Algorithms and Programming Tools for Image Processing on the MPP

Report for the Period
May 1984 to November 1985

Anthony P. Reeves
School of Electrical Engineering
Cornell University
Ithaca, New York 14853

Work Supported by NASA Grant NAG 5-403

CORNELL UNIVERSITY

SCHOOL OF ELECTRICAL ENGINEERING
ITHACA, NEW YORK 14853

Algorithms and Programming Tools for Image Processing on the MPP

Report for the Period
May 1984 to November 1985

Anthony P. Reeves
School of Electrical Engineering
Cornell University
Ithaca, New York 14853

Work Supported by NASA Grant NAG 5-403

Algorithms and Programming Tools for Image Processing on the MPP

Report for the Period
May 1984 to November 1985

Anthony P. Reeves
School of Electrical Engineering
Cornell University
Ithaca, New York 14853

Work Supported by NASA Grant NAG 5-403

Summary

The work reported here was conducted by a number of students at Cornell University and myself. A major contribution was made by Cristina Mahon (previously Cristina Moura); her masters thesis is included as Appendix A. The work for this grant falls into two main categories: algorithms for the MPP and the Parallel Pascal Development system. A number of novel algorithms for data arbitrary data mappings, permutations, and image rotation including interpolation have been developed and implemented on the MPP. A program development system has been developed for both the MPP and conventional serial computers. This system greatly simplifies the development of high level language programs for the MPP. Furthermore, it allows programs to be developed and tested on any conventional computer. This environment consists of a set of system programs and a library of general purpose Parallel Pascal functions.

A detailed description of the data mapping and rotation algorithms for the MPP is given in section 3 of Cristina's thesis which is included as Appendix A. These algorithms have been published [1] and a copy of this paper has been included as Appendix B. The specification of the Parallel Pascal language has now been published [2, 3] These papers are included as Appendices C and D. The documentation for the Parallel Pascal Development system is given in Appendix E and a description of using this system on the MPP is given in section 2 of Appendix A.

In addition to the development of the reported algorithms and software this grant provided us with the opportunity to be the pioneer remote users of the MPP. Most of the work on the MPP was done from Cornell over a telephone line which in

itself absorbed a significant amount of the available funds. We also made several visits to NASA; on two occasions seminars were presented on the Parallel Pascal environment. Considerable discussions were held wiht NASA on the development of the I/O system for the MPP and other aspects of the high level language environment. The complete Parallel Pascal development system has been installed on the MPP host and has been distributed by us to a number of remote sites. It has been ported to a number of conventional computers including VAX systems running either VMS or UNIX.

Some of the highlights of the results of this research are listed below.

Image Processing Algorithms

The following algorithms are described in detail in section 3 of Appendix A.

Data Mapping

A fast heuristic arbitrary data mapping algorithm has been developed. For most mappings this is much faster than other techniques such as sorting. This has been implemented for both regular (128 x 128) and large ((n * 128) x (m * 128) two dimensional arrays.

Matrix Rotation

Fast matrix rotation algorithms have been implemented based on the above data mapping function. The nearest neighbor algorithm ahs been tested on the MPP. Large matrix nearest neighbor rotation, and interpolation schemes have been developed and tested on conventional computers but have not yet run on the MPP.

Interpolation

A high speed interpolation algorithm has been implemented for bilinear and bicubic interpolation for image rotations (any angle) and small matrix warps. These algorithms work on the development system but have not yet been tested on the MPP.

The Parallel Pascal Development System

Compiler Command

A command file has been written both for the MPP and the development system which, with a simple noninteractive command, compiles a program, makes all the library links, and in the case of the MPP, loads the program onto the system.

See section 2 of Appendix A for details.

Library Preprocessor

Standard Pascal does not have any library facilities. A general purpose library preprocessor has been developed which works for both the MPP compiler and the development system. See Appendices D and E for details. The preprocessor looks for library subprograms first in named files, then in the local directory, and finally in a system library directory. The MPP compiler version of the preprocessor also examines a special MPP system library before the general system library. This library contains system library programs which have been modified to overcome deficiencies in the MPP compiler. This library preprocessor can be used in conjunction with the assembly language library feature which is built into the MPP Parallel Pascal compiler. It is also able to work in conjunction with any library facilities that are available with a local Pascal compiler that is used with the development system.

Parallel Pascal Translator

The translator is the heart ot the development system. It translates a Parallel Pascal program into standard Pascal for execution on a conventional serial computer; See Appendix E for details. The translator is a Pascal program with over 8000 lines of code. It has been in a very stable form for over a year now. It still has some limitations but these are now well documented. In addition to being used by this and other MPP research groups it has also been used in a Parallel Processing course which has now been offered three times with an enrollment of about 50 students each time.

The System Library

A set of general purpose library programs have been developed. All of these programs run correctly on the development system and nearly all of them have been tested on the MPP. Documentation for these programs is given in Appendix E.

General Utilities

Programs in the general utilities group include a parallel random number generator, an index generator, simplified I/O functions and a parallel ceiling function.

Masked Reduction Functions

It is frequently necessary to apply a reduction function to a subregion of an ar-

ray. The masked reduction library functions are similar to the primitive reduction functions except that a Boolean array mask parameter is required.

Large Array Utilities

The large array utilities are shift and rotate functions that operate on matrices which have dimensions that are multiples of 128. Both the crinkled and blocked data structures are supported. There are also shift and rotate functions which treat a 128 x 128 array as a vector of 16384 elements. These functions do not use the hardware spiral interconnections; the use of the regular mesh interconnections is faster for multiple element shifts.

Near Neighbor Convolution Functions

Many low level image processing applications require convolutions between images and small kernel matrices. Programs in this group simplify the entry of small matrices and the application of these matrices to image arrays.

Pyramid Operations

A pyramid convolution function has been implemented; this is a three dimensional convolution function which operates on the 13 near neighbors of a pyramid data structure. This data structure is embedded within an MPP array. Functions are also available for both the vertical and horizontal data shift operations that are associated with pyramid algorithms.

References

1. A. P. Reeves and C. H. Moura, "Data Mapping and Rotation Functions for the Massively Parallel Processor," *Proceedings of Computer Architecture for Pattern Analysis and Image Database Management*, pp. 412-419 (November 1985).

2. A. P. Reeves, "Parallel Pascal: An Extended Pascal for Parallel Computers," *Journal of Parallel and Distributed Computing* 1 pp. 64-80 (1984).

3. A. P. Reeves, "Parallel Pascal and the Massively Parallel Processor," pp. 230-260 in *The Massively Parallel Processor*, ed. J. Potter, MIT Press (1985).

Appendix A

PROGRAMMING TOOLS AND ALGORITHMS FOR THE MASSIVELY PARALLEL PROCESSOR

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

by

Cristina Helena Francfort de Sellos Moura

August 1985

ABSTRACT

The Massively Parallel Processor (MPP) is a SIMD computer with 16384 processing elements connected in a 128 x 128 mesh. The MPP is programmed in a high level language called Parallel Pascal, a superset of standard Pascal used to program parallel computers. This thesis describes system and programming tools for the Parallel Pascal development system and the MPP-compiler system for the MPP. Some of these tools were developed to help remote users, while others were developed to give users a way to work around features not yet implemented on the MPP.

The organization of the MPP is ideal for problems which involve near neighbor interactions. In this thesis we present a general algorithm for implementing arbitrary permutations and mappings on such systems as well as matrix rotation schemes for nearest neighbor, bilinear interpolation and bicubic spline interpolation mappings. An algorithm for the Ising model and its implementation on the MPP using Monte Carlo techniques is also discussed. Such an application is well suited for the MPP because it uses mainly near neighbor interactions and Boolean operations. Results and timings obtained from a direct implementation of those algorithms on the MPP are also reported.

BIOGRAPHICAL SKETCH

Cristina Helena Francfort de Séllos Moura was born in Rio de Janeiro, Brazil in November 26, 1962. She started her undergraduate work in Electrical Engineering at Ecole Polytechnique Féminine in Sceaux, France in 1980. At the end of the first year she transferred to the University of Miami, Florida. She graduated Summa Cum Laude in May 1984 with a Bachelor of Science degree in Electrical Engineering and a second major in French.

In 1984, she came to Cornell University to pursue a Master of Science in Electrical Engineering. She graduated from Cornell in August 1985.

She is a member of Tau Beta Pi, Eta Kappa Nu and IEEE.

DEDICATION

Aos meus pais,

minha irmã Márcia

e meu noivo Hugh

sem os quais esta tese

não teria sido possível

ACKNOWLEDGEMENTS

I would like to thank Professor Anthony P. Reeves, my committee chairman, for his encouragement, guidance, and assistance in my research and writing of this thesis and as a graduate student in general. I would also like to thank Professor Christopher C. Pottle for serving on my committee.

I am grateful to the Schlumberger Foundation for providing me with the 1984-85 fellowship.

I would like also to specially thank my family and my fiance Hugh for their continuous love, encouragement and support through sometimes difficult times.

I thank also Paul Chau for his help in formatting my thesis. Thanks are also due to all my colleagues in room 209 for their support and advice. They helped make this year not only instructive, but also fun.

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

LIST OF TABLES

LIST OF TABLES (continued)

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

This thesis is concerned with programming tools and parallel algorithms created for the Massively Parallel Processor (MPP) located at NASA Goddard Space Center. The goals of this thesis are to create a user-friendly environment for high-level language parallel algorithm development, as well as research the issues involved in implementing certain algorithms on the MPP and compare the expected results with the achieved results.

Digital computers as we know them today have always had a certain degree of parallelism, first for reliability purposes, and later also for greater performance. Parallelism, for the purpose of increased performance, can be classified into the following types. [1].

- serial by bit
- serial by character
- "parallel word" but serial instruction
- parallel instruction execution/access
- parallel instruction execution units
- parallel instructions (SIMD)
- parallel instructions (MIMD)

The term parallel processing is usually used to refer to the last two types of parallelism listed above, since those are the types that allow parallel work on more than one set of data at any single time. An SIMD (single-instruction stream, multiple-data streams) computer is composed of several processors which perform the same operation on different pieces of data. An MIMD (multiple-instruction streams, multiple-data streams) computer, on the other hand, is composed of several processors performing different operations on different pieces of data.

These two approaches underlie different purposes. SIMD computers are lockstep processing type computers, while MIMD computers are asynchronous processing type. That difference tends to make SIMD computers more specialized than MIMD, since they do not allow for independence between their processors. The Massively Parallel Processor (MPP) produced by Goodyear Aerospace for NASA Goddard Space Center belongs to the SIMD class of processors. It was developed for image processing of satellite data.

1.1. The MPP Architecture

The block diagram of the MPP is shown in Figure 1.1. It consists of the following five main components. The Array Unit (ARU) processes two dimensional arrays of data. It is controlled by the PE Control Unit which executes parts of the user program that contain array operations. The Staging Memory handles the array I/O by both storing and rearranging array data. The Main Control Unit (MCU) executes the application program, performs scalar operations and calls on the PE control unit to perform the array operations of the application program. This is the main interface unit to the host machine. Finally, the host computer, a VAX 11/780, serves as the communication unit between the user and the MPP. [2].

1.1.1. The Array Unit:

The component that makes the MPP special is the ARU. Logically, it contains 16384 bit processing elements (PE's) organized as a 128 by 128 square and operating at a basic cycle of 100 nsec. Physically, it contains an extra 128 by 4 rectangle of PE's that is used for reliability purposes. When a PE fault is detected the ARU can be reconfigured by replacing the 128 by 4 rectangle containing the faulty PE by the extra rectangle. Each PE supports boolean and arithmetic operations, is maskable and is capable of routing data to its orthogonal neighbors. The speed of some typical operations can be seen in table 1.1. [3].

The MPP is constructed using a total of 2112 VLSI chips. This total includes the spare column of chips for redundancy. Each chip contains eight PE's configured in a 2 by 4 array,

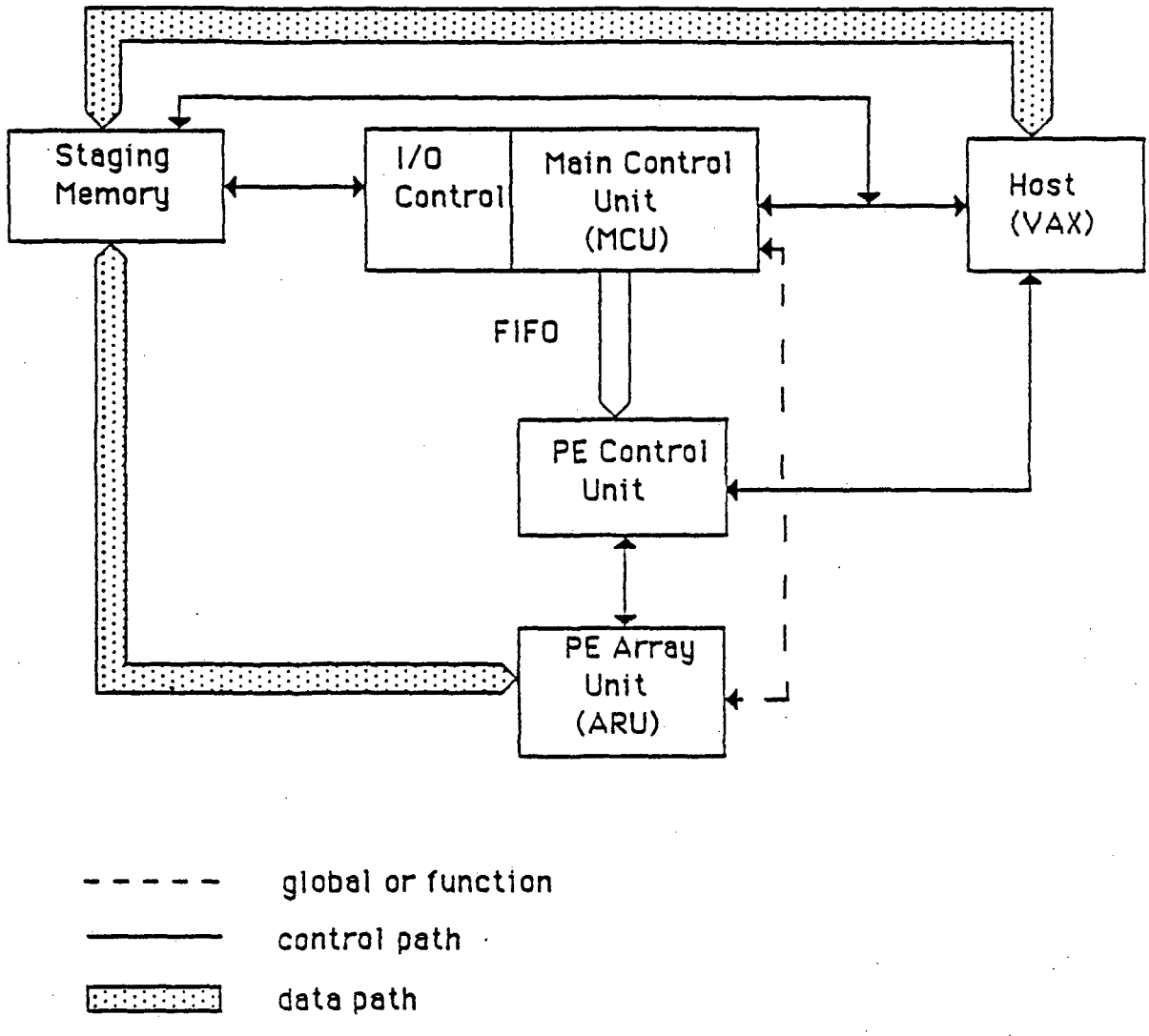


Figure 1.1. MPP Block Diagram

Table 1.1. Speed of Typical Operations

OPERATIONS	EXECUTION SPEED*
ADDITION OF ARRAYS	
8-BIT INTEGERS (9-BIT SUM)	6553
12-BIT INTEGERS (13-BIT SUM)	4428
32-BIT FLOATING-POINT NUMBERS	470
MULTIPLICATION OF ARRAYS (ELEMENT-BY-ELEMENT)	
8-BIT INTEGERS (16-BIT PRODUCT)	1861
12-BIT INTEGERS (24-BIT PRODUCT)	910
32-BIT FLOATING-POINT NUMBERS	291
MULTIPLICATION OF ARRAY BY SCALAR	
8-BIT INTEGERS (16-BIT PRODUCT)	2824
12-BIT INTEGERS (24-BIT PRODUCT)	1489
32-BIT FLOATING-POINT NUMBERS	373

*MILLION OPERATIONS PER SECOND

an eight bit bi-directional data port, and a disable circuit that is used to disconnect the chip from its east-west neighbors. This last feature is useful when repairing the array using the redundant PE's. The chip does not include PE memory so as not to complicate the chip design, to allow the MPP to use more memory per PE at a faster access time and to allow future expansion of PE memory without chip redesign.

Each PE contains six single bit registers (A,B,C,G,P,S), a variable length shift register, a full adder and some combinatorial logic. The logic diagram of a PE is shown in Figure 1.2. [3]. The PE has four subunits that have independent control but share a common clock. They are: logic and routing, arithmetic, I/O, and masking subunits. They are interconnected by a bi-directional data bus which also connects to external PE memory.

The logic and routing subunit is formed by the P register and some supporting combinatorial logic. Register P can be logically combined with the state of the data bus and the result would be stored in register P itself. When the routing is enabled the state of one of the P registers in the north, south, east or west neighbor PE's is latched in P. This allows the routing of data between neighboring PE's.

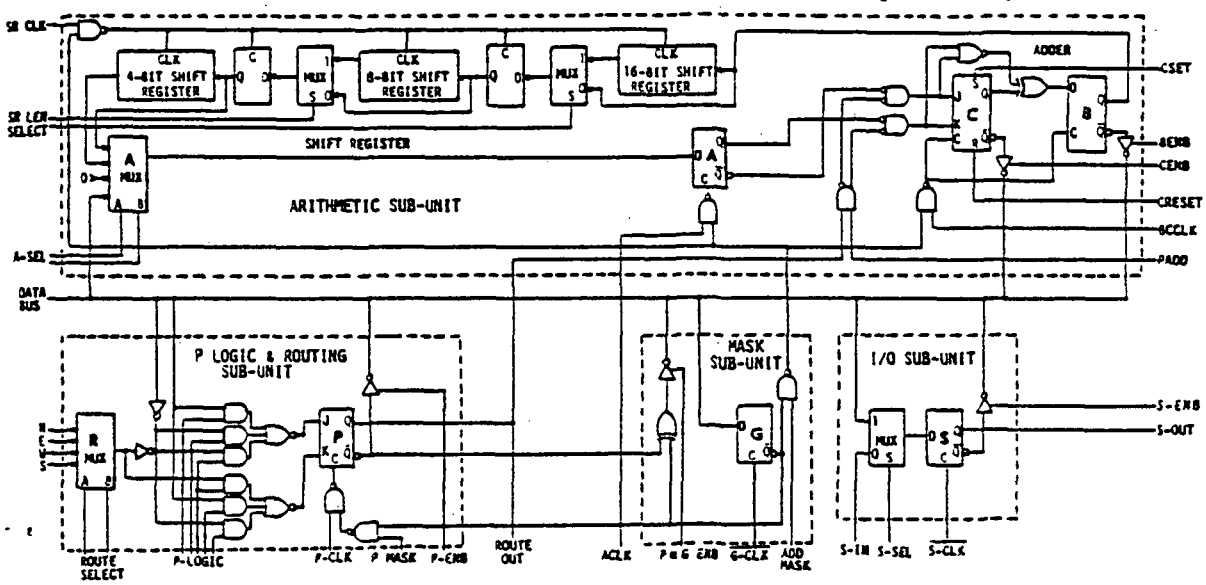


Figure 1.2. PE Logic Diagram

The **arithmetic subunit** contains a serial-by-bit adder formed by registers B and C and a variable length shift register whose output may be stored in A. The inputs for the adder are registers P and A. Register A can be loaded directly from the data bus. Register B stores the least significant bit and register C the carry. The variable length shift register is used to improve the multiply and divide operation times.

The **I/O subunit** is formed by the S register and a two input multiplexor which selects the input from either the data bus or the S register of the PE west neighbor. The data enters the array coming from the west.

Finally, the **masking subunit** is formed by the G register which decides which PE's are active. A PE will be active when its G register is enabled. It allows routing and arithmetic operations to be masked separately. The ARU, composed of all the PE's, is controlled by the PE control unit.

1.1.2. The PE Control Unit:

The PE control unit is a microprogrammed control unit with 64-bit words. It performs all the basic array arithmetic operations of the application program, leaving no idle ARU cycles. [4]. It also determines the connectivity between opposite edges of the ARU through a 3-bit code held in the topology register. Before the application program starts execution, the PE control unit memory is loaded from the host computer.

The address of the bit-planes to be processed by the PE's is generated by the bit-plane address generator in the PE Control Unit and then broadcasted to all PE random-access memories in the ARU. The PE Control Unit is connected to the Main Control Unit by a first-in-first-out buffer called the call queue. The call queue holds calls to PE control routines from the Main Control Unit, which allows the Main Control Unit to work ahead on the application program without waiting for a call to the PE control unit to complete.

Scalar information associated with an MCU call is put into a common register from the call queue at the beginning of the called routine. At the end of the called routine, the contents of the common register can be transmitted back to the MCU. The PE control unit, if specified, can combine certain common register bits with the control lines by performing a logical-OR operation. Since the MCU can initialize the common register when a PE control unit routine is entered, this capability allows the main control to specify certain PE operations. An example of an application of the logical-OR function is array multiplication. It is useful for setting the lengths of the PE shift registers. Without this logical-OR capability, a different multiplication routine would be needed for each shift register length. The logical-OR is also useful to change the common register when the contents of this register have to be sent back to the MCU.

The PE control unit allows operations like a global OR function to be performed on all the elements of the array. Such an operation is extremely useful to the programmer since it allows conditions to be determined based on the status of the whole array.

1.1.3. The Staging Memory:

The staging memory is connected to the I/O ports of the ARU and to a host computer. It acts not only as a buffer to the ARU data but it also allows the arrays of data to be reordered.

Since the ARU ports transfer the data in a bit-sequential order, that is the most (or least) significant bit of 16384 elements followed by the next bit of 16384 elements, etc., the reordering provided by the staging memory is necessary to organize the data in the normal order of satellite imagery used in the host.

1.1.4. The Main Control Unit:

The Main Control Unit (MCU) is essentially a high-speed 16-bit minicomputer. It contains 50 16-bit registers. [5]. Most of those registers are used to communicate with the PE control unit and through the I/O control unit to the Staging Memory. The main control memory of the MCU is loaded by the host computer before an application program starts execution. It holds instructions and scalar data for the MCU. This memory is also shared with the I/O control unit for the Staging Memory. It holds I/O programs for the I/O control unit. The MCU communicates with the PE control unit through a FIFO buffer which was explained in section 1.1.2.

1.1.5. The host machine:

The host computer is a VAX 11/780 running the VAX-VMS operating system. An application program is split between the host computer and the MCU; the programmer specifies which sections of the program are to be run on the MCU.

On the MCU programs have direct high speed access to the ARU, however there are some important limitations. First the MCU only has 32 Kwords of memory, second the MCU has no access to system peripheral devices other than the MPP and finally the MCU has no hardware for scalar floating point arithmetic. On the VAX there is no restrictions on memory space, full access to all peripherals (except direct access to the ARU) and a floating point

accelerator. Currently, the VAX is also shared with other users which can seriously affect performance. The programmer must decide the trade-offs in splitting an application program. A significant time penalty in transferring control and data through the UNIBUS link which connects these two systems must also be considered.

1.2. Outline

The goal of this work was to develop algorithms in a high-level language that would take advantage of the MPP architecture described above. The high-level language for the MPP is Parallel Pascal [6]. which is a superset of standard Pascal for programming parallel computers. In Parallel Pascal all conventional expressions are extended to array data types, that is, arrays can be treated as units.

The development system used to create Parallel Pascal programs is divided in two major parts. First, a Parallel Pascal to standard Pascal translator used in conjunction with a library preprocessor, both developed under the UNIX operating system, allow for algorithm development and testing on a serial computer. This system was adapted to run also under VAX-VMS, the operating system used by the MPP VAX 11/780 host computer. The translator and the system created around it allows the initial program development to be done in any serial computer running UNIX or VMS. After this initial phase, an MPP compiler system running under VMS is used to prepare the programs for execution on the MPP. This part of the work is done on the host machine.

These systems as well as other tools used for algorithm development are described in greater detail in chapter 2. In chapters 3 and 4, algorithms that were implemented on the MPP and performance measurements obtained are discussed. The algorithms include parallel image rotations and an Ising model. For these algorithms expected results are compared to obtained results, so that conclusions on how well these algorithms performed can be drawn.

CHAPTER 2

SYSTEM AND PROGRAMMING TOOLS

The algorithms implemented in this thesis were written in Parallel Pascal. Parallel Pascal offers efficiency, portability as well as error detection and diagnosis facilities, features which when combined create a more pleasant environment than assembly language for program development on the MPP. [6].

Parallel Pascal contains three basic extensions to standard Pascal: [7].

1) expressions involving whole arrays are permitted;

2) the **where - do - otherwise** control statement is available. This statement ia a parallel version of the **if - then - else** statement; the control expression must evaluate to a boolean array. All array assignments within the controlled statements must be conformable with the control array and are masked by it.

3) there are array reduction functions available. These functions reduce arrays according to the *or*, *and*, *minimum*, *maximum*, *sum* or *product* operations.

Programming in Parallel Pascal on the MPP involves using a development system on a serial computer to do the preliminary algorithm development work and then, using a compiler to check for the limitations and restrictions of the MPP. The advantages of this two-step approach are that the initial work can be done in any serial computer running UNIX or VMS and having a Parallel Pascal development system, that is, independently of the MPP, there are no compile time restrictions as the ones encountered when using the Parallel Pascal compiler for the MPP, and, finally, different parallel array sizes can be used to test some of the algorithms without being restricted to the 128 by 128 MPP size. This two-step approach is specially useful for users that are not in the same location as the MPP.

2.1. MPP Limitations

Several programs written in standard Parallel Pascal that had successfully passed the first phase had to be significantly modified to run on the MPP hardware. Modifications were necessary because of the intrinsic limitations of the MPP. Table 2.1 contains a list of the more significant limitations.

One of the limitations of the MPP is that the programmer has to explicitly assign programs, procedures and functions either to the host or to the MPP. This is done through a compiler switch {$H}. Parts of the program that will run on the host are preceded by a {$H+} and parts that will run on the MPP are preceded by a {$H-}. These two instructions may only appear before the **program**, **procedure** and **function** statements. This compiler switch or target machine option affects the programming unit to which it is attached and all syntactically inner functions or procedures, except when those inner functions or procedures are preceded by their own target machine options.

2.2. Software Structure

The work in this thesis represents the first project on the MPP which: a) involved the development of large programs and libraries and b) was conducted by remote access to the system, i.e. a 1200 baud telephone line from Cornell. Several system tools had to be developed for effective remote use of the MPP.

Table 2.1. MPP Limitations.

