maintain the prismoid analogy introduced in the previous section. A data collection and presentation method will be referred to as a *monitor*. These generally appear as either graphical meters or simple state information provided in response to a query. A few examples are cited below.

In his aptly-named Software Oscilloscope, Katseff [6] provides time-changing data for each processor on the amount of time spent on user, system, and idle, or waiting, time. This data has proven useful for determining load balancing information about particular parallel programs. The Software Oscilloscope provides dynamic performance-oriented information about each processor and is clearly in the monitor class, but does not provide data on the interconnection network or the structure of interprocess communications.

In Poker, a number of useful views of a program running on a particular architecture are provided to the user, although animation and performance feedback appear to be minimal. Thus, Poker provides a number of facets, but the only monitors are the indications of current state contained in each node. This environment has to be laboriously customized in order to handle a new architecture[19] and cannot be used as an exploratory tool.

In PARET, a number of facets are possible for a system, one for each flow graph in the model, and these flow graphs are animated and alterable during simulation. Simple monitors allow the user to select the components of the facet being observed to get the current state information. A limited selection of dynamic monitors (meters) are provided that present resource use as a function of time for a sliding time window. In addition, summary statistics of a simulation are collected.

A typical PARET window is divided into four parts: a *display* area where a single flow graph at a time can be observed, a *meter* area where a set of user-selected meters is displayed, a *local control panel* that is specific to the flow graph presently in the display where the simulation is controlled, and a *global control panel* where universal aspects of the model and its simulation are controlled. PARET also contains some visual features for representing complex graphs. These additions were intended to allow the application of the PARET visuals to large systems: expansion of selected portions of a displayed flow graph (selected magnification), and grouping of nodes into supernodes (hierarchical structure). Neither of these requires any changes in the underlying graph model. The first has been implemented by displaying a flow graph in a set of *viewports* each of which are controlled by separate movable, resizable *reticles* on the full flow graph (the display area shown in Figure 2). In the second method, although we *visually* represent a group of primitive nodes by a single node, the back end (simulator or machine) continues to treat the primitive nodes individually and has no notion of the supernode.

To adequately support massively parallel environments, PARET needs an improved suite of selectable and customizable performance meters, a variety of permissible methods for connecting the various flow graph views of a system, and the ability to replay trace information gained from real machines. Although all of these extensions are possible, it may be preferable to develop a new tool based loosely on PARET and incorporating facets and monitors from other tools.

## AN EXAMPLE MODEL

In this section, a model is presented for a Linda machine [20], an architecture that supports the Linda parallel programming language by emulating a type of shared memory particular to Linda, called *tuple space*. In the Linda language, all processes may access objects (tuples) in the shared space (see Figure 3). One characterization of architectural performance is the total number of data tuples that can be handled by the implementation over some period of time, as well as the average amount of time for the shared tuple space to respond to requests for particular tuples. This will be presented as an example of creating a rough model and beginning to refine it. In addition, this example will show methods of representing shared memory within the present constraints of PARET and will propose additions to the model.

The Linda machine has N horizontal buses, called *tuple beams*, and N vertical buses called *inverse beams*, as shown in Figure 4. At the $N^2$ intersections of these beams are Linda Nodes, each of which interfaces to a CPU. At each Linda Node, there is an *In Processor*, an *Out Processor*, and a *Tuple Memory* as shown in Figure 5. We begin by depicting these elements of each Linda Node by three PARET nodes as shown in Figure 6. This ignores some of the details of the actual Linda Node, but maintains the features of interest for our performance study. We place buffers inside each of the nodes; in the In and Out Processors, they hold data (PARET tokens) waiting to be sent on the inverse beam or tuple beam, respectively. For the Tuple Memory, the buffer represents the physical memory with PARET tokens representing tuples.

Since PARET does not have a construct for representing a bus structure, the *beams* are represented by PARET nodes. Further, rather than using a model of a Linda program to drive the architecture, a simpler model, generating tuples of randomly selected sizes at random intervals, is used to get initial performance data. For this purpose, a random *token generator* node is used to create tuples for the Out Processor, and another random token generator node creates requests for the In Processor. These two additions are shown as dashed nodes in Figure 6. The random tuple intervals are centered around a mean value referred to as the *grain size* of the computation. The undashed nodes in Figure 6 form a Linda Node, or in PARET, a cluster. An initialization function was written which builds a system model given the number of buses desired. Thus, the number of processors in the model is easily scaled by making a change in the input model description file.

Although creating a model of arbitrary size is easy, scaling the visual information is more problematic. The flow graph becomes increasingly more difficult to observe and interpret as N grows large. One approach to reducing the visual complexity is to encapsulate the PARET nodes of a single Linda Node into a supernode, a PARET feature allowing a set of nodes to be visually gathered into a single representation, but the number of these supernodes still grows as the square of N. In addition, we can use the reticle and viewport system to isolate sections of the graph.

To introduce another facet to the model, a second graph is added (shown in Figure 7), showing a programmer's conceptual view of the Linda machine. Here the entire tuple space is modeled as a shared memory node attached to $N^2$ user processes. These process nodes contain the generator nodes from the original graph, and PARET ethers [21] are used to pass tokens between graphs. A more sophisticated model can be created if the process nodes represent actual user program graphs instead of the generator nodes. These graphs can be connected to the process node by ethers, or the process node can be a "grouped" representation of that portion of the user graph that makes up the process.

The average response time can be compiled as a function of the number of processors in the architecture and the grain size, or the average time between tuple and request generation. A set of these values is currently being compiled.

## CONCLUSIONS

Tools for massively parallel systems require a very basic and flexible underlying model. Flow graphs have proven efficacious in many concurrent modeling methodologies. Since effective tools will require a "consensus approach", a marriage of the various current approaches to representing parallel systems and monitoring their performance, techniques for permitting the interaction of a number of flow graph models in a flexible tool environment are required. PARET was shown to be a tool which lays important groundwork for such a tool and an example implementation of a scalable architecture model in PARET was shown to illustrate its use.

### Acknowledgements

Thanks are due to Venkatesh Krishnaswamy for suggestions on the Linda machine model and to John Edmark for the display layout of the Linda machine model.

### References

1. Lawrence Snyder, "Parallel Programming and the Poker Programming Environment," *Computer*, vol. 17, no. 7, pp. 27-36, July, 1984.
2. Kevin Gates and David Socha, "Programming NCUBEs with a Graphical Parallel Programming Environment versus and Extended Sequential Language," *Proceedings of the Second Conference on Hypercube Multiprocessors*, pp. 17-27, Knoxville, TN, September 29, 1986.
3. Zary Segall and Larry Rudolph, "PIE: A Programming and Instrumentation Environment for Parallel Processing," *IEEE Software*, pp. 22-37, November, 1985.
4. Dharma P. Agrawal, Virendra K. Janakiram, and Grish C. Pathak, "Evaluating the Performance of Multicomputer Configurations," *Computer*, vol. 19, no. 5, pp. 23-37, May 1986.
5. ParaSoft Corporation, "Profile: A Profiling System for Parallel Computers," in *external communication*, ParaSoft Corporation, Mission Viejo, CA, 1988.
6. Howard P. Katseff, "The Software Oscilloscope: A Real-Time Execution Monitor for Multiprocessor Applications," *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 316-318, Madison, WI, May 5-6, 1988.
7. Kathleen M. Nichols and John T. Edmark, "Modeling Multicomputer Systems with PARET," *IEEE Computer*, vol. 21, no. 5, pp. 39-48, May, 1988.
8. Steven P. Levitan, "Evaluation Criteria for Communication Structures in Parallel Architectures," *1985 International Conference on Parallel Processing*, pp. 147-154, 1985.
9. Susan J. Eggers and Randy H. Katz, "A Characterization of Sharing in Parallel Programs and Its Application to Coherency Protocol Evaluation," *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pp. 373-382, Honolulu, HI, May 30 - June 2, 1988.
10. Steven E. Lucco and Kathleen M. Nichols, "A Performance Analysis of Two Parallel Programming Methodologies in the Context of MOS Timing Simulation," *Proceeding of the Spring 1987 Compcon*, February, 1987.
11. H.T. Kung, "When will the widespread use of parallel computers become a reality?," *Architecture Plenary*, IEEE Computer Society and IEEE Circuits and Systems Society, Rye Brook, New York, October 3, 1988.
12. Robert G. Babb II, "Parallel Processing with Large-Grain Data Flow Techniques," *Computer*, vol. 17, no. 7, pp. 55-61, July, 1984.
13. David C. DiNucci and Robert G. Babb, "Practical support for Parallel Programming," in *Proceedings of the Twenty-First Annual Hawaii Conference on System Sciences*, vol. 2, pp. 109-118, IEEE Computer Society Press, 1988.
14. V.P. Srini and B.D. Shriver, "A Methodology for Designing and Modeling Reconfigurable Systems," *International Journal of computer and Information Sciences*, vol. 13, no. 5, pp. 339-394, 1984.
15. C.V. Ramamoorthy and Gary S. Ho, "Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 5, pp. 440-449, September, 1980.
16. Tadao Murata, "Synthesis of Decision-Free Concurrent Systems for Prescribed Resources and Performance," *IEEE*

*Transactions on Software Engineering*, vol. SE-6, no. 6, pp. 525-530, November, 1980.

17. K. Hwang and J. Ghosh, "Hypernet: A Communication-Efficient Architecture for Constructing Massively Parallel Computers," *IEEE Transactions on Computers*, vol. C-36, no. 12, pp. 1450-1466, December, 1987.

18. John L. Gustafson and Gary R. Montry, "Programming and Performance on a Cube-Connected Architecture," *Digest of Papers: Spring Compcon*, pp. 97-100, San Francisco, CA, February 29,1988.

19. Lawrence Snyder and David Socha, "Poker on the Cosmic Cube: The First Retargetable Parallel Programming Language and Environment," *Proceedings of the International Conference on Parallel Processing*, pp. 628-635, August, 1986.

20. S. Ahuja, N.J. Carriero, D.H. Gelernter, and V. Krishnaswamy, "Matching Language and Hardware for Parallel Computation in the Linda Machine," *IEEE Transactions on Computers*, vol. 37, no. 8, pp. 921-929, August, 1988.

21. Kathleen M. Nichols and John T. Edmark, "PARET: An Integrated, Visual Tool for the Study of Parallel Systems," *Proceedings of the ICCD*, IEEE Computer Society and IEEE Circuits and Systems Society, Rye Brook, New York, October 3, 1988.