1) The two lower dimensions of a parallel array have to be 128 by 128.
2) The local PE memory size is 1024 bits.
3) Programs, procedures and functions have to be assigned to the host or the MPP by the programmer.
4) The local MCU memory size is 65K bytes.

2.2.1. Development System Tools

The development system is used to develop and test Parallel Pascal programs in a serial environment. It consists of two programs: *extern*, a Pascal external function preprocessor, and *ppt*, the Parallel Pascal translator. Used in conjunction with them is a set of library functions and a standard Pascal compiler.

For a Parallel Pascal program to run on a serial computer, it has to go through the following steps (see figure 2.1):

- the program *extern* will replace all the calls to external procedures and functions by the procedures themselves, which are read from a set of libraries.

- the output of *extern* will be translated by *ppt*, the Parallel Pascal translator. Any compiler detected errors will be reported in a listing at this stage. The output will be a program in conventional Pascal.

- the program is compiled by a standard Pascal compiler.

In order to allow a user to easily translate a program from Parallel Pascal to Pascal, a command file was created. By simply typing

pp filename.pp

the user's program will be sent through *extern* and *ppt* and a listing file as well as a compiled and linked program will be created.

The command *pp* on VMS is the same command used for the development system at Cornell University. It was chosen so that all the documentation already available at Cornell could be used without modifications at NASA. However, at NASA, outside the account REEVES, used to develop this system, another command name must be used since *pp* invokes the local Parallel Pascal compiler used in conjunction with the MPP. It has not yet been decided which of the two systems will retain the name *pp*. The listings of the command file for the VMS system can be found in section 6.1 of appendix A. Its corresponding shell script in UNIX can be found in section 6.2 of appendix A.

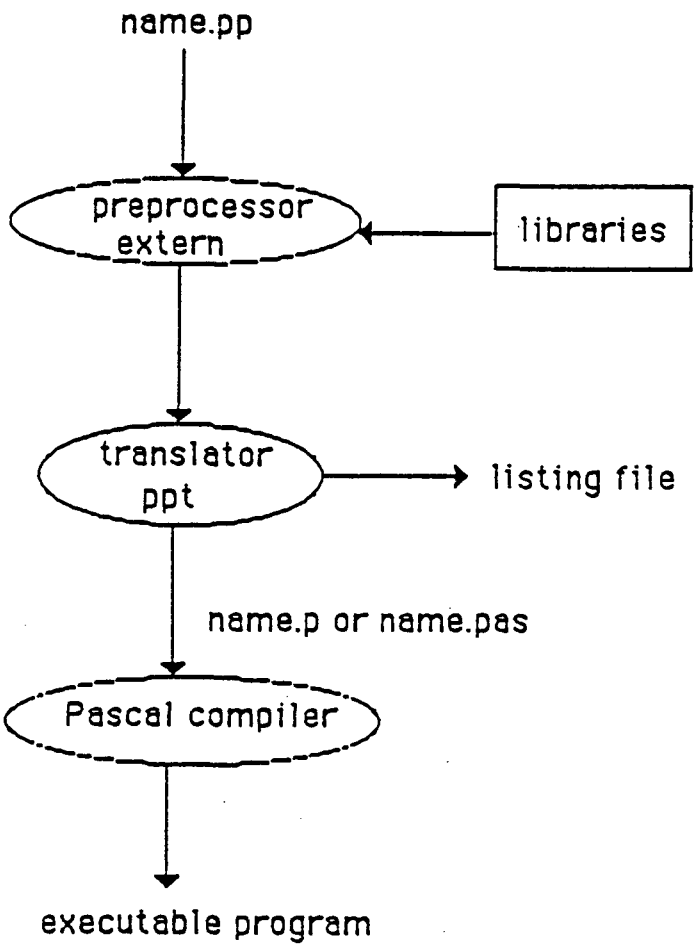


Figure 2.1. Steps of Parallel Pascal Development System

There are two sets of libraries containing Parallel Pascal functions and procedures. One of them is defined to be used as part of the development system in conjunction with *extern*, while the other is targeted for use directly on the MPP and is accessed through a modified version of *extern* called *mppextern*.

The only difference between *extern* and *mppextern* is that *mppextern* checks for a special MPP library first and, if none is found, uses the standard library. *Extern* uses only the standard library. Both libraries contain the same set of functions; however, the MPP-targeted functions were modified to fit some of the limitations and restrictions of the MPP. Section 2.3

will describe in detail some known MPP restrictions and the changes to programs written in standard Parallel Pascal required by them.

2.2.2. MPP-Compiler System

After an algorithm has been tested on the host using the development system, the program can be adapted to use the MPP for its array computations. The MPP is considered as an attached processor to the VAX 11/780. This section will discuss how to compile, assemble, link and execute a Parallel Pascal program on the VAX 11/780 and MPP systems. The features added in order to create a more user-friendly environment for those users attempting to use the MPP are also described.

Parallel Pascal programs can execute on the VAX 11/780, the MPP or on a combination of both. [8]. Compiler switches can be used to specify on which system the code will execute; however, all main programs begin execution on the host machine.

The first system tool created implements all the steps needed to compile a program for the MPP. To use this tool all that is required is typing the command:

rmpp filename The listing for the command file can be found in section 6.3 of appendix A.

This command file does library preprocessing using *mppextern* to substitute all calls to external functions by the body of these functions. The new file created has a *t* appended in front of the old filename. A second command file, called *repl*, has been developed to do just the library preprocessing step. The syntax for this command is:

repl filename.pp

In *rmpp*, once the automatic preprocessing is finished the resulting file is compiled, assembled and linked according to the list of commands below:

$ PP *filename* or $PPDEV *filename*

$ MACRO *filename*

$ MCL *filename*

$ MPPLINK *filename*

$ CADLNK *filename, filename_stb*

Figure 2.2 shows a flowchart of the different commands executed and all the different files which are created. Each command includes several qualifiers to further specify which actions should be taken by the system. These are provided by the command file.

The first command: *PP filename* when used inside the account REEVES would cause problems since *PP* is locally defined to mean the Parallel Pascal translator and not the Parallel Pascal compiler, which is what we want in this context. Fortunately, the people in charge of the Parallel Pascal compiler suggest the use of the command *PPDEV* instead of *PP* when calling the Parallel Pascal compiler. *PPDEV* corresponds to the compiler in the system development directory and which has some of the bugs of *PP* already fixed. Since we use *PPDEV* each time the compiler is called there is no possible ambiguity. To my knowledge, no one uses anymore the command *PP* when referring to the Parallel Pascal compiler.

The compiler inputs Parallel Pascal source code and outputs P-code. The code generator is invoked by the same command and it uses the P-code to produce MACRO (VAX assembly language) and MCL (MCU assembly language) code. The code generator will only execute if the compiler has not detected any errors. [8].

The files generated by the compiler are:

- *filename.lis*: Parallel Pascal listing file
- *filename.pcd*: P-code file

The compiler switches to produce the *lis* and *pcd* files are default. These switches can be turned off by including {$L-} and {$C-} respectively in the program. However, in the absence of a P-code file the code generator will not be able to run.

Another of the compiler switches {$H} was mentioned before in section 2.1. It allows the programmer to specify which parts of the program will run on the MCU and which parts

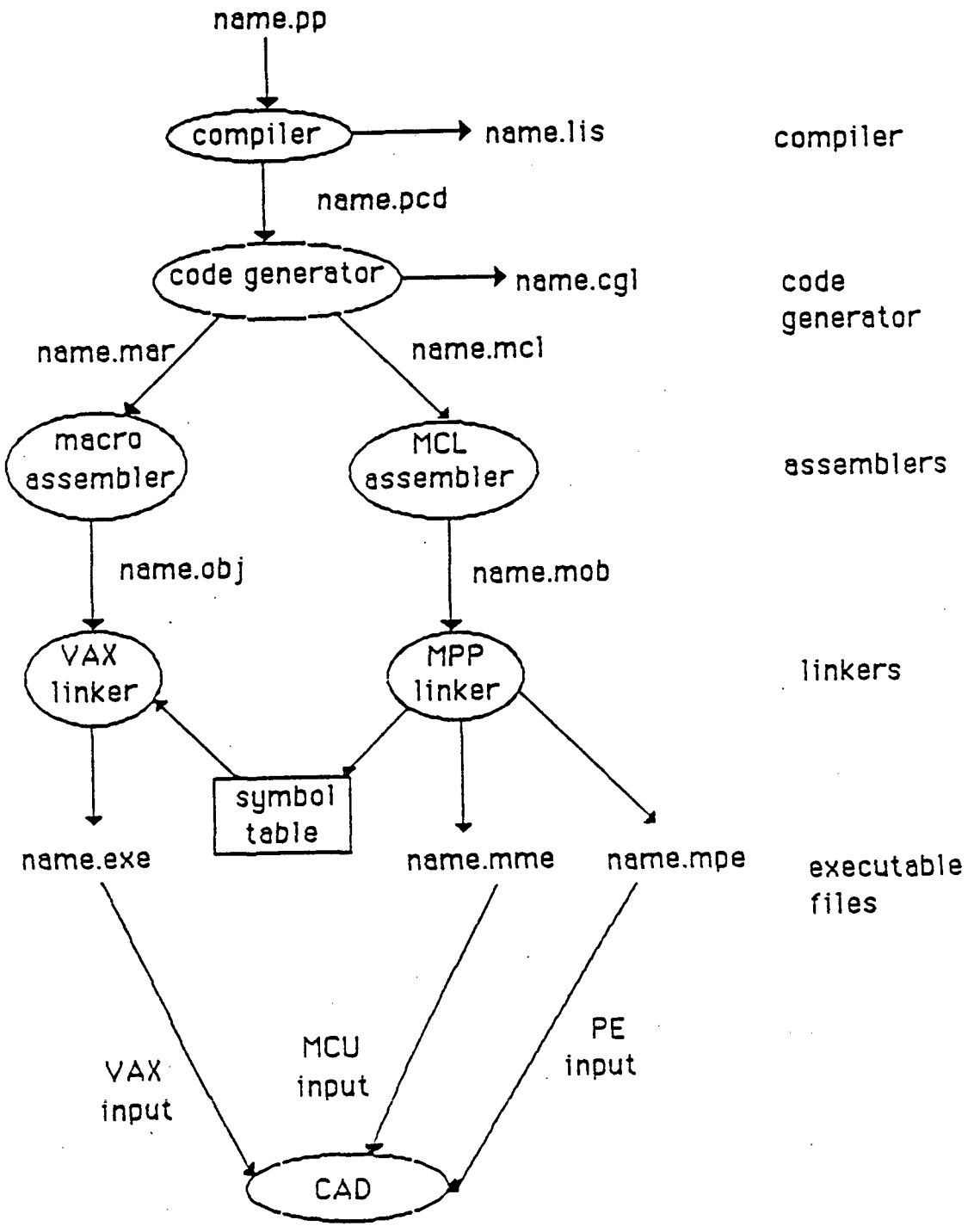


Figure 2.2. Flowchart of MPP-Compiler System

will run on the host. The switch {$D+} enables symbol debugging information to be passed to the assemblers. The default is {$D-}, that is, option disabled. More information on these

switches and others can be found in the Parallel Pascal User's guide. [8]. The code generator, which executes after the compiler, produces the following files:

- *filename.cgl*: a debug file
- *filename.mar*: VAX assembly code
- *filename.mcl*: MPP assembly code.

After the assembly files are produced, two different assemblers need to be used. One is the VAX 11/780 assembler and the other one is the MPP assembler.

The VAX assembler is called by the command:

$ MACRO filename or filename.MAR

where MAR is the default extension for files to be assembled by the VAX assembler. Its output is an object file with extension OBJ.

The MPP assembler is invoked by:

$ MCL filename or filename.MCL

where MCL is the default extension for files to be assembled by the MPP assembler and stands for Main Control Language. By default it outputs an object file filename.MOB. An MCL listing can also be created by using the /LIS qualifier.

The next step is to link the assembled programs. Two links are necessary. The first one, for the MCU-resident code, provides a symbol table that will be used in conjunction with the second link. It is invoked by the command *MPPLINK*. The second link is called *CADLNK* and is used to link together the symbols from the MCU code to the ones from the VAX resident code. These two links have to be performed in this order.

The symbol table created by the MCU linker contains symbol definitions for all global symbols in the image. It is created by default if an executable file is created. Its extension is STB. This symbol table is used as input to the subsequent VAX linker. The VAX linker is called by:

$ CADLNK filename(.OBJ), filename.STB, (PPDEVRUN/lib)

When the VAX linker is invoked, the host-resident object module is linked to the global symbols defined by the MPP linker. An extra runtime library can also be linked with the program. For the moment, to use this library the statement added to the CADLNK command is PPDEVRUN/lib.

The MPPLINK produces two files, filename.MME and filename.MPE. Filename.MME and filename.MPE contain the executable code for the MCU and the PE array respectively. The VAX linker produces filename.EXE, which contains the executable code for the VAX.

To execute the program the command

$ CAD filename(.MME) filename(.MPE) filename(.EXE)

has to be given. If compilation, assembly and linkage didn't produce any errors, then that command can be typed after successful completion of the *rmpp* command, which includes all the commands described above.

The CAD command will initiate a CAD (Control and Debug) session. [9]. The Control and Debug (CAD) program is an interactive program that allows the user to control the MPP. Its debugging portion allows the programmer to:

a) load MPP programs.

b) control execution of the program.

c) display data.

d) detect, locate and patch errors.

The total program compilation takes a long time. Therefore, it is usually worthwhile to execute the compilation in batch mode. This frees the terminal for further work, which is especially helpful when all the communication is done over a long distance phone line.

In order to submit a program for batch processing all that is required is to create a command file that sets the default directory to the directory where the file is located and then gives the command to be executed in batch. The general format of this file is:

$ *sd [directory]*

$! command to be executed

Once this file exists the batch task is initialized by typing:

batchm command_file.

The command *batchm* is a simplification that can be used instead of the system command *submpp* followed by several qualifiers. It automatically uses the qualifiers best suited for someone working over the phone line and therefore without direct access to hard copies of log files produced during the batch job.

2.3. MPP Compiler Restrictions

Modifications to standard Parallel Pascal programs for them to run on the MPP due to the limitations listed in table 2.1 were anticipated. However, we found out that since the system is still being developed, there are also several restrictions to the theoretically possible operations. Table 2.2 contains a list of the main compiler restrictions. Substantial additional program modifications were necessary and new programming tools had to be developed.

Table 2.2. MPP Compiler Restrictions.

1) The MPP has no high-level I/O, all high-level I/O is done on the host.
2) MPP-host interface restrictions
   - An MPP-resident routine can only call other MPP-resident routines.
   - An MPP-resident routine may not reference a variable defined in a host-resident routine.
   - Non-parallel arrays cannot be converted to parallel arrays or vice-versa.
   - Parallel arrays can only be passed by reference.
   - A maximum of one kilobytes can be passed at a time.
   - Arguments passed cannot be used as loop counters. (supposedly fixed)
3) Several primitive functions are not currently implemented on the MPP array, e.g. *ord*, *transpose*.
4) The number zero cannot be used as a parameter in the functions shift and rotate.
5) No good timing mechanism to time long operations.
6) Parallel array elements cannot be individually indexed.

For example, since for the moment there is no high-level language I/O on the MPP, and copying of parallel arrays to the VAX is not currently implemented, a mechanism to conveniently inspect parallel array data was necessary. Therefore, a function that accesses any element of an array was created. This function is called *accessmpp*. Its inputs are the matrix whose value is to be accessed and two integers which are the coordinates of the value to be accessed. The top left corner of the matrix has coordinates (1,1). The output from the access function is a single value of the same base type as the matrix. It is the value of the matrix at the coordinates requested. There is also a version of *accessmpp* for boolean matrices. The boolean access function is called *accessmppb*. The two library functions can be found in sections 6.4 and 6.5 of Appendix A.

Both these functions create a temporary mask. This mask is all zeros, except for the position defined by the input coordinates. The value selected by the mask is stored in a matrix whose other values are zero for *accessmpp* and false for *accessmppb*. By using the *sum* and *or* reduction functions, respectively for *accessmpp* and *accessmppb*, the temporary matrix is reduced to a single value which constitutes the output of the access functions. To read several values out of a matrix the access functions have to be called several times. The approach of transferring a single value at the time from the MCU to the host, instead of transferring a whole array, was chosen to avoid problems do to the limited size of arrays that can be passed between the two machines.

The access function is only a first step that allows programmers to verify small parts of their matrix results, given the current constraints imposed by the MPP system. More powerful tools need to be developed to create a high-level I/O for the MPP.

Another restriction is that an MPP-resident routine may not reference a variable defined in a host-resident routine. The solution is to explicitly pass to the MPP-resident routine all variables in the host-routine that have to be referenced.

For the problem with the shift and rotate functions not being able to receive directly the number zero as a parameter, the fix used was to create a local variable and assign to it the

integer zero. Then, that variable was used when invoking the shift or rotate functions.

Possible solutions to the lack of a timing function able to measure long time intervals on the MPP are still being studied. The current timing function is limited by the size of the internal counter. A solution would be to reset the internal counter before it overflows and before exiting the improved printing function, calculate the total time. Another different approach, would involve the performance monitor of CAD. A command file would obtain the memory locations of the lines between which a timing measurements is to done, and would pass those values to the performance monitor. The performance monitor would then set breakpoints at those locations and call the timer.

Once the programs developed were modified to take into account the restrictions and limitations of the MPP, it was possible to verify their correctness directly on the MPP and to obtain some performance measurements. The next two chapters present implementations and performance measurements for image permutation and rotation algorithms and an Ising model.

CHAPTER 3

PERMUTATION AND ROTATION ALGORITHMS

The only permutation which is directly implemented by the MPP is the near neighbor rotate (or shift). The direction of the rotation may be in any of the four cardinal directions. In Parallel Pascal the main permutation functions are multi-element rotate and shift functions; other permutations are built on these primitives.

The rotate function takes as arguments the array to be shifted and a displacement foe each of the array dimensions. For example consider a one dimensional array *a* specified by

```
a: array [0..n] of integer;
```

The rotate statement

```
a := rotate( a, 5);
```

is equivalent to

```
for i := 0 to n do
    a[i] := a[(i + 5) mod (n + 1)];
```

The rotation utilizes the toroidal end around edge connections of the mesh. The *shift* function is similar except that the mesh is not toroidally connected and zeroes are shifted into elements at the edge of the array; therefore, the shift function is not a permutation function in the strict sense. The concept of the rotate and shift functions extend to n dimensions; on the MPP the last two dimensions of the array correspond to the parallel hardware dimensions and are executed in parallel, higher dimension operations are implemented in serial. The cost of the rotate function is dependent on the distance rotated. It also depends on the size of the data elements to be permuted.

3.1. Matrix Permutation

The matrix permutation algorithm which is the basis upon which the rotation algorithm is constructed, is a general algorithm for implementing arbitrary permutations of a two dimensional matrix on mesh connected parallel processors. [7]. It is also capable of performing any onto mapping. It uses a heuristic approach to reduce the execution time.

The permutation of a matrix a is specified by two coordinate matrices c and r which have similar dimensions to a. The permuted matrix b also has the same dimensions as a. For a matrix element b[i,j] the corresponding elements r[i,j] and c[i,j] specify the row and column indices respectively of where the related element of a is located. That is, the permutation is specified by

$$b[i,j] := a[\, r[i,j], c[i,j]\,]$$

More formally, the data arrays involved in the permutation are specified by:

a,b : **array** [1..nrow,1..ncol] of data; {where data is any base type}

r : **array** [1..nrow,1..ncol] of 1..nrow;

c : **array** [1..nrow,1..ncol] of 1..ncol;

In order to compute the relative distance that the data must be moved, two pixel element identifying matrices *idr* and *idc* are precomputed. They contain the following:

idr[i,j] := i;

idc[i,j] := j;

for all i,j.

The relative distances to be moved are then specified by

rr := (*r*-*idr*) **mod** *nrow*;

rc := (*c*-*idc*) **mod** *ncol*;

In a permutation the data may be shifted in any of the four quadrants in order to reach a specified destination. However, in the following algorithms only positive data shifts are

considered, i.e. in the up and left directions. The other three quadrants are covered by using modulo arithmetic for shift distance calculations and implementing data movement with the rotation function which utilizes the end around mesh connections. We have investigated a modified heuristic algorithm which checks all four quadrants and moves in the optimal direction. Fewer data shift operations are required but the overhead due to checking alternative directions is significantly higher.

3.1.1. A Simple Permutation Algorithm

A simple naive algorithm to achieve an arbitrary permutation is to slide a over all the possible positions of b, assigning the specified elements of a to each element of b when they are in the correct position.

```
for i:= 1 to nrow do
    begin
       for j:= 1 to ncol do
            begin
               where (rr = i) and (rc = j) do
                              b := a;
               a := rotate(a, 0, 1);
            end;
       a := rotate(a, 1, 0);
    end;
```

This algorithm involves $O(n^2)$ operations for an n x n matrix.

3.1.2. The Heuristic Algorithm

In many permutations which occur in practice there are well defined patterns for the data. For example, near neighbor shifts are trivial with complexity $O(1)$, perfect shuffles can be implemented in $O(n)$ time. The heuristic algorithm attempts to take advantage of the fact that rr and rc will be the same or similar for many elements. This is particularly true for operations such as matrix warping.

The algorithm first slides (rotates) a as many locations up and left as possible such that future backtracking will not be necessary. If any element of a is correctly positioned over b

(i.e. (rr = 0) and (rc = 0)), then b is updated. Otherwise, atr, which is a copy of the current version of a, is slid in the upwards direction until all outstanding elements of b, for which the current rc = 0, are satisfied. The algorithm then shifts as far as possible up and left again and repeats the above procedure until all elements of the result mask are false, i.e. b is complete.

The following variables are used in the algorithm:

Variable declaration

mask,masktr : **array** [1..*nrow*, 1..*ncol*] of **boolean**;

atr : **array** [1..*nrow*, 1..*ncol*] of data;

rrt : **array** [1..*nrow*, 1..*ncol*] of 0..*nrow*;

ri, *rit*, *lastrit* : 0..*nrow*;

ci : 0..*ncol*;

Variable functions:

mask : the result mask, true values indicate elements of b which have not yet received the correct element of a.

ri, *ci* : row and column distances for the up-left move.

masktr : a version of *mask* to process one column.

rit : a version of *ri* used to process one column.

atr : a version of a used to process one column.

rrt : a version of *rr* used to process one column.

lastrit : the last value of *rit*.

The Parallel Pascal version of the heuristic algorithm is as follows:

```
lastrit :=0;
b := a;
mask := (rr <> 0) or (rc <> 0);

while any(mask, 1 ,2) do
      { iterate until the permutation is complete }
      begin
          ri := min(rr, 1, 2);
          ci := min(rc, 1, 2);
          { move up and left as far as possible }
          a := rotate(a, ri, ci);
          rr := rr - ri;
          rc := rc - ci;
          masktr := (rr = 0) and (rc = 0);
          { satisfy elements for the current position }
          if any(masktr, 1, 2) then
              atr := a
          else
              {satisfy each element for the given column}
              begin
                  where rc = 0 do
                      rrt := rr
                  otherwise
                      rrt := nrow;
                  rit := min(rrt, 1, 2);
                  masktr := rrt = rit;
                  { the next seven statements implement }
                  { the statement atr = rotate (a, rit, 0) }
                  { but also take advantage of the previous }
                  { shifts }
                  if ci <> 0 then
                      begin
                          atr := a;
                          lastrit := 0;
                      end;
                  atr := rotate( atr, rit - lastrit, 0 );
                  lastrit := rit;
              end;
          { update b for the current location of a }
          where masktr do
              begin
                  b := atr;
                  rr := nrow;
                  rc := ncol;
                  mask := false;
              end;
      end;
```

This algorithm is bounded by n^2 iterations. However, this must be considered a loose bound since we currently do not know a permutation which would require all n^2 iterations.

The algorithm requires one iteration for a positive single element shift permutation but n-1 operations for a negative shift since the rotate is in the wrong direction.

3.1.3. Algorithm Cost

The cost of the naive algorithm is proportional to the number of rotate operations, i.e. $(n-1)^2$ * (the cost of a one element rotate operation plus two comparison operations). The heuristic algorithm has two major cost components: the rotate operations as noted before and the (min) reduction functions. The reduction functions are used to compute the multi-element distance for moves. In the tables for the performance of the algorithm, both the total number of element rotates and the total number of reduction operations are given.

The relative cost of a rotation and reduction is both system and data size dependent. For the MPP, the cost of a reduction function is in the order of 4.2 μs whereas the single element rotation of 32-bit data requires in the order of 3.2 μs to 9.6 μs depending upon the number of successive rotate operations. Therefore, the reduction functions may represent a significant portion of the computation cost. With careful low level programming the reduction operations can be overlapped with data rotate operations such that their effective cost is in the order of 1.4 μs. If the MPP was augmented with a small amount of additional hardware similar to that outlined in[10] then the reduction time could be reduced to 1.5 μs over half of which could could be overlapped with data rotation operations. The heuristic algorithm always requires less iterations and rotations than the naive algorithm; however, the additional overhead of the reduction function may make it less efficient in some instances.

3.1.4. Permutation Results

The results of some permutations performed in order to obtain rotated matrices of size 32 x 32 are given in table 3.1 and 3.2. These rotations are into mappings rather than permutations (see section 3.3.1 for details). For comparison, the naive algorithm requires 961 iterations, 961 rotate operations and zero reductions for any 32 x 32 matrix permutation or mapping.

Table 3.3 contains the results for perfect shuffle permutations for different size matrices. The result for perfect and inverse shuffles are identical for any matrix size.

The perfect shuffle is an example of a permutation which does not exhibit the locality property. The number of algorithm iterations needed to implement shuffles directly is $(n-1)^2$. However, the separability property of the two dimensional shuffle is not being used. If we use the permutation algorithm to the permutation in two stages, i.e. first shuffle the rows and then shuffle the columns, then $n-1$ iterations are needed for each permutation. Therefore, the perfect shuffle when implemented directly has complexity $O(n^2)$, but when computed in two stages the algorithm is much more effective and has $O(n)$ complexity. The results of implementing the perfect shuffle as two separable shuffles are also given in Table 3.3. Table 3.4 shows the results of a random permutation; this demonstrates that the heuristic is not effective when the mapping does not possess the locality property.

3.2. Large Arrays

Frequently the data to be processed by a parallel processor will be in the format of arrays which exceed the fixed range of parallelism of the hardware. Therefore, it is necessary to have special algorithms that will deal with large arrays by breaking them down into blocks manageable by the hardware, without loosing track of the relationships between different

Table 3.1: Cost for a near neighbor rotation on a 32 x 32 matrix centered at 16 16.

Angle of rotation	Matrix rotation mapping cost		
	iterations	rotations	reductions
0	0	0	0
15	124	562	340
30	262	620	748
45	505	837	1464
60	741	1022	2163
75	724	1019	2119
90	528	1007	1552

Table 3.2: Cost for a near neighbor rotation on a 32 x 32 matrix centered at 1 1.

Angle of rotation	Matrix rotation mapping cost		
	iterations	rotations	reductions
0	0	0	0
15	235	304	666
30	437	517	1266
45	625	683	1825
60	779	829	2292
75	889	956	2631
90	993	992	2946

Table 3.3: Cost for perfect shuffle permutations for different matrix sizes.

matrix size	Direct Shuffle Cost			Separable Shuffle Cost		
	iterations	rotations	reductions	iterations	rotations	reductions
4 x 4	9	12	22	6	6	6
8 x 8	49	56	134	14	14	14
16 x 16	225	240	646	30	30	30
32 x 32	961	992	2822	62	62	62

Table 3.4: Permutation cost for a random permutation.

matrix size	Random Permutation		
	iterations	rotations	reductions
32 x 32	630	995	1851

blocks.

One scheme, which is frequently used on the MPP, is to partition the large array into blocks which are conveniently stored in a four dimensional array. The range of the first dimension of this array specifies the number of blocks in each row of the large matrix and the range of the second dimension specifies the number of blocks in each column. Given a conceptual large matrix

mx : **array** [0..x,0..y] of btype;

which is to be stored in an array *a* of type

array [1..n, 1..m, 1..p, 1..q] **of** btype;

Element *i,j* of the large matrix is mapped into the array *a* as specified by

mx[i,j] = *a* [1+*i* **div** p, 1+*j* **div** q, 1+*i* **mod** p, 1+*j* **mod** q]

For example, a 512 x 256 matrix could be stored in eight blocks as:

la : **array** [1..4,1..2,1..128,1..128] **of real**;

This data structure allows blocks to be manipulated independently. However, it still preserves the positional relationships of those blocks in the original large matrix.

To simplify the manipulation of large arrays on the MPP, two Parallel Pascal library functions *lrotate* and *lshift* have been developed. These functions take an array argument and two displacement arguments, like the primitive matrix rotate and shift functions, however, in this case the array argument is a four dimensional array which is treated like a conceptually large matrix.

Many programs can be converted to operate on blocked rather than conventional matrices by simply replacing all instances of rotate and shift with lrotate and lshift respectively. This is true for the permutation programs presented; however, in the case of the heuristic permutation algorithm, this is not a very efficient solution. A better method is to scan through the result blocks and perform permutations on only the input blocks that contribute to the current result block being processed. This algorithm is shown below.

```
var
    la,lb: array [1..n,1..m,1..nrow,1..ncol] of data;
    lr,lc: array [1..n,1..m,1..nrow,1..ncol] of index;

begin
    for i = 1 to n do
        for j = 1 to m do
            begin {process each result block}
                rb := 1 + lr[i,j] div nrow;
                cb := 1 + lc[i,j] div ncol;
                ro := 1 + lr[i,j] mod nrow;
                co := 1 + lc[i,j] mod ncol;
                for k = 1 to n do
                    for l = 1 to m do
                        begin {consider each input block}
                            maskb := (rb = k) and (cb = l);
                            if any(maskb, 1, 2) then
                                where maskb do
                                    lb[i,j] := perm2(la[k,l],
                                     ro,co,maskb);
                        end;
            end;
end;
```

Perm2 is the heuristic algorithm presented previously with the modification that the initial mask value is passed as an argument. That is, only elements selected by the mask are permuted. An additional speedup is achieved by this since the heuristic works much better when only a subset of elements are to be permuted.

3.3. Matrix Rotation

The permutation function described above serves as the basis for a general rotation algorithm. Three rotation techniques are considered: nearest neighbor, bilinear interpolation and bicubic interpolation. A rotation is specified by three parameters: the location of the origin of rotation (r_0,c_0) and the rotation angle θ . The starting point of all rotation algorithms is the generation of the mapping matrices r and c from these parameters.

3.3.1. Nearest Neighbor

The nearest neighbor algorithm is simply an into mapping in which a result element is assigned the value of the nearest rotated matrix element. In this case the new row and

column coordinate matrices, r and c, are defined as follows:

$$r[i,j] = \text{round}((c_0 - j)sin(\theta) + (i - r_0)cos(\theta) + r_0)$$

$$c[i,j] = \text{round}((j - c_0)cos(\theta) + (i - r_0)sin(\theta) + c_0)$$

for all i and j; any values of the result which have near neighbors outside the range of the input matrix are set to zero.

In performing a rotation, some elements are rotated off the result matrix and some elements are selected which are outside of the input matrix. In our algorithm result elements for the latter case are simply set to zero. Therefore we have a permutation in which a subset of the input elements map into a subset of the result elements; the size of these subsets depends upon the angle and origin of the rotation. The rotation is achieved by using the valid elements of r and c with the heuristic permutation algorithm.

3.3.2. *Bilinear Interpolation*

For the bilinear interpolation algorithm a result element is computed from a weighted sum of the four rotated input matrix elements which surround it. There are two possible approaches to implementing this scheme. First, we can compute four permutations; each permutation acquiring one of the four neighbors for each element. This is called *multiple permutations*. This was the approach used until now.

The second method is to perform one permutation and then seek the local neighborhood of the rotated input matrix for the other near neighbors. The idea being that a local search will require less computation than four complete rotations especially when the angle of rotation is large. The local neighborhood of a single rotated matrix does not contain a complete set of the near neighbor elements of the input matrix; some are lost due to grid spacing differences. A complete set can be guaranteed, however, if we also include the local neighborhood of a slightly perturbed, rotated input matrix. This scheme is called the *double permutation* method; both rotated matrices can be computed simultaneously with a single execution of

a slightly modified heuristic permutation algorithm. If we map a result element back to the input matrix, it will be surrounded by four elements P1 - P4 as shown in figure 3.2. These points are moved to the processing element associated with the result P and the interpolation is then computed in parallel.

For the interpolation algorithms, the matrices, rp and cp, contain the actual locations of the rotated elements.

$$rp[i,j] = (c_0 - j)(\sin\theta) + (i - r_0)(\cos\theta) + r_0$$

$$cp[i,j] = (j - c_0)(\cos\theta) + (i - r_0)(\sin\theta) + c_0$$

for all i and j.

The coordinates of the near neighbors are as follows:

$$r_{top} = \left\lfloor (c_0 - j)(\sin\theta) + (i - r_0)(\cos\theta) + r_0 \right\rfloor$$

$$c_{left} = \left\lfloor (j - c_0)(\cos\theta) + (i - r_0)(\sin\theta) + c_0 \right\rfloor$$

$$r_{bottom} = r_{top} + 1$$

$$c_{right} = c_{left} + 1$$

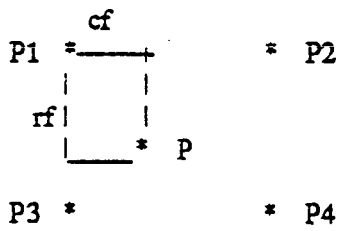


Figure 3.1. Bilinear Interpolation

The interpolation fractions are:

$$rf = rp - r_{top}$$

$$cf = cp - c_{left}$$

Once the points P1 - P4 have been obtained the interpolated result is computed as follows:

$$P = (1 - cf)*(1 - rf)*P1 + (1 - rf)*cf*P2 + (1 - cf)*rf*P3 + cf*rf*P4$$

The algorithm for the multiple permutation approach is as follows:

```
begin
        r := r_top;
        c := c_left;
        b := coef1 * perm(a, r, c);
        { value of top left neighbor }
        c := c + 1 ;
        b := coef2 * perm(a, r ,c) + b;
        { value of top right neighbor }
        r := r + 1 ;
        b := coef3 * perm(a, r, c) + b;
        { value of bottom right neighbor }
        c := c - 1 ;
        b := coef4 * perm(a, r, c) + b;
        { value of bottom left neighbor }
end;
```

Perm is the heuristic permutation function. The final result of rotating a is stored in the matrix b.

The double permutation approach uses a modified permutation function which creates the following matrices:

$$b[i,j] := a[\, r[i,j],\, c[i,j]]$$

and

$$d[i,j] := a[\, r[i,j],\, c[i,j] + 1\,]$$

where

a is the original matrix

b is the rotated matrix

d is the shifted rotated matrix

r is the row coordinate matrix

and

c is the column coordinate matrix

To avoid loosing the value of the center of rotation, when the origin is located to the right of the matrix center, the matrix is shifted left and, therefore, in the equation defining matrix *d* we substitute a negative one for the constant one.

The second step in the double permutation algorithm is a local search performed on both rotated and shifted rotated matrices in order to find all the values of the elements needed for the interpolation. The local search has a constant maximum cost for any size matrix. It therefore has an advantage over the multiple permutations approach, since every permutation in that approach will become more costly as the matrix size increases.

For the worst case rotation angle ($\theta = 45$), it has been determined that a local search in a 5 x 5 window is sufficient to yield the values of all the elements needed to perform a bilinear interpolation. The local search strategy implemented in our algorithm is a spiral search. The elements are selected by comparing their row and column coordinates to those needed. Once they match, their values can be obtained from the rotated matrix or from the rotated shifted matrix.

3.3.3. Cubic Interpolation

The cubic interpolation version of the rotation algorithm is a simple extension of the bilinear interpolation scheme. The first step, finding the coordinate matrices *r* and *c*, is identical to the bilinear interpolation case. After obtaining these matrices, the values of sixteen neighbor points must be acquired. If the *multiple permutations* approach is used, then sixteen separate permutations will be required. However, with the *double permutation* scheme only a small

extension of the bilinear algorithm is needed. Instead of using a 5 x 5 window, which is the case when four points have to be found, a 7 x 7 window is necessary to find sixteen points. However, since the element will have rotated in a specific direction, the search window can be reduced to a 7 x 5 window. Each row of points needed will use a different 7 x 5 window of search.

Once all the values needed are found, the bicubic interpolation itself is done by, first, performing a cubic interpolation for each of the four rows and, then, performing a fifth cubic interpolation on the row points obtained.

As shown in figure 3.3, the reference point for the cubic interpolation computed in step one is P6. The first four cubic interpolations are performed to obtain points pa, pb, pc and pd. The fifth one yields the value of point P.

3.3.4. Test Results

The local search performed in the double permutation algorithm has a constant maximum cost for any size matrix. For any matrix the maximum cost is 100 rotations and 25 reductions for the bilinear interpolation method and 552 rotations and 525 comparisons for the cubic interpolation.

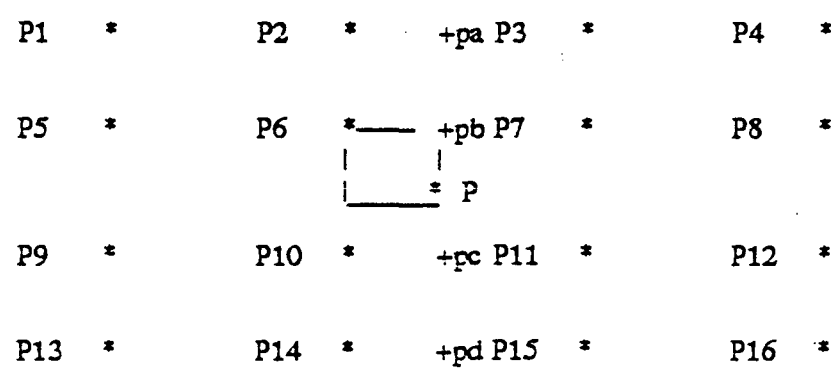


Figure 3.2. Bicubic Interpolation

The results for rotations using bilinear interpolation for a 32 x 32 matrix are given in table 3.5 and 3.6 for different centers of rotation. The results of rotations applying cubic interpolation are given in tables 3.7 and 3.8.

3.4. The MPP Implementation of the Rotation Algorithm

Some simple modifications of the rotation programs were necessary for them to run on the MPP. However, after these changes were made it was possible to obtain values for the number of rotations and reduction functions required during the permutation and, when it

Table 3.5: Cost of bilinear interpolated rotation centered at coordinates 16 16.

angle of rotation	Double perm.		Multiple perm.	
	rotations	reductions	rotations	reductions
0	1188	184	4092	649
15	850	374	2488	1416
30	1011	766	2544	2978
45	1448	1503	3292	5916
60	1858	2174	4084	8602
75	1846	2151	4081	8491
90	1793	2001	4090	7920

Table 3.6: Cost of bilinear interpolated rotation centered at coordinates 1 1

angle of rotation	Double perm.		Multiple perm.	
	rotations	reductions	rotations	reductions
0	13	3	4	6
15	657	703	1342	2705
30	1050	1302	2035	5093
45	1389	1834	2683	7238
60	1702	2329	3291	9230
75	1931	2669	3471	10627
90	2086	2971	3968	11780

Table 3.7: Cost of cubic interpolated rotation centered at coordinates 16 16

angle of rotation	Double perm.		Multiple perm.	
	rotations	reductions	rotations	reductions
0	1640	684	16368	2608
15	1302	874	10072	5757
30	1463	1266	10394	12021
45	1900	2003	13300	23668
60	2310	2674	16341	34371
75	2298	2651	16315	33928
90	2245	2501	16360	31664

Table 3.8: Cost of cubic interpolated rotation centered at coordinates 1 1

angle of rotation	Double perm.		Multiple perm.	
	rotations	reductions	rotations	reductions
0	552	525	272	30
15	1109	1203	7683	11132
30	1502	1802	9723	20589
45	1841	2334	11815	29104
60	2154	2829	13793	37047
75	2383	3169	15316	42586
90	2538	3471	15872	47088

exists, the interpolation phases. Tables 3.9 through 3.12 contain those values for the near neighbor and the bilinear interpolation rotations. These values show that the heuristic permutation and rotation algorithms are efficient when compared to the naive algorithm. From these values, expected times, included in tables 3.13 to 3.16, were calculated. It was also possible to obtain timings for the total running time of the program. The total time includes the host running time, the MPP running time and also the time necessary to set up all calls to the MPP. Several runs were effected to obtain the average values in tables 3.13 through 3.16. The values for the total time obtained in the several runs varied approximately 1.5 sec in each direction. Such a large variation is to be expected in the total time since it involves the host running time which is highly dependent on the system load. The host running time involves

only the input of the parameters necessary for the rotation like the center and the angle of rotation.

The running time on the MPP is calculated using timing functions created by Jim Abeles at NASA Goddard Space Center. These timing functions are:

pfm_init

pfm_start(name:integer);

pfm_stop(name:integer)

pfm_close

They use the CAD performance monitor to calculate the time taken by a section of code. The first function to be called is *pfm_init*. It clears a counter register and attaches the performance monitor to the MCU. The section of code to be timed has to be placed between a *pfm_start* call and a *pfm_stop* call, both using the same integer to identify the section. When a call to *pfm_start* is found, the counter starts being incremented. Once the corresponding call to *pfm_stop* is reached, a call to the VAX is made, the VAX reads the value from the counter register in the MCU and the difference between the starting and final values of the counter is computed. The values accumulated for the different sections of code timed and their total are printed when *pfm_close* is called. At that time the performance monitor is also detached from the MCU.

Tables 3.13 and 3.14 consist of the timings for the near neighbor rotation, tables 3.15 and 3.16 for the bilinear interpolation rotation. The MPP times include a variable overhead of approximately 0.1 sec for each call made to an MCU-resident procedure. This overhead was determined by timing a call to a dummy MCU-resident procedure. This dummy procedure contained no statements in it. This overhead occurs because when an MCU-resident routine is called the UNIBUS link between the host machine and the MCU is used. Therefore, the transfer of about 1000 bytes involved in calling a routine takes approximately 1 msec. To call a routine and then return would take twice that time, taken into account only the transfer of information. However, to this has to be added also the time necessary to interface with the

VAX operating system. For the near neighbor and bilinear interpolation rotations, four calls to MCU-resident routines were made in each case. These calls explain the difference of approximately 0.4 sec between the MPP times and the estimated times, which otherwise were very similar.

The estimated times were calculated by considering the cost in microseconds of several operations. These costs are based on an instruction cycle time for memory access and operation on the MPP of 100 ns. For example, a shift or rotate operation costs 2n where n is 0.1 μ sec for Boolean, 0.8 μ sec for integer and 3.2 μ sec for real variables. A cost of 2n was assigned to elementary Boolean operations. Arithmetic operations were assigned a cost of 3n when they involved two operands and an assignment.

A comparison between the ratios of the number of shifts to the n^2 shifts for a 32 by 32 matrix and a 128 by 128 matrix, where n is the size of the matrix, demonstrates that even if the size of the matrix increases, the ratios remain very similar for the same rotation angles which is a good characteristic of this algorithm. Figures 3.4 and 3.5 show these ratios.

Table 3.9: Cost for a near neighbor rotation on a 128 x 128 matrix centered at 1 1.

Angle of rotation	Matrix rotation mapping cost	
	rotations	reductions
0	0	0
15	5114	7506
30	8495	14046
45	11103	20050
60	13504	25121
75	15516	28802
90	16256	32385

Table 3.10: Cost for a near neighbor rotation on a 128 x 128 matrix centered at 64 64.

Angle of rotation	Matrix rotation mapping cost	
	rotations	reductions
0	0	0
15	9183	3594
30	10446	7800
45	13589	15777
60	16378	23359
75	16375	24331
90	16319	16384

Table 3.11: Cost for a bilinear interpolation rotation on a 128 x 128 matrix centered at 1 1.

Angle of rotation	Matrix rotation mapping cost	
	rotations	reductions
0	101	25
15	9009	7555
30	15689	14107
45	21246	20034
60	26228	21192
75	30023	28893
90	32614	32410

Table 3.12: Cost for a bilinear interpolation rotation on a 128 x 128 matrix centered at 64 64.

Angle of rotation	Matrix rotation mapping cost	
	rotations	reductions
0	101	25
15	11510	3628
30	14648	7826
45	21556	15800
60	28282	23378
75	28763	24382
90	24676	16409

Table 3.13: Timing of near neighbor rotation on a 128x128 matrix centered at 1 1.

Angle of rotation	Matrix rotation timing (sec.)		
	total CPU time	MPP time	Estimated time
0	5.49	0.54	0.01
15	5.56	0.61	0.04
30	5.41	0.47	0.07
45	5.50	0.58	0.10
60	5.46	0.55	0.12
75	5.15	0.58	0.14
90	5.42	0.48	0.16

Table 3.14: Timing of near neighbor rotation on a 128x128 matrix centered at 64 64.

Angle of rotation	Matrix rotation timing (sec.)		
	total CPU time	MPP time	Estimated time
0	5.43	0.42	0.01
15	5.36	0.42	0.03
30	5.21	0.47	0.05
45	5.50	0.52	0.08
60	5.35	0.54	0.12
75	5.37	0.56	0.13
90	5.45	0.47	0.09

Table 3.15: Timing of bilinear interpolated rotation on a 128x128 matrix centered at 1 1.

Angle of rotation	Matrix rotation timing (sec.)		
	total CPU time	MPP time	Estimated time
0	5.54	0.57	0.01
15	5.64	0.65	0.05
30	5.78	0.82	0.08
45	5.45	0.48	0.10
60	5.51	0.54	0.13
75	5.61	0.62	0.16
90	5.54	0.57	0.19

Table 3.16: Timing of bilinear interpolated rotation on a 128x128 matrix centered at 64 64.

Angle of rotation	Matrix rotation timing (sec.)		
	total CPU time	MPP time	Estimated time
0	5.50	0.52	0.01
15	5.39	0.44	0.03
30	5.41	0.46	0.05
45	5.53	0.53	0.10
60	5.44	0.59	0.14
75	5.46	0.59	0.15
90	5.41	0.48	0.10

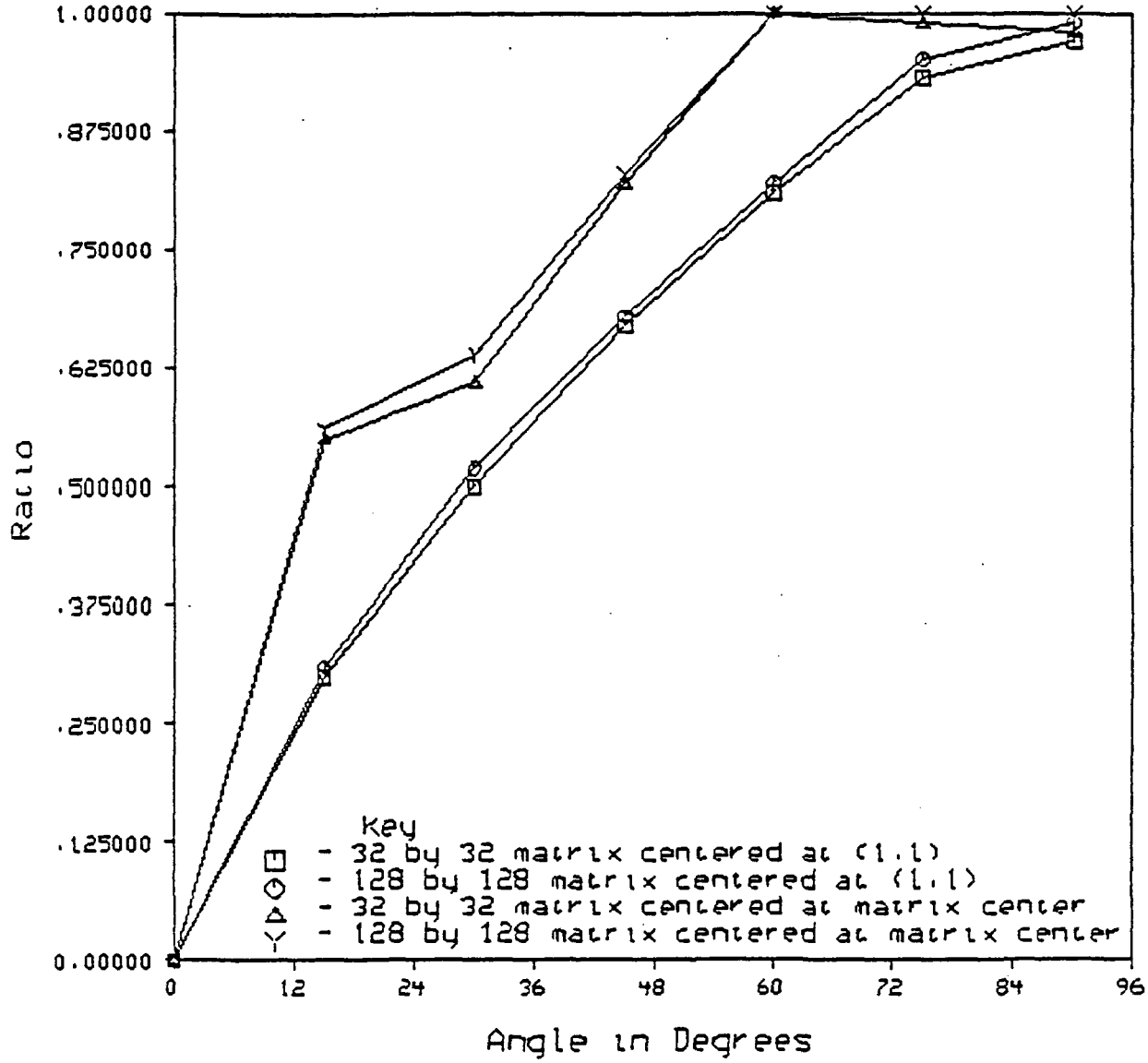


Figure 3.3. Ratio of number of rotations to n^2 rotations for near neighbor rotation with center of rotation at 1 1 and at matrix center.

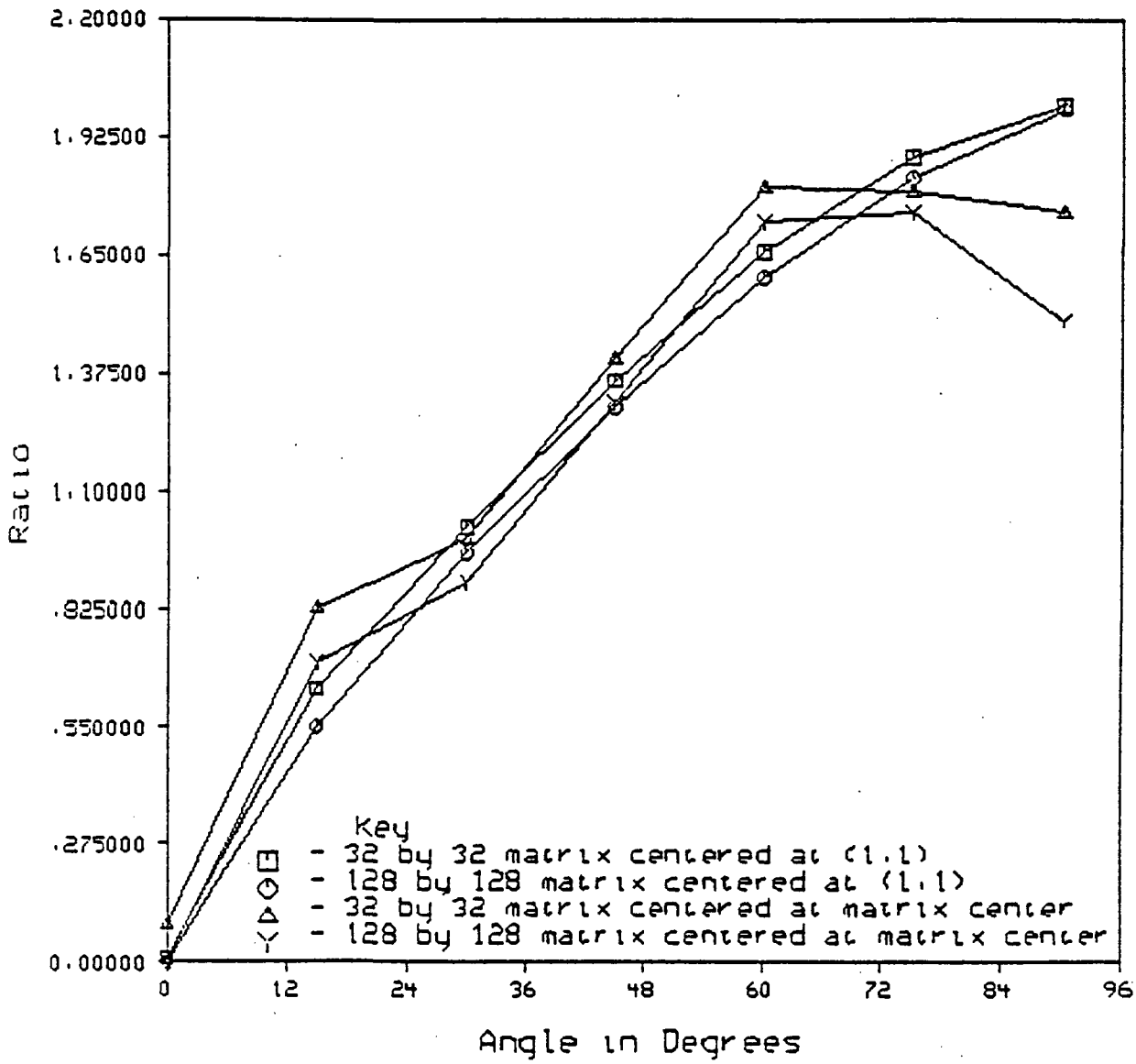


Figure 3.4. Ratio of number of rotations to n^2 rotations for bilinear interpolation rotation with center of rotation at 1 1 and at matrix center.