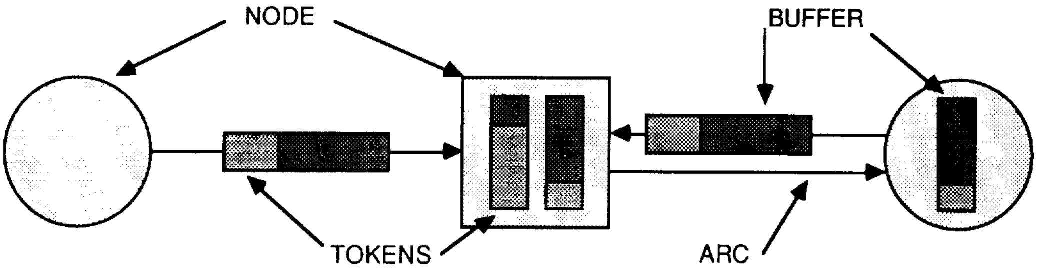


Figure 1. Components of a Model

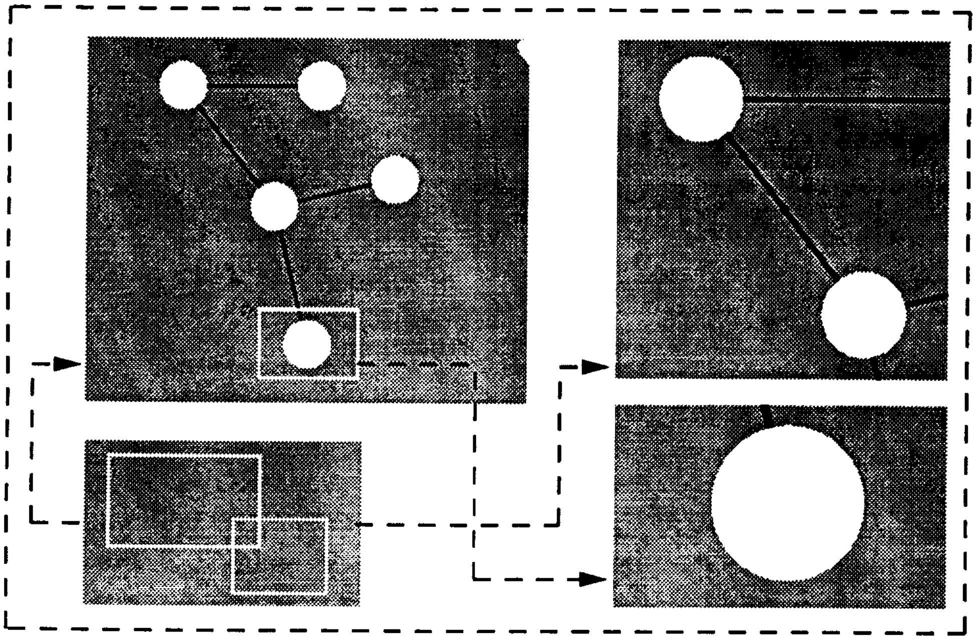


Figure 2. Reticle and Viewport Display

ORIGINAL PAGE
BLACK AND WHITE PHOTOGRAPH

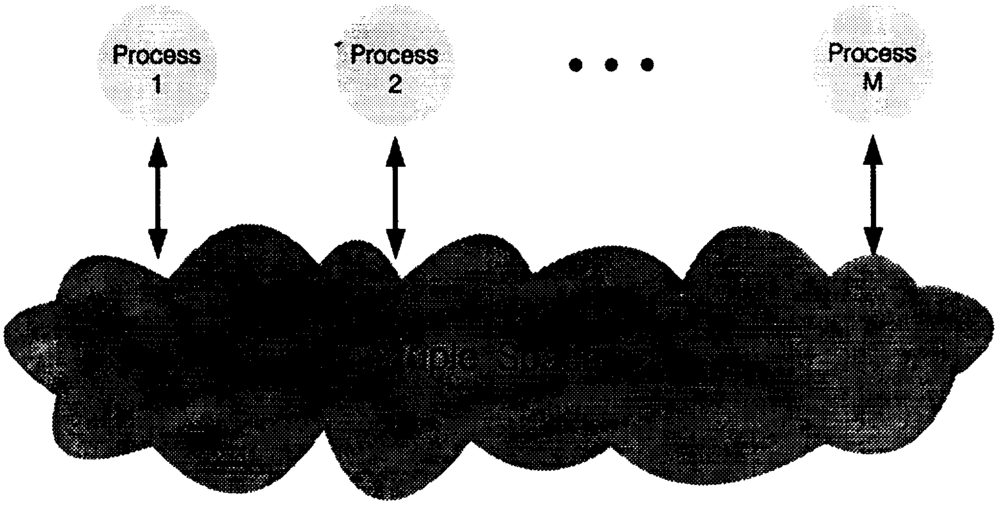


Figure 3. Linda Paradigm - Virtual Shared Memory

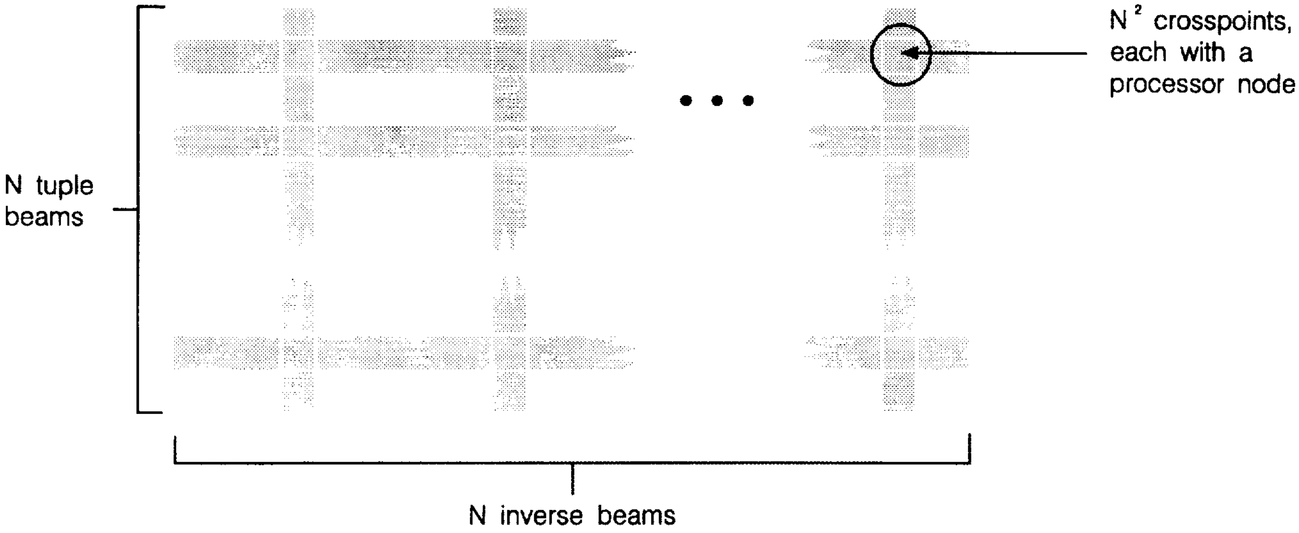


Figure 4. Linda Machine Architecture

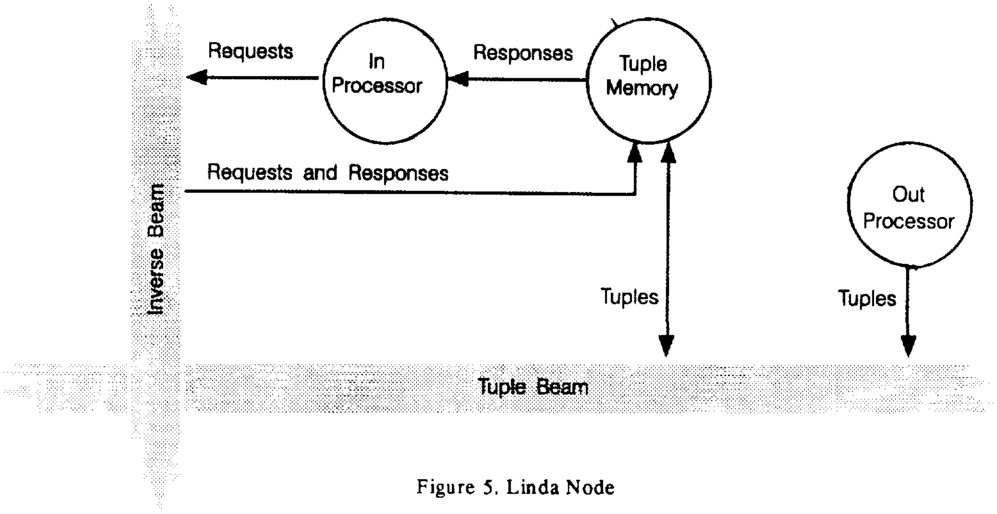


Figure 5. Linda Node

ORIGINAL PAGE
BLACK AND WHITE PHOTOGRAPH

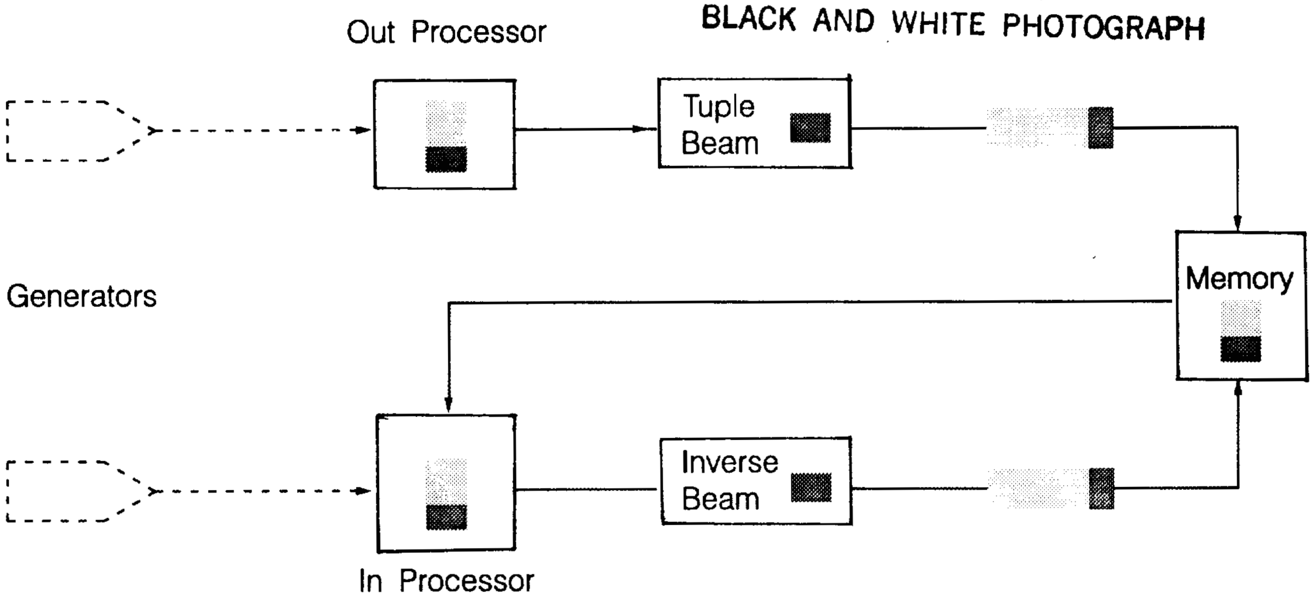


Figure 6. PARET Model of a Linda Node

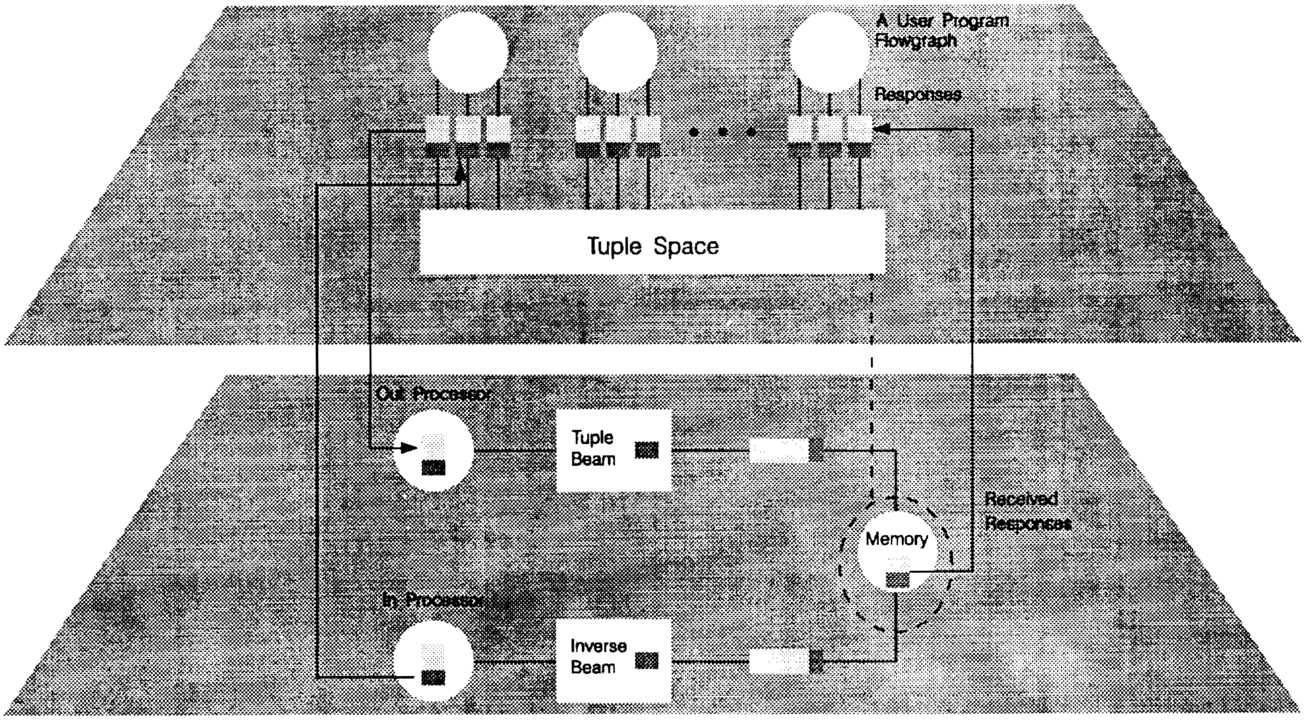


Figure 7. A More Complex System Model

# PROGRAMMING CONSIDERATIONS IN THE DESIGN AND USE OF A SIMD IMAGE COMPUTER

Allan L. Fisher
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Peter Highnam
Schlumberger Doll Research
Old Quarry Road
Ridgefield, CT 06877

## ABSTRACT

The CMU Scan Line Array Processor (SLAP) is a SIMD architecture designed for image computation and similar applications. A SLAP includes a serial control processor and a long vector of word-parallel processing elements. Such an architecture is compact, intrinsically capable of high clock rates, and scales very well with improvements in chip technology. In this paper we demonstrate that with a modicum of compilation complexity a SLAP becomes a versatile tool, efficiently supporting a number of useful programming models: *position-independent* (low-level image processing operations), *scan-line* (intermediate-level image processing and graphics), and *systolic*. We discuss the impact of programming issues on overall system architecture, specifically with respect to hierarchical control structures, the handling of concurrent I/O streams, and the importance of considering whole applications. Such considerations are critical to the success of highly parallel systems, particularly those designed for imbedded applications, yet often are treated as afterthoughts. The high level SLAP programming language SLANG is introduced. A prototype hardware and software system are under construction.

*Keywords:* compilation, conditionals, directionals, image-processing, linear array, programming models, real-time, SIMD.

## INTRODUCTION

There have been a large number of image computers, their architectures running the gamut of architectural diversity. The most common form of processor control found in such systems is SIMD. In general SIMD systems have had essentially a grid topology, from Unger's early work (Ref. 14) through more recent machines such as the DAP (Ref. 9), CLIP4 (Ref. 3), and the MPP (Ref. 11). More recent SIMD systems have either provided a much richer connection structure (the Connection Machine (Ref. 2)) or have elected to provide an even simpler topology, the linear array (AIS-5000 (Ref. 12), CLIP7A (Ref. 10), SLAP (Ref. 7)). All image computers have to deal with high bandwidth I/O requirements and the unusual programming models that their native mode presents. In this paper we examine both issues within the context of the CMU Scan Line Array Processor.

A Scan Line Array Processor (Refs. 6, 7) (SLAP) is a SIMD machine comprising a long linear array of word-parallel processing elements (PEs), whose instruction stream is organised and sequenced by a flexible conventional processor (the controller). The SIMD model and the point-to-point connectivity of the vector guarantee that communication between PEs is both synchronous and fast. The nearest-neighbor connectivity of the vector permits easy advantage to be taken of increased on-chip gate counts. Input, output and computation are pipelined, permitting high data bandwidth with almost zero cost in terms of lost processing cycles. The SLAP is intended to be a support processor for computer perception applications that require flexible, high throughput processing. Generally, the input is in the form of a continuous image stream. Images from optical sensors are currently typically 512 × 512 with 8 bit pixels, received scanline by scanline at a rate of 30 images per second. The prototype under construction includes custom 2$\mu$ CMOS processor chips. Each chip holds

This research was sponsored by the Defense Advanced Research Project Agency (DOD), ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory under Contract F33615-84-K-1520.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

CH2649-2/89/0000/0683$01.00 © 1988 IEEE

four processors. The prototype can deliver up to 500 sixteen bit integer operations per pixel while maintaining video-rate throughput. The prototype's peak (internal) aggregate bandwidth is approximately 16 Gbyte $s^{-1}$, with an execution peak of over 4 billion sixteen bit integer operations $s^{-1}$.

We describe the hardware and software architectures of a SLAP emphasising programming considerations. The common theme is the appropriate allocation of responsibility within the system. One result of this approach is that, in contrast to the majority of SIMD image computers, the SLAP has a two-level controller. The next Section contains a brief review of the system goals and premises. In Section we examine the hardware structure. Section includes an overview of the software architecture. A high-level language, *SLANG*, an augmented imperative language, has been developed for use with the SLAP. We conclude with a status review of the system that is now being built.

## GOALS AND PREMISES

A SLAP is an image computation accelerator for a general-purpose Host. As such the general purpose facilities it must provide are limited. Rather, a SLAP must be able to conveniently offload computations and return results with minimal interference to independent Host computation. We expect Host – SLAP interactions to be on the order of one video frame (about 30ms). This is a significant amount of time on a current workstation. The Host is probably ill-equipped to handle video rate I/O and definitely unable to provide the computation bandwidth required.

Certain characteristics of image computations have been set up as goals.

- *Data and computation rates.* Video rate throughput and video synchronisation is necessary. Concurrently there must be high enough instruction rate counts to perform interesting computations *online*.

- *Communication of results and parameters.* There must be a convenient mechanism for moving low bandwidth information between the Host and the SLAP.

- *Pixel-level processing abilities.* There is no need to provide floating point operations for the vast majority of image computations. Integers are appropriate. Provision of features to ameliorate the inflexibility of SIMD can significantly increase the machine's power. For example, sorting benefits greatly from local address computations and local context control.

- *Image storage and access.* Vision systems frequently need to be able to hold an image for several operations. Such facilities must be provided in an appropriate manner. A number of computations can achieve significant performance increases by the provision of transposed versions of an image. Speed gains can also be realised using region of interest selection. Numerous algorithms require access to several images simultaneously.

- *Feedback.* Iterative operations require that there be a mechanism summarising an aspect of computation state.

There are a number of successful image processing libraries (eg., SPIDER (Ref. 13)). The functionality of the routines in such a library meet the needs of a large number of applications. A successful system should also provide a library together with the appropriate "glue". Furthermore, particularly in a research environment, it must be easy to create efficient solutions that can be added to such a library. By this we mean that a high level language should be available, together with a compiler that can generate good quality code.

Unusual machines do not obviously provide familiar programming models. The SLAP has at its core a very long vector of SIMD processors. One of our goals in the SLANG language is to support programming models encouched in a fairly familiar high level imperative language and to make implicit the task of synchronisation with the concurrent I/O streams. Compilation for a parallel machine is not trivial. The SLANG language includes some novel concepts that appear to provide a compiler with a significant amount of information for easily realised optimisations.

In applications such as mobile robotics where the computation engine must be an integral component it is very important that the system be both physically compact and power-efficient. These are implicit in our focus on a VLSI implementation of a linear array.

## SYSTEM COMPONENTS

We describe the SLAP hardware architecture in terms of the requirements imposed on the system, how they can be dealt with by certain types of subsystem, and the control mechanisms needed at runtime. There are three major subsystems, the *controller*, the *sequencer* and the PE vector. The Host communicates directly

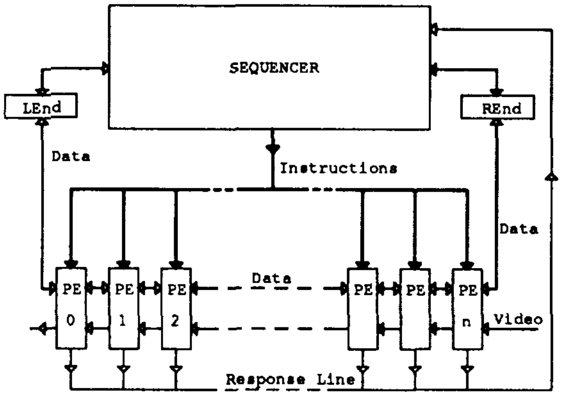


Figure 1. A SLAP System

only with the controller over a VME bus. Code and data memories (and a number of control registers) are mapped into the Host address space. Interaction at this level is coordinated on the controller by programs written in C. In our prototype system (Ref. 8) the controller is a Motorola 68030. A SLAP system sketch is shown in Figure 1.

The controller is not fast enough to provide the vector instruction sequencing which in the prototype is about 8 MHz. This task is performed by the sequencer. The sequencer also contains a 16 bit ALU constructed from bit-slice devices (IDT chip set). The sequencer is thus able to perform substantial runtime computation. The primary function of the sequencer requires that it operate in lockstep with the vector. Thus, the sequencer and the vector can (as in the ILLIAC IV (Ref. 1)) compute simultaneously, communicating data without any synchronisation overhead. In fact, by *mimicking* the operation set of a PE in the sequencer we have a very convenient mechanism for off-loading global runtime computations to the sequencer for concurrent execution with the vector. Data is broadcast to the vector by insertion into the instruction stream. There is also a bidirectional data link between the sequencer and the vector at the latter's extremes.

In the remainder of this section we will briefly describe the I/O subsystem, the image storage mechanisms, and the abilities of the Processing Elements.

## I/O

As explained elsewhere (Ref. 6) a SLAP can efficiently handle video-rate I/O within the PE vector. (Briefly, there is a distinct video route along the vector that can operate independently of instruction exection.) The pixel level synchronisation problem is avoided by the common clock of the SIMD system and the short point-to-point connections of the vector. Synchronisation at the row and frame level are handled by the sequencer. The appropriate video conditions are made available to the sequencer branch unit.

Communication with the Host is low bandwidth. The Host provides starting addresses and parameters, the SLAP returns low volume results. Host communication requires being able to communicate with whatever protocol the Host assumes. This is handled by the controller, programmed in the C language with the corresponding selection of libraries to draw upon.

## IMAGE STORAGE

Within the SLAP images are held in two different ways. There are video buffers under the direct control of the sequencer and there is a significant amount of memory within the vector itself. Commercial video buffers typically require separate boards and must be controlled via the Host bus. The cost of endowing the controller with the ability to directly control such buffers was held to be too high. The alternative way to control such buffers is to pass requests to the Host to perform the operations. This is unwieldy. The SLAP controller includes a number of custom video buffers. Each is able to hold one standard image ( 512 x 512 x 8 bits ), supports region of interest operations directly, and can be accessed transposed. Each buffer is physically compact and is under the direct control of the sequencer. The prototype SLAP includes four such buffers. Video rate I/O can be directed to and from the buffers.

Access to buffer images is fast, but there are operations that are able to proceed much faster than video rate. Thus, restricting image inputs and outputs to the buffers is an unnecessary impedance. There are a number of PEs per processor chip. Each such chip has an associated static RAM that is used to provide a secondary storage level for the resident PEs. In the prototype each PE has 8 KBytes of storage (permitting 16 images to be held within the vector). Access to this storage is achieved by co-opting the video and neighbor pathways of the vector, under sequencer control. This mechanism permits the PEs to operate on (and generate) images at a much finer operation grain than is efficiently permitted by the buffers.

## PROCESSING ELEMENT

The PEs perform the bulk of the computation. Pixel data arrives and leaves the PE via the video path across the vector. The shared instruction stream is broadcast to the PE chips by the sequencer. In the prototype the instructions are 27 bits wide. A single on-chip decoder expands the instruction to provide the control signals for the coresident PEs. All PEs operate using the same instruction clock as the sequencer, in the prototype this is 125 ns.

In contrast to the majority of modern SIMD systems a PE has a multi-bit word (16 bits in the protoype). This design decision lets the basic operation unit of a PE be a reasonable match to common precision needs. There are also a number of PE abilities that are reasonable to give a bit-parallel processor but not a bit-serial design. These include local addressing abilities, significant hardware assistance for multiplicative operations, and rotate/shift abilities. From the programming perspective these features provide a PE with significant operational flexibility and power.

A PE has a local register file (32 words in the prototype) that can use local or broadcast addresses. This addressing ability is also extended to the offchip image storage mentioned earlier. In the SIMD model the PEs are unable to branch independently. The bit-serial PE solution is to maintain a stack of condition bits whose top bit is used to determine whether or not a state change can take place. The stack values are manipulated by condition operators. Smaller SIMD machines (in PE count) such as the ILLIAC IV are able to centralise a similar facility. The SLAP PE supports structured conditionals using a *sleep counter*. A counter value of zero permits state changes within the PE's register file and offchip storage. An instruction field is dedicated to support this facility. This form of state control is efficient in both hardware and computation time, the conditional manipulation takes place concurrently with operations and has a resolution of a single instruction.

The PE has a rotate/shift unit (*RSU*) which can take both amount and direction of operation from the PE. This permits easy access to locally determined fields. The PE ALU performs full word additive and logical operations in a single instruction cycle. The ALU operation set includes variants on the arithmetic operations to support multi-precision operations with minimal overhead. Multiplicative operations are supported in the ALU by Booth-type logic. A 2 bit Booth step is used to perform a 16 x 16 multiply in about 8 cycles (generating a 32 bit result), for example. A division or modulos operation takes approximately 16 cycles. The multiplicative operations are actually sequenced by the on-chip decoder. The operation field of an instruction starts an operation by changing decoder state, the decoder then issues the appropriate control bits each cycle to the coresident ALUs until told to stop. During such an operation the instruction stream arriving at the chip can include operations for other PE components. Utilisation of this ability is expected to require no more compiler sophistication than a simple form of delayed branch analysis.

Neighbor communication is provided by a third functional unit in the PE datapath. The operations it supports transfer (and receive) a single word of data in one cycle. Additionally, longer sequences of neighbor communication steps can be performed using a dedicated instruction field. A feedback path is supported for the sequencer to sample global state. There is a single bit wired-OR line that appears within the PE as a single register. Writing a value to that register can change the value examined by the controller within a couple of cycles.

A single prototype PE is sketched in Figure 2. There are two 16 bit buses (**A** and **B**) and a dual-ported (read and write) register file. The three functional units read operands from **A** and **B**, placing the result word onto **B**. The functional units have operation sets that are symmetric in their use of the operand buses. The units are the integer ALU, RSU, and the communication unit. An instruction has three phases: place a value on **A**; compute a result value onto **B**; store from **B**. Each phase takes one cycle, three instructions can be executing (different stages) concurrently.

# PROGRAMMING

A Host process invocation of a SLAP operation employs a construct semantically similar to a routine call or thread. The Host program can be written in any reasonable language. The SLAP is controlled by a user-supplied C program, the *harness*. The harness is unable to directly address components of the PE vector state. Rather, the harness is responsible for invoking sequences of code modules that execute in the sequencer and the vector. A high level language, *SLANG* is provided for the modules. The decision to use C to link SLANG modules together removes the need to make SLANG a completely general purpose language that controls the SLAP system for an entire application. A module cannot invoke another module directly, this permits a module to have complete control of the sequencer and the vector. Data can be passed from one module to the next by parameters relayed via the harness and by leaving image data in the offchip memory and the external image buffers.

The SLAP is a highly parallel machine. The constructs of SLANG have been chosen to facilitate the expression and optimisation of image computations. The SLANG compiler generates code for the

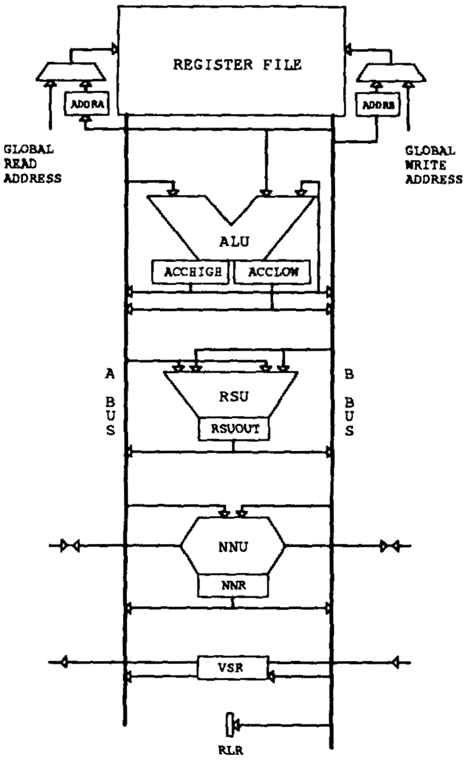


Figure 2. SLAP Processing Element

sequencer and the vector. The language is similar to C or Pascal with constructs and mechanisms to provide appropriate programming models and make parallelism and video synchronisation implicit. In this section we begin with a brief discussion of hierarchical control in image computation and then outline important aspects of SLANG and the compilation process.

## CONTROL

In applications that involve image computation one can reasonably identify a three level hierarchy of control. At the highest level are the broad algorithmic decisions. For example, whether to run filter X or filter Y. The variety and frequency (frame) of such decisions make the use of a general-purpose language and processor appropriate. In the SLAP system this role is carried out by the controller (a 68030) running a harness (in C).

During an image computation there are decisions to be made on every row (e.g., video synchronisation) or at least several times per frame (e.g., adapting a filter size near the image vertical limits). This type of decision involves significant changes in control flow. In the SLAP system this is appropriate for the sequencer. At the lowest level are the decisions that must be made on a per pixel basis (e.g., threshold testing). The SLAP PE is responsible for these decisions that must be made several times per scanline using local data. This latter class of decisions can be represented using control flow constructs that (in the SLAP PE) are mapped into conditional manipulations. They can frequently be mapped into arithmetic operations instead.

In SLANG the lower two levels of decision are implicitly identified by the compiler using information about the variables involved in the controlling expression. The lockstep synchronisation of the PEs and the sequencer greatly facilitate the mapping.

## COMPUTATION LOCALE

A SLANG variable is integer and can be scalar or array in the usual senses. A third attribute *locality* is also included in each variable declaration. A variable defined to be **local** has an instance in every PE. A variable defined to be **global** exists uniquely within the sequencer. This locality information is used by the compiler to place computations. Thus, if all the operands of an operator are **global** then the operation is scheduled to take place in the sequencer. If the operands are all **local** then the computation will be carried out using PE-local data at every PE. If the operand localities are a mix then the **global** operands will be broadcast to the PEs where the operation will take place. This notion of locality propagates through expressions in the obvious way.

In a SLANG program the locality of the expression controlling an **if** statement determines whether the sequencer's general branching mechanism or the PE conditional manipulation is used. The programmer does not need explicitly distinguish the two in any other way. The iterative SLANG constructs, **while**, **repeat** and **for** all require **global** controlling expressions.

The compiler employs locality information in optimisation. Constant propagation is a conventional compiler optimisation that permits the computation of constant expressions during compilation. By using SLANG locality information this technique can be carried one step further to place computations that might otherwise have been carried out on the vector into the sequencer (which mimics the PE ALU and RSU operation set). The sequencer and the vector perform arithmetic computations concurrently and in general this code motion is useful because most of the execution time is usually spent in the vector.

## COMMUNICATION

Communication of data laterally within the vector is expressed using a class of unary operators that we call *directionals*. Syntactically they appear as single word symbols, **left** and **right**. For example, "**left** *expr*" yields the value of the expression *expr* computed at the PE's left neighbor. There are two other directionals (**up** and **down**) useable within certain contexts as explained below. The set of four directionals have been demonstrated (Ref. 4) to provide a compiler for a SIMD machine with significant opportunities for optimisation.

## PROGRAMMING MODELS

The SLANG language provides three distinct models for programming image operations on a SLAP. The models differ in the way that video synchronisation and the specification of the computation are related. The first model is, in a sense, the most basic for the system. Input and output image(s) are held in offchip memory and the vector is able to perform random access to the data (i.e., the video pathway is not connected to external devices).

Access to the image buffers held within the controller and to "live" video sources and sinks requires significant sequencer assistance. There are two programming models that can achieve this in a parsimonious manner. The first model is a variant of the simple first scheme. The programmer writes code that is to be executed once per scanline. This code is then imbedded within a **scanline pass** statement. The interpretation of the resulting construct is that the scanlines of the input and output image(s) specified with the statement are to be transported to and from the appropriate PEs as required. In a sense this is the native mode for the SLAP system, capable of making good use of the pipelining of the video shift register in the vector.

A very large number of the low-level image-processing operations can be cast as "position-independent" programs. In this scheme a programmer writes code that executes at one pixel position. Within this code values that are computed at other positions can be specified using relative offsets. A single value generated by the code is the result value for the position. An interpreter is required to perform the correct mapping. The Apply language (Ref. 15), is of this type. The third SLANG programming model is a generalisation of this simple form, permitting the programmer to use the results of arbitrary computations at other grid positions (not just input pixel values). In the SLANG language, position independent programming is achieved using a **grid pass**.

Of the three image computation models we have found that all three have their place. Some computations simply cannot be phrased well as **grid** programs, but are easily written as **scanline** code (e.g., histogramming and the Hough Transform (Ref. 5)). The ability to write SLANG code that does not require even the implicit video synchronisation permits the programmer to realise the power of the vector unencumbered, as a systolic array with programmable cells. From the implementation viewpoint, a difference between the models is that the **scanline** model uses variables that are retained between real image scanlines, in the **grid** model each set of variables is implicitly declared anew for each pixel position. Use of a **pass** construct provides the compiler with a good deal of information about the structure of the computation defined.

## COMPILATION

We have already alluded to significant optimisation opportunities exploiting directional and **pass** information. Here we briefly examine the target machine for the SLANG compiler. Each SLANG module is invoked to run on a machine that generally retains very little state between modules. The code words generated include sequencer and PE instruction fields. The vector and the PE vector run in lockstep, making static allocation of tightly cooperative computations across the two subsystems straightforward.

The image memory in the vector is allocated (and deallocated) by software on the *controller* to provide image storage. Applications create images by invoking the controller routine which returns an address. This address is then passed to modules, identifying images. SLANG symbolic variables have to be allocated within the register files and memories in both the sequencer and the PEs. No special techniques are required.

The mapping of **pass** code to something amenable to code generation on the vector is straightforward for the **scanline** model but a little more complicated for the **grid**. The latter code is basically remapped so that all references to data that has not yet arrived are removed. The resulting code is then implemented using circular buffers.

## EXAMPLES

Two examples are shown below, keywords are shown capitalised. The first is a very simple SLANG module that thresholds an image "on-the-fly". As we mentioned above, this would be actually a very inefficient use of the SLAP, because the majority of the available computation cycles at the video rate are not being used.

```
SLANG simple_threshold ( t )
BEGIN
   VAR t : GLOBAL ;
   PASS ( SCANLINE ,  IN ( a EXTERN ) ,
                     OUT ( b EXTERN ) )
     b := a >= t
END.
```

The second example is more interesting, demonstrating fragments of the code for a 2D FFT-like computation, the Walsh-Hadamard computation. A harness component might be

```
  allocate_image( & start ) ;
  column_transform( start ) ;
  allocate_image( & t_start ) ;
  vector_transpose( start , t_start ) ;
  column_transform( t_start ) ;
```

The vector_transpose module transposes an image in place in the vector. That operation takes about one frame time. The SLANG column_transform module is sketched below. SLANG does not currently have routines, column_shuffle and butterfly are macros.

```
SLANG column_transform ( start )
BEGIN
   column_shuffle( start ) ;
   FOR level := 0 TO 8 DO
     FOR i := 0 TO 255 DO
       butterfly( start , level , i )
END.
```

# STATUS

The prototype's $2\mu$ CMOS chips, each of which contain four PEs have been fabricated and are now being tested. The boards, sequencer, and controller have been designed. A first version of the SLANG compiler has been built. It does not yet perform the **grid pass** mapping or perform substantial optimisations (a number of which have been tested independently). A compiler for Apply to SLANG mapping is close to completion.