CHAPTER 4

THE ISING MODEL

Another application program developed for the MPP is an Ising model simulation. The Ising model is a simplified model of a magnet in which the atomic magnetic moments or spins are arranged on a lattice and can point only up or down. To completely specify the microscopic state of the system, it is enough to assign a value of +1 (up) or -1 (down) to spins assigned to each point on the lattice. The Ising model is a suitable model for implementation on the MPP since interactions between spins can only occur with the nearest neighbors on the lattice. Such a structure maps well onto the physical configuration of the MPP.

4.1. The MPP Implementation

The purpose of this Ising model simulation is to calculate the energy as the sum over a lattice of the product of neighboring spins divided by the number of interactions between those spins. This energy is related to the strength of a spin-spin interaction k as well as to the strength of the interaction with an imposed magnetic field h.

The lattice chosen for this simulation is a three-dimensional array of spins of dimensions 128 by 128 by 128.

Since the Ising model has only two possible values for the spins: -1 and +1, the energy could be easily calculated by summing over the four possible states of spin interaction. However, the number of states is an exponential function of the number of spins. For example, even a small 10 by 10 lattice already has 2^{100} configurations to sum over. The lattice we consider in this simulation will have $2^{2097152}$ possible configurations.

It is obvious that a direct approach to this problem is not possible. Another solution would be to randomly select states and, based on them, estimate the sum. Unfortunately, the

majority of states in most systems with a large number of particles do not make a significant contribution, being energetically unfavorable. [11]. Therefore, random sampling is an inefficient and impractical solution.

The approach taken in most cases and also in this simulation is to generate states with the probability they would have in nature. The method is called a Monte Carlo simulation because random numbers are an important feature of this approach. A probabilistic model of the system is used to determine numerical solutions. In other words, instead of analyzing the system as a whole, we can analyze the behavior of a large number of individual particles. Those particles exhibit a random behavior based on the model chosen. By considering each particle as a separate experiment and taking the statistical average of many experiments, we can obtain the necessary information about the system. [12].

The algorithm chosen can be summarized as follows:

- generate a sequence of configurations by modifying each spin according to the energy change, its neighbors influence and a comparison with a random number.
- for each state, which is formed by a sequence of configurations and depends on the value of the spin-spin interaction, calculate the average total energy.

In more details for our simulation to generate different configurations we start at an arbitrary state. That arbitrary state is first created by initializing a set of 40-bit shift registers used to generate random numbers and a set of bits, in our case four bits, that will be used as a feedback at the next step to generate the new set of random numbers.

The bits of the shift register set are initialized and modified depending on the results of a random number generator. The bits of the integer returned are assigned to the shift registers. There are no limits set on the size of the random integer returned by the random number generator.

At high temperatures the spin-spin interaction k is small and the spins are almost independent. On the other hand, at low temperatures the spin-spin interaction k is large and

the probability that the spins will be found in the same state increases. Therefore, after the first arbitrary state has been defined, for every new value of k a new table of probabilities for a positive spin has to be created. That table gives the probability of a positive spin for each possible sum of neighbors according to the energy associated with the spin-spin interaction and the imposed magnetic field interaction h.

In this simulation, we calculate the effect of the six nearest neighbor spins on each spin on every point on the lattice. The six nearest neighbors are those located to the up, down, east, west, south and north directions. The possible spin sums for the neighbors range from -6 to +6. The probabilities for every sum are an exponential function of the spin-spin interaction and the magnetic field interaction. They are defined as follows:

$$prob(i) = \frac{e^{ki+h}}{e^{ki+h}+e^{-ki+h}}$$

where i is one of the possible sums.

After the probability table is initialized, the spins in the three-dimensional lattice can be recalculated. Our simulation allows us to define the number of configurations or times the spin lattice will be recalculated. That determines how many configurations will form a state and so, how many configurations will be involved in determining the energy at each state.

To recalculate the spins in the lattice we proceed plane by plane. There are 128 planes and each one of them is a 128 by 128 matrix. For each plane the sum of spins of the six nearest neighbors is calculated at each site. Since this simulation was implemented on the MPP, a parallel processor with dimensions 128 by 128, all those sums can be calculated in parallel.

After the sum of the neighboring spins is calculated, a new state is calculated by using the set of bits and shift registers defined at the beginning of the simulation. It is here that the set of bits is used, by exclusive-oring them with the old register array, to create the new register array.

Then, the spins are reset according to the probability table. However, all the spins are not reset at once. A boolean mask is initialized to a checkerboard pattern and applied to each one of the planes of the lattice. Where that mask is true the spins are reset according to the spin sum of their neighbors. Once all the possible spin sums have been considered, the mask is inverted.

At that point the same sequence of events, from the computing of all the neighbor sums to the resetting of the spins over the masked lattice, is repeated. At that point a new next state has been generated. For each one of the states created the average total energy for each spin-spin interaction is calculated by taking the mean of the sums over the configurations generated.

Two different implementations of the Ising model were timed on the MPP. Their difference was simply on the way the calls were made to the MCU-resident routines. In one version, all the calls to the MCU routines were grouped under one general procedure which was called by the VAX. In the other version, each one of the MCU routines were called directly from the VAX. The only real difference between these two approaches when timing is concerned is that the second version takes approximately 0.1 sec for each MCU procedure call. In this program, this is not a very significant difference.

Different parts of the Ising model were timed. Table 4.1 lists average times in microseconds for several sections of the Ising program. Section 1 is the random number generator that returns a positive random integer. This random number generator is called 16384 times to completely initialize the random matrix. Section 2 corresponds to the initialization of the table of probabilities which gives the probability of a positive spin for each possible sum of neighbors. This table is reinitialized every time the strength of the spin-spin interaction changes. Section 3 calculates for a given bit plane the sum of spins of neighbors at each site. It is computed the number of iterations selected and, for each iteration, it is computed 128 times so that all bit planes are covered. Section 4 resets the spins according to the probabilities calculated in section 2. It is computed the same number of times as section 3.

Table 4.1. Timings of sections of the Ising program

Sections of Ising	Measured Times (μ sec)	Expected Times(μ sec)
Section 1	16.2	27
Section 2	4431.8	4500
Section 3	185.2	168
Section 4	32290.2	32330

The expected values were very close to the obtained values when the operations involved mainly array manipulations.

CHAPTER 5

CONCLUSION

This thesis presented both tools and algorithms adapted to the Massively Parallel processor. The system and programming tools were created to be used in conjunction with the Paralel Pascal development and MPP-compiler systems.

The tools include a command file that combines library preprocessing, translation from Parallel Pascal to standard Pascal and compilation in one step, a command file that simplifies the compilation, assembly and linking of Parallel Pascal programs and access functions to overcome the current lack of high-level I/O functions to transfer parallel arrays from the MPP to the host machine.

All the features mentioned above create a user friendly environment for program development and debugging. However, it is still necessary to add new functions to increase the usefulness of the system. Examples of such functions are a more complete MPP I/O system and either a preprocessor or a modification of the Parallel Pascal compiler that would indicate when functions used by a program are not yet implemented.

Using the development system and tools mentioned above some algorithms were coded and tested directly on the MPP hardware after being developed on the serial host. An effective heuristic algorithm was developed for arbitrary permutations and data mappings for the MPP was presented. This algorithm is very effective when large arrays are to be processed, when few elements are to be moved or when many elements share a similar motion like it is the case with small angle rotations and well behaved permutations. An effective technique for matrix rotation interpolation involving a local search scheme and based on the above permutation algorithm was also studied.

It was possible to obtain results and timing measurements from the MPP for the algorithms mentioned above. The timing results aggree well with the estimated times. The estimated times were computed using average times for the matrix assignements, rotations and reduction functions, and the number of rotations and reductions performed during the permutation and interpolation phases of the rotation algorithms.

Another algorithm implemented on the MPP was the Ising model. This model is ideal for implementation on the MPP since interactions between atomic magnetic moments or spins only occur with the nearest neighbors on the lattice and most operations are on Boolean data.

APPENDIX A

6.1. VMS command file used to implement *pp* command

```
$!
$!   compile a parallel program using ppt
$!
$ echo   := write sys$output
$ extern := $sys$usr1:[reeves.ppascal]extern
$ ppt    := $sys$usr1:[reeves.ppascal]ppt
$!
$ on error then goto err
$!
$ int = 0
$ lst :=
$ ppf :=
$ lbs :=
$ !foreach i ($argv)
$ count = 1
forloop:
$ !switch ($i)
$ arg = p'count'
$ if arg .eqs. "" then goto endfor
$ !case -i:
$ if arg .nes. "-I" then goto c2
$       int = 1
$       goto endsw
$ !case -s:
c2:
$ if arg .nes. "-S" then goto default
$       lst := 'arg'
$       goto endsw
default:
$       name = "'fSparse(arg,,"NAME")'"
$       ext  = "'fSparse(arg,,"TYPE")'"
$       dir  = "'fSparse(arg,,"DIRECTORY")'"
$       if ext .nes. ".PP" then goto pp
$               ppf := 'arg'
$               goto endsw
$!      else
pp:
$               lbs := 'lbs' " " 'arg'
endsw:
$ count = count + 1
$ goto forloop
endfor:
$!
$ name = "'fSparse(ppf,,"NAME")'"
$ ext  = "'fSparse(ppf,,"TYPE")'"
$ dir  = "'fSparse(ppf,,"DIRECTORY")'"
$!
$ if ext .nes. ".PP" then exit !no .pp file so do nothing
```

```
$ c = "''dir''name"
$ b = "''c'.p"
$ d = "''c'.obj"
$ l = "''c'.lis"
$ echo "*** Pascal Library Processor and ***"
$ echo "*** Parallel Pascal Translator  ***"
$ !extern $lst $lbs  <$ppf |  ppt  >$b
$ ass/user 'ppf' sys$input
$ ass/user pptemp.tmp sys$output
$ extern 'lst' 'lbs'
$ ass/user pptemp.tmp sys$input
$ ass/user 'b' sys$output
$ ass/user sys$error terminal
$ ass/user 'l' pplist
$ ppt
$ deass sys$input
$ deass sys$output
$ del pptemp.tmp.*
$ !tail -1 pplist | grep -s "No errors"
$ !
$ !if ($status == 0) then
$ open/read pplist pplist.dat
$ newline =""
loop1:
$ oldline = newline
$ read/end_of_file = exitloop pplist newline
$ goto loop1
exitloop:
$ close pplist
$ if oldline .nes.
    "Syntax Analysis Complete, No errors detected." then goto err

$ if int .ne. 1  then goto comp
$       echo "***     Pascal Translator     ***"
$ !     pi -w $b
$       echo "No interpreter on VMS."
$       exit
$ !else
comp:
$       echo "***  Pascal Compiler and Linker  ***"
$       pas 'b'
$      link 'd'
$       exit
err:
$ echo "***       No Compilation       ***"
$ del pptemp.tmp.*
$ exit
```

6.2. UNIX shell file used to implement *pp* command

```
#
#   compile a parallel program using ppt
#
set int = 0
set lst = ""
```

```
set ppf = ""
set lbs = ""
foreach i ($argv)
switch ($i)
case -i:
        set int = 1
        breaksw
case -s:
        set lst = $i
        breaksw
default:
        if( $i:e =~ pp) then
                set ppf = $i
        else
                set lbs = ($lbs $i)
        endif
        breaksw
endsw
end
if( $ppf:e !~ pp) exit  #no .pp file so do nothing
set b = $ppf:r.p
set c = $ppf:r
echo "*** Pascal Library Processor and ***"
echo "*** Parallel Pascal Translator  ***"
extern $lst $lbs . <$ppf | ppt  >$b
tail -1 pplist | grep -s "No errors"
if ($status == 0) then
if ($int == 1 ) then
echo "***      Pascal Translator      ***"
pi -w $b
else
echo "***      Pascal Compiler        ***"
pc $b -w -o $c
endif
else
echo "***      No Compilation         ***"
endif
```

6.3. VMS command file used to implement *rmpp* command

```
$!     BUILD THE NECESSARY FILES TO EXECUTE A PROGRAM
$!     ON THE MPP.
$!
$!       Get the name of the file
$!
$ if p1.eqs."" then inquire "file : " p1
$ pa = ""p1"" + ".pp"
$ tempa = "t" + ""p1"" + ".pp"
$ assign/user 'pa' sys$input
$ assign/user 'tempa' sys$output
$ run [reeves.mpp]mppextern
$ deassign sys$input
$ deassign sys$output
$ temp = "t" + ""p1""
$!
```

```
$!     Execute the compiler/code-generator
$!
$ PPDEV 'temp'
$!
$!        Delete unnecessary files.
$!
$ DELETE 'temp'.pcd;*,'temp'.cgl;*
$!
$!     Assemble the vax program.
$!
$ MACRO 'temp'
$!
$!     Assemble the mpp program.
$!
$ MCL 'temp'/lis/obj
$!
$!        Link the mpp programs.
$!
$ MPPLINK/deb [devaney.prim_release]prlfix.psl/lib/map -
$!
$!        Link the vax routines with the mpp symbol tables.
$!
$CADLNK 'temp','temp'.stb,PPDEVRUN/lib
$!
$!          The following message informs you how
$!        to run the program. It is not executed
$!          from this procedure!
$!
$ CLR
$ WRITE SYSSOUTPUT "To run the program type : CAD "temp' "temp' "temp'"
$EXIT
```

6.4. *Printmpp* Function

```
#printmpp

{ function printmpp (x,y : integer; matrix : plr):btype; extern; }
{                       $1  $2          $3   $0          }

var
        mask : parallel array [1..128,1..128] of boolean;
        temp : $3;
        result : $0;
      zero : integer;

begin (* of printmpp *)

      zero := 0;
        mask := true;
        mask := not(shift(mask,-1,zero) or shift(mask,zero,-1));
        mask := shift (mask,-x+1,-y+1);
        temp := 0;
        where mask do
          temp := matrix;
        result := sum (temp,1,2);
```

```
        printmpp := result;

end; (* of printmpp *)
```

6.5. *Printmppb* Function

```
#printmppb

{ function printmpp (x,y : integer; matrix : plb):boolean; extern; }
{                        $1  $2          $3   $0              }
{ where plb is a parallel two-dimensional array of type boolean    }


var
        mask : parallel array [1..128,1..128] of boolean;
        temp : $3;
        result : $0;
      zero : integer;

begin (* of printmpp *)

      zero := 0;
        mask := true;
        mask := not(shift(mask,-1,zero) or shift(mask,zero,-1));
        mask := shift (mask,-x+1,-y+1);
        temp := false;
        where mask do
          temp := matrix;
        result := or (temp,1,2);
        printmpp := result;

end; (* of printmpp *)
```

REFERENCES

[1] Paul B. Schneck, "Issues in Parallel Computing: A Non-Euclidian Examination," *Proceedings of the 1979 International Conference on Parallel Processing*, (August 1979).

[2] W. C. Meilander, "History of Parallel Processing at Goodyear Aerospace," *Proceedings of the 1981 International Conference on Parallel Processing*, (August 1981).

[3] John Burkley, "MPP VLSI Multiprocessor Integrated Circuit Design," *Proceedings of the 1982 International Conference on Parallel Processing*, (August 1982).

[4] *MPP PE Control Unit*June 1983.

[5] *MPP Main Control Unit*April 1983.

[6] A. P. Reeves, "Parallel Pascal: An extended Pascal for Parallel computers," *Journal of Parallel and Distributed Computing* 1(1984).

[7] A.P. Reeves and C.H. Moura, "Permutation and Rotation Functions for the Massively Parallel Processor," in *Computing Structures and Image Processing*, ed. K. Preston,Academic Press (in press) O.

[8] *Parallel Pascal User's Guide MUD220*,ie f"Version 1"" Version 1

[9] *Control and Debug (CAD) Users Manual*April 1983.

[10] A. P. Reeves, "On Efficient Global Information Extraction Methods For Parallel Processors," *Computer Graphics and Image Processing* 14 pp. 159-169 (1980).

[11] S. Kirkpatrick and R. H. Swendsen, "Statistical Mechanics and Disordered Systems," *Communications of the ACM* 28 pp. 363-364 (April 1985).

[12] L. D. Kovach, *Computer-oriented Mathematics*, Holden-Day,Inc., San Francisco (1964).

OMIT TO APP. E

Appendix B

DATA MAPPING AND ROTATION FUNCTIONS FOR THE MASSIVELY PARALLEL PROCESSOR

Anthony P. Reeves and Cristina H. Francfort de Sellos Moura

School of Electrical Engineering
Cornell University
Ithaca, New York 14853

The Massively Parallel Processor is a SIMD computer with 16384 processing elements connected in a 128 x 128 mesh. Such an organization is ideal for problems which involve near neighbor iterations, but for other problems which involve other data mappings it is often considered to be inefficient. In this paper a general algorithm for implementing arbitrary permutations and mappings on such systems is presented. Efficient matrix rotation algorithms based on this permutation function are also discussed. Nearest neighbor, bilinear interpolation and bicubic spline interpolation schemes are considered. These algorithms are extended for the case when the matrix to be processed is larger than the parallel hardware dimensions.

INTRODUCTION

A convenient way to interconnect a very large number of processors is in a two dimensional grid or mesh; this interconnection arrangement is very simple to implement, has a cost which increases linearly with the number of processors and is very suitable for a large number of algorithms. An example of such a system is the Massively Parallel Processor [1] which involves 16384 bit-serial processors organized in a 128 x 128. The MPP is programmed in a high level language called Parallel Pascal [2].

The only permutation function which is directly implemented by the MPP is the near neighbor rotate (or *shift*). The direction of the rotation may be in any of the four cardinal directions. In Parallel Pascal the main permutation functions are multi-element rotate and shift functions; other permutations are built on these primitives.

The *rotate function* takes as arguments the array to be shifted and a displacement for each of the arrays dimensions. For example consider a one dimensional array *a* specified by

```
a: array [0..n] of integer;
```

The rotate statement

```
a := rotate( a, 5);
```

is equivalent to

```
for i := 0 to n do
    a[i] := a[(i + 5) mod (n + 1)];
```

The rotation utilizes the toroidal end around edge connections of the mesh. The *shift* function is similar except that the mesh is not toroidally connected and zeroes are shifted into elements at the edge of the array; therefore, the shift function is not a permutation function in the strict sense. The concept of the rotate and shift functions extend to n dimensions; on the MPP the last two dimensions of the array correspond to the parallel hardware dimensions and are executed in parallel, higher dimension operations are implemented in serial. The cost of the rotate function is dependent on the distance rotated. It also depends on the size of the data elements to be permuted.

There is no simple known way to decompose an arbitrary permutation into a minimal sequence of operations on an MPP like system. In this paper a heuristic algorithm is described. The algorithm exploits the local consistency of data which occurs in many practical applications. An effective application of this algorithm to matrix rotation is presented. For some permutations, such as the perfect shuffle, which do not directly exhibit this consistency property, the algorithm may not be very effective.

For many applications the physical dimensions of the parallel hardware are smaller than the dimensions of the array to be processed. In this case the data array is processed as a set of blocks. An extension of the permutation algorithm to deal with this situation is discussed.

The program and algorithm examples given in this paper use the Parallel Pascal notation. This notation involves three extensions to standard Pascal:

1) expressions involving whole arrays are permitted;
2) the **where - do - otherwise** control statement is available. This statement is a parallel version of the **if - then - else** statement; the control expression must evaluate to a Boolean array. All array assignments within the controlled statements must be conformable with the control array and are masked by it.
3) the functions *any* and *min* are the array reduction functions or and minimum respectively.

MATRIX PERMUTATIONS

The matrix permutation algorithm presented in this paper is a general algorithm for implementing arbitrary permutations of a two dimensional matrix on mesh connected parallel processors. It is also capable of performing any onto mapping. It uses a heuristic approach to reduce the execution time.

The permutation of a matrix *a* is specified by two coordinate matrices *c* and *r* which have similar dimensions to *a*. The permuted matrix *b* also has the same dimensions as *a*. For a matrix element *b[i,j]* the corresponding elements *r[i,j]* and *c[i,j]* specify the row and column indices respectively of where the related element of a is located. That is, the permutation is specified by

$$b[i,j] := a[\, r[i,j], c[i,j] \,]$$

More formally, the data arrays involved in the permutation are specified by:

```
a,b : array [1..nrow,1..ncol] of data;
    {where data is any base type}
r : array [1..nrow,1..ncol] of 1..nrow;
c : array [1..nrow,1..ncol] of 1..ncol;
```

In order to compute the relative distance that the data must be moved, two pixel element identifying matrices *idr* and *idc* are precomputed. They contain the following:

```
idr[i,j] := i;
idc[i,j] := j;
for all i,j
```

The relative distances to be moved are then specified by

```
rr := (r-idr) mod nrow;
rc := (c-idc) mod ncol;
```

In a permutation the data may be shifted in any of the four quadrants in order to reach a specified destination. However, in the following algorithms only positive data shifts are considered, i.e. in the up and left directions. The other three quadrants are covered by using modulo arithmetic for shift distance calculations and implementing data movement with the rotation function which utilizes the end around mesh connections. We have investigated a modified heuristic algorithm which checks all four quadrants and moves in the optimal direction. Fewer data shift operations are required but the overhead due to checking alternative directions is significantly higher.

A simple permutation algorithm

A simple naive algorithm to achieve an arbitrary permutation is to slide a over all the possible positions of b, assigning the specified elements of a to each element of b when they are in the correct position.

```
for i:= 1 to nrow do
    begin
      for j:= 1 to ncol do
          begin
            where (rr = i) and (rc = j) do
                        b := a;
            a := rotate(a, 0, 1);
          end;
      a := rotate(a, 1, 0);
    end;
```

This algorithm involves $O(n^2)$ operations for an n x n matrix.

The Heuristic algorithm

In many permutations which occur in practice there are well defined patterns for the data. For example, near neighbor shifts are trivial with complexity $O(1)$, perfect shuffles can be implemented in $O(n)$ time. The heuristic algorithm attempts to take advantage of the fact that rr and rc will be the same or similar for many elements. This is particularly true for operations such as matrix warping.

The algorithm first slides (rotates) a as many locations up and left as possible such that future backtracking will not be necessary. If any element of a is correctly positioned over b (i.e. $(rr = 0)$ and $(rc = 0)$) then b is updated. Otherwise, atr, which is a copy of the current version of a, is slid in the upwards direction until all outstanding elements of b, for which the current $rc = 0$, are satisfied. The algorithm then shifts as far as possible up and left again and repeats the above procedure until all elements of the result mask are false, i.e. b is complete.

The following variables are used in the algorithm :

Variable declaration
```
mask,masktr : array [1..nrow, 1..ncol] of boolean;
atr : array [1..nrow, 1..ncol] of data;
rrt : array [1..nrow, 1..ncol] of 0..nrow;
ri, rit, lastrit : 0..nrow;
ci : 0..ncol;
```

Variable functions:

mask : the result mask, true values indicate elements of b which have not yet received the correct element of a.
ri, ci : row and column distances for the up-left move.
masktr : a version of *mask* to process one column.
rit : a version of *ri* used to process one column.
atr : a version of a used to process one column.
rrt : a version of *rr* used to process one column.
lastrit : the last value of *rit*.

The Parallel Pascal version of the heuristic algorithm is as follows:

```
lastrit :=0;
b := a;
mask := (rr <> 0) or (rc <> 0);

while any(mask, 1..2) do
   begin  { iterate until the permutation is complete }
     ri := min(rr, 1, 2);
     ci := min(rc, 1, 2);
     a := rotate(a, ri, ci);  { move up and left as far as possible }
     rr := rr - ri;
     rc := rc - ci;
     masktr := (rr = 0) and (rc = 0);
     if any(masktr, 1, 2) then  { satisfy elements for the
                                  current position }
       atr := a
     else
       begin     {satisfy each element for the given column}
         where rc = 0 do
           rrt := rr
         otherwise
           rrt := nrow;
         rit := min(rrt, 1, 2);
         masktr := rrt = rit;
         { the next seven statements implement }
         { the statement atr = rotate (a, rit, 0) }
         { but also take advantage of the previous shifts  }
         if ci <> 0 then
           begin
             atr := a;
             lastrit := 0;
           end;
         atr := rotate( atr, rit - lastrit, 0 );
         lastrit := rit;
       end;
     where masktr do {update b for the current location of a}
       begin
         b := atr;
         rr := nrow;
         rc := ncol;
         mask := false;
       end;
   end;
```

This algorithm is bounded by n^2 iterations. However, this must be considered a loose bound since we currently do not know a permutation which would require all n^2 iterations. The algorithm requires one iteration for a positive single element shift permutation but n-1 operations for a negative shift since the rotate is in the wrong direction.

Algorithm Cost

The cost of the naive algorithm is proportional to the number of rotate operations, i.e. $n^2 * 2$ (the cost of a one element rotate operation plus two comparison operations). The heuristic algorithm has two major cost components : the rotate operations as noted before and the (min) reduction functions. The reduction functions are used to compute the multi-element distance for moves. In the tables for the performance of the algorithm, both the total number of element rotates and the total number of

reduction operations are given.

The relative cost of a rotation and reduction is both system and data size dependent. For the MPP, the cost of a reduction function is in the order of 4.2 μs whereas the single element rotation of 32-bit data requires in the order of 3.2 μs to 9.6 μs depending upon the number of successive rotate operations. Therefore, the reduction functions may represent a significant portion of the computation cost. With careful low level programming the reduction operations can be overlapped with data rotate operations such that their effective cost is in the order of 1.4 μs. If the MPP was augmented with a small amount of additional hardware similar to that outlined in [3] then the reduction time could be reduced to 1.5 μs over half of which could could be overlapped with data rotation operations. The heuristic algorithm always requires less iterations and rotations than the naive algorithm; however, the additional overhead of the reduction function may make it less efficient in some instances.

Permutation Results

The results of some permutations performed in order to obtain rotated matrices of size 32 x 32 are given in Table I and II. These rotations are into mappings rather than permutations (see the matrix rotation section for details). For comparison, the naive algorithm requires 1024 iterations, 1024 rotate operations and zero reductions for any 32 x 32 matrix permutation or mapping. Table III contains the results for perfect shuffle permutations for different size matrices. The result for perfect and inverse shuffles are identical for any matrix size.

TABLE I: Cost for a near neighbor rotation on a 32 x 32 matrix centered at 16 16.

Angle of rotation	Matrix rotation mapping cost		
	iterations	rotations	reductions
0	0	0	0
15	124	562	340
30	262	620	748
45	505	837	1464
60	741	1022	2163
75	724	1019	2119
90	528	1007	1552

TABLE II: Cost for a near neighbor rotation on a 32 x 32 matrix centered at 1 1.