# SUMMARY

We have briefly described a SIMD image manipulation machine and the fundamentals of a high level language for it. The system includes a computational component, called here the sequencer, that is not usually found in SIMD architectures. We have indicated how it is well exploited. The language appears to be able to support convenient programming models with very efficient mappings to the hardware. The Processing Elements are designed with novel features to add flexibility to an SIMD system.

# REFERENCES

1. G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes. The ILLIAC IV Computer. *IEEE Transactions on Computers*, C-17(8):746–757, August 1968.

2. Thinking Machines Corporation. *Connection Machine Model CM-2 Technical Summary*. Technical Report HA87-4, April 1987.

3. M. J. B. Duff. CLIP 4: a large scale integrated circuit array parallel processor. In *3rd International Joint Conference on Pattern Recognition*, pages 728–733, 1976.

4. A. L. Fisher and P. T. Highnam. Communication and code optimization in SIMD programs. In *International Conference on Parallel Processing*, August 1988.

5. A. L. Fisher and P. T. Highnam. Computing the Hough Transform on a Scan Line Array Processor. In *IEEE Computer Society Workshop on Computer Architectures for Pattern Analysis and Machine Intelligence*, October 1987.

6. A. L. Fisher and P. T. Highnam. Real-Time Image Processing on Scan Line Array Processors. In *IEEE Computer Society Workshop on Computer Architectures for Pattern Analysis and Image Database Management*, November 1985.

7. A. L. Fisher, P. T. Highnam, and T. E. Rockoff. Scan Line Array Processors : Work in Progress. In *DARPA Image Understanding Workshop*, April 1988.

8. A. L. Fisher and J. A. Zsarnay. System support for a VLSI SIMD image computer. In *IEEE Workshop on VLSI Signal Processing*, November 1988.

9. P. M. Flanders, D. J. Hunt, S. F. Reddaway, and D. Parkinson. Efficient high speed computing with the Distributed Array Processor. In *High speed computer and algorithm organization*, pages 113–127, Academic Press, 1977.

10. T. J. Fountain, K. N. Matthews, and M. J. B. Duff. The CLIP7A Image Processor. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(3):310–319, May 1988.

11. J. L. Potter, editor. *The Massively Parallel Processor.* The MIT Press, 1985.

12. L. A. Schmitt and S. S. Wilson. The AIS-5000 Parallel Processor. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(3):320–330, May 1988.

13. H. Tamura, S. Sakane, F. Tomita, and N. Yokoya. Design and implementation of SPIDER – a transportable image processing software package. *Computer Vision, Graphics and Image Processing*, 23(3):273–294, 1983.

14. S. H. Unger. A computer oriented towards spatial problems. *Proceedings of the IRE*, 46:1744–1750, 1958.

15. R. S. Wallace, J. A. Webb, and I. C. Wu. Machine-independent image processing: performance of Apply on diverse architectures. In *3rd International Conference on Supercomputing*, May 1988.

# ILLUSTRATIONS

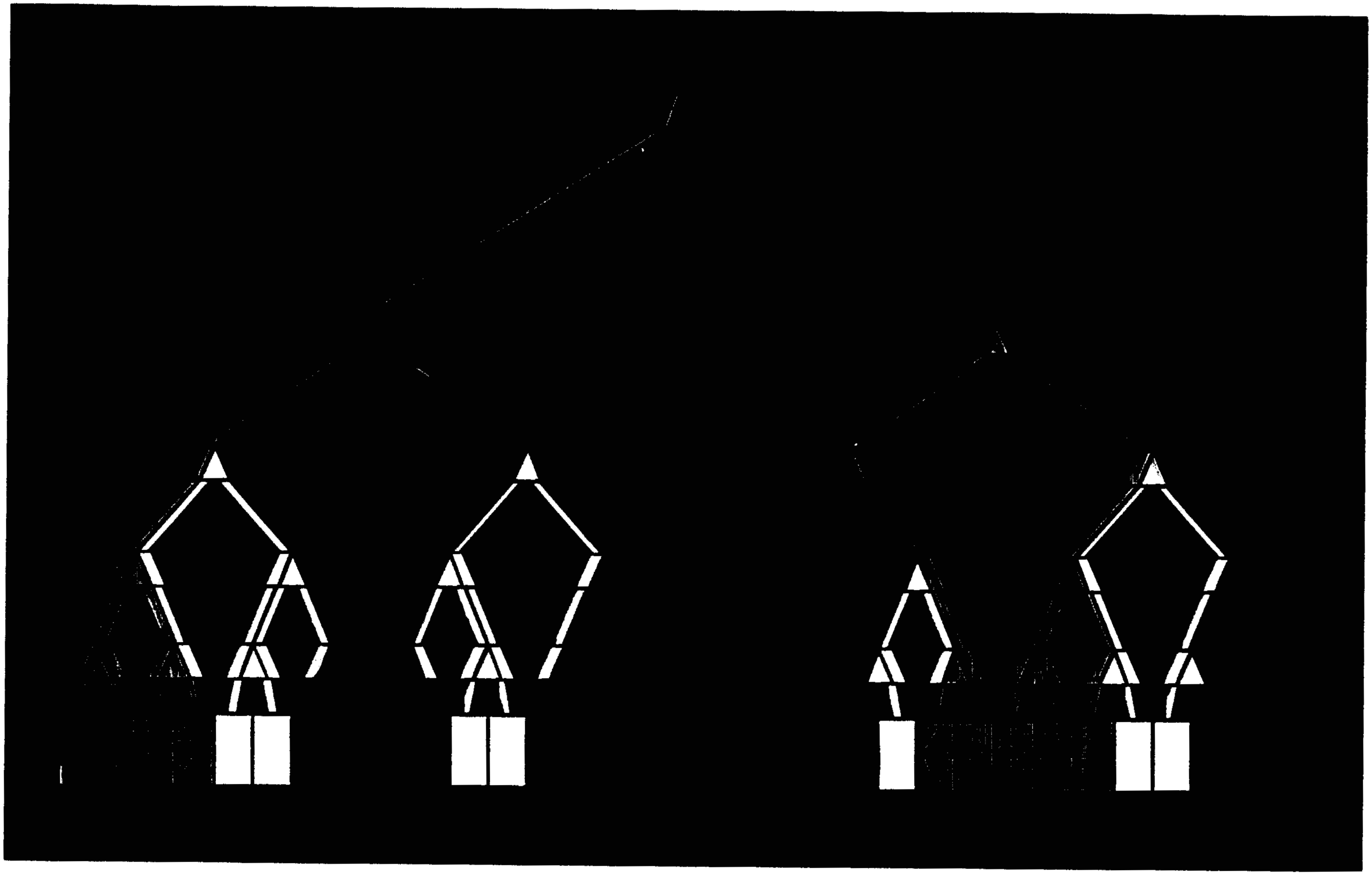


Figure 2
The result of partitioning, emphasizing the distinction between the physical tree structure and the tree structures of the resource groups.

**Plate I.** Dynamically Allocating Sets of Fine-Grained Processors to Running Computations

**Plate II**. Scan Line Graphics Generation on the Massively Parallel Processor

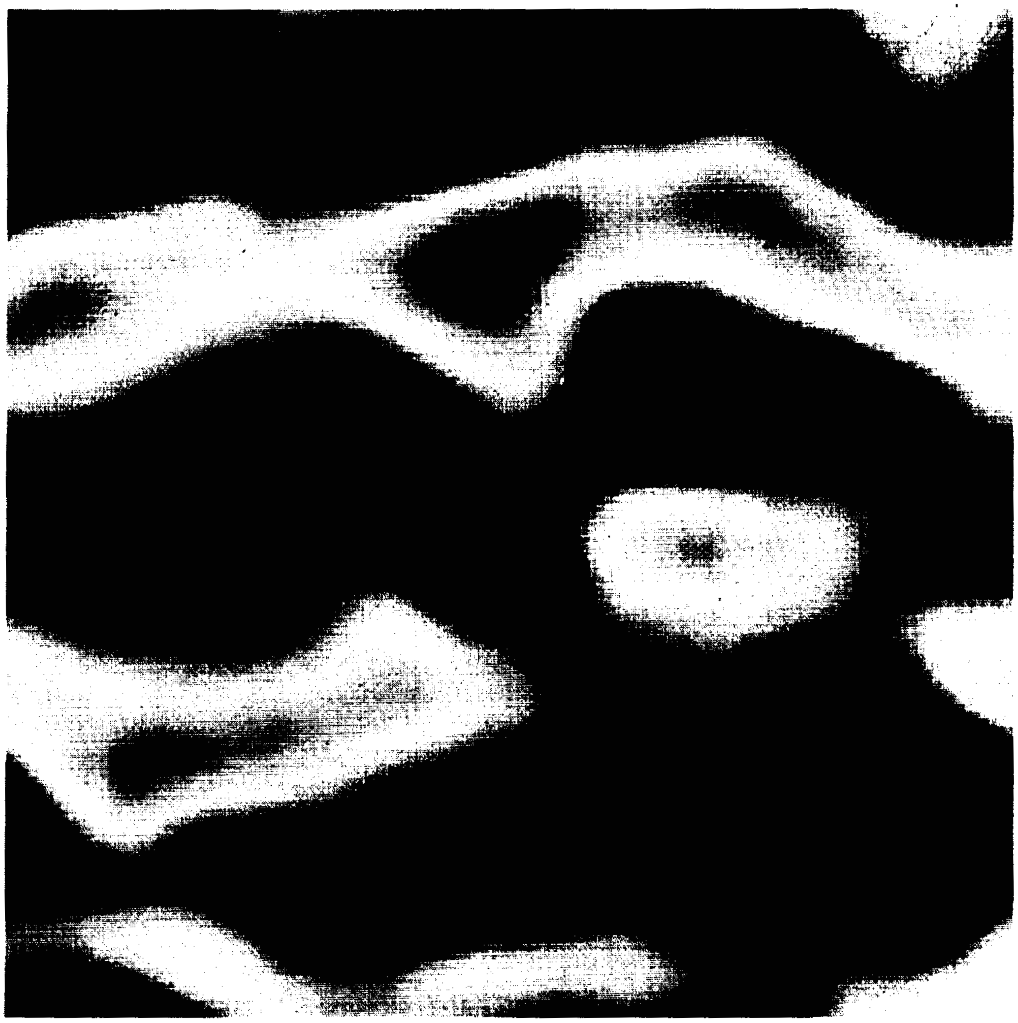


Figure 5. Color contour plot of electric potential during a beam plasma instability.

**Plate III**. A Parallel Particle-In-Cell Model for the Massively Parallel Processor

ORIGINAL PAGE
COLOR PHOTOGRAPH

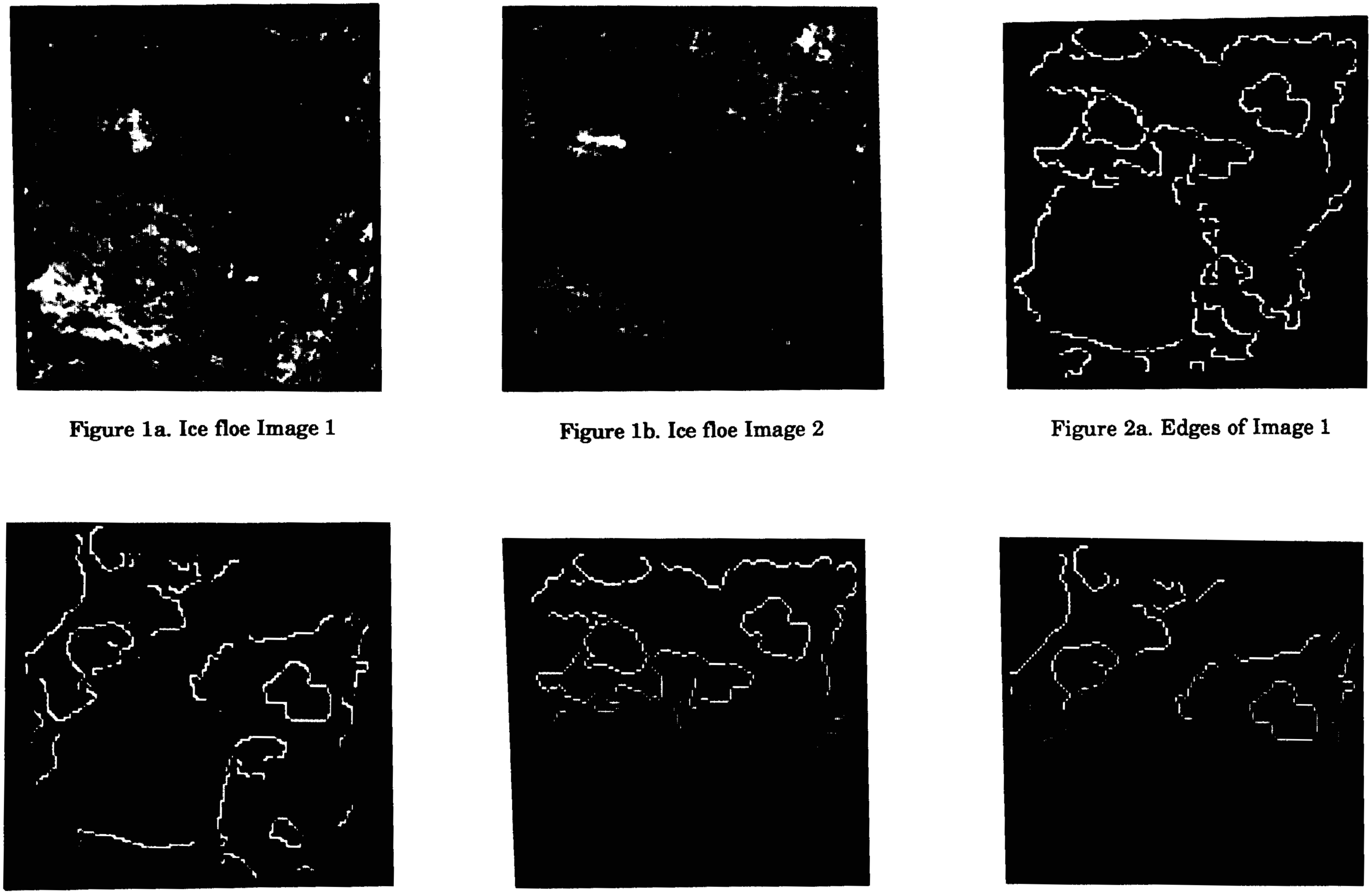


Figure 1a. Ice floe Image 1

Figure 1b. Ice floe Image 2

Figure 2a. Edges of Image 1

Figure 2b. Edges of Image 2

Figure 3a. Connected components of Image 1

Figure 3b. Connected components of Image 2

ORIGINAL PAGE
COLOR PHOTOGRAPH

Plate IV. Parallel Algorithm for Determining Motion Vectors in Ice Floe Images by Matching Edge Features I