Angle of rotation	Matrix rotation mapping cost		
	iterations	rotations	reductions
0	0	0	0
15	235	304	666
30	437	517	1266
45	625	683	1825
60	779	829	2292
75	889	956	2631
90	993	992	2946

The perfect shuffle is an example of a permutation which does not exhibit the locality property. The number of algorithm iterations needed to implement shuffles directly is $(n-1)^2$. However, the separability property of the two dimensional shuffle is not being used. If we use the permutation algorithm to the permutation in two stages, i.e. first shuffle the rows and then shuffle the columns, then $n-1$ iterations are needed for each permutation. Therefore, the perfect shuffle when implemented directly has complexity $O(n^2)$, but when computed in two stages the algorithm is much more effective and has $O(n)$ complexity. The results of implementing the perfect shuffle as two separable shuffles are also given in Table III. Table IV shows the results of a random permutation; this demonstrates that the heuristic is not effective when the mapping does not posses the locality property.

TABLE IV: Permutation cost for a random permutation.

matrix size	Random Permutation		
	iterations	rotations	reductions
32 x 32	630	995	1851

LARGE ARRAYS

Frequently the data to be processed by a parallel processor will be in the format of arrays which exceed the fixed range of parallelism of the hardware. Therefore, it is necessary to have special algorithms that will deal with large arrays by breaking them down into blocks manageable by the hardware, without loosing track of the relationships between different blocks.

One scheme, which is frequently used on the MPP, is to partition the large array into blocks which are conveniently stored in a four dimensional array. The range of the first dimension of this array specifies the number of blocks in each row of the large matrix and the range of the second dimension specifies the number of blocks in each column. Given a conceptual large matrix

$$mx : \text{array } [0..x, 0..y] \text{ of btype;}$$

which is to be stored in an array a of type

$$\text{array } [1..n, 1..m, 1..p, 1..q] \text{ of btype;}$$

Element i,j of the large matrix is mapped into the array a as specified by

$$mx[i,j] = a[1+i \text{ div } p, 1+j \text{ div } q, 1+i \text{ mod } p, 1+j \text{ mod } q]$$

For example, a 512 x 256 matrix could be stored in eight blocks as:

$$la : \text{array } [1..4, 1..2, 1..128, 1..128] \text{ of real;}$$

This data structure allows blocks to be manipulated independently. However, it still preserves the positional relationships of those blocks in the original large matrix.

To simplify the manipulation of large arrays on the MPP, two Parallel Pascal library functions *lrotate* and *lshift* have been developed. These functions take an array argument and two dis-

TABLE III: Cost for perfect shuffle permutations for different matrix sizes.

matrix size	Direct Shuffle Cost			Separable Shuffle Cost		
	iterations	rotations	reductions	iterations	rotations	reductions
4 x 4	9	12	22	6	6	6
8 x 8	49	56	134	14	14	14
16 x 16	225	240	646	30	30	30
32 x 32	961	992	2822	62	62	62

placement arguments, like the primitive matrix rotate and shift functions, however, in this case the array argument is a four dimensional array which is treated like a conceptually large matrix.

Many programs can be converted to operate on blocked rather than conventional matrices by simply replacing all instances of rotate and shift with lrotate and lshift respectively. This is true for the permutation programs presented; however, in the case of the heuristic permutation algorithm, this is not a very efficient solution. A better method is to scan through the result blocks and perform permutations on only the input blocks that contribute to the current result block being processed. This algorithm is shown below.

```
var
    la,lb: array [1..n,1..m,1..nrow,1..ncol] of data;
    lr,lc: array [1..n,1..m,1..nrow,1..ncol] of index;

begin
    for i = 1 to n do
        for j = 1 to m do
            begin {process each result block}
                rb := 1 + lr[i,j] div nrow;
                cb := 1 + lc[i,j] div ncol;
                ro := 1 + lr[i,j] mod nrow;
                co := 1 + lc[i,j] mod ncol;
                for k = 1 to n do
                    for l = 1 to m do
                        begin {consider each input block}
                            maskb := (rb = k) and (cb = l);
                            if any(maskb, 1, 2) then
                                where maskb do
                                    lb[i,j] := perm2 (la [k,l],
                                        ro, co, maskb);
                        end;
            end;
end;
```

Perm2 is the heuristic algorithm presented previously with the modification that the initial mask value is passed as an argument. That is, only elements selected by the mask are permuted. An additional speedup is achieved by this since the heuristic works much better when only a subset of elements are to be permuted.

Table V contains the results of the rotation mapping for the case where a 32 x 32 matrix is considered to consist of 4 x 4 blocks of 8 x 8 elements. The Large Perm results are from using the lrotate approach and the *perm2* results are for the block scanning algorithm.

TABLE V: Comparison of perm2 and large blocked permutation for a rotation centered at coordinates 1 1.

angle of rotation	Perm2		Large perm.	
	rotations	reductions	rotations	reductions
0	13	3	0	0
15	657	703	6768	7184
30	1050	1302	7616	10704
45	1389	1834	8352	13152
60	1702	2329	6896	11248
75	1931	2669	4112	5808
90	2086	2971	1856	1408

TABLE VI: Comparison of perm2 and large blocked permutation for a rotation centered at coordinates 16 16.

angle of rotation	Perm2		Large perm.	
	rotations	comparisons	rotations	comparisons
0	1188	184	3472	1552
15	850	374	7616	7552
30	1011	766	11520	15408
45	1448	1503	16544	25232
60	1858	2174	21328	33552
75	1846	2151	25056	37760
90	1793	2001	22544	33120

MATRIX ROTATION

One application of the permutation function is matrix rotation mapping. Three rotation techniques are considered : nearest neighbor, bilinear interpolation and bicubic interpolation. A rotation is specified by three parameters : the location of the origin of rotation (r_0, c_0) and the rotation angle θ . The starting point of all rotation algorithms is the generation of the mapping matrices r and c from these parameters.

Nearest neighbor

The nearest neighbor algorithm is simply an into mapping in which a result element is assigned the value of the nearest rotated matrix element. In this case the new row and column coordinate matrices, r and c, are defined as follows:

$$r[i,j] = \text{round}((c_0 - j)sin(\theta) + (i - r_0)cos(\theta) + r_0)$$

$$c[i,j] = \text{round}((j - c_0)cos(\theta) + (i - r_0)sin(\theta) + c_0)$$

for all i and j; any values of the result which have near neighbors outside the range of the input matrix are set to zero.

In performing a rotation, some elements of the result matrix are rotated and some elements are selected which are outside of the input matrix. In our algorithm result elements for the latter case are simply set to zero. Therefore we have a permutation in which a subset of the input elements map into a subset of the result elements; the size of these subsets depends upon the angle and origin of the rotation. The rotation is achieved by using the *valid elements of r and c with the heuristic permutation algorithm.*

Examples of nearest neighbor rotation are shown in Fig. 1. for an 8 x 8 matrix with the origin of rotation located at (1, 1). For a small angle of rotation most of the results have valid mapped values; however, the heuristic algorithm is very effective for this case because there is little movement of the data or *rr* and *cr* are the same for many elements.

When the rotation angle is large there is a lot of data rearrangement but only a few elements are to be moved with the rotation located at (1, 1). For the naive algorithm 49 iterations are needed for all rotations.

Bilinear Interpolation

For the bilinear interpolation algorithm a result element is computed from a weighted sum of the four rotated input matrix elements which surround it. There are two possible approaches to implementing this scheme. First, we can compute four permutations; each permutation acquiring one of the four neighbors for each element. This is called *multiple permutation.*

The second method is to perform one permutation and then seek the local neighborhood of the rotated input matrix for the

11	12	13	14	15	16	17	18
21	22	23	24	25	26	27	28
31	32	33	34	35	36	37	38
41	42	43	44	45	46	47	48
51	52	53	54	55	56	57	58
61	62	63	64	65	66	67	68
71	72	73	74	75	76	77	78
81	82	83	84	85	86	87	88

Input Matrix

11	0	0	0	0	0	0	0
22	12	0	0	0	0	0	0
22	23	14	0	0	0	0	0
33	24	25	15	0	0	0	0
44	35	25	26	17	0	0	0
55	45	36	27	27	18	0	0
55	56	47	37	28	0	0	0
66	57	57	48	0	0	0	0

45 degree rotation

35 iterations

11	12	13	0	0	0	0	0
21	22	23	14	15	16	17	18
31	32	33	24	25	26	27	28
42	43	43	34	35	36	37	38
52	53	54	45	46	47	48	0
62	63	64	55	56	57	58	0
72	73	74	65	66	67	68	0
82	83	84	75	76	77	78	0

10 degree rotation

5 iterations

11	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0

90 degree rotation

14 iterations

Figure 1. Sample Nearest Neighbor Rotations

other near neighbors. The idea being that a local search will require less computation than four complete rotations especially when the angle of rotation is large. The local neighborhood of a single rotated matrix does not contain a complete set of the near neighbor elements of the input matrix; some are lost due to grid spacing differences. A complete set can be guaranteed, however, if we also include the local neighborhood of a slightly perturbed, rotated input matrix. This scheme is called the *double permutation* method; both rotated matrices can be computed simultaneously with a single execution of a slightly modified heuristic permutation algorithm.

If we map a result element back to the input matrix it will be surrounded by four elements P1 - P4 as shown in Fig. 2. These points are moved to the processing element associated with the result P and the interpolation is then computed in parallel.

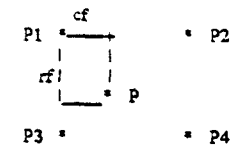


Figure 2. Bilinear interpolation

For the interpolation algorithms, the matrices, rp and cp, contain the actual locations of the rotated elements.

$$rp[i,j] = (c_0 - j)\sin\theta) + (i - r_0)\cos\theta) + r_0$$
$$cp[i,j] = (j - c_0)\cos\theta) + (i - r_0)\sin\theta) + c_0$$

for all i and j.

The coordinates of the near neighbors are as follows:

$$r_{top} = \left| (c_0 - j)\sin\theta) + (i - r_0)\cos\theta) + r_0 \right|$$

$$c_{left} = \left| (j - c_0)\cos\theta) + (i - r_0)\sin\theta) + c_0 \right|$$

$$r_{bottom} = r_{top} + 1$$

$$c_{right} = c_{left} + 1$$

The interpolation fractions are:

$$rf = rp - r_{top}$$

$$cf = cp - c_{left}$$

Once the points P1 - P4 have been obtained the interpolated result is computed as follows:

$$P = (1 - cf)*(1 - rf)*P1 + (1 - rf)*cf*P2 + (1 - cf)*rf*P3 + cf*rf*P4$$

The algorithm for the *multiple permutation* approach is as follows:

```
begin
    r := r_top;
    c := c_left;
    b := coef1 * perm(a, r, c); { value of top left neighbor }
    c := c + 1;
    b := coef2 * perm(a, r, c) + b; { value of top right
    neighbor }
    r := r + 1;
    b := coef3 * perm(a, r, c) + b; { value of bottom right
    neighbor }
    c := c - 1;
    b := coef4 * perm(a, r, c) + b; { value of bottom left
    neighbor }
end;
```

Perm is the heuristic permutation function. The final result of rotating a is stored in the matrix b.

The *double permutation* approach uses a modified permutation function which creates the following matrices:

$$b[i,j] := a[r[i,j], c[i,j]]$$

and

$$d[i,j] := a[r[i,j], c[i,j] + 1]$$

where

a is the original matrix
b is the rotated matrix
d is the shifted rotated matrix
r is the row coordinate matrix

and

c is the column coordinate matrix

To avoid loosing the value of the center of rotation, when the origin is located to the right of the matrix center, the matrix is shifted left and, therefore, in the equation defining matrix d we substitute a negative one for the constant one.

The second step in the *double permutation* algorithm is a local search performed on both rotated and shifted rotated matrices in order to find all the values of the elements needed for the interpolation. The local search has a constant maximum cost for any size matrix. It therefore has an advantage over the *multiple permutations* approach, since every permutation in that approach will become more costly as the matrix size increases.

For the worst case rotation angle ($\theta = 45$), it has been determined that a local search in a 5 x 5 window is sufficient to yield the values of all the elements needed to perform a bilinear interpolation. The local search strategy implemented in our algorithm is a spiral search. The elements are selected by comparing their row and column coordinates to those needed. Once they match, their values can be obtained from the rotated matrix or from the rotated shifted matrix.

Cubic Interpolation

The cubic interpolation version of the rotation algorithm is a simple extension of the bilinear interpolation scheme. The first step, finding the coordinate matrices r and c, is identical to the bilinear interpolation case. After obtaining these matrices, the values of sixteen neighbor points must be acquired. If the *multiple permutations* approach is used, then sixteen separate permutations will be required. However, with the *double permutation* scheme only a small extension of the bilinear algorithm is needed.

Instead of using a 5 x 5 window, which is the case when four points have to be found, a 7 x 7 window is necessary to find sixteen points. However, since the element will have rotated in a specific direction, the search window can be reduced to a 7 x 5 window. Each row of points needed will use a different 7 x 5 window of search.

Once all the values needed are found, the bicubic interpolation itself is done by, first, performing a cubic interpolation for each of the four rows and, then, performing a fifth cubic interpolation on the row points obtained.

As shown in Fig. 3, the reference point for the cubic interpolation computed in step one is P6. The first four cubic interpolations are performed to obtain points pa, pb, pc and pd. The fifth one yields the value of point P.

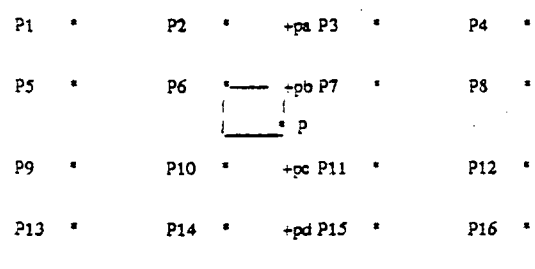


Figure 3. Cubic Interpolation

Test Results

The local search performed in the *double permutation* algorithm has a constant maximum cost for any size matrix. For any matrix the maximum cost is 100 rotations and 25 reductions for the bilinear interpolation method and 552 rotations and 525 comparisons for the cubic interpolation.

The results for rotations using bilinear interpolation for a 32 x 32 matrix are given in Table VII and VIII for different centers of rotation. The results of rotations applying cubic interpolation are given in Tables IX and X.

TABLE VII: Cost of bilinear interpolated rotation centered at coordinates 16 16

angle of rotation	Double perm.		Multiple perm.	
	rotations	reductions	rotations	reductions
0	1188	184	4092	649
15	850	374	2488	1416
30	1011	766	2544	2978
45	1448	1503	3292	5916
60	1858	2174	4084	8602
75	1846	2151	4081	8491
90	1793	2001	4090	7920

TABLE VIII: Cost of bilinear interpolated rotation centered at coordinates 1 1

angle of rotation	Double perm.		Multiple perm.	
	rotations	reductions	rotations	reductions
0	13	3	4	6
15	657	703	1342	2705
30	1050	1302	2035	5093
45	1389	1834	2683	7238
60	1702	2329	3291	9230
75	1931	2669	3471	10627
90	2086	2971	3968	11780

TABLE IX: Cost of cubic interpolated rotation centered at coordinates 16 16

angle of rotation	Double perm.		Multiple perm.	
	rotations	reductions	rotations	reductions
0	1640	684	16368	2608
15	1302	874	10072	5757
30	1463	1266	10394	12021
45	1900	2003	13300	23668
60	2310	2674	16341	34371
75	2298	2651	16315	33928
90	2245	2501	16360	31664

TABLE X: Cost of cubic interpolated rotation centered at coordinates 1 1

angle of rotation	Double perm.		Multiple perm.	
	rotations	reductions	rotations	reductions
0	552	525	272	30
15	1109	1203	7683	11132
30	1502	1802	9723	20589
45	1841	2334	11815	29104
60	2154	2829	13793	37047
75	2383	3169	15316	42586
90	2538	3471	15872	47088

RESULTS USING THE MPP

The results for 32 x 32 matrices reported in this paper were obtained with a Parallel Pascal Translator which translates Parallel Pascal into standard Pascal for program development [4]. Some of these functions have also been run on the MPP; in this case for a 128 x 128 array. In Table XI and Table XII results for near neighbor rotations are given and in tables XIII and XIV results for bilinear interpolation rotations are given.

Table XI: Cost for a near neighbor rotation on a 128 x 128 matrix centered at 1 1.

Angle of rotation	Matrix rotation mapping cost	
	rotations	reductions
0	0	0
15	5114	7506
30	8495	14046
45	11103	20050
60	13504	25121
75	15516	28802
90	16256	32385

In Figs. 4 and 5 comparisons are given between the 32 x 32 and 128 x 128 results. The cost shown is the ratio of the number of shift operations required by the heuristic algorithm over the number of shifts required by the simple algorithm for a single permutation (i.e., n^2). These figures show that there is a very good correspondence between the the results for the different size matrices. That is, for the rotation algorithm the improvement achieved with the heuristic algorithm is a constant which which is independent of matrix size.

Table XII: Cost for a near neighbor rotation on a 128 x 128 matrix centered at 64 64.

Angle of rotation	Matrix rotation mapping cost	
	rotations	reductions
0	0	0
15	9183	3594
30	10446	7800
45	13589	15777
60	16378	23359
75	16375	24331
90	16319	16384

Table XIII: Cost for a bilinear interpolation rotation on a 128 x 128 matrix centered at 1 1.

Angle of rotation	Matrix rotation mapping cost	
	rotations	reductions
0	101	25
15	9009	7555
30	15689	14107
45	21246	20034
60	26228	21192
75	30023	28893
90	32614	32410

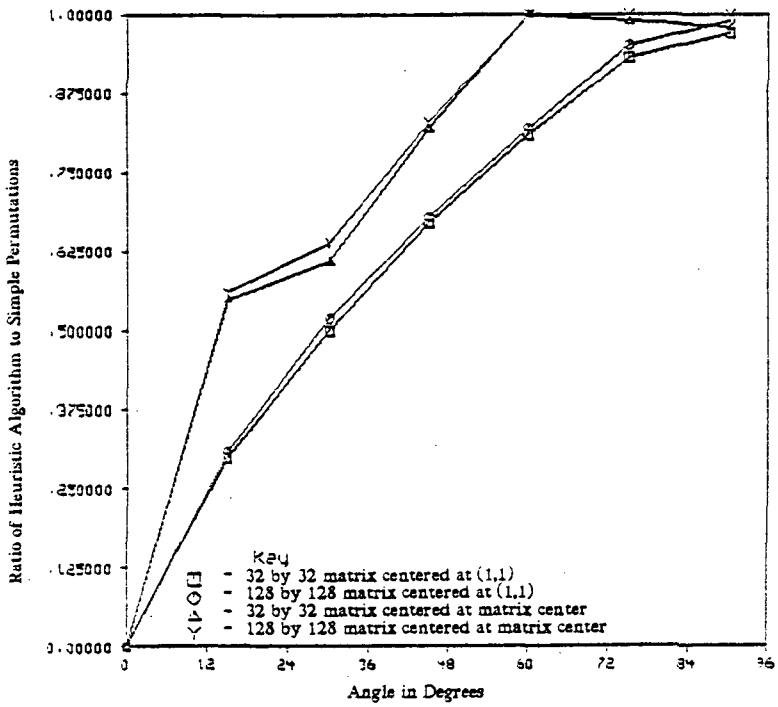


Figure 4. Ratio of the number of rotations to n^2 rotations for near neighbor rotation with the center of rotation at (1,1) and at the matrix center.

Table XIV: Cost for a bilinear interpolation rotation on a 128 x 128 matrix centered at 64 64.

Angle of rotation	Matrix rotation mapping cost	
	rotations	reductions
0	101	25
15	11510	3628
30	14648	7826
45	21556	15800
60	28282	23378
75	28763	24382
90	24676	16409

CONCLUSION

An effective heuristic algorithm for arbitrary permutations and data mappings for mesh connected SIMD processors has been presented. This algorithm is particularly suited to the following conditions:

1. When only a few elements are to be moved.

2. When many elements share a similar motion, e.g. small angle matrix rotation and warping.

3. When large arrays are to be processed.

It is less suitable when the permutation or mapping is dense and does not have the locality property. The effectiveness of this algorithm over a naive algorithm depends upon the system implementation parameters, and the size of the data to be manipulated.

An effective technique for matrix rotation interpolation has been presented which involves a local search scheme. Excellent results have been obtained especially for bicubic interpolation.

REFERENCES

1. K. E. Batcher, "Design of a Massively Parallel Processor," *IEEE Transactions on Computers* C-29(9) pp. 836-840 (September 1981).

2. A. P. Reeves, "Parallel Pascal: An extended Pascal for Parallel computers," *Journal of Parallel and Distributed Computing* 1 pp. 64-80 (1984).

3. A. P. Reeves, "On Efficient Global Information Extraction Methods For Parallel Processors," *Computer Graphics and Image Processing* 14 pp. 159-169 (1980).

4. A. P. Reeves, "Parallel Pascal Development System," *Cornell University Technical Report*, (January, 1985).

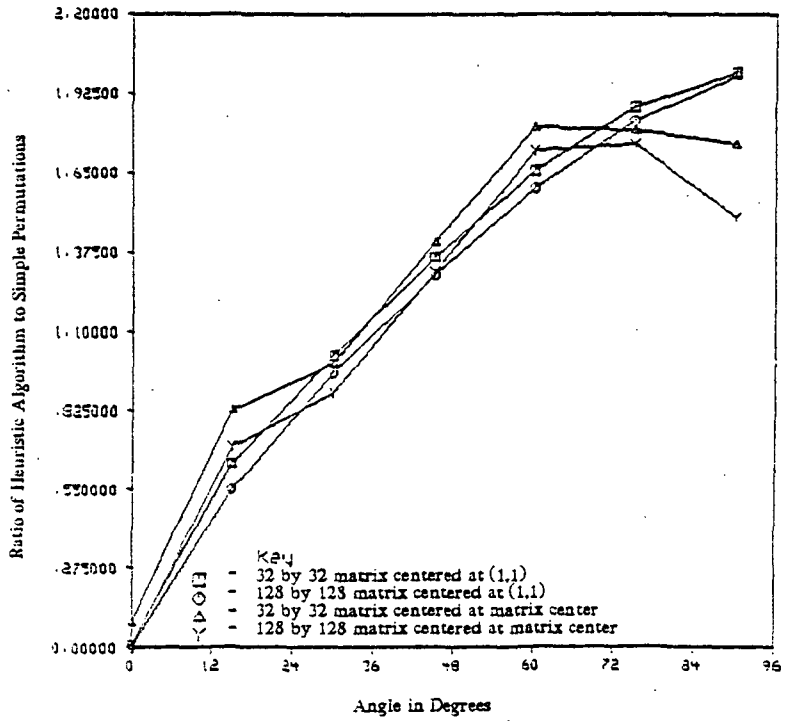


Figure 5. Ratio of the number of rotations to n^2 rotations for bilinear interpolation rotation with the center of rotation at (1,1) and at the matrix center.

OURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 1, 64–80 (1984)

Parallel Pascal: An Extended Pascal for Parallel Computers

ANTHONY P. REEVES

School of Electrical Engineering, Cornell University, Ithaca, New York 14853

Parallel Pascal is an extended version of the conventional serial Pascal programming language which includes a convenient syntax for specifying array operations. It is upward compatible with standard Pascal and involves only a small number of carefully chosen new features. Parallel Pascal was developed to reduce the semantic gap between standard Pascal and a large range of highly parallel computers. Two important design goals of Parallel Pascal were efficiency and portability. Portability is particularly difficult to achieve since different parallel computers frequently have very different capabilities.

1. INTRODUCTION

There is a large class of mainly scientific problems which require the ıvailability of highly parallel processors in order to compute results in a ·easonable amount of time. Many diverse parallel computer architectures ıave been designed for these problems, which usually involve either pipeline ır processor array schemes. A common feature of these architectures is their ıbility to perform some very high speed operations on arrays of data. How-:ver, there is a very large variation between different architectures in the set ıf array operations which can be efficiently implemented and the size of the ırrays which can be efficiently processed. The term "parallel computer" in his paper is used to indicate systems which have high-speed array-processing :apabilities through hardware parallelism. Parallel Pascal is a programming anguage for such parallel computers.

Conventional serial high-level programming languages are difficult to im-ılement efficiently on parallel computers. Most parallel computers are cur-ently programmed in either assembly language or a machine-dependent pecial version of Fortran. The main advantages of a general high-level anguage for parallel computers, such as Parallel Pascal, are portability, ıetter error detection and diagnosis facilities, and efficiency. Efficiency must ıe a prime consideration since with any parallel system extra hardware is ıeing used to achieve a high-speed performance. Portability is perhaps the

0743-7315/84 $3.00

most difficult goal to achieve while maintaining efficiency since different parallel systems have very different data permutation capabilities.

There are three fundamental classes of operations on array data which are frequently implemented as primitives on array computers but which are not available in conventional programming languages; these are: data reduction, data permutation, and data broadcast. These operations have been included as primitives in Parallel Pascal.

Data reduction operations reduce the number of elements in the data by applying an operator between array elements and returning the result. For example, a typical reduction operation is to compute a vector which contains the sums over the rows of a matrix. Parallel Pascal reduction functions are described in Section 4. Data selection may be considered to be a special type of reduction function; however, selective data assignment requires a different syntactic structure. Data selection and selective data assignment use a similar syntactic form in Parallel Pascal and are described in Section 3.

Data permutation and data broadcast functions rarely need to be used in conventional programming languages since random access to array elements is usually adequately dealt with by index expressions. In a parallel system it is usually much more efficient to do a data permutation in parallel. Data permutation and broadcast functions are described in Section 5.

A single parallel control statement, the "where" statement, is defined in Parallel pascal. It is similar to an "if" statement but with an array control variable. The where statement is described in Section 6. A method of accessing the individual bits of array data elements, which is possible with bit-serial parallel computers, is outlined in Section 7.

Parallel Pascal was originally designed as a high-level language for NASA's Massively Parallel Processor (MPP), which was constructed by Goodyear Aerospace [1]. The MPP has a single-instruction unit and 16,384 processing elements organized in a 128 × 128 matrix with near-neighbor interconnections. Other parallel computers with a similar interconnection scheme include Illiac IV and the DAP. Parallel Pascal is also suitable for parallel computers with a less restrictive interconnection scheme such as is found with many pipeline systems, for example. In this case, it may be necessary to implement some additional data-mapping functions in order to take advantage of all the parallel computer's capabilities. This is discussed further in Section 5.3. An in-depth description of Parallel Pascal and other aspects of the MPP is given in [2] and also in [3].

A Parallel Pascal-to-standard Pascal translator has been developed to allow initial experimentation with different language features. This translator is now being used for program development of Parallel Pascal programs on conventional serial computers.

A compiler has been developed [3, 2] which converts a Parallel Pascal program into a parallel P-code form. Interesting features of this P-code language are outlined in Section 8. A parallel P-code code generator for the

MPP has been developed by Computer Sciences Corporation. Parallel Pascal is the first operational high-level language for the MPP.

A considerable effort was made to maintain the concepts of standard Pascal in designing the extensions for Parallel Pascal. However, there were several features of standard Pascal which cause problems in the parallel computer context. These are discussed in Section 9.

2. Parallel Expressions

In standard Pascal the only aggregate array operation defined for arrays is to copy one array to another. For example, given the definition

```
var a, b, c: array [1..10] of integer;
```

the following statement is valid and means copy all elements of array *b* to array *a*.

```
a := b;
```

In Parallel Pascal all conventional expressions are extended to array data types. In a parallel expression all operations must have conformable array arguments. A scalar is considered to be conformable to any type of compatible array and is conceptually converted to a conformable array with all elements having the scalar value. For example, the statement

```
a := b + c + 1;
```

is equivalent to

```
for i := 1 to 10 do
    a[i] :=b[i] + c[i] + 1;
```

In many highly parallel computers there are at least two different primary memory systems; one in the host and one in the processor array. Parallel Pascal provides the reserved word **parallel** to allow programmers to specify the memory in which an array should reside. In standard Pascal an array type is specified with the syntax

```
type newtype = array [indextype] of eltype;
```

where indextype specifies the number and range of the array dimensions and eltype specifies the type of the array elements. A parallel array type is specified with the syntax


```
type newtype = parallel array [indextype] of eltype;
```

The parallel specifier exists only to provide information to the compiler as to the variables' usage. In all usage in the language a parallel array is indistinguishable from a conventional array. In some systems there is no distinction between host and processor memories; then the parallel specifier has no effect. In any case, a compiler may decline to store the array where requested.

3. Array Selection

Selection of a portion of an array by selecting either a single index value or all index values for each dimension is frequently used in many parallel algorithms, e.g., to select the *i*th row of a matrix which is a vector. Specification of a single index value is the standard indexing method in standard Pascal. In Parallel Pascal all index values can be specified by eliding the index value for that dimension. For example, given the definition

```
var a,b: array [1..5, 1..10] of integer;
```

in Parallel Pascal the statement

```
a[,1] := b[,4];
```

assigns the fourth column of *b* to the first column of *a*. It is interesting to note that standard Pascal already permits the assignment of whole arrays and of subarrays when the rightmost dimensions have been completely elided; therefore, the following are valid statements in standard Pascal:

```
a := b;
a[1] := b[2];
```

The second statement means assign the second row of *b* to the first row of *a*; in Parallel Pascal this could also be specified by

```
a[1,] := b[2,];
```

3.1. *Subrange Constants*

It is sometimes necessary to move data between arrays with different dimensions. In Parallel Pascal subarrays consisting of consecutive sets of elements may be specified. If subarrays with other than consecutive elements are required then they must be packed into the consecutive form with permutation functions. The concept of a constant subrange is introduced in order to specify a consecutive subset of index values.

The syntax for the constant subrange is

const identifier = low..high;

where low and high are either literals or previously defined constant identifiers.

3.2. *Subrange Indexing and Array Packing*

Subrange constants may be used to index an array in Parallel Pascal. The general syntax for a subrange index is

array-identifier[offset @ subrange-constant]

where offset is an optional conventional scalar index expression. The ordered set of indices specified by a subrange index is the result of adding the value of the offset expression to the values implied by the subrange constant. For example, given the definition

var *a*, *b*: **array** [1..10] **of** integer;

the statement

a[@2..6] := *b*[3 @ 1..5];

is functionally equivalent to

for *i* := 1 **to** 5 **do**
  a[*i* + 1] := *b*[*i* + 3];

The main reason for the introduction of subrange indexing was to permit blocks of data to be transferred between arrays having different dimensions. It was not designed to be a tool for algorithm development. An alternative specification of this operation, which was considered, was to use a new standard procedure, called replace, which is analogous in style to the standard Pascal pack and unpack functions. A possible syntax for replace is

replace(*a*, *Oa*1, *Ra*1, *Oa*2, *Ra*2, . . . , *b*, *Ob*1, *Rb*1, . . .)

where *Oa*1 is the offset of the first dimension of array *a*, *Ra*1 is the range of the first dimension of array *a*, etc..

This syntax works well for vectors but results in a proliferation of arguments for higher-dimensional arrays (four for each dimension). The more syntactically complex subrange indexing scheme was chosen because it was considered to be more readable for multidimensional arrays.

3.3. *Array Conformability*

In standard Pascal, data items combined together in an expression must be type compatible. In Parallel Pascal, array data items in a parallel expression must also be conformable, i.e., have the same rank (number of dimensions) and the same range in each dimension. For example, given the definitions

var *a*, *b*: **array** [1..10] **of** integer;
  c: **array** [0..9] **of** integer;

the statement

a := *a* + *b*;

is conformable, while the statement

a := *b* + *c*;

is not conformable since the specified ranges of *b* and *c* are different.

While the exact range conformability requirement is in keeping with the strong typing concepts of standard Pascal, there are occasions when the action specified by the above statement is useful. The range requirement can be explicitly circumvented by using subrange indexing. For example, the statements

a := *b* + *c*[@0..9];
a[@1..10] := *b*[@1..10] + *c*;
a[@1..10] := *b*[@1..10] + *c*[@0..9];

are all conformable and have the same effect.

4. Reduction Functions

Array reduction operations are achieved with a set of standard functions in Parallel Pascal which are listed in Table I.

TABLE I
Reduction Functions

Syntax	Meaning
sum(array, *D*1, *D*2, . . . , *Dn*)	Reduce array with arithmetic sum
prod(array, *D*1, *D*2, . . . , *Dn*)	Reduce array with arithmetic product
all(array, *D*1, *D*2, . . . , *Dn*)	Reduce array with Boolean AND
any(array, *D*1, *D*2, . . . , *Dn*)	Reduce array with Boolean OR
max(array, *D*1, *D*2, . . . , *Dn*)	Reduce array with arithmetic maximum
min(array, *D*1, *D*2, . . . , *Dn*)	Reduce array with arithmetic minimum

The first argument of a reduction function specifies the array to be reduced and the following arguments specify which dimensions are to be reduced. A dimension is specified by a constant expression; the first dimension is dimension 1. The dimension parameters must be constant expressions so that the shape of the result is known at compile time.

For example, given the the definitions

```
var
    a : array[1..10,1..5] of integer;
    b : array[1..10] of integer;
    c : integer;
```

the following are correct Parallel Pascal statements

```
b := sum(a, 2);          (* sum the rows of a *)
c := sum(a, 1, 2);       (* sum all elements of the array a *)
c := max(b, 1);          (* find the maximum value of b *).
```

Each dimension parameter of a reduction function implies that there will be one less dimension in the result array; a scalar is considered to be an array without any dimensions in this context.

5. Permutation and Distribution Functions

One of the most important features of a parallel programming language is the facility to specify parallel array data permutation and distribution operations. In Parallel Pascal four such operations are available as primitive standard functions; however, for some Parallel Processors it may be necessary to specify more primitive functions for efficiency. The standard Parallel Pascal functions for data permutation and distribution are given in Table II.

TABLE II
Permutation and Distribution Functions

Syntax	Meaning
shift(array, *S1*, *S2*, . . . , *Sn*)	End-off shift data within array
rotate(array, *S1*, *S2*, . . . , *Sn*)	Circularly rotate data within array
transpose(array, *D1*, *D2*)	Transpose two dimensions of array
expand(array, dim, range)	Expand array along specified dimension

5.1. *Shift and Rotate*

The shift and rotate primitives are found in many parallel hardware architectures and, also, in many algorithms. The shift function shifts data by the amount specified for each dimension and shifts zeros (null elements) in at the edges of the array. Elements shifted out of the array are discarded. The rotate function is similar to the shift function except that data shifted out of the array are inserted at the opposite edge so that no data are lost. The first argument to the shift and rotate functions is the array to be shifted; then there is an ordered set of parameters, each of which specifies the amount of shift in its corresponding dimension. There must be as many shift parameters as there are dimensions in the array; the first shift parameter is associated with the first dimension of the array.

For example, given the definitions

```
var
    a, b : array [1..5, 0..9] of integer;
    c, d : array [0..9] of integer;
```

the statement

```
a := shift(b, 0, 3);
```

is functionally equivalent to

```
for i := 1 to 5 do
  begin
    for j := 0 to 6 do
      a[i,j] := b[i, j + 3];
    for j := 7 to 9 do
      a[i, j] := 0;
  end;
```

and the statement

```
c := rotate(d, 3);
```

is functionally equivalent to

```
for i := 0 to 9 do
  c[i] := d[(i + 3) mod 10];
```

5.2. *Transpose and Expand*

While transpose is not a simple function to implement with many parallel architectures, a significant number of matrix algorithms involve this function; therefore, it has been made available as a primitive function in Parallel Pascal. The first parameter to transpose is the array to be transposed and the following two parameters, which are constant expressions, specify which dimensions are to be interchanged. If only one dimension is specified then the array is flipped about that dimension.

The main data distribution function in Parallel Pascal is expand. This function increases the rank of an array by one by repeating the contents of the array along a new dimension. The first parameter of expand specifies the array to be expanded; the second parameter, a constant expression, specifies the number of the new dimension and the last parameter; a subrange or a subrange type specifies the range of the new dimension.

This function is used to maintain a higher degree of parallelism in a parallel statement; this may result in a clearer expression of the operation and a more direct parallel implementation. In a conventional serial environment such a function would simply waste space.

For example, given the definitions of a, b, and c as specified in Section 5.1, the following statement adds a vector to all rows of a matrix

$$a := b + \text{expand}(c, 1, 1..5);$$

The above statement is functionally equivalent to

```
for i := 1 to 5 do
    a[i,] := b[i,] + c;
```


5.3. *Other Functions*

While the shift and rotate functions are adequate primitive functions for highly parallel computers such as the MPP, which have near-neighbor mesh interconnections, different primitive functions may be necessary for other architectures. For example, consider an architecture which can directly implement a perfect shuffle permutation and the algorithm to be implemented is the Fast Fourier Transform (FFT). If a perfect shuffle function is available in the high-level language then the FFT can be very clearly and efficiently specified, whereas a specification using only shift and rotate would be very difficult to write and the compiler would have to do a clever code optimization in order to achieve an efficient implementation.

A solution which is consistent with the framework of Parallel Pascal is to define the shuffle as a primitive permutation function for this computer architecture. Portability may be maintained by writing a library function for the shuffle using only shift and rotate primitives. In this way an algorithm written

architecture. The efficiency of such a ported algorithm may not be very high on the MPP and some reprogramming of the algorithm may be necessary.

Given the wide diversity of parallel processor architectures it is perhaps unreasonable to expect that a single algorithm specification can be efficiently compiled for all systems. The scheme proposed above for Parallel Pascal permits simple porting of all algorithms to different architectures for verification and provides a consistent notation for reprogramming an algorithm to tune it for a particular architecture. This scheme is made possible because Parallel Pascal uses the conventional function syntax to specify permutation primitives. In this way the use of new primitive functions is syntactically indistinguishable from using a user-defined permutation function.

6. Conditional Execution

An important feature of any parallel programming language is the ability to have an operation operate on a subset of the elements of an array. In standard Pascal each array element is processed by a specific sequence of statements and there are a variety of program control structures for the repeated or selective execution of statements. In Parallel Pascal the whole array is processed by a single statement; therefore, an extended program control structure is needed.

In the initial specification for Parallel Pascal all the standard Pascal control structures were extended to accept an array control expression. These extended control structures proved to be very difficult to exactly specify within the framework of standard Pascal. Furthermore, the need for the looping control structures is much less in an environment which allows parallel expressions. For example, the APL programming language, which involves a very powerful array expression capability, does not have a direct program loop control structure. In the test algorithms that were programmed only the extended **if** statement was frequently used. The other control structures were dropped from the language and the conditional execution statement was renamed **where** due to semantic differences with the standard **if** statement.

The syntax of the Parallel Pascal **where** statement is as follows:

```
where array-expression do
    statement
otherwise
    statement
```

where array-expression is a Boolean-valued array expression and statement is a Parallel Pascal statement. The **otherwise** and the second controlled statement may be omitted.

The execution of a **where** structure is defined as follows. First, the controlling expression is evaluated to obtain a Boolean array (mask array). Next, the first controlled statement is evaluated. Array assignments are masked according to the Boolean control array. If there is an **otherwise** statement it is then evaluated; in this case array assignments are masked with the inverse of the control array.

For example, given the definition

```
var a, b, c : array [1..10] of integer;
```

the expression

```
where a < b do
    c := b
otherwise
    c := a;
```

is functionally equivalent to

```
for i := 1 to 10 do
  if a[i] < b[i] then
    c[i] := b[i]
  else
    c[i] := a[i];
```

The main semantic difference between the **where–do–otherwise** structure and the **if–then–else** structure is that with the former both controlled statements are evaluated, independent of the value of the control expression, while with the latter only one of the two controlled statements is evaluated.

Where statements may be nested provided that all of the controlling array expressions are type compatible. Other standard Pascal control statements can also be nested within **where** statements. Any array variable which appears on the left-hand side of an assignment within a **where**-controlled statement must be type compatible with the controlling array expression. Assignments to other than array variables in a **where** statement are in no way affected by the **where** statement. The effect of a **where** statement is local to the procedure or function in which it occurs; that is, it does not affect the execution of any procedures or functions called from within a **where** statement or an **otherwise** statement.

The **where** statement provides Parallel Pascal with a *conditional assignment facility*. An alternative to conditional assignment which was not implemented in Parallel Pascal is *conditional evaluation*. In this case, all the operations are masked; only data elements which contribute to mask selected results are processed. This is conceptually more pleasing since only needed computations are specified, whereas, with conditional assignment, extra computations are conceptually performed which are discarded by the assignment operation.

Conditional evaluation can also be useful for catching or avoiding exceptional conditions. For example, the following statement will successfully execute if any element of a is zero and conditional evaluation rules are obeyed but will cause a divide-by-zero error if conditional assignment is used since the reciprocal of a is computed for all elements.

```
where a <> 0 do r := 1/a;
```

While conditional evaluation provides some additional capabilities, it also introduces some semantic difficulties. The main problem occurs when an array expression is passed to a procedure or function. What values are passed for those elements for which the controlling expression is false? Another problem arises with the use of standard functions which alter the shape of arrays. At what point is the masking applied?

7. Bit-Plane Indexing

A feature of several current highly parallel computers such as the MPP and the DAP is that arithmetic is conducted at the bit level rather than the word or number level. That is, the computer "word" or bit plane manipulated by these computers is a single-bit slice through all elements in the array being processed.

Some algorithms can be made considerably more efficient for these computers if specified at the bit plane level. Bit-plane indexing was added to Parallel Pascal to enable a programmer to conveniently specify most of these special algorithms without resorting to an assembly code subroutine.

A bit-plane index is specified by the last item in an index expression and is separated from other indices by a colon. The result of a bit-plane-indexed array has a Boolean element type. For example, given the definition

```
var a: array [1..5, 1..10] of integer;
var b: array [1..5, 1..10] of Boolean;
```

then the statement

```
b := a[:0];
```

is equivalent to

$$b := \text{odd}(a);$$

The next example subtracts one from the selected array element if necessary to make it exactly divisible by 2.

$$a[3, 1:0] := \text{false};$$

The least significant or first bit-plane is always bit-plane 0. Programming with bit-plane indexing requires a knowledge of the internal number representation of the parallel processor and is a highly nonportable feature. Furthermore, bit-plane indexing on a processor which does not operate at the bit level is usually very inefficient.

8. Parallel P-Code

The Parallel Pascal compiler consists of a syntax analysis "front end" and a code generation "back end." These two phases of the compiler communicate through an intermediate language called Parallel P-code [4]. The compiler and the P-code are based on the P-4 Pascal Compiler [5].

Parallel P-code has the following extensions relative to standard P-code for parallel languages and computers. First, it provides a mechanism by which nonprimitive types may be specified. This is needed because parts of an array may be specified which do not have an explicit type declaration. For example, the column of a two-dimensional array may be selected and used within a parallel expression; however, this column has not been previously defined by an explicit type statement.

Second, Parallel P-code provides an abstract addressing scheme for allocating and referencing automatically allocated variables. This is done to permit optimizing stages and the code generator to determine the memory system in which a data item should reside. In many parallel computers there are at least two data memory systems, the host memory and the parallel computer memory.

Third, Parallel P-code provides mechanisms for operating upon arrays, array subsets, and individual array elements. Fourth, it provides a symbolic mechanism for defining and referencing the fields of a **record** structure. This is needed because of the abstract addressing scheme of Parallel P-code. Finally, it facilitates conditional assignment, i.e., the **where** statement, by providing mechanisms for establishing, altering, and removing a Boolean mask array. A mask stack facility is supported for nested **where** statements.

9. Problems with Standard Pascal

The initial Pascal language as designed by Wirth is described in [6]; since then, the ISO standard Pascal has been specified, a very readable account of which is given by Cooper [7], and an IEEE/ANSI standard has also been specified [8]. There were two fundamental problems with standard Pascal which made it difficult to use in the parallel processor context. First, strong typing, one of the prime features of Pascal, made it very difficult to write general array subprograms which could operate on more than one size of array. Second, there is no library facility or separate compilation facility in standard Pascal. However, a large number of support functions are needed for many of the anticipated applications.

9.1. *Strong Typing*

Strong typing is supported with varying degrees in standard Pascal. For example, the range of integers is well defined and may be redefined; however, for real numbers there is no specification of either range or precision. Array types are very strongly typed; two total arrays are only conformable if they share a common type declaration. This has to be relaxed in the conformability of the **where** statement in Parallel Pascal. A controlling array must be of element type Boolean, while the controlled arrays may be of any element type. If two arrays have different element types then they must have completely separate type declarations.

A second fundamental problem is how to define a procedure or function which can deal with arrays of different sizes. The conformant array parameter described in the ISO standard (for Pascal level 1), but not in the ANSI standard, is a very good solution to this problem, especially in the parallel environment. In this scheme, array type information is explicitly specified with each array parameter. A subprogram may be specified to accept several different types for an argument; however, all these types are known at compile time, permitting an efficient code to be generated for each of them. This may be especially important for some array computers where the algorithm for the generated code may depend upon the array dimensions.

9.2. *Library Subprograms and Separate Compilation*

Standard Pascal has no library facility; all subprograms, i.e., procedures and functions, must be present in the source program. A library preprocessor was developed to allow the use of libraries without violating the rules of standard Pascal. The header line of a library subprogram is specified in the source program with an **extern** directive. The library preprocessor replaces the **extern** directive with the appropriate subprogram body. The type information for the library subprogram is extracted from the declaration statement in the source program. Therefore, library subprograms can be written to work with any user-specified array type.

If a library subprogram is to be used for more than one array type in the same block, then a subprogram declaration statement for each unique argument type is necessary. Each unique version of the subprogram is identified by a user-specified extension to the subprogram name in both declaration and usage.

For example, consider the ceiling function as defined below:

```
function ceiling(x:xtype) : rtype;
begin
    where x < 0.0 do
        ceiling := trunc(x)
    otherwise
        where x-trunc(x) = 0.0 do
            ceiling := trunc(x)
        otherwise
            ceiling := trunc(x) + 1;
end;
```

The following program fragment illustrates how more than one version of this function could be specified for the library preprocessor.

```
. . .
type
    ar = array [1..10] of real;
    ai = array [1..10] of integer;
    br = array [1..8, 1..8] of real;
    bi = array [1..8, 1..8] of integer;
function ceiling.a(x:ar) :ai; extern;
function ceiling.b(x:br) :bi; extern;
var
    ax:ar; ay:ai; bx:br; by:bi;
begin
. . .
```



```
    ay := ceiling.a(ax);
    by := ceiling.b(bx);
. . .
```

The simple library preprocessor does not solve the separate compilation problem: all requested library subprograms must be recompiled whenever a change is made to the main program. However, it is an expedient solution to the library problem which will work with all Parallel Pascal compilers. External, partially compiled subprograms could be inserted at the P-code level or at the code generator level of a compiler. However, they should be inserted before the optimization stage so that specific parallel computer sensitivities to different array sizes may be considered.

10. Conclusion

A version of the Pascal programming language for parallel computers has been developed which requires very few new language features. One of the main features of this language is that permutations are achieved with conventional function forms. In this way it is simple to introduce new permutation functions for the efficient programming of a new parallel computer, when necessary, without changing the language.

The obvious extension to standard Pascal was to allow operations on complete, and partially selected, arrays. Subrange constants were introduced to permit data to be transferred between arrays of different sizes. Several features were developed with current parallel computer architectures in mind. These include: the parallel specifier for systems with more than one data memory, conditional assignment instead of conditional evaluation, and bit indexing for computers with bit-serial arithmetic.

The strong typing of arrays in standard Pascal was found to be a problem in some cases and had to be relaxed. Also, a library management capability was considered to be very important. One of the attractive features of standard Pascal is that as many decisions as possible are made at compile time rather than at run time. This is a vitally important concept in any parallel language, and has been maintained in Parallel Pascal, since runtime decisions on factors such as array size can be very costly on parallel computers.

A version of the P-code intermediate language has been developed for Parallel Pascal. This contains two major changes from standard P-code. First, addressing is symbolic, rather than by direct memory offsets; this permits a code generator to select the best memory system for the data. Second, new operators have been introduced to deal with aggregate data structures.

ACKNOWLEDGMENTS

I gratefully acknowledge the assistance of John Bruner, who helped specify the language and wrote the P-code compiler; Mark Poret and Tony Brewer, who developed the Parallel Pascal translator; and Steve Elias, who developed the library preprocessor. Most of this work was supported by NASA Grant NAG 5-3.

REFERENCES

1. Batcher, K. E. Design of a Massively Parallel Processor. *IEEE Trans. Comput.* C-29, Sept. 1981), 836–840.
2. Reeves, A. P., and Bruner, J. D. The language Parallel Pascal and other aspects of the Massively Parallel Processor. Cornell University Tech. Rep., Dec. 1982.
3. Bruner, J. D. Efficient implementation of a high-level language on a bit-serial parallel matrix processor. Ph.D. thesis, Purdue University, 1982.
4. Bruner, J. D., and Reeves, A. P. A parallel P-Code for Parallel Pascal and other high level languages. *1983 International Conference on Parallel Processing*, Aug. 1983.
5. Nori, K. V., Ammann, U., Jensen, K., and Naegeli, H. The Pascal (P) compiler—Implementation notes. Institut for Informatik, Eidgenoessische Technische, Zurich, 1975.
6. Jensen, K., and Wirth, N. *Pascal User Manual and Report*. Springer-Verlag, New York/Berlin, 1976.
7. Cooper, D. *Standard Pascal Users Reference Manual*. Norton, New York, 1983.
8. *American National Standard Pascal Computer Programming Language*. IEEE, New York, 1983.

:d. J.L. Potter, MIT Press, 1985
)p. 230-260.

Appendix D

PARALLEL PASCAL AND THE MASSIVELY PARALLEL PROCESSOR

Anthony P. Reeves
School of Electrical Engineering
Cornell University
Ithaca, New York 14853

INTRODUCTION

Parallel Pascal is an extended version of the Pascal programming language which is designed for the convenient and efficient programming of parallel computers. Parallel Pascal was designed with the MPP as the initial target architecture. It is the first high level programming language to be implemented on the MPP.

The Parallel Pascal language is outlined in the first section of this chapter; then language restrictions on the current MPP compiler imposed by the MPP architecture are discussed. Finally, algorithm techniques for efficiently programming the MPP are presented.

Conventional serial high-level programming languages are difficult to efficiently implement on parallel computers. Most parallel computers are currently programmed in either assembly language or a machine-dependent special version of Fortran. The main advantages of a general high-level language for parallel computers, such as Parallel Pascal, are portability, better error detection and diagnosis facilities, and efficiency. Efficiency must be a prime consideration since with any parallel system extra hardware is being used to achieve a high speed performance. Portability is perhaps the most difficult goal to achieve while maintaining efficiency since different parallel systems have very different data permutation capabilities. The efficiency of working programs can be enhanced, in some cases, by reprogramming a small number of critical procedures in assembly code.

There are three fundamental classes of operations on array data which are frequently implemented as primitives on array computers but which are not available in conventional programming languages, these are: data reduction, data permutation and data broadcast. These operations have been included as primitives in Parallel Pascal.

The design of Parallel Pascal was directed towards the MPP; however, it is also suitable for other parallel computers with a similar interconnection scheme such as Illiac IV and the DAP. Parallel Pascal is also suitable for parallel computers with a less restrictive interconnection scheme such as is found with many pipeline systems for example. In this case, it may be necessary to implement some additional data mapping functions in order to take advantage of all the parallel computers capabilities. A more detailed discussion of the language design is given in [1]. An in depth description of Parallel Pascal and other aspects of the MPP is given in [2] and also in [3].

A Parallel Pascal to standard Pascal translator has been developed to allow initial experimentation with different language features. This translator is now being used for program development of Parallel Pascal programs on conventional serial computers.

A compiler has been developed [3, 2] which converts a Parallel Pascal program into a parallel p-code form. Interesting features of this p-code language include: a mechanism for non-primitive data types needed because of subarrays, an abstract addressing scheme for automatically allocated variables to permit the code generator to decide the appropriate host for the code, mechanisms for operating on arrays and subarrays, and a symbolic scheme for referencing fields of a record structure. An description of this p-code is given in [4] and also in [2]. A parallel p-code code generator for the MPP has been developed by Computer Sciences Corporation.

PARALLEL EXPRESSIONS

In Parallel Pascal all conventional expressions are extended to array data types. In a parallel expression all operations must have conformable array arguments. A scalar is considered to be conformable to any type compatible array and is conceptually converted to a conformable array with all elements having the scalar value. For example, given the definition

```
var a, b, c: array [1..10] of integer;
```

the following statement

```
a := b + c + 1;
```

is equivalent to

```
for i := 1 to 10 do
        a[i] := b[i] + c[i] + 1;
```

In many highly parallel computers including the MPP there are at least two different primary memory systems; one in the host and one in the processor array. Parallel Pascal provides the reserved word parallel to allow programmers to specify the memory in which an array should reside. In standard Pascal an array type is specified with the following syntax

```
type newtype = array [indextype] of eltype;
```

where indextype specifies the number and ranges of the array dimensions and eltype specifies the type of the array elements. A parallel array type is specified with the syntax

```
type newtype = parallel array [indextype] of eltype;
```

The parallel specifier exists only to provide information to the the compiler as to the variables usage. In all usage in the language a parallel array is indistinguishable from a conventional array. In some systems there is no distinction between host and processor memories, then the parallel specifier does not have any effect. In any case, a compiler may decline to store the array where requested.

ARRAY SELECTION

Selection of a portion of an array by selecting either a single index value or all index values for each dimension is frequently used in many parallel algorithms; e.g., to select the ith row of a matrix which is a vector. Specification of a single index value is the standard indexing method in standard Pascal. In Parallel Pascal all index values can be specified by eliding the index value for that dimension. For example, given the definition

```
var a,b: array [1..5,1..10] of integer;
```

in Parallel Pascal the statement

```
a[,1] := b[,4];
```

assigns the fourth column of b to the first column of a. The following are valid statements in standard Pascal

```
a := b;
```

```
a[1] := b[2];
```

The second statement means assign the second row of b to the first row of a; in Parallel Pascal this could also be specified by

```
a[1,] := b[2,];
```

SUBRANGE CONSTANTS

It is sometimes necessary to move data between arrays with different dimensions. In Parallel Pascal subarrays consisting of consecutive sets of elements may be specified. If subarrays with other than consecutive elements are required then they must be packed into the consecutive form with permutation functions. The concept of a constant subrange is introduced in order to specify a consecutive subset of index values.