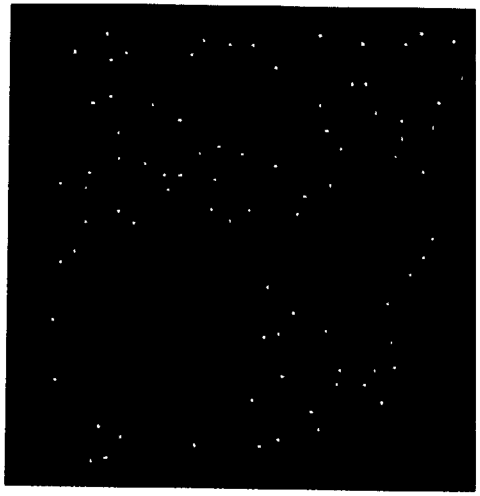


Figure 4a.Corners of Image 1

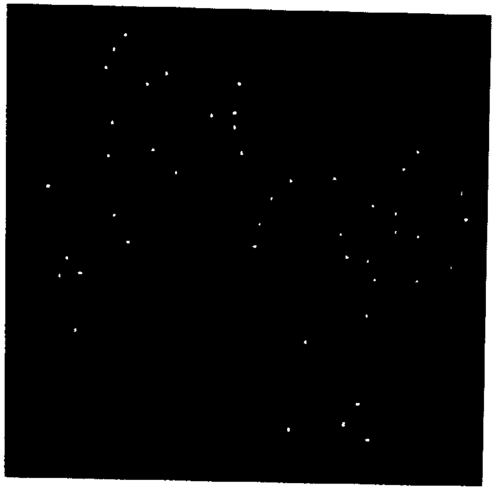


Figure 4b. Corners of Image 2

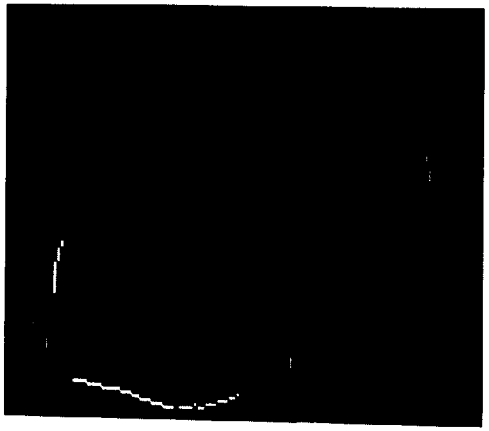


Figure 5a. LEN of Image 1

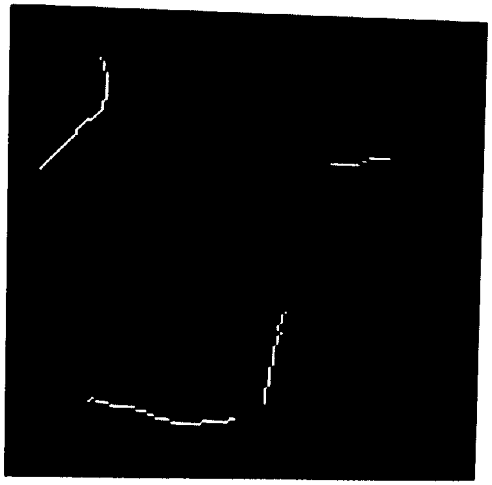


Figure 5b. LEN of Image 2

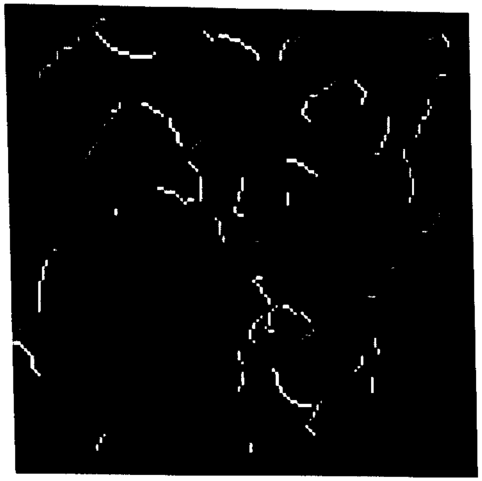


Figure 6a. DIR of Image 1

Figure 6b. DIR of Image 2

**Plate V.** Parallel Algorithm for Determining Motion Vectors in Ice Floe Images by Matching Edge Features II

ORIGINAL PAGE
COLOR PHOTOGRAPH

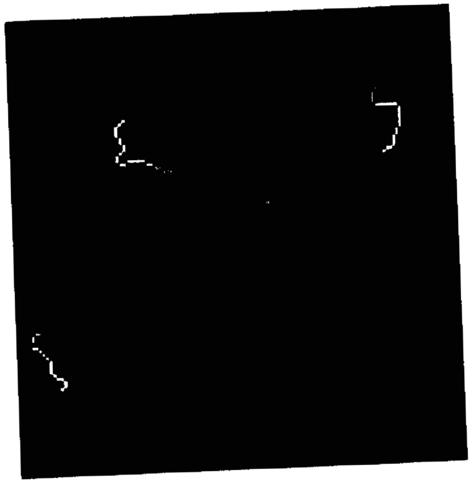


Figure 7a.SEQ of Image1

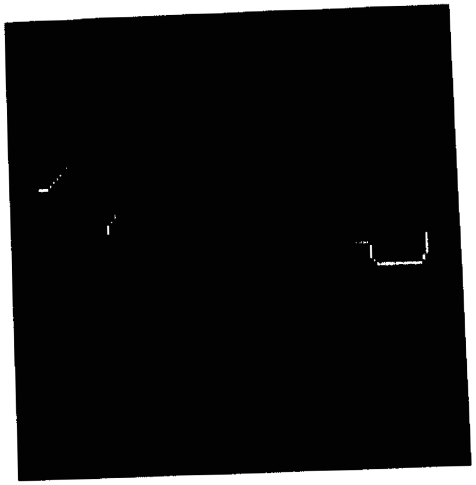


Figure 7b. SEQ Image 2

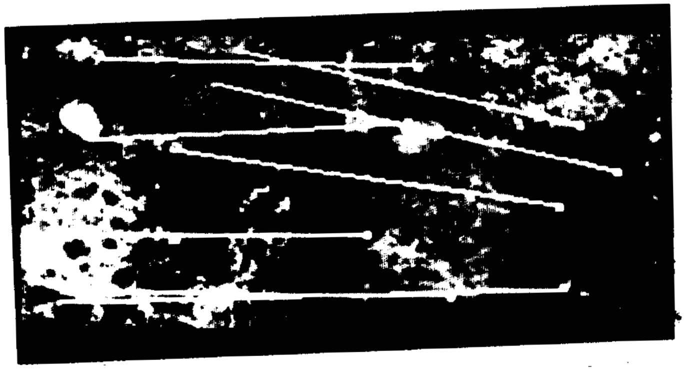


Figure 8. Matched pairs of Image 1 and Image 2

Figure 9a. Test Image 1

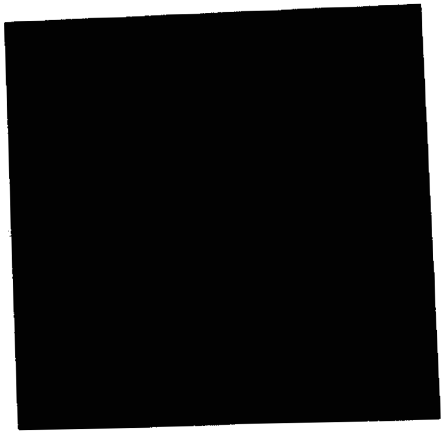


Figure 9b. Test Image 2

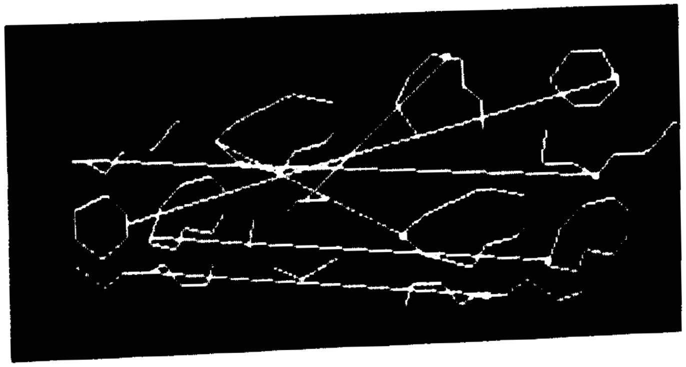


Figure 10. Matched pairs of Test Image 1 and Test Image 2

ORIGINAL PAGE
COLOR PHOTOGRAPH

**Plate VI**. Parallel Algorithm for Determining Motion Vectors in Ice Floe Images by Matching Edge Features III

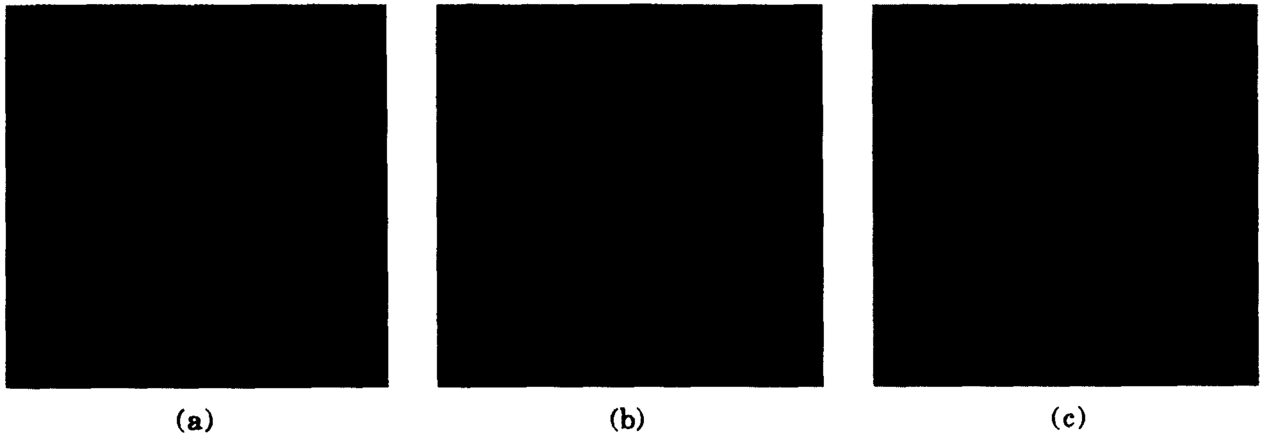


**Figure 1.** (a) Original TM image. (b) Segmented TM image. (c) Difference between (a) and (b) plus a bias (to create a positive image).

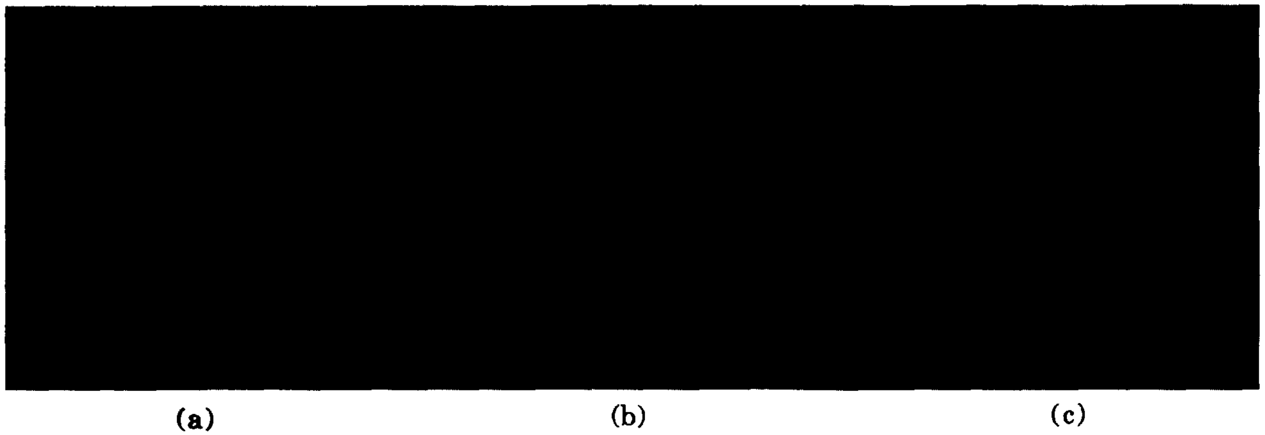
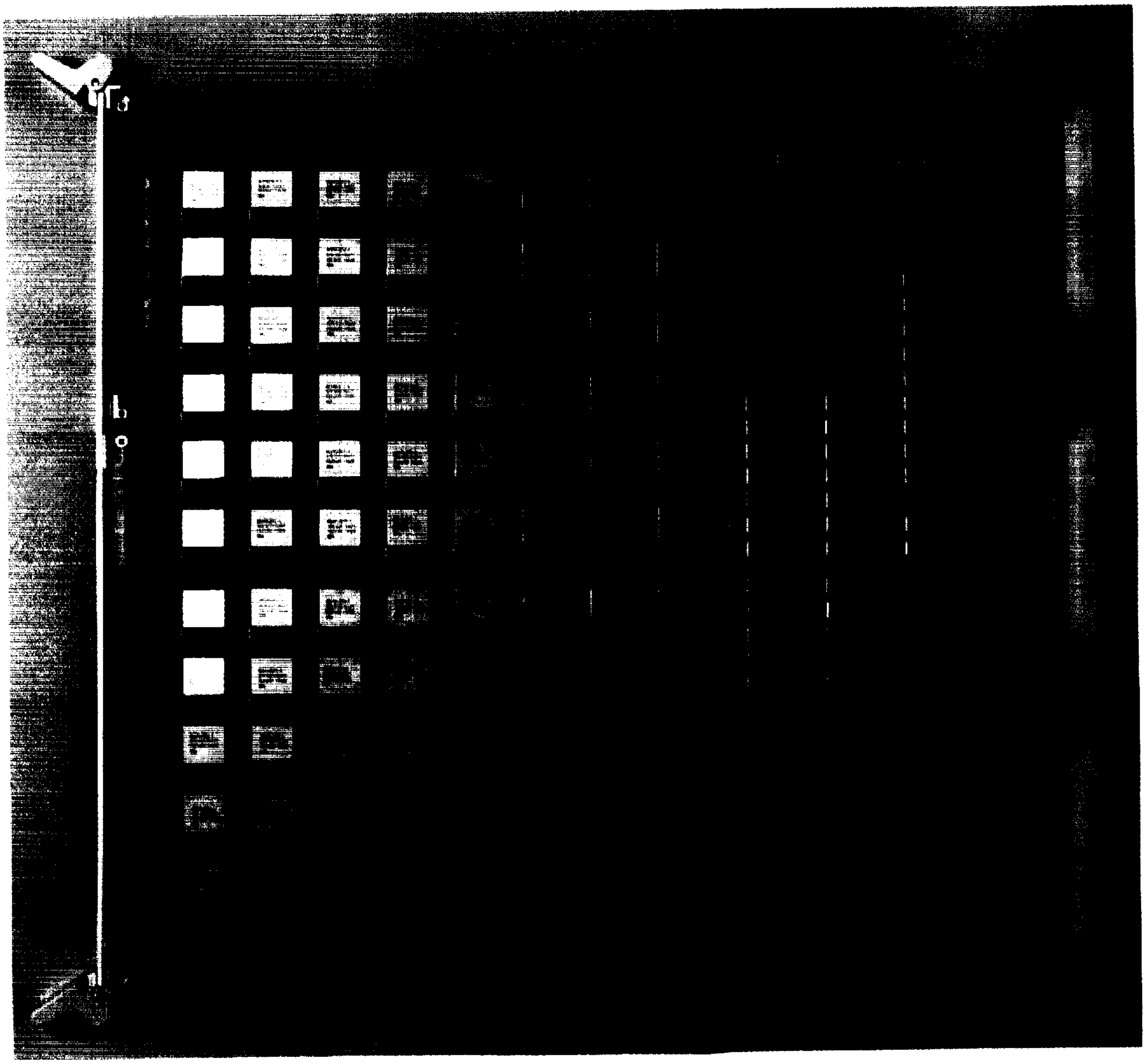


Figure 2. **Pseudo-colored class label maps with "water/marsh" shown as blue, "forest" shown** as orange, "residential" shown as green. (a) Bayes classification of the original TM image. (b) Ground truth map. (c) Bayes classification of segmented TM image.