The syntax for the constant subrange is

```
const identifier = low_high;
```

where low and high are either literals or previously defined constant identifiers.

SUBRANGE INDEXING AND ARRAY PACKING

Subrange constants may be used to index an array in Parallel Pascal. The general syntax for a subrange index is

```
array-identifier[ offset @ subrange-constant ]
```

where offset is an optional conventional scalar index expression. The ordered set of indices specified by a subrange index is the result of adding the value of the offset expression to the values implied by the subrange constant. For example, given the definition

```
var a, b: array [1..10] of integer;
```

the statement

```
a[@2..6] := b[3 @ 1..5];
```

is functionally equivalent to

```
for i := 1 to 5 do
   a[i + 1] := b[i + 3];
```

The main reason for introducing subrange indexing was to permit blocks of data to be transferred between arrays having different dimensions. It was not designed to be a tool for algorithm development.

ARRAY CONFORMABILITY

In standard Pascal, data items combined together in an expression must be type compatible. In Parallel Pascal, array data items in a parallel expression must also be conformable, i.e. have the same rank (number of dimensions) and the same range in each dimension. For example, given the definitions

```
var a, b: array [1..10] of integer;
    c: array [0..9] of integer;
```

the statement

```
a := a + b;
```

is conformable, while the statement

```
a := b + c;
```

is not conformable since the specified ranges of b and c are different.

While the exact range conformability requirement is in keeping with the strong typing concepts of standard Pascal, there are occasions when the action specified by the above statement is useful. The range requirement can be explicitly circumvented by using subrange indexing. For example, the statements

```
a := b + c[@0..9];
a[@1..10] := b[@1..10] + c;
a[@1..10] := b[@1..10] + c[@0..9];
```

are all conformable and have the same effect.

REDUCTION FUNCTIONS

Array reduction operations are achieved with a set of standard functions in Parallel Pascal which are listed in table 1.

Table 1: *Reduction Functions*

Syntax	Meaning
sum(array, D1, D2, ..., Dn)	reduce array with arithmetic sum
prod(array, D1, D2, ..., Dn)	reduce array with arithmetic product
all(array, D1, D2, ..., Dn)	reduce array with Boolean AND
any(array, D1, D2, ..., Dn)	reduce array with Boolean OR
max(array, D1, D2, ..., Dn)	reduce array with arithmetic maximum
min(array, D1, D2, ..., Dn)	reduce array with arithmetic minimum

The first argument of a reduction function specifies the array to be reduced and the following arguments specify which dimensions are to be reduced. A dimension is specified by a constant expression; the first dimension is dimension 1. The dimension parameters must be constant expressions so that the shape of the result is known at compile time.

For example, given the the definitions

```
var
    a: array[1..10,1..5] of integer;
    b: array[1..10] of integer;
    c: integer;
```

the following are correct Parallel Pascal statements

```
b := sum(a, 2);    (* sum the rows of a *)
```

```
c := sum(a, 1, 2); (* sum all elements of the array a *)
c := max(b, 1);    (* find the maximum value of b *)
```

Each dimension parameter of a reduction function implies that there will be one less dimension in the result array; a scalar is considered to be an array without any dimensions in this context.

PERMUTATION AND DISTRIBUTION FUNCTIONS

One of the most important features of a parallel programming language is the facility to specify parallel array data permutation and distribution operations. In Parallel Pascal four such operations are available as primitive standard functions; however, for some Parallel Processors it may be necessary to specify more primitive functions for efficiency. The standard Parallel Pascal functions for data permutation and distribution are given in table 2.

Table 2: Permutation and Distribution Functions

Syntax	Meaning
shift(array, S1, S2, ..., Sn)	end-off shift data within array
rotate(array, S1, S2, ..., Sn)	circularly rotate data within array
transpose(array, D1, D2)	transpose two dimensions of array
expand(array, dim, range)	expand array along specified dimension

SHIFT AND ROTATE

The shift and rotate primitives are found in many parallel hardware architectures and also, in many algorithms. The shift function shifts data by the amount specified for each dimension and shifts zeros (null elements) in at the edges of the array. Elements shifted out of the array are discarded. The rotate function is similar to the shift function except that data shifted out of the array is inserted at the opposite edge so that no data is lost. The first argument to the shift and rotate functions is the array to be shifted; then there is an ordered set of parameters, each one specifies the amount of shift in its corresponding dimension. There must be as many shift parameters as there are dimensions in the array; the first shift parameter is associated with the first dimension of the array.

For example, given the definitions

```
var
  a, b: array [1..5,0..9] of integer;
  c, d: array [0..9] of integer;
```

the statement

```
a := shift(b, 0, 3);
```

is functionally equivalent to

```
for i := 1 to 5 do
  begin
    for j := 0 to 6 do
      a[i,j] := b[i,j+3];
    for j := 7 to 9 do
      a[i,j] := 0;
```

```
    end;
```

and the statement

```
    c := rotate(d, 3);
```

is functionally equivalent to

```
    for i := 0 to 9 do
      c[i] := d[(i + 3) mod 10];
```

TRANSPOSE AND EXPAND

While transpose is not a simple function to implement with many parallel architectures, a significant number of matrix algorithms involve this function; therefore, it has been made available as a primitive function in Parallel Pascal. The first parameter to transpose is the array to be transposed and the following two parameters, which are constant expressions, specify which dimensions are to be interchanged. If only one dimension is specified then the array is flipped about that dimension.

The main data distribution function in Parallel Pascal is expand. This function increases the rank of an array by one by repeating the contents of the array along a new dimension. The first parameter of expand specifies the array to be expanded, the second parameter, a constant expression, specifies the number of the new dimension and the last parameter, a subrange or a subrange type, specifies the range of the new dimension.

This function is used to maintain a higher degree of parallelism in a parallel statement; this may result in a clearer expression of the operation and a more direct parallel implementation. In a conventional serial environment such a function would simply waste space.

For example, given the definitions of a, b, and c as specified in section 5.1 the following statement adds a vector to all rows of a matrix

```
    a := b + expand(c, 1, 1..5);
```

The above statement is functionally equivalent to the following

```
    for i := 1 to 5 do
      a[i,] := b[i,] + c;
```

CONDITIONAL EXECUTION

An important feature of any parallel programming language is the ability to have an operation operate on a subset of the elements of an array. In standard Pascal each array element is processed by a specific sequence of statements and there are a variety of program control structures for the repeated or selective execution of statments. In Parallel Pascal the whole array is processed by a single statement; therefore, an extended program control structure is needed.

The syntax of the Parallel Pascal where statement is as follows:

```
where array-expression do
  statement
otherwise
  statement
```

where array-expression is a Boolean valued array expression and statement is a Parallel Pascal statement. The otherwise and the second controlled statement may be omitted.

The execution of a where structure is defined as follows. First, the controlling expression is evaluated to obtain a Boolean array (mask array). Next, the first controlled statement is evaluated. Array assignments are masked according to the boolean control array. If there is an otherwise statement it is then evaluated; in this case array assignments are masked with the inverse of the control array.

For example, given the definition

```
var a, b, carray [1..10] of integer;
```

the following expression

```
where a < b do
   c := b
otherwise
   c := a;
```

is functionally equivalent to

```
for i := 1 to 10 do
  if a[i] < b[i] then
    c[i] := b[i]
  else
    c[i] := a[i];
```

The main semantic difference between the where-do-otherwise structure and the if-then-else structure is that with the former both controlled statements are evaluated, independent of the value of the control expression, while with the latter only one of the two controlled statements is evaluated.

Where statements may be nested provided that all of the controlling array expressions are type compatible. Other standard Pascal control statements can also be nested within where statements. Any array variable which appears on the left hand side of an assignment within a where controlled statement must be type compatible with the controlling array expression. Assignments to other than array variables in a where statement are in no way affected by the where statement. The effect of a where statement is local to the procedure or function in which it occurs; that is, it does not affect the execution of any procedures or functions called from within a where statement or an otherwise statement.

BIT-PLANE INDEXING

A feature of several current highly parallel computers such as the MPP is that arithmetic is conducted at the bit level rather than the word or number level. That is, the computer "word" or bit plane manipulated by these computers is a single bit slice through all elements in the array being processed.

Some algorithms can be made considerably more efficient for these computers if specified at the bit plane level. Bit-plane indexing was added to Parallel Pascal to enable a programmer to conveniently specify most of these special algorithms without resorting to an assembly code subroutine.

A bit-plane index is specified by the last item in an index expression and is separated from other indices by a colon. The result of a bit-plane indexed array has a Boolean element type. For example, given the definition

```
var a: array [1..5,1..10] of integer;
var b: array [1..5,1..10] of Boolean;
```

then the statement

```
b := a[:0];
```

is equivalent to

```
b := odd(a);
```

The next example subtracts one from the selected array element if necessary to make it exactly divisible by 2.

```
a[3,1:0] := false;
```

The least significant or first bit-plane is always bit-plane 0. Programming with bit-plane indexing requires a knowledge of the internal number representation of the parallel processor and is a highly non portable feature. Furthermore, bit-plane indexing on a processor which does not operate at the bit level is usually very inefficient.

LIBRARY SUBPROGRAMS AND SEPARATE COMPILATION

Standard Pascal has no library facility; all subprograms i.e., procedures and functions, must be present in the source program. A library preprocessor was developed to allow the use of libraries without violating the rules of standard Pascal. The header line of a library subprogram is specified in the source program with an extern directive. The library preprocessor replaces the extern directive with the appropriate subprogram body. The type information for the library subprogram is extracted from the declaration statement in the source program. Therefore, library subprograms can be written to work with any user specified array type.

If a library subprogram is to be used for more than one array type in the same block, then a subprogram declaration statement for each unique argument type is necessary. Each unique version of the subprogram is identified by a user specified extension to the subprogram name in both declaration and usage.

For example, consider the ceiling function as defined below:

```
function ceiling(xxtype) : rtype;
begin
   where x < 0.0 do
      ceiling := trunc(x)
   otherwise
      where x-trunc(x) = 0.0 do
         ceiling := trunc(x)
      otherwise
         ceiling := trunc(x)+1;
end;
```

The following program fragment illustrates how more than one version of this function could be specified for the library preprocessor.

```
...
type
     ar = array [1..10] of real;
     ai = array [1..10] of integer;
```

```
      br = array [1..8, 1..8] of real;
      bi = array [1..8, 1..8] of integer;
function ceiling.a(x:ar) :ai; extern;
function ceiling.b(x:br) :bi; extern;
var
    ax:ar; ay:ai; bx:br; by:bi;
begin
...
    ay := ceiling.a(ax);
    by := ceiling.b(bx);
...
```

The simple library preprocessor does not solve the separate compilation problem: all requested library subprograms must be recompiled whenever a change is made to the main program. However, it is an expedient solution to the library problem which will work with all Parallel Pascal compilers. External, partially compiled subprograms could be inserted at the p-code level or at the code generator level of a compiler. However, they should be inserted before the optimization stage so that specific parallel computer sensitivities to different array sizes may be considered.

MPP COMPILER RESTRICTIONS

The Parallel Pascal compiler for the MPP currently has several restrictions. The most important of these is that the range of the last two dimensions of a parallel array are constrained to be 128; i.e., to exactly fit the parallel array size of the MPP. It is possible that language support could have been provided to mask the hardware details of the MPP array size from the programmer; however, this would be very difficult to do and efficient code generation for arbitrary sized arrays could not be guaranteed. Matrices which are smaller than 128 x 128 can usually be fit into a 128 x 128 array by the programmer. Frequently, arrays which are larger than 128 x 128 are required and these are usually fit into arrays which have a conceptual size which is a multiple of 128 x 128.

A large matrix of dimensions (m * 128) x (n * 128) is specified by a four dimensional array in the MPP version of Parallel Pascal which has the dimensions m x n x 128 x 128. There are two fundamental methods for packing the large matrix data into this four dimensional array, this packing may be directly achieved by the staging memory in both cases. In the "crinkled" packing scheme a m x n matrix of adjacent large matrix elements is assigned to each PE; adjacent submatrices are assigned to adjacent PE's. More formally, element (i,j) of the large matrix is mapped to location [i mod m, j mod n, i div m, j div n] of the four dimensional array.

The alternative packing scheme, called "blocked" packing, assigns adjacent large matrix elements to adjacent PE's in blocks of 128 x 128. The large matrix is represented by a m x n matrix of adjacent 128 x 128 blocks. More formally, element (i,j) of the large matrix is mapped to location [i div 128, j div 128, i mod 128, j mod 128] of the four dimensional array.

The best method of large array packing is application dependent which is one reason that large arrays are not handled automatically by the compiler.

Programming with large matrices stored as four dimensional arrays is very simple in Parallel Pascal. In general, programs developed for a single 128 x 128 array are easily modified to deal with large packed matrices. Simple arithmetic expressions directly extend to higher dimensioned arrays, reduction functions may require additional dimension specifiers. Shift and rotate operations require special consideration since care must be taken to correctly transfer data between the boundaries of the submatrices. Generic library functions have been written in Parallel Pascal to deal with large matrices. The functions lshift and lrotate will correctly manipulate large matrices stored with the blocked packing scheme and the functions crshift and crrotate will deal with matrices stored with the crinkled packing scheme.

The hardware organization of the MPP currently imposes some further language restrictions. These could be removed with a more advanced version of the code generator. Host programs for the MPP can be run either on the main control unit (MCU) or on the VAX; in the latter case the MCU simply relays commands from the VAX to the PE array. The advantages of running on the VAX is a good programming environment, floating point arithmetic support and large memory (or virtual memory). The advantage of running on the MCU is more direct control of the MPP array.

Compiler directives are used to specify if the generated code should run on the MCU or the VAX. With the current implementation of the code generator, only complete procedures can be assigned to the MCU and only programs on the MCU can manipulate parallel arrays. There are several other language restrictions for programs which are run on the MCU such as no conventional I/O. Therefore, the programmer must isolate sections of code which deal with the PE array in procedures which are directed to the MCU. A better strategy might be to run the majority of the host program code on the VAX with only small sections which deal with PE array on the MCU, then there would be no language restrictions for the programmer but the code generator would be more complex.

LIBRARY PROGRAM DEVELOPMENT

Initial experience with developing algorithms for the MPP indicate that library functions must frequently be developed in three different forms for maximum efficiency. The basic form is a pure Parallel Pascal algorithm which takes the greatest advantage of the parallel features of the language. This form will run directly on the MPP array for arrays with the last two dimensions being 128 x 128.

The second form is for large arrays on the MPP, i.e., arrays with dimensions which are exact multiples of 128 x 128. In many cases, such as the near neighbor operations described in the next section, the transformation to this form from the previous form is very simple. In general, shift and rotate functions are replaced by lshift and lrotate library functions.

The third form is for functions which are to be implemented on the VAX host computer. While parallel algorithms will run correctly on the host computer such algorithms do not take advantage of the direct indexing capabilities of the VAX and much more efficient serial algorithms may be possible. Furthermore, any algorithms which use the bit-indexing language features can usually be much more efficiently reprogrammed for the VAX since it does not have an efficient bit indexing mechanism.

LIBRARY PACKAGES

The Parallel Pascal language provides basic efficient orthogonal primitives for developing application programs. However, it is expected that for any specific application area application directed primitive library functions will be required. Several application library packages have already been developed for Parallel Pascal. These include: large matrix shift and rotate, near neighbor functions, a general permutation function which is used for matrix rotation and polynomial warping, a parallel random number generator, and pyramid data structure functions.

In the next two sections, programming techniques will be illustrated with examples from the near neighbor package and the permutation package.

THE NEAR NEIGHBOR LIBRARY PACKAGE

The near neighbor package illustrates how Parallel Pascal can provide a convenient environment for specifying application primitives; the need for different forms of the same function is also demonstrated. A near neighbor (nn) operation is one in which each result element of a matrix is computed by a function of only locally adjacent elements in a

corresponding input matrix. Near neighbor operations are frequently used in image processing applications. Several high level languages for image processing include such operations as basic primitives.

The basic unit frequently used in near neighbor operations is a small matrix (3 x 3 to 7 x 7) of constant values. The first nn library function, called mx3, is used for specifying 3 x 3 matrices. The use of this function is illustrated in the following example.

```
type
    mtype = array [1..3, 1..3] of integer;
function  mx3(v00, v01, v02,
              v10, v11, v12,
              v20, v21, v22: integer): mtype; extern;
var  mc: mtype;
...
mc := mx3( -1, -1, -1,
           -1,  8, -1,
           -1, -1, -1);
```

The matrix mc is set with all boundary elements to -1 and the center element to 8 as is pictorially shown. A typical filtering operation is to convolve a large image matrix with a small kernel matrix; the generic library function conv is designed for this operation. A possible definition for conv is shown below:

```
type ...
    mx = parallel array [1..128, 1..128] of eltype;
function conv(matrix:mx; kernel:mtype):mx;
    var i,j: integer;
        sum: mx;
    begin
        sum:=0;
        for i:= 1 to 3 do begin
            for j:= 1 to 3 do begin
              if (kernel[i,j] <> 0) then
              sum := sum + kernel[i,j] * shift(matrix,i-2,j-2);
            end;
        end;
        conv := sum
    end;
var ...
    ma, mb:mx;
```

the convolution of ma with the kernel mc is specified by

```
mb := conv(ma, mc);
```

The contents of the kernel may also be expressed in the same statement; e.g.,

```
mb := conv(ma, mx3(0, 1, 0,
                   0, 0, 1,
                   0, 1, 0);
```

Sparse kernels with only a few 1 elements occur frequently in some applications. In these cases programmers often prefer to specify the kernel in a short-hand form consisting of a list of the cardinal directions of the 1's. The library function mxd converts such a list to the 3 x

3 matrix; using this function the above statement may be rewritten as

```
mb := conv(ma, mxd([N, E, S]));
```

The large matrix version of conv is simply specified by changing the word shift to lshift of crshift, as is appropriate, and redefining mx to be a four dimensional array. Conv is also reasonably organized for a serial processor. An improvement may be possible in this case by explicitly writing the loops so that the convolution is computed in one pass through the data since better use of a cache memory would result.

THE PERMUTATION LIBRARY PACKAGE

The need for more than one version of a library function for the same operation is illustrated with the permutation function. The matrix permutation function has three arguments: a data matrix to be permuted, a row matrix which indicates in which row from the data matrix the corresponding result element is to be obtained and a column matrix which indicates in which column the result is to be obtained. The function returns the permuted matrix (in fact any data mapping is possible). A serial version of this function is shown below:

```
type
    pa = array [lo1..hi1, lo2..hi2] of integer;

function perm2s(mx:pa; r:pa; c:pa):pa;
var
   i, j: integer;
begin
   for i := lo1 to hi1 do
     for j := lo2 to hi2 do
        perm2s[i,j] := mx[r[i,j],c[i,j]];
end;
```

This function is efficiently programmed for a serial computer such as the VAX host and would execute in $O(n^2)$ time for a n x n data matrix. In contrast, this would be a very poor algorithm for a parallel array. A single data transfer would require $O(n)$ time since only near neighbor shifts are possible; therefore, the total algorithm would require $O(n^3)$ time.

We have developed a parallel algorithm for this task which attempts to move all the data together as much as possible. On a parallel processor with $O(n^2)$ PE's this algorithm still has a worst case time complexity of $O(n^2)$ but for many structured permutations such as rotation and warping the complexity will be closer to $O(n)$.

The parallel algorithm will execute much more slowly on a serial processor than the serial algorithm since the serial algorithm makes direct use of the serial processors indexing capability. The parallel algorithm can be easily extended to large matrices which are multiples of 128 x 128 by directly replacing shift operations with lshift operations; however, this will not be optimal with respect to the number of shift operations needed. For example, with the blocked storage scheme a horizontal shift which is a multiple of 128 steps requires no shift operations at all since the data is already in the correct PE. A large matrix version of the algorithm is currently under development which will attempt to minimize the actual number of shift operations executed.

EXTENDED I/O

For many applications the standard I/O facilities of Pascal will be adequate. The staging buffer of the MPP can be very useful for directly performing certain data permutations; these permutations cannot be directly specified in the basic language of Parallel Pascal. A high level

language facility for using the staging buffer has not yet been implemented; a proposal for how the staging buffer may be programmed and used is outlined below.

There are three new capabilities, made possible with the staging buffer, which we would like to specify in the language.

1. File reformatting
   Raw data files may not be in the correct format for the MPP array. For example we may wish to do large matrix packing or select one band of a multiband file.
2. Sub-array file I/O
   In some cases it is useful to assemble a large array from sub arrays (which may be smaller than 128 x 128).
3. Data permutations
   The staging buffer is capable of implementing a large range of data permutations. In some cases it is effective to transfer data from the PE array through the staging buffer and back to the array in order to achieve a data permutation.

These new capabilities could be made available by introducing two new procedures to Parallel Pascal and relaxing one of the Pascal I/O constraints as indicated below:

FILE REFORMATTING

File reformatting can be achieved by the "reformat" procedure which specifies a reordering of the dimensions of an array structure. For example consider that we have a disk file of a set of 128 x 128 images with 6 bands. The file is declared as follows:

```
var f: file of parallel array [1..6, 1..128, 1..128] of 0..255;
```

This will work correctly if each image is stored on the disk as a sequence of 6 consecutive 128 x 128 matrices. If the data is stored in pixel interleaved format (i.e. as a sequence of 6 element pixels) then the staging buffer must be set to do the format conversion; this can be achieved with the following call to the reformat procedure:

```
reformat(f, 2, 3, 1);
```

This specifies that the ordering of the dimensions for the data on the disk is 2, 3, 1; i.e., the disk file array has the shape 128 x 128 x 6.

SUB-ARRAY I/O

In general, array I/O is done with a file having records of type array; an I/O operation then specifies the transfer of a complete array. Sub-array I/O may be achieved by relaxing the Pascal restriction that complete file data types must be read or written. The range of the dimensions of the subarray must match the last dimensions of the file array. For example, consider the 6 band image file described above and the following array declarations.

```
type ar = parallel array[1..128, 1..128] of 0..255;
var  a: ar;
     b: array [1..6] of ar;
```

With the conventional Pascal I/O restrictions only whole images can be read; i.e., read(f, b) is a valid statement whereas read(f, a) is not since a is not the same type as the file type.

In the extended I/O scheme, read(f, a) is permitted and reads the bands of an image one at a time. Parallel array files having the last dimensions smaller than 128 x 128 may be declared in which case a variable in an I/O statement such as a must have a subrange index for conformability.

DATA PERMUTATIONS

Data permutations can be achieved with a "link" procedure which links two files. Linked files form a virtual channel through the staging buffer and, in general, do not require any disk space. The data permutation is specified by reformatting the two files.

The syntax for link is:

```
link(f,g);
```

where f and g are files.
Link also implies a "rewrite" on file f and a "reset" on file g.

For example: a transpose permutation could be specified as follows:

```
type
    at = parallel array [1..128, 1..128] of real;
var
    fa, fb: file of at;
    a, b: at;
...
reformat(fa, 2, 1);
link(fa, fb);
write(fa, a);
read(fb, b);
```

The above set of statements is equivalent to

```
b := transpose(a, 2, 1);
```

CONCLUSION

A version of the Pascal programming language for parallel computers has been developed which required very few new language features. One of the main features of this language is that permutations are achieved with conventional function forms. In this way it is simple to introduce new permutation functions for the efficient programming of a new parallel computer, when necessary, without changing the language.

No attempt was made with the first implementation of the MPP compiler to hide the 128 x 128 dimensions of the PE array. This was considered to be necessary in order to ensure that efficient algorithms are developed and also ensure that the very limited local memory is not squandered. Programming tools have been outlined for programming larger arrays. A future compiler may hide these details from the user as effective programming techniques are better understood.

The only extensions needed for Parallel Pascal to effectively use the MPP hardware are the I/O extensions which consist of two new procedures and the relaxation of a standard Pascal constraint. These enable the staging buffer to be effectively utilized for data permutations and file reformatting.

Parallel Pascal provides convenient orthogonal efficient high level primitives on which to build application programs. It is more difficult to efficiently program a parallel computer than a serial computer; therefore, the establishment of library packages for application oriented primitives is even more important than for the conventional serial case. Some of the programming techniques for Parallel Pascal have been outlined in the context of packages which are currently being developed.

ACKNOWLEDGEMENTS

I gratefully acknowledge the assistance of John Bruner who helped specify the language and wrote the P-code compiler, Mark Poret and Tony Brewer who developed the Parallel

Pascal translator, and Steve Elias who developed the library preprocessor. Most of this work was supported with NASA grant NAG 5-3.

REFERENCES

1. A. P. Reeves, "Parallel Pascal: An extended Pascal for Parallel computers," *Journal of Parallel and Distributed Computing* 1 pp. 64-80 (1984).

2. A. P. Reeves and J. D. Bruner, "The Language Parallel Pascal and Other Aspects of the Massively Parallel Processor," Cornell University Technical Report (December 1982).

3. J. D. Bruner, "Efficient Implementation of a High -level Language on a Bit-Serial Parallel Matrix Processor," Ph.D. Thesis, Purdue University (1982).

4. J. D. Bruner and A. P. Reeves, "A Parallel P-Code for Parallel Pascal and Other High Level languages," *1983 International Conference on Parallel Processing*, pp. 240-243 (August 1983).

Appendix E

PARALLEL PASCAL DEVELOPMENT SYSTEM

Anthony P. Reeves
School of Electrical Engineering
Cornell University
Phillips Hall
Ithaca, New York 14853

Version 1.0

JANUARY 1985

PARALLEL PASCAL DEVELOPMENT SYSTEM

CONTENTS

Parallel Pascal Development System

Anthony P. Reeves
School of Electrical Engineering
Cornell University
Ithaca, New York 14853

Abstract

The Parallel Pascal Development System enables Parallel Pascal programs to be developed and tested on a conventional computer. It consists of several system programs, including a Parallel Pascal to standard Pascal translator, and a library of Parallel Pascal subprograms. The library includes subprograms for using Parallel Pascal on a parallel system with a fixed degree of parallelism, such as the Massively Parallel Processor, to conveniently manipulate arrays which have different dimensions than the hardware. Programs can be conveniently tested with small sized arrays on the conventional computer before attempting to run on a parallel system.

Introduction

This manual is organized in two sections: the first describes the system programs and the second describes available library subprograms. The library documentation is organized alphabetically on the name of the library function; when more than one function is described, the first listed name is used. A subject based index, which groups the library subprogram names under logically ordered application headings, is given at the end of the manual.

In general, a user of this system need be aware of only one program, called *pp*, which is used to compile and link Parallel Pascal programs on a conventional computer system. In order to use this system, a user should read the documentation on pp, which is in section 1 of this manual, and also be familiar with the Parallel Pascal programming language, which is a superset of the Pascal programming language. The Parallel Pascal extensions to Pascal are described in *ppascal* which is also in section 1 of this manual; a more detailed discussion of the language is given in [1]. An in depth description of Parallel Pascal and other aspects of the MPP is given in [2] and also in [3]. For information on standard Pascal, the initial Pascal language as designed by Wirth is described in [4] since then the ISO standard Pascal has been specified, a very readable account of which is given by Cooper [5], and an IEEE/ANSI standard has also been specified [6].