**Plate VII.** Image Segmentation by Iterative Parallel Region Growing With Applications to Data Compression and Image Analysis

ORIGINAL PAGE
COLOR PHOTOGRAPH



GAPP Cell

**Plate VIII**. The Geometric Arithmetic Parallel Processor I

ORIGINAL PAGE
COLOR PHOTOGRAPH

Martian Terrain

**Plate IX.** The Geometric Arithmetic Parallel Processor II

# ATTENDEES

PRECEDING PAGE BLANK NOT FILMED

# FRONTIERS '88 ATTENDEES LIST

M. Abdelguerfi
University of Detroit
Engineering Dept.
4001 West McNichols
Detroit, MI 48221

Maurice Aburdene
Bucknell University
Electrical Engineering Dept.
Lewisburg, PA 17837

Raj Acharya
SUNY at Buffalo
Electrical & Computer Engineering
Buffalo, NY

Gus S. Adhar
University of Maryland
Baltimore, MD 21228

Merigot Alain
Un. Paris Sud
I.E.F. - Bat 220
Orsay 91405
France

Ted Albert
U.S. Geological Survey
806 National Center
Reston, VA 22092

Bruce Alper
Active Memory Technology, Inc.
16802 Aston St., Suite 103
Irvine, CA 92714

Ned Anderson
Digital Equipment Corp.
75 Reed Rd.
Hudson, MA 01749

Paul B. Anderson
Planning Research Corp.
3521 Launcelot Way
Annandale, VA 22003

Gary Anthes
Federal Computer Week
3110 Fairview Park Drive
Falls Church, VA 22046

Mark Antonelli
DEC
77 Reed Rd.
Hudson, MA
(508) 568-5607

R.A. Athale
BDM
7915 Jones Branch Dr.
McLean, VA 22102

James Lee Bagby
George Mason University
1805 Post Oak Terrace
Reston, VA 22091

Chuck Baldwin
University of South Carolina
CPSA, Computer Services
Columbia, SC 29208

Alex L. Bangs
Institute for Defense Analyses
1801 N. Beauregard St.
Alexandria, VA 22311

Suzanne Banqhart
Dept. of Defense
9800 Savage Rd.
Ft. George G. Meade, MD 20755-6000

Herb S. Barad
Tulane University
Department of Electrical Engineering
News Orleans, LA 70118

Juan Bargagna
George Mason University
3965 Fairfax Sq.
Fairfax, VA 22031

John A. Barnden
Computing Research Lab
New Mexico State University
Box 30001
Las Cruces, NM 88003

Stan Barouch
Fed. Comp. Week
3110 Fairview Park Dr.
Falls Chuch, VA 22046

Bill Bass
Hughes Aircraft Co.
1768 Business Center Dr.
4th Floor
Reston, VA 22090

Don Becker
Harris Corporation
P.O. Box 37, MS 3A-1912
Melbourne, FL 32902

Robert W. Bee
Automation Research Systems
4480 King St., Suite 500
Alexandria, VA 22302

Carol Bee-Latty
Thinking Machines Corporation
4705 Langdrum Ln.
Chevy Chase, MD 20815

Don Blevins
MCNC
3021 Cornwallis Rd.
Research Triangle Park, NC 27709

Judith O. Berkey
George Mason University
9123 Peabody St.
Manassas, VA 22110

Semyon Berkovich
Allied-Signal Aerospace Tech. Center
9140 Old Annapolis Road
MD 108
Columbia, MD 21045-1998

Philip J. Bernhard
Clemson University
405 College of Nursing Building
Clemson, S.C. 29634

H. Scott Berryman
Yale University
51 Prospect
New Haven, CT 06158

Steve Bershader
MRJ Inc.
10455 White Granite Dr.
Oakton, VA 22124

Hugo F. Bezdek
NOAA/AOML
4301 Rickenbacker Causeway
Miami, FL 33149

Lloyd I. Biscomb
Vitro Corp.
4452 Burlington Pl,. NW
Washington, D.C. 20016

John Blakemore
Texas Instruments, NPDD
P.O. Box 660246, MS 8605
Dallas, Texas 75266

T. William Blank
Masspar, Inc.
2840 San Tomas, Suite 140
Santa Clara, CA 95051

Guy E. Blelloch
MIT Artificial Int. Lab.
Cambridge, MA 02139

John Blinka
The Goodyear Tire & Rubber Co.
P.O. Box 3531
Akron, OH 44309-3531

James Blue
National Institute of Standards & Technology
Gaithersburg, MD 20899

Bruce Blundell
US Army Engineer Topographic Lab.
ATTN: Sl-TE
Bldg. 2592
Fort Belvoir, VA 22060-5546

Eugene C. Boman
United Technologies Research Center
MS 129-47 Silver Lane
East Hartford, CT 06118

Dr. Haran Boral
MCC
3500 West Balcones Center Drive
Ausin, TX 78759

Kevin W. Bowyer
University of South Florida
Dept. of Computer Science
Tampa, FL 33620

Stephen W. Boyack
Dept. of Defense
9741-202 Clocktower Lane
Columbia, MD

Lance Boyd
Mercury Computer Systems
7855 Guilford Rd.
Suite 120
Columbia, MD 21046

Larry Boyer
Naval Research Lab
Code 4691
Washington, DC 20375

Jane Brandenburg
George Mason University
8015 Orange Plank
Springfield, VA 22153

Moshe Braner
Cornell University
Theory Center
265 Olin Hall
Ithaca, NY 14853

Alfred B. Brenner
Supercomputing Research Center
4380 Forbes Boulevard
Lanham, MD 20706

A.R. Briggs
208 Farmgate Lane
Silver Spring, MD 20904

David Britton
Trident Systems, Inc.
3554 Chain Bridge Rd., Ste. 200
Fairfax, VA 22030

Alan Broder
The MITRE Corporation
7525 Colshire Drive
McLean, VA 22102

Richard N. Brooke
E-Systems - Melpar Division
11225 Waples Mill Rd.
Fairfax, VA 22030

Joe R. Brown
Martin Marietta Electronic Systems
P.O. Box 628007
Orlando, FL 32862

Daniel Broyles
George Mason University
5614 Rathbone Pl.
Springfield, VA 22151

Lisa A. Burgess
U.S. Geological Survey
804 National Center
12201 Sunrise Valley Dr.
Reston, VA 22092

Tim Busse
Computer Sciences Corporation
3160 Fairview Park Drive
Falls Church, VA 22042

Lowell Campbell
University of Idaho
Electrical Engineering
BEL 213
Moscow, Idaho 83843

Tim Cannon
George Mason University
5060 Queenswood Dr.
Burke, VA 22015-1529

Mark Carlotto
TASC
55 Walkers Brook Dr.
Reading, MA 01867

David A. Carlson
Supercomputing Research Center
4380 Forbes Blvd.
Lanham, MD 20706

Jay W. Chalmers
Los Alamos National Laboratory
M.S. B267
Los Alamos, NM 87545

Robert Chamberlain
Supercomputing Research Center
4380 Forbes Blvd.
Lanham, MD 20706

Mark E. Chambers
DOD
10558 Faulkner Ridge Circle
Columbia, MD 21044

Art Charlesworth
University of Richmond
Richmond, VA 23173

Chi-Yuan Chin
General Electric Company
Corp. Research Dev.
P.O. Box 8, KWC-528
Schenectady, NY 12301

G.E. Christensen
Dept. of Elec. Eng.
Washington University
St. Louis, Missouri 63130

Cecilia Chu
NORDA
Code 115, SSC, MS
Stennis Space Center, MS 39529

Nhi Anh Chu
The MITRE Corporation
7525 Colshire Drive
McLean, Virginia 22102-3481

Joseph Claman
U.S. Geological Survey
804 National Center
12201 Sunrise Valley Dr.
Reston, VA 22092

Raymond E. Cline
Sandia National Laboratories
P.O. Box 969
Livermore, CA 94551-0969

Eugene L. Cloud
Martin Marietta Electronic Systems
MP 1304
Orlando, FL 32862

David M. Cohen
Dept. of Computer Sciences
101 MacLean Hall
University of Iowa
Iowa City, Iowa 52245

Jacques Cohen
Brandeis University
Comp. Science Dept.
South St.
Waltham, MA 02254

Ron Cook
Eastman Kodak
B82A Rm 342 Research Labs
Rochester, NY 14650-2163

Ronald Coleman
Polytechnic University
333 Jay Street
Brooklyn, NY 11201

John B. Corliss
NASA/GSFC
Code 635
Greenbelt, MD

Rob Creecy
U.S. Census Bureau
SRD 3215-4
Washington, DC 20233

Daniel H. Cress
Waterways Experiment Station
Box 631
Vicksburg, MS 39180

Charles Croft
Geodynamics
P.O. Box 5267
Springfield, VA 22150

Clifford Cummings
Jet Propulsion Laboratory, M/S 382E
1500 Planning Research Drive
McLean, VA 22102

Thomas Curtis
CDSI
5443 Taussig Rd.
Bladensburg, MD 20710

Ernest Daddio
STX
4400 Forbes Blvd.
Lanham, MD 20706

Charles A. Dallas
Martin Marietta Astronautics
P.O. Box 179, M.S H4372
Denver, CO 80201

Edward W. Davis
North Carolina State University
Dept. of Computer Science
Box 8206
Raleigh, N.C. 27695

Marvin Denicoff
Thinking Machines Corporation
245 First Street
Cambridge, MA 02142

Etienne Deprit
Naval Research Lab, Code 8242
4555 Overlook Ave.
Washington, D.C. 20375

Victor V. Derefinko
Eastman Kodak Com
1447 St. Paul St.
Rochester, NY 14653-7205

Judith Devaney
STX
4400 Forbes Blvd.
Lanham, MD 20706

John DiFrancesco
SAIC
26679 West Agoura Rd., Suite 200
Calabasas, CA 91302

Susan R. Dickey
New York University
Courant Institute
251 Mercer St.
New York, NY 10012

Carl F. Diegert
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185

Kevin Doherty
E-Systems Melpar Division
7700 Arlington Blvd.
Falls Church, VA 22046

Dr. Jack Dongarra
Argonne National Laboratory
Argonne, IL

John E. Dorband
NASA/Goddard
Code 635
Greenbelt, MD 20771

Richard N. Draper
Supercomputing Research Center
4380 Forbes Blvd.
Lanham, MD 20706

Therese A. Dumoulin
Naval Surface Warfare Center
Code G45
Silver Spring, MD 20903-5000

Dr. James A. Earl
University of Maryland
Dept. of Physics & Astronomy
College Park, MD 20742

Mark Edwards
GTE Gov't. Systems
1700 Research Blvd
Rockville, MD 20850

P.J. Edwardson
Condensed Matter Physics Branch
Naval Research Lab.
Washington, DC 20375

Kemal Efe
Univ. of Southwestern Louisiana
CACS, USL, 2-Rex Street
P.O. Box 44330
Lafayette, LA 70504-4330

Robert Ehrlich
George Mason University
Physics Dept.
Fairfax, VA 22030

Mohamed El-Sharkawy
Bucknell University
Dana Building
Lewisburg, PA 17837

Stephen Engel
DEC
75 Reed Rd.
Hudson, Mass.

Paul M. Engelhart
Rome Air Development Center
COEE
Griffiss AFB, NY 13441

Frederick H. Esch
US Army Engineer Topographic Lab.
ATTN: SL-T
Bldg. 2592
Fort Belvoir, VA 22060-5546

Ronald Estes
STX
4400 Forbes Blvd.
Lanham, MD 20706

David Evans
Visionary Systems Inc.
25 Science Park
New Haven, CT 06511

Ai Chun Fang
NASA HQ, Code EI
600 Independence Ave., SW
Washington, DC

Donald W. Fausett
Florida Institute of Technology
316 George St., S.W.
Vienna, VA 22180

James W. Feeney
IBM
1701 N. St., Dept. V71
Endicott, NY 13760

R.B. Ferguson
E.I. DuPont
600 Eagle Run Rd.
Newark, DE 19707

Carmelo Ferrigno
U.S. Geological Survey
804 National Center
12201 Sunrise Valley Dr.
Reston, VA 22092

Charles M. Fiduccia
General Electric R & D Center
Schenectady, NY 12309

Michael Lee Finney
FAX Group
Rt. 11, N.
Troutville, VA 24175

G.R. Fischer
University of Florida
CIS Dept.
Gainesville, FL 32611

James R. Fischer
NASA/GSFC
Mail Code 635
Greenbelt, MD 20771

Allan L. Fisher
Dept. of Comp. Science
Carnegie Mellon Univesity
Pittsburgh, PA 15213

Jeff Fleischer
CIA
Washington, DC 20505

Judith L. Flippen-Anderson
Naval Research Laboratory
Code 6030 - Naval Res. Lab.
Washington, DC 20375-5000

Mark D. Forest
U.S. Government
Rome Air Development Center
Rome, NY 13441

Jose Fortes
Purdue University
School of Electrical Engineering
W. Lafayette, IN 47907

Dick Foster
Myrias Research Corporation
900 Park Plaza
10611-98 Ave
Edmonton AB TSK-2P9,
CANADA

Scott Fouse
Technowledge Federal Systems, Inc.
501 Marin St., Suite 214
Thousand Oaks, CA 91360

Jeff Frank
Amber Engineering, Inc.
5756 Thornwood Drive
Goleta, CA 93117

Robert G. Frantz
TRW
M5-1027, 1 Space Park
Redondo Beach, CA 90278

R. Don Freeman
AT&T Bell Labs
Room 2B-302
600 Mountain Ave.
Murray Hill, NJ 07974

Richard F. Freund
Naval Ocean Systems Center
271 Catalina Blvd.
San Diego, CA 92064

Jeff Fried
Laboratory for Computer Science
MIT
Cambridge, MA 02139

Gideon Frieder
Syracuse University
School of Computer & Information Science
Syracuse, NY 13244

Ophir Frieder
Bell Comm. Research
445 South Street
MRE 2D297
Mooristown, NJ 07960-1960

Jerrard B. Gaertner
Price Waterhouse
National Auditing Services
1100, boul. Dorchester ouest
Montreal, CANADA H3B 2G4

M.M. Garber
Martin Marietta Elec. Systems
P.O. Box 555837, MP 1304
Orlando, FL 32855-5837

Steven Geldman
Time/Life Books
777 Duke St.
Alexandria, VA 22314

Bill J. Georges
Newport News Shipbuilding
4101 Washington Ave.
Dept. C62, B521-2
Newport News, VA 23607

Thomas E. Gerasch
Sparta Inc.
7926 Jones Branch Dr.
McLean, Virginia 22102

Jeffrey Gerstenberger
Eastman Kodak Research Labs
1999 Lake Avenue
Rochester, NY 14650-5075

Joydeep Ghosh
Univ. of Texas
ENS 516, U.T.
Austin, TX 78712-1084

Helen Gill
The MITRE Corporation
MS Z645, 7525 Colshire Drive
McLean, VA 22107

Walter Gleaton
U.S. Geological Survey
509 National Center
Reston, VA 22092

Dr. Maya Gokhale
Supercomputing Research Center
4380 Forbes Blvd.
Lanham, MD 20706

Alwyn E. Goodloe
UNISYS
P.O. Box 181
Quantico, VA 22134

Jesse M. Gordon
University of Michigan
Department of EECS
The University of Michigan
Ann Arbor, MI 48109-2122

T.R. Gowrishankar
George Mason University
4400 University Drive
Fairfax, VA 22030

T.W. Greenwood
IBM
8250 Vernon Dr.
Manassas, VA 22110

Robert Grondalski
Digital Equipment Corp.
75 Reed Rd.
Hudson, MA 01749

Chester Grosch
Old Dominion University
1130 Manchester Ave.
Norfolk, VA 23508

Peter D. Gross
Computer Sciences Corporation
3160B Fairview Park Drive
Falls Church, VA 22042

J. Anthony Gualtieri
NASA/GSFC
Code 635, Bldg. 28
Greenbelt, MD 20771

David Guarino
SAIC
5151 E. Broadway, Suite 900
Tucson, AZ 85716

Joseph Guzek
Hughes Aircraft
4th Floor
1768 Business Center Dr.
Reston, VA 22090

Dr. Milton Halem
NASA/GSFC
Code 630
Greenbelt, MD 20771

J. Storrs Hall
Rutgers University
Dept. of Computer Science
New Brunswick, NJ 08903

Russell Hall
Merrill Pickard Anderson & Byre
2 Palo Alto Sq., Suite 425
Palo Alto, CA 94306

Magnus M. Halldorsson
Rutgers University
40 Marvin Lane
Piscataway, NJ 08854

Lisa Hamet
NASA/Goddard Space Flight Center
Code 635
Greenbelt, MD 20771

John W. Hansen
MRJ, Inc.
10145 White Granite Dr., Suite 200
Oakton, VA 22124

Dr. James Hardy
Whitney/Demos Productions
Culver City, CA

Max Halperin
Stanford University
Stanford, CA 94305

Dr. Joe M. Harris
Sandia National Laboratories
P.O. Box 969
Livermore, CA 94551-0969

Jonathan Harris
Digital Equipment Corp.
75 Reed Rd.
Hudson, MA 01749

Wendy Harris
Northeast Parallel Architectures Center
Syracuse University
250 Machinery Hall
Syracuse, NY 13244-1260

Harold M. Hastings
Hofstra University
Dept. of Mathematics
Hempstead, NY 11550

Anton J. Haug
Martin Marietta Aero & Naval Systems
103 Chesapeake Pk. Plaza
MPE 435
Baltimore, MD 21046

Robert Heaton
Microelectronics Center of North Carolina
Cornwalis Rd.
P.O. Box 12889
RTP, NC 27709

Jean Hecquard
SUNY at Buffalo
525 Minnesota Avenue
Buffalo, NY 14215

Paul J. Hedrick
Temple University
1525 Heather Hollow Circle #11
Silver Spring, MD 20904

Daniel B. Heifetz
Supercomputing Research Center
4380 Forbes Blvd.
Lanham, MD 20706

Paul Hertz
Naval Research Lab
Code 4121.5
Washington, DC 20375

Joe Hicklin
20 Main
Wayland, MA

Mac Hicks
MITRE
6103 Ivy Lane
Suite 304
Greenbelt, MD 20770

Peter Highnam
Schlumberger Doll Research
Old Quarry Rd.
Ridgefield, CT 06877

Ping Ho
STX
4400 Forbes Blvd.
Lanham, MD 20706

Stephen W. Hodson
Los Alamos National Laboratory
Ms B267
Los Alamos, NM 87545

R. Michael Hord
MRJ, Inc.
10455 White Granite Dr.
Oakton, VA 22124

Regan E. Howard
Fairchild Space Co.
20301 Century Blvd., MS A-10
Germantown, MD 20874

Geoffrey A. Howe
Rensselaer Polytechnic Institute
1C Russet Oaks
Ballston Lake, NY 12019

W.J. Hsu
MSU
A-734 Wells Hall, MSU
E. Lansing, MI 48824

Karen Humes
University of Arizona
Dept. of Hydrology and Water Resources
Tucson, AZ 85721

Paul Hunt
Martin Marietta
6633 St. Martin Pl.
Orlando, FL 32812

Paul Hunter
Naval Research Lab
Code 5305
Washington, DC 20375-5000

Bev Huntsberger
University of South Carolina
CPSA, Computer Services
Columbia, SC 29208

Terrance Huntsberger
University of South Carolina
Department of Computer Science
Columbia, SC 29208

Jon T. Hurd
David Sarnoff Research Center
CN 5300
Princeton, NJ 08543-5300

Peter Hyde
Space Telescope Science Inst.
Johns Hopkins University
3700 San Martin Dr.
Baltimore, MD 21218

G. Iazeolla
University of Rome
Electronic Engineering Dept.
via O. Ratmondo
00173 Roma,
Italy

Richard Iliff
USAF/SDIO/S/ES
The Pentagon
Washington, DC 20301

Kenneth Iobst
Supercomputing Research Center
4380 Forbes Blvd.
Lanham, MD 20706

David Izraelevitz
The Analytic Sciences Corporation
55 Walkers Brook Dr.
Reading, MA 01867

Daniel Jablonski
Supercomputing Research Center
4380 Forbes Blvd.
Lanham, MD 20706

Laura L. Jackson
Mississippi State University
Dept. of Elec. Eng.
P.O. Drawer EE
Mississippi State, MS 39762

Elaine M. Jacobson
Supercomputing Research Center
4380 Forbes Blvd.
Lanham, MD 20706

Kenneth P. Jacobson
Mass Par Inc.
2840 San Tomas Expwy, #140
Santa Clara, CA 95051

Joseph Jaja
Univ. of Maryland
Electrical Eng.
College Park, MD 20762

Mark Jankins
The Aerospace Corp.
MI-102, P.O. Box 92957
Los Angeles, CA 90009

Anne Jansen
Washington Technology
1953 Gallows Rd.
Suite 130
Vienna, VA 22180

Keith Johnson
Amber Engineering, Inc.
5756 Thornwood Drive
Goleta, CA 93117

Brian L. Jordan
5414 Charleston Woods Dr.
Fairfax, VA 22032

Dr. G.R. Joubert
Ned. Philips' Bedr. B.V.
Building HKF-p826 P.O. Box 218
5600 MD Eindhoven
The Netherlands

Moon-Seog Jun
U.M.B.C.
912-#C Apt. Hooper Ave.
Baltimore, MD 21224

B. Kamgar-Parsi
University of Maryland
College Park, MD 20742

Behrooz Kamgar-Parsi
George Mason University
9338 Cherry Hill Rd., #103
College Park, MD 20740

Joel Kaplan
Morgan Stanley and Co.
1251 Avenue of the Americas, 19th floor
New York, NY 10020

O. Karmin
Dept. of Elec. Eng.
Washington University
St. Louis, Missouri 63130

Alan Karp
IBM
18100 Frederick Pike
Gaithersburg, MD 20879

Joel Katzin
SEDC
9170 Rumsey Rd.
Columbia, MD

James P. Kelley
The MITRE Corporation
7525 Colshire Drive
McLean, VA 22102

Robert G. Kennington
George Mason University
1311 St. Paul's Way
Crownsville, MD 21032-2115

D. Kim
Univ. of Southern Calif.
Dept. of Elec. Eng.-Sys.
Los Angeles, CA 90089-0781

Chung-ta King
New Jersey Inst. of Tech.
Dept. of CIS
Newark, N.J. 07102

W. Worth Kirkman
The MITRE Corporation
7525 Colshire Drive
McLean, VA 22102-3481

Camille S. Klein
9459 Shouse Dr.
Vienna, VA 22180

Todd A. Kline
Naval Air Development Center
Code 5051
Warminster, PA 18974

Stanley P. Knight
David Sarnoff Research Center
CN 5300
Princeton, NJ 08543-5300

Kathy Knobe
COMPASS
550 Edgewater Dr.
Wakefield, MA 01880

Phil Koopman
Carnegie Mellon Univ.
5551 Beacon St.
Pittsburgh, PA 15217

Marta Kosarchyn
Hewlett-Packard Company
5301 Stevens Creek Blvd
Santa Clara, CA 95052

Bill Kraynek
F.I.U.
Tamiami Traic
Miami, FL 33173

Sridhar Krishnamurthy
University of Maryland
Systems Research Center
College Park, MD 20770

R. Kruger
SAIC
5151 East Broadway, Suite 900
Tucson, AZ 85711

Bradley C. Kuszmaul
MIT, Thinking Machines
8 Florence Rd.
Waltham, MA 02154

Chris Kuszmaul
Masspar, Inc.
2840 San Tomas
Suite 140
Santa Clara, CA 95051

Christopher Landaver
The Aerospace Corp.
P.O. Box 92957
Los Angeles, CA 90009

Dr. Faiza Lansing
Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, CA 91109

Linda M. Lawson
East Tennessee State University
Mathematics Dept.
Box 22309A
Johnson City, TN 37660

Jacqueline Le Moigne-Stewart
Martin Marietta Laboratories
1450 South Rolling Road
Baltimore, MD 21227-3898

R.M. Lea
Brunel University
Kingston Lane
Uxbridge Middx UB83PH
UK

Steven Lederman
Supercomputing Research Center
4380 Forbes Blvd.
Lanham, MD 20706

Philip Lem
SI-TE ETL
Telegraph & Leaf Rds.
Fort Belvoir, VA 22060-5546

Dar-Ren Leu
University of Houston
Department of Computer Science
4800 Calhoun Road
Houston, TX 77004

Christopher Lewis
Naval Research Lab
4555 Overlook Ave.
Washington, DC 20375

John W. Lewis
GE CRCD
P.O. Box 8, Rm. KWD244
Schenectady, NY 12309

Yiging Liang
George Mason University
Rm. 103, S&T
4400 University Drive
Fairfax, VA 22030

Dr. Chin S. Lin
Southwest Research Institute
6220 Culebra Rd.
San Antonio, TX 78238

J.E. Lin
George Mason University
Fairfax, VA 22030

John Lindelow
SETS, Inc.
1110 University Ave, #507
Honolula, HI 96826

Helen M Ling
NADC
Street Road
Warminster, PA 18974

Yue-Sheng Liu
New York University
251 Mercer St.
New York, NY 10012

Louis Lome
US Government/DOD
4513 N. 11th St.
Arlington, VA 22201

Dr. Lyle Long
Lockheed
Box 551, Dept. 7011
Burbank, CA 91520

G.M. Loseke
E-Systems, Inc.
P.O. Box 1056, CBN 132
Greenville, TX 75401

T. Michael Louden
The MITRE Corporation
7525 Colshire Drive
McLean, VA 22102-3481

Cheng-Chang Lu
Kent State University
Department of Mathematical Sciences
Kent, OH 44242-0001

Boris Lubachevsky
AT & T Bell Laboratories
600 Mountain Avenue
2C-121
Murray Hill, NJ 07974

Joan D. Lukas
Compass, Inc.
550 Edgewater Drive
Wakefield, MA 01880

William I. Lundgren
GE Aerospace
Bldg. 145, Rte 38
Moorestown, NJ 08057

Richard MacDonald
U.S. Geological Survey
804 National Center
12201 Sunrise Valley Dr.
Reston, VA 22092

Donald A. MacLennan
Hanovia, Division of Canrad, Inc.,
100 Chestnut Street
Newark, NJ 07105

Raj Kumar Mahadevan
University of Alabama
P.O. Box #6169
Dept. of Elec. Eng.
Tuscaloosa, AL 35486

Allen D. Malony
University of Illinois
305 Talbot Lab,
104 S. Wright St.
Urbana, IL 61801

Xin Mao
School of Information Technology & Eng.
George Mason University
4400 Univ. Dr.
Fairfax, VA 22030

Richard Marciano
218 E. Washington, #3W
Iowa City, Iowa 52240

Manohar Mareboyana
NASA/GSFC
Code 636
Greenbelt, MD 20771

Massimo Maresca
DIST-Univ. of Geneva
via Opera Pia 11A
Genova, Italy, 16145

Charles Marshall
Morgan Stanley & Co.
1251 Avenue of the Americas, 19 floor
New York, NY 10020

Douglas R. Martin
Department of Defense
1509 Winding Brook Way
Baltimore, MD 21207

Dr. Robert Martino
NIH
Bldg. 12A, Rm. 2019
Bethesda, MD 20892

Dr. Michael Mascagni
NIH
Bldg. 31, Rm. 4B-54
Bethesda, MD 20892

Richard Masline
Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, CA

Robert A. Masumrua
Naval Research Lab
Code 6304
Washington, DC 20375-5000

Robert Mattheyses
GE
P.O. Box 8
Schenectady, NY 12301

Isaac Mayergoyz
University of Maryland
College Park, MD 22742

David F. McAllister
NC State University
Box 5206
Raleigh, NC 27695

Carolyn McCreary
American University
4400 Mass. Ave., NW
Washington, DC 20016

Russ McFadden
McDonald Douglas Astronomics
5301 Bolsa Ave.
Bldg. 22-2 Dept. W362
Huntington Beach, CA 92647

Kathryn S. McKiney
Student @ Rice University
7000 Greenbriar #32
Houston, TX 77030

Edward P. McMahon
MRJ, Inc.
10455 White Granite Dr.
Oakton, VA 22124

D. McPherson
Atomic Energy of Canada, Ltd.
Chalk River
Ontario KOJ 1P0
CANADA

Kenneth McVearry
NPAC - Syracuse University
250 Machinery Hall
Syracuse, N.Y. 13244

Judith McWilliams
GE
4041 N. First St.
San Jose, CA 95134

J. Mehat
Institut d'Electronique Fondamentale
Batiment 220
Universite Paris Sud
91405 Orsay Cedex,
France

Joseph Melia
MIT Lincoln Lab
PO Box 73
Lexington, MA

Bernard L. Menezes
University of Maryland
College Park, MD 20742

A. Merigot
Institut d'Electronique Fondamentale
Batiment 220
Universite Paris Sud
91405 Orsay Cedex,
France

James Bret Michael
George Mason University
4400 University Drive
Fairfax, VA 22030

John Michener
SIEMENS Corporate Research, Inc.
775 College Road East
Princeton, NJ 08540

David Middleton
ICASE
MS 132C, NASA LaRC
Hampton, VA 23665-5225

Scott F. Midkiff
VPI
Dept. of Electrical Engineering
Blacksburg, Virginia 24061-0111

Calvin S. Miles
Systems Control Technology
475 School Street S.W., DC 1030
Washington, DC 20024

Brad Miller
DEC
77 Reed Rd.
Hudson, MA 01745

Peter Miller
DVP, Inc.
2401 Research Blvd., Suite 101
Rockville, MD 20850

Ronnie Mills
STX
4400 Forbes Boulevard
Lanaham, MD 20706

Barbara Moore
MIT AI Lab
NE 43-826
545 Technology Square
Cambridge, MA 02139

Peter G. Moore
Naval Research Lab
Code 6321
Washington, DC 20375

Osama A. Morad
Digital Analysis Corp.
1889 Preston White Dr.
Reston VA, 22091

Robert E. Morley
Washington University - St. Louis
#1 Brookings Dr., Box 1127
St. Louis, MO 63117

Andrew P. Mullhaupt
University of New Mexico
Dept. of Math
Albuquerque, NM 87131

Walid Najjar
USC/ Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90242

Wayne Nation
Purdue University
Electrical Engineering Building
West Lafayette, IN 47907

Philip M. Neches
Teradata Corporation
12945 W. Jefferson Blvd.
Los Angeles, CA 90066

David A. Nelson
US Air Force
AFOISR
Bolling AFB, DC 20332

Bill Nestlerode
University of South Carolina
CPSA, Computer Services
Columbia, SC 29208

Kathleen M. Nichols
Apple Computer, Inc.
20525 Mariani Ave.
Cupertino, CA 95014

Jay P. Norris
Naval Research Laboratory
Code 4121.3
Washington, DC 20375-5000

Mark J. Norton
GE Aerospace, ATL
Moorestown Corp. Center
Moorestown, NJ 08057

John T. O'Donnell
Indiana University
Computer Science Dept.
101 Lindley Hall
Bloomington, IN 47405

M. O'Keefe
Purdue University
School of Elec. Eng.
W. Lafayette, IN 47907

Dr. Mark A. Olson
The BDM Corp.
7915 Jones Branch Dr.
McLean, VA 22102

Yoshie Ono
Hitachi American, Ltd.
950 Elm Avenue
Suite 160
San Bruno, CA 94066

Torstein Opsahl
MRJ, Inc.
10467 White Granite Drive
Oakton, VA 22124

Darryl S. Paden
Loral Defense Systems
1210 Massillon Rd.
Akron, Ohio 44315

Bland Painter
FAX Group
P.O. Box 477
Troutville, VA 24175

David R. Palmer
Office of the Director NOAA/AOML
4301 Kickenbacker Cswy.
Miami, FL 33149

Connie Pankratz
ORD/AMRD
706 Ames
Washington, DC 20505

Osborne Parchment
Morgan/NASA Digital Imaging Project
PO Box 2178
Wheaton, MD 20902

Jo Ann Parikh
Southern Conn. State Univ.
501 Crescent St.
New Haven, CN 06515

Dennis Parkinson
Active Memory Technology
65 Suttons Park Ave.
Reading, RG6 1AA, England

R.G. Peck
NADC
Code-5051
Warminster, PA 18974

Robert L. Peebles
Digital Equipment Corp.
40 Old Bolton Rd.
Stow, MA 01775

Shietung Peng
UMBC
Wilkens Ave.
Catonsville, MD 21224

A.T. Perlik
MRJ, Inc.
10467 White Granite Drive
Oakton, VA 22124

Ivars Peterson
Science News
1719 N. Street, NW
Washington, DC 20036

Tom Phillips
1911 Wintergreen Ct.
Reston, VA 22091

Douglas Photiadis
David Taylor Research Center
Code 1965
Bethesda, MD 20084-5000

Ryszard Pisarski
NASA/GSFC
Code 636
Greenbelt, MD 20771

Prof. Tomaso Poggio
Massachusetts Institute of Technology
Cambridge, MA

Stephen D. Post
Emhart PRC
1500 Planning Research Dr.
McLean, VA 22102

Jerry Potter
Kent State University
Kent, OH 44242

Michael W. Powers
US Army Engineer Topographic Lab.
ATTN: CEETL-SL-TE
Bldg. 2592
Fort Belvoir, VA 22060-5546

V. Prasanna-Kumar
University of Southern California
Sal-344, Dept. of EE-Systems
Los Angeles, CA 90089-0781

Karbhari Prasmanti
New Jersey Inst. of Technology
143 Chestnut St.
Bridgewater, NJ 08807

Tammy Preston-Boyd
Naval Research Lab
Code 5326.1
4555 Overlook Ave., SW
Washington, DC 20375-5000

Daniel V. Pryor
Supercomputing Research Center
4380 Forbes Blvd.
Lanham, MD 20706

William Pugh
University of Maryland
Dept. of Computer Science
Silver Spring, MD 20742

Jeanne Pujanauski
George Mason University
1323 N. Lynnbrook Dr.
Arlington, VA 22201

David M. Race
E-Systems
1200 S. Jupiter Rd.
Garland, Texas 75042

James D. Radosevich
NASA HQ
Code NTD
Washington, DC 20546

Iqbal S. Rai
Goodyear Tire & Rubber Co.
P.O. Box 3531
Akron, Ohio 44309-3531

H.K. Ramapriyan
NASA/GSFC
Code 636
Greenbelt, MD 20771

Stewart F. Reddaway
Active Memory Technology, Inc.
65 Suttons Park Ave.
Reading, RG6 1AA, England

Anthony Reeves
Cornell University
410 Phillips Hall
Ithaca, NJ 14853

Prof. John Reif
Duke University
3112 Devon Rd.
Durham, NC 27707

Michael D. Rice
Mathematics Dept.
Wesleyan University
Middletown, VA 06457

Teresa Haynes Rice
East Tennessee State University
Box 23830A
Johnson City, TN 37614

John P. Riganati
Supercomputing Research Center
4380 Forbes Blvd.
Lanham, MD 20706

G.X. Ritter
University of Florida
CIS Dept.
Gainesville, FL 32611

Louis G. Robinson
The Spang Robinson Reports
P.O. Box 1432
Manchester, MA 01944

Amos Rohrer
EG & G WASCI
1396 Pickard St.
Rockville, MD 20850

Azriel Rosenfeld
University of Maryland
College Park, MD 20742

Thomas Roskan
George Washington Univ.
Dept. of Management SC
Washington, DC 20052

Ray A. Rothrock
Venrock Associates
30 Rockefeller Plaza, Room 5508
New York, NY 10112

Badrinath Roysam
Washington University
6251 San Bonita Ave.
St. Louis, MO 63105

George Rumney
NASA Goddard
Greenbelt, MD 20771

T. Rus
University of Iowa
Dept. of Computer Science
Iowa City, Iowa 52242

Tom Ryan
SAIC
7014 Alicent Ct.
McLean, VA 22101

Narayan Sahoo
Dept. of Physics
SUNY at Albany
1400 Washington Avenue
Albany, NY 12222

Douglas Sakal
Computer Sciences Corporation
3160 Fairview Park Dr.
Falls Church, VA 22042

Ken Sallenger
University of South Carolina
CPSA, Computer Services
Columbia, SC 29208

Nidal Sammur
Oklahoma State University
P.O. Box 2376
Stillwater, OK 74076

Dr. Jerry L. Sanders
Parallelogram
71 Collingwood Ave.
London NI0 3EE
UK

Jay Santry
STX
GSFC Bldg #28
Code 6350
Greenbelt, MD 20771

Marlyn M. Sarry
DOD
Fort Meade, MD 20755

Shigeru Sasaki
Carnegie Mellon University
Computer Science Department
Pittsburgh, PA 15213

Dale R. Satran
NASA
Code NTD
Washington, DC 20546

Micahel P. Satyshur
U.S. Naval Research Lab
4555 Overlook Ave., SW
Washington, DC 20375-5000

S. Sayre
SAIC
5151 East Broadway Suite 900
Tucson, AZ 85711

David H. Schaefer
George Mason University
Fairfax, VA

Willie Schatz
Datamation
4451 Albemarle St. NW
Washington, D.C. 20016

Dr. Paul B. Schneck
Supercomputing Research Center
4380 Forbes Blvd.
Lanham, MD 20706

Howard Schneider
Cite de la Sante de Laval
149 Finchley Rd.
Montreal, Quebec H3X3A3
Canada

Raymond J. Schneider
George Mason University
12623 Holkein Dr.
Herndon, VA 22071

Daniel S. Seewer
Dept. of Defense
10377 College Sq.
Columbia, MD 21044

Stephen B. Seidman
George Mason University
4400 University Dr.
Fairfax, Virginia 22030

Edward Seiler
STX
Mail Code 635
Goddard Space Flight Center
Greenbelt, MD 20771

A. Sengupta
Univ. of South Carolina
Dept. of Computer Science
Intell. Systems Lab.
Columbia, SC 29208

Dr. H. Thomas Sharp
Lockheed Aeronautical Systems Company
25115 Avenue Stanford, Box 1
Valencia, CA 91355

Thomas J. Sheffler
The MITRE Corporation
7525 Colshire Drive
McLean, VA 22102-3481

Sam Shepard
Martin Marietta
6303 Ivy Lane
Greenbelt, MD 20770

Gideon Shichman
IBM/T.J. Watson Research Center
Dept. 543/Room 38-209
Yorktown Heights, NY 10598

Tom Shook
U.S. Army
203 Enrikons Rd.
Sterling, VA 22170

Prof. H.J. Siegel
Purdue University
E.E. School
West Lafayette, IN 47907

Claude Sigel
DEC
P.O.Box 80
Albuquerque, NM 87103

T.J. Sullivan
Dept. of Elec. Eng.
Washington University
St. Louis, Missouri 63130

S.D. Simmes
SAIC
5151 E. Broadway
Suite 900
Tucson, AZ 857211-3796

David Simmons
Martin Marietta
8926 Curry Ford Place
Orlando, FL 32825

Raj Kumar Singh
UNC - Computer Science
CB# 3175 Sitterson Hall
Chapel Hill, NC 27599

Donald J. Skala
US Army Engineer Topographic Lab.
ATTN: CEETL-SL-TE
Bldg. 2592
Fort Belvoir, VA 22060-5546

William Slough
University of Missouri
Computer Science Dept.
Columbia, MO 65211

Bruce T. Smith
University of NC
Sitterson Hall
Chapel Hill, NC 27707

Capt. P.G. Smith
MAGTF Warfighting Center
WF13A, MCCDC
Quantico, VA 22314-5001

Kurt R. Smith
Washington University
700 S. Euclid
St. Louis, MO 63110

Arun Sood
George Mason University
4400 University Dr.
Fairfax, VA 22030

T. Sos
SAIC
5151 East Broadway Suite 900
Tucson, AZ 85711

S.J. Spoerry
George Mason University
10256 Battlefield Dr.
Manassas, VA 22110

Dr. Steven Squires
DARPA
Arlington, VA

Dr. Andrew J. Srokowski
Lockheed Aeronautical Systems Company
25115 Avenue Stanford, Box 1
Valencia, CA 91355

Dr. Guy Steele
Thinking Machines Corporation
245 First St.
Cambridge, MA 02142

Laurence Stein
E-Systems
7700 Arlington Blvd.
Falls Church, VA 22046

Mark Stevens
STX
4400 Forbes Blvd.
Lanham, MD 20706

Brad Stewart
Inst. for Parallel Computation
Thornton Hall,
UVA
Charlottesville, VA 22901

Bert Still
University of South Carolina
CPSA, Computer Services
Columbia, SC 29208

L.R. Owen Storey
NASA HQ, Code ES
600 Independence Avenue S.W.
Washington, DC 20546

Joe Straub
Matrik Technology Support
8355 Alvord St.
McLean, VA 22102

Dr. James Strong
NASA/GSFC
Greenbelt, Maryland 20770

Pradipa Subramaniam
The University of Akron
Akron, OH 44325

John Sui
Dow Jones & Co., Inc.
Rt. 1 at Ridge Road
South Brunswick, NJ 08852

Andrew H. Sung
New Mexico Tech
Computer Science Dept.
Socorro, NM 87801

David Sung
GE
P.O. Box 8048
Philadelphia, PA 19101

Myung H. Sunwoo
Computer and Vision Research Center
The University of Texas at Austin
Austin, Texas 78712

Prof. Daniel Tabak
Dept. Electr. & Computer Eng.
George Mason University
Fairfax, VA 22030

Doron Tal
School of Computer Science
University Park
Miami, FL 33199

Thomas F. Tarr
Naval Surface Warfare Center
White Oak
Silver Spring, MD 20903-5000

Kosmo Tatlias
Atlantic Aerospace Electronic Corp.
6404 Ivy Lane, Suite 300
Greenbelt, MD 20770

Iraj Tavakoli
George Mason University
4400 University Dr.
Fairfax, VA 22030

Anthony Terrano
Rutgers University
Computer Engineering
Piscataway, NJ 08855

Xavier Thibault
ONERA-CERT
2, Avenue Edouard Belin
B.P. 4025
Toulouse, 31400, France

James C. Tilton
NASA/GSFC
Code 636
Greenbelt, MD 20771

Suminto To
Mississippi State University
P.O. Box 1558
Mississippi, MS 39762

Sherryl Tomboulian
ICASE
NASA Langley Research Center
Hampton, VA 23665

Russell R. Tuck
Univ. of North Carolina
Computer Science Dept. CB 3175
Chapel Hill, N.C. 27599

C.J. Turner
SAIC
5151 E. Broadway
Suite 900
Tucson, AZ 85711-3796

Leonard Uhr
University of Wisconsin
Computer Science Dept.
Madison, WI 53706

James Utt
MacAulay Brown, Inc.
3915 Germany Lane
Dayton, OH 45431

Jane Van Fossen
Office of Naval Technology
Code 227
800 N. Quincy St., BCT #1
Arlington, VA 22217-5000

Stephen Van Trees
SIMPACT ASSOCIATES
11306 Gatesborogh Lane
Reston, Virginia 22091-3605

Neal Vanderkept
University of Maryland
11356 Cherry Hill Rd., #301
Beltsville, MD 20705

David Velten
Digital Equipment Corp.
75 Reed Rd.
Hudson, MA 01749

Steven Venable
Martin Marietta Elec. Systems
P.O. Box 555837, MP 1304
Orlando, FL 32855-5837

Alan S. Waksman
Plex Systems, Inc.
353 Lexington Avenue
New York City, NY 10016

J.L. Walker, Jr.
Northeastern University
Boston, MA 02115

J.L. Wallace
Atlantic Research
7401 Boston Blvd.
Springfield, VA 22153

Donald P. Walsh
U.S. Patent & Trademark Office
P.O. Box 2251
Arlington, VA 22202

Chia-Jiu Wang
University of Colorado
Dept. of Electrical Engineering
Colorado Springs, CO 80933-7150

Pearl Wang
George Mason University
CS Department
Fairfax, Virginia 22030

Robert Wasilausky
Naval Ocean System Center
C423
San Diego, CA 92152

Scott Weidman
MRJ, Inc.
10455 White Granite Dr.
Oakton, VA 22124

John Weidner
The MITRE Corporation
7525 Colshire Drive
McLean, VA 22102-3481

Charles R. Weisbin
Oak Ridge National Laboratory
P.O. Box 2008
Oak Ridge, TN 37831-6364

Kelly Wheeler
4028 Estabrook Dr.
Annandale, VA 22003

John Whelchel
E Systems Melpar Div.
7700 Arlington Blvd.
Falls Church, VA 22046

Marc Willebeek-LeMair
Cornell
412 Hanshaw Rd.
Ithaca, NY 14850

Lowell Williams
MacAulay Brown, Inc.
3915 Germany Lane
Dayton, OH 45431

Joseph N. Wilson
University of Florida
CIS Dept.
Rm. E301 CSE Bldg.
Gainesville, FL 32611

L. Winter
SAIC
5151 East Broadway Suite 900
Tucson, AZ 85711

K.B. Winterbon
Atomic Energy of Canada, Ltd.
Chalk River, Ontario
CANADA KOJ 1J0

Kel Winters
Montana State University
900 Poane Rd.
Bozeman, MT

Chwan-Hwa (John) Wu
Auburn University
Electrical Engineering Dept.
Auburn, AL 36849

Zhiwei Xu
Rutgers University
P.O. Box 909
Piscataway, NJ 08855

Dan Yahiel
Grumman Data Systems
6862 Elm Street
McLean, VA 22091

Len Yenchanis
Yenchanis Consulting Group
582 Hawthorne Sq.
N. Massapepna, NY 11758

Y. Yomdin
The Institute for Advanced Study
Princeton, NJ 08540

Jason Yoon
Privac Inc.
3713 S. George Mason Dr.
Falls Church, VA 22041

Stanley Young
Southwest Research Inst.
6220 Culebra - Div. 5
San Antonio, TX 78238

Ernest Zaidman
Naval Research Lab
Washington, DC 20375

Mark Zischke
TRW
1633 Westwind Way
McLean, VA 22102

# AUTHOR INDEX

# AUTHOR INDEX

PRECEDING PAGE BLANK *NOT FILMED*

PAGE 726 INTENTIONALLY BLANK

# TITLE INDEX

# TITLE INDEX