Library Subprograms

A set of library subprograms, written in Parallel Pascal, is available with this system. Documentation for these subprograms is given in section 2 of this manual. These subprograms will be automatically inserted into a users program if the correct library file comment specifier is given. See the detailed documentation for each library function to obtain this specifier. User library subprograms may also be easily used; see the documentation on *extern* in section 1 of this manual for the details of library program formats.

On a system with a fixed degree of parallelism, such as the Massively parallel Processor (MPP), the last dimensions of parallel arrays are fixed by the hardware (the last two dimensions must be 128 x 128 on the MPP). Arrays for such systems are categorized into four types: regular, which exactly match the hardware; large, which have dimensions which are an exact multiple of the hardware dimensions; small, which have dimensions smaller than the hardware dimensions; and huge, which are too large to fit into fast primary memory and must be accessed in regular sized blocks.

Regular arrays require no special treatment. Large arrays are organized as four dimensional structures; a set of library subprograms makes the programming of these arrays very similar to regular arrays. Programming for small arrays is usually very simple and application dependent and is currently left to the programmer. Huge arrays are very difficult to deal with in a general sense; some utilities are available for accessing arbitrarily located chunks from large arrays which should be useful for these applications. In some cases different algorithms should be used depending on where the program is to run; i.e., on the host computer or the parallel hardware. in some cases versions of the library functions to run on a serial host computer are also available.

Implementation

This development system involves three programs *pp, ppt and extern*. Pp is a command file which controls the compilation process; versions of pp are available for VAX-VMS and UNIX. It first invokes extern to insert the bodies (in Parallel Pascal source code) of any library subprograms and then invokes ppt to translate the complete Parallel Pascal program into standard Pascal. If no errors are detected by ppt then the translated program is compiled and linked with the host computers standard Pascal compiler and linker.

Ppt is a Parallel Pascal to Pascal translator, it is written in Pascal. It will only translate a subset of the Parallel Pascal language; see the documentation on ppt for restrictions.

Extern is a library preprocessor; it is written in C. A problem with the standard Pascal language is that there is no subprogram library facility. Library subprograms are referenced in a user program by a formal declaration statement and an *extern* directive. Additional information may be passed with the extern statement to allow, for example, a library subprogram to be used with different sized arrays. Library subprograms are specified as a Parallel Pascal body with a dummy declaration statement. Users can easily develop their own library subprograms.

This development system has been implemented on VAX-11/780 computers for both VMS and UNIX operating systems. It should work on any UNIX system which has a large enough memory and a Pascal compiler. Ppt has also been run on a Perkin Elmer computer. For a different computer system a Pascal compiler is necessary to compile ppt and to run the code it generates, pp will have to be rewritten for the host operating system and, for the automatic insertion of library subprograms, a C compiler will be necessary to compile extern.

References

1. A. P. Reeves, "Parallel Pascal: An extended Pascal for Parallel computers," *Journal of Parallel and Distributed Computing* 1 pp. 64-80 (1984).
2. A. P. Reeves and J. D. Bruner, "The Language Parallel Pascal and Other Aspects of the Massively Parallel Processor," Cornell University Technical Report (December 1982).
3. J. D. Bruner, "Efficient Implementation of a High -level Language on a Bit-Serial Parallel Matrix Processor," Ph.D. Thesis, Purdue University (1982).
4. K. Jensen and N. Wirth, *Pascal User Manual and Report*, Springer-Verlag (1976).
5. D. Cooper, *Standard Pascal Users Reference Manual*, W. W. Norton (1983).
6. , *American National Standard Pascal Computer Programming Language*, IEEE (1983).

NAME

extern — pascal external subprogram preprocessor

SYNOPSIS

extern [-s] lib1 lib2 libn <infile >outfile

DESCRIPTION

Extern reads a source program on the standard input and replaces external function references with the source code from library files. Library subprograms are specified in the source program by a function or procedure statement followed by an *extern* directive. Up to 100 string constants (parameters) may be substituted for in the source code for each function/procedure. The modified source program is written to standard output.

The external function source code is taken from library files which may be specified in the command line for extern as shown in the synopsis or in library commands in the program text. A program library command has the following format:

{$library libname }

or

(*$library libname *)

where libname is the name of a library file. The program library commands are in the form of Pascal comments so they we have no effect on a subsequent Pascal or Parallel Pascal compiler.

The library file names usually have a .pl extension. *Extern* first checks if a specified library file name (or path name) is in the current directory. If no file is found, it then checks for the same name with a .pl extension. It then checks in the system library directory first with the file name as specified and then with a .pl extension. A library not found error is reported if the file has still not been found.

Extern is designed to be used with any Pascal compiler; however, it has a special listing feature when used with the Parallel Pascal Translator ppt(1). By default, it inserts ppt flags (in comment statements) into the output file which prevents the inserted subprogram bodies from appearing in the listing file. In this way, the listing file corresponds to the original source file in both content and line numbers; any errors detected in the library functions will still be reported. If the *-s* option is specified then all of the output file will appear in the subsequent listing file.

Extern is rather primitive and does not understand much about Pascal syntax. External function and procedure declarations must start on a new line and the the declaration and extern directive, with any required additional parameters, should appear consecutively without any comment statements. (Comment statements bounded by '{}' which do not contain any semicolon symbols ';' should be o.k.). The key words 'function', 'procedure', 'extern' and 'library' are only recognized if typed in lower case letters.

MULTIPLE INSTANCIES OF A FUNCTION

There are many cases when we may wish to have the same subprogram defined to operate on arguments having different types in the same program block. Extern has a facility for defining multiple versions of a library subprogram. The function name in the source program can be followed by a dot (.) and a single character version identifier, e.g. function conv3.0, conv3.1. All instances of the subprogram usage in the source program body must have the correct version identifier.

LIBRARY FILE FORMAT

Each function in a library file is delimited by a pound sign (#), as follows:

```
#func1
```

```
{local vars, body of func1, including arbitrary references to
 the variables $0 through $99 as needed}
#func2
{local vars, body of func2, including arbitrary references to
 the variables $0 through $99 as needed}
#end
```

The library function specification is similar to a pascal subprogram without the procedure or function statement as this is provided by the source program.

The $ variables are used to specify types which are taken from the declaration of the function in the source program. Types or constants which do not appear in the argument list may be specified after the extern statement. The following example illustrates how $ variables are obtained from the function definition in the source file.

```
function conv3(var1:type1,var2:type2):returntype; extern type3, type4,type5;
```

$0 is always the return type of the function. In this case,
$1=type1, $2=type2, $3=type3, $4=type4, $5=type5.

Extern will also preprocess external procedure definitions; in this case, $0 is not defined.

AUTHORS

Steve Elias, Anthony P. Reeves

BUGS

key words are only recognized in lower case letters.
external functions and procedures must each be declared starting on a new line.
external function and procedure declarations should not contain comments.

NAME

pp — Parallel Pascal translator and compiler

SYNOPSIS

pp name [-i] [-s] [lib1 lib2 ...]

DESCRIPTION

Pp is the Parallel Pascal compiler. If given a file name ending in .pp it will compile the file and load it into an executable file having the same name as the original file without the .pp extension.

If the *-i* option is specified then the Pascal translator pi is used; the interpreter code is loaded in to the file obj for interpretation by px. For program development with small data sets the -i option is strongly recommended as it is usually much faster.

If any file names without a .pp extension are specified then these are assumed to be library files. Relevent subprograms will be extracted from these files by the extern(1) library processor.

If the *-s* option is specified then the complete library functions will be written to the listing file "pplist". In this case the listing line numbers will not correspond to the source program line numbers.

Pp uses the Parallel Pascal translator *ppt* to convert the named file into a conventional Pascal program which is stored in a file having the same name as the input file but with a .p instead of the .pp extension. A listing of the translated program including any error messages is stored in *pplist*. There are several Parallel Pascal language restrictions with the translator; see ppt(1) for details.

The translated program is compiled with pc by default and by pi if the -i option is specified. If any errors are detected by ppt no further compilation takes place.

FILES

file.pp	input file
file.p	translated Pascal file
file	output file from pc
obj	output file from pi
pplist	Parallel Pascal listing

SEE ALSO

ppt(1), pc(1), pi(1), px(1), extern(1)

AUTHOR

Anthony P. Reeves

NAME

ppt — Parallel Pascal translator

SYNOPSIS

ppt <infile >outfile

DESCRIPTION

Ppt is a Parallel Pascal translator which translates a Parallel Pascal program into a conventional Pascal program. Ppt is itself written in Pascal (based on the P4 compiler) for portability.

Ppt reads a Parallel Pascal source program from the standard input and writes the translated program onto the standard output. The listing of the source program, including any error messages, is stored in a file called pplist. Any errors are also reported to the terminal. Ppt only translates a subset of the Parallel Pascal language; see the bugs section for details of the language restrictions. WARNING - The first pass of ppt checks for a valid Parallel Pascal program but does not check for all the restrictions and limitations of the second pass.

Translator options are written as comments intermixed throughout the source program. Comments are designated as option comments by a $-character as the first character of the comment as follows:

(*$<option sequence> <any comment> *)

Example: (*$l+,m- *)

The option sequence uses commas to separate options. Each option consists of a letter indicating the option desired and either a plus (+) or minus (-) to indicate whether to activate or deactivate the option.
The following options are presently available:

l - Create a listing of the source program as the syntax analysis is being conducted. Default is l+.

m - Map upper case letters to lower case letters. (All reserved words and standard functions or procedures are only recognized in lower case.) Default is m-.

c - Output both the translated code and the original code to the standard output. (The original code is written out within comments to not affect the actual program.) Default is c-.

uxxx -
Limit the line length of the translated output to xxx characters where possible. The range of acceptable values is from 80 to 132 characters. Default is u132.

The options take effect directly after being issued and therefore they may be selectively applied to the program sections.

Throughout the design of the translator emphasis was put on the production of reliable and predictable translations and not on the production of efficient translations. There are some necessary optimizations which were implemented that greatly reduce the amount of both data space and instruction space used. These included temporary variable reusage and type matching for Parallel Pascal functions.

An additional feature of ppt is the ability to declare a function or procedure as being external to the program where it is called. The syntax for an external declared function or procedure is shown below:

procedure identifier (parameter list); extern;

- or -

function identifier (parameter list): result_type; extern;

FILES

infile	Parallel Pascal inputfile
outfile	Pascal output file
pplist	file containing listing

DIAGNOSTICS

A detected error is indicated by placing an arrow followed by a number (which corresponds to the appropriate error message) below the source line where the error was detected. For each error number issued throughout the source program the corresponding error message is printed out at the end of the listing. Error messages for Standard Pascal were obtained from the Pascal User Manual and Report by Jensen and Wirth, the error messages added for the Parallel Pascal features are listed below.

350: must be parallel array type
351: illegal type for parallel array
360: parallel arrays not compatible
361: parallel array not compatible with controlling array
362: result must be parallel type
363: parallel array not allowed
364: function result type must be parallel array
365: dimension not compatible with array
366: integer constant expected
367: at least one dimension expected
368: bit index type must be integer
369: error in number of standard function arguments
370: subrange exceeds array index limits
371: set type not compatible with array index type
372: index type must be scalar, subrange or set
373: bit indexing not allowed
374: illegal array type for bit indexing

AUTHORS

Anthony P. Reeves, Tony M. Brewer, Steve Elias, Mike Vernick

RESTRICTIONS

1. Constant subranges are not implemented
2. Bit indexing is only implemented for types integer and subrange.
3. Parallel arrays cannot have more than nine dimensions.
4 The index type of a parallel array must be integer.
5 All parallel identifiers declared by ppt have "pll" as a prefix to the identifier name. Therefore a user identifier should not begin with this prefix. Also, standard parallel functions are translated to have the same name but ending with a numeral; therfore names of this form sould be avoided.
6 Input and output: Read, readln, write and writeln may have at most one parallel parameter which must be the first parameter of the I/O procedure except for possibly a file specifier. If readln or writeln is used, then each element of the array would be read or written using readln or writeln respectively not just the last element. All other

Standard Pascal I/O facilities are restricted to non-parallel use.

7 Identifiers are limited to a maximum of 10 characters.

Parallel Pascal: Summary

Anthony P. Reeves
School of Electrical Engineering
Cornell University
Ithaca, New York 14853

INTRODUCTION

Parallel Pascal is an extended version of Pascal for programming parallel computers. This summary briefly describes the extensions which have been made to Pascal.

PARALLEL EXPRESSIONS

In Parallel Pascal all conventional expressions are extended to array data types. In a parallel expression all operations must have conformable array arguments. A scalar is considered to be conformable to any type compatible array and is conceptually converted to a conformable array with all elements having the scalar value. For example, given the definition

```
var a, b, c: array [1..10] of integer;
```

the following statement

```
a := b + c + 1;
```

is equivalent to

```
for i := 1 to 10 do
        a[i] := b[i] + c[i] + 1;
```

In many highly parallel computers including the MPP there are at least two different primary memory systems; one in the host and one in the processor array. Parallel Pascal provides the reserved word **parallel** to allow programmers to specify the memory in which an array should reside. In standard Pascal an array type is specified with the following syntax

```
type newtype = array [indextype] of eltype;
```

where indextype specifies the number and ranges of the array dimensions and eltype specifies the type of the array elements. A parallel array type is specified with the syntax

```
type newtype = parallel array [indextype] of eltype;
```

The parallel specifier exists only to provide information to the the compiler as to the variables usage. In all usage in the language a parallel array is indistinguishable from a conventional array. In some systems there is no distinction between host and processor memories, then the parallel specifier does not have any effect. In any case, a compiler may decline to store the array where requested.

ARRAY SELECTION

Selection of a portion of an array by selecting either a single index value or all index values for each dimension is frequently used in many parallel algorithms; e.g., to select the ith row of a matrix which is a vector. Specification of a single index value is the standard

indexing method in standard Pascal. In Parallel Pascal all index values can be specified by eliding the index value for that dimension. For example, given the definition

```
var a,b: array [1..5,1..10] of integer;
```

in Parallel Pascal the statement

```
a[,1] := b[,4];
```

assigns the fourth column of b to the first column of a. The following are valid statements in standard Pascal

```
a := b;
a[1] := b[2];
```

The second statement means assign the second row of b to the first row of a; in Parallel Pascal this could also be specified by

```
a[1,] := b[2,];
```

SUBRANGE CONSTANTS

It is sometimes necessary to move data between arrays with different dimensions. In Parallel Pascal subarrays consisting of consecutive sets of elements may be specified. If subarrays with other than consecutive elements are required then they must be packed into the consecutive form with permutation functions. The concept of a constant subrange is introduced in order to specify a consecutive subset of index values.

The syntax for the constant subrange is

```
const identifier = low..high;
```

where low and high are either literals or previously defined constant identifiers.

SUBRANGE INDEXING AND ARRAY PACKING

Subrange constants may be used to index an array in Parallel Pascal. The general syntax for a subrange index is

```
array-identifier[ offset @ subrange-constant ]
```

where offset is an optional conventional scalar index expression. The ordered set of indices specified by a subrange index is the result of adding the value of the offset expression to the values implied by the subrange constant. For example, given the definition

```
var a, b: array [1..10] of integer;
```

the statement

```
a[@2..6] := b[3 @ 1..5];
```

is functionally equivalent to

```
for i := 1 to 5 do
```

```
a[i + 1] := b[i + 3];
```

The main reason for introducing subrange indexing was to permit blocks of data to be transferred between arrays having different dimensions. It was not designed to be a tool for algorithm development.

ARRAY CONFORMABILITY

In standard Pascal, data items combined together in an expression must be type compatible. In Parallel Pascal, array data items in a parallel expression must also be conformable, i.e. have the same rank (number of dimensions) and the same range in each dimension. For example, given the definitions

```
var a, b: array [1..10] of integer;
    c: array [0..9] of integer;
```

the statement

```
a := a + b;
```

is conformable, while the statement

```
a := b + c;
```

is not conformable since the specified ranges of b and c are different.

While the exact range conformability requirement is in keeping with the strong typing concepts of standard Pascal, there are occasions when the action specified by the above statement is useful. The range requirement can be explicitly circumvented by using subrange indexing. For example, the statements

```
a := b + c[@0..9];
a[@1..10] := b[@1..10] + c;
a[@1..10] := b[@1..10] + c[@0..9];
```

are all conformable and have the same effect.

REDUCTION FUNCTIONS

Array reduction operations are achieved with a set of standard functions in Parallel Pascal which are listed in table 1.

Table 1: Reduction Functions

Syntax	Meaning
sum(array, D1, D2, .., Dn)	reduce array with arithmetic sum
prod(array, D1, D2, .., Dn)	reduce array with arithmetic product
all(array, D1, D2, .., Dn)	reduce array with Boolean AND
any(array, D1, D2, .., Dn)	reduce array with Boolean OR
max(array, D1, D2, .., Dn)	reduce array with arithmetic maximum
min(array, D1, D2, .., Dn)	reduce array with arithmetic minimum

The first argument of a reduction function specifies the array to be reduced and the following arguments specify which dimensions are to be reduced. A dimension is specified by a constant expression; the first dimension is dimension 1. The dimension parameters must be constant expressions so that the shape of the result is known at compile time.

For example, given the the definitions

```
var
   a: array[1..10,1..5] of integer;
   b: array[1..10] of integer;
   c: integer;
```

the following are correct Parallel Pascal statements

```
b := sum(a, 2);     (* sum the rows of a *)
c := sum(a, 1, 2);  (* sum all elements of the array a *)
c := max(b, 1);     (* find the maximum value of b *)
```

Each dimension parameter of a reduction function implies that there will be one less dimension in the result array; a scalar is considered to be an array without any dimensions in this context.

PERMUTATION AND DISTRIBUTION FUNCTIONS

One of the most important features of a parallel programming language is the facility to specify parallel array data permutation and distribution operations. In Parallel Pascal four such operations are available as primitive standard functions; however, for some Parallel Processors it may be necessary to specify more primitive functions for efficiency. The standard Parallel Pascal functions for data permutation and distribution are given in table 2.

Table 2: Permutation and Distribution Functions

Syntax	Meaning
shift(array, S1, S2, ..., Sn)	end-off shift data within array
rotate(array, S1, S2, ..., Sn)	circularly rotate data within array
transpose(array, D1, D2)	transpose two dimensions of array
expand(array, dim, range)	expand array along specified dimension

SHIFT AND ROTATE

The shift and rotate primitives are found in many parallel hardware architectures and also, in many algorithms. The shift function shifts data by the amount specified for each dimension and shifts zeros (null elements) in at the edges of the array. Elements shifted out of the array are discarded. The rotate function is similar to the shift function except that data shifted out of the array is inserted at the opposite edge so that no data is lost. The first argument to the shift and rotate functions is the array to be shifted; then there is an ordered set of parameters, each one specifies the amount of shift in its corresponding dimension. There must be as many shift parameters as there are dimensions in the array; the first shift parameter is associated with the first dimension of the array.

For example, given the definitions

```
var
   a, b: array [1..5,0..9] of integer;
```

```
c, d: array [0..9] of integer;
```

the statement

```
a := shift(b, 0, 3);
```

is functionally equivalent to

```
for i := 1 to 5 do
 begin
  for j := 0 to 6 do
     a[i,j] := b[i,j+3];
  for j := 7 to 9 do
     a[i,j] := 0;
 end;
```

and the statement

```
c := rotate(d, 3);
```

is functionally equivalent to

```
for i := 0 to 9 do
  c[i] := d[(i + 3) mod 10];
```

TRANSPOSE AND EXPAND

While transpose is not a simple function to implement with many parallel architectures, a significant number of matrix algorithms involve this function; therefore, it has been made available as a primitive function in Parallel Pascal. The first parameter to transpose is the array to be transposed and the following two parameters, which are constant expressions, specify which dimensions are to be interchanged. If only one dimension is specified then the array is flipped about that dimension.

The main data distribution function in Parallel Pascal is expand. This function increases the rank of an array by one by repeating the contents of the array along a new dimension. The first parameter of expand specifies the array to be expanded, the second parameter, a constant expression, specifies the number of the new dimension and the last parameter, a subrange or a subrange type, specifies the range of the new dimension.

This function is used to maintain a higher degree of parallelism in a parallel statement; this may result in a clearer expression of the operation and a more direct parallel implementation. In a conventional serial environment such a function would simply waste space.

For example, given the definitions of a, b, and c as specified in section 5.1 the following statement adds a vector to all rows of a matrix

```
a := b + expand(c, 1, 1..5);
```

The above statement is functionally equivalent to the following

```
for i := 1 to 5 do
   a[i,] := b[i,] + c;
```

CONDITIONAL EXECUTION

An important feature of any parallel programming language is the ability to have an operation operate on a subset of the elements of an array. In standard Pascal each array element is processed by a specific sequence of statements and there are a variety of program control structures for the repeated or selective execution of statments. In Parallel Pascal the whole array is processed by a single statement; therefore, an extended program control structure is needed.

The syntax of the Parallel Pascal **where** statement is as follows:

```
where array-expression do
  statement
otherwise
  statement
```

where array-expression is a Boolean valued array expression and statement is a Parallel Pascal statement. The **otherwise** and the second controlled statement may be omitted.

The execution of a **where** structure is defined as follows. First, the controlling expression is evaluated to obtain a Boolean array (mask array). Next, the first controlled statement is evaluated. Array assignments are masked according to the boolean control array. If there is an otherwise statement it is then evaluated; in this case array assignments are masked with the inverse of the control array.

For example, given the definition

```
var a, b, c:array [1..10] of integer;
```

the following expression

```
where a < b do
  c := b
otherwise
  c := a;
```

is functionally equivalent to

```
for i := 1 to 10 do
  if a[i] < b[i] then
    c[i] := b[i]
  else
    c[i] := a[i];
```

The main semantic difference between the **where-do-otherwise** structure and the **if-then-else** structure is that with the former both controlled statements are evaluated, independent of the value of the control expression, while with the latter only one of the two controlled statements is evaluated.

Where statements may be nested provided that all of the controlling array expressions are type compatible. Other standard Pascal control statements can also be nested within **where** statements. Any array variable which appears on the left hand side of an assignment within a where controlled statement must be type compatible with the controlling array expression. Assignments to other than array variables in a where statement are in no way affected by the where statement. The effect of a where statement is local to the procedure or function in which it occurs; that is, it does not affect the execution of any procedures or functions called from within a where statement or an otherwise statement.

BIT-PLANE INDEXING

A feature of several current highly parallel computers such as the MPP is that arithmetic is conducted at the bit level rather than the word or number level. That is, the computer "word" or bit plane manipulated by these computers is a single bit slice through all elements in the array being processed.

Some algorithms can be made considerably more efficient for these computers if specified at the bit plane level. Bit-plane indexing was added to Parallel Pascal to enable a programmer to conveniently specify most of these special algorithms without resorting to an assembly code subroutine.

A bit-plane index is specified by the last item in an index expression and is separated from other indices by a colon. The result of a bit-plane indexed array has a Boolean element type. For example, given the definition

```
var a: array [1..5,1..10] of integer;
var b: array [1..5,1..10] of Boolean;
```

then the statement

```
b := a[:0];
```

is equivalent to

```
b := odd(a);
```

The next example subtracts one from the selected array element if necessary to make it exactly divisible by 2.

```
a[3,1:0] := false;
```

The least significant or first bit-plane is always bit-plane 0. Programming with bit-plane indexing requires a knowledge of the internal number representation of the parallel processor and is a highly non portable feature. Furthermore, bit-plane indexing on a processor which does not operate at the bit level is usually very inefficient.

NAME

allm — masked all reduction functions

SYNOPSIS

```
{ $library reduce.pl }

function allm1( arg, mask: vec ): boolean; extern;
function allm2( arg, mask: mat ): boolean; extern;
function allm3( arg, mask: arr3 ): boolean; extern;
function allm4( arg, mask: arr4 ): boolean; extern;
function allm5( arg, mask: arr5 ): boolean; extern;
```

TYPES

vec: a vector of type boolean
mat: a matrix of type boolean
arr3: a three dimensional array of type boolean
arr4: a four dimensional array of type boolean
arr5: a five dimensional array of type boolean

DESCRIPTION

These functions reduce the first argument where there are true elements in the boolean mask second argument. All dimensions are automatically reduced and the final result is always a scalar value. The range of the dimensions of arg and mask must match. The following functions are available:

allm1 all reduction for vector arguments

allm2 all reduction for matrix arguments

allm3 all reduction for three dimensional arrays

allm4 all reduction for four dimensional arrays

allm5 all reduction for five dimensional arrays

NAME

anym — masked any reduction functions

SYNOPSIS

```
{ $library reduce.pl }

function anym1( arg, mask: vec ): boolean; extern;
function anym2( arg, mask: mat ): boolean; extern;
function anym3( arg, mask: arr3 ): boolean; extern;
function anym4( arg, mask: arr4 ): boolean; extern;
function anym5( arg, mask: arr5 ): boolean; extern;
```

TYPES

vec: a vector of type boolean
mat: a matrix of type boolean
arr3: a three dimensional array of type boolean
arr4: a four dimensional array of type boolean
arr5: a five dimensional array of type boolean

DESCRIPTION

These functions reduce the first argument where there are true elements in the boolean mask second argument. All dimensions are automatically reduced and the final result is always a scalar value. The range of the dimensions of arg and mask must match. The following functions are available:

anym1 any reduction for vector arguments

anym2 any reduction for matrix arguments

anym3 any reduction for three dimensional arrays

anym4 any reduction for four dimensional arrays

anym5 any reduction for five dimensional arrays

NAME

blint — Bilinear interpolation procedure for a matrix

SYNOPSIS

{$library blint.pl}

procedure blint(ro,co: real; rp,cp: pli; rf,cf: plr; msk: plb; b,d: plr; var result: plr); extern mxrowl, mxrowh, mxcoll, mxcolh;

TYPES

plr = array [mxrowl..mxrowh,mxcoll..mxcolh] of btype;
pli = array [mxrowl..mxrowh,mxcoll..mxcolh] of itype;
plb = array [mxrowl..mxrowh,mxcoll..mxcolh] of boolean;
Where btype is any type and itype is an integer or subrange base type

EXTERN CONSTANTS

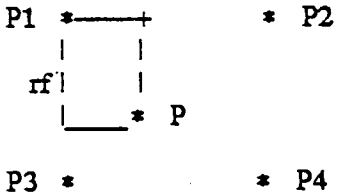
mxrowl = the smallest row number of the input matrix
mxrowh = the largest row number of the input matrix
mxcoll = the smallest column number of the input matrix
mxcolh = the largest column number of the input matrix

DESCRIPTION

Blint is a procedure that can be used in conjunction with irotate (see rotation(2)). It performs a local search to find three out of four vertices of a square that contains the point whose value we want to interpolate. The fourth point is the top left corner of the square and is immediately defined from the matrices rp and cp (row and column coordinates) and matrix b (value of the point). Matrix d is a horizontally shifted version of matrix b. Matrices rf and cf contain the coefficients used to perform the interpolation. The false values in matrix msk indicate the positions in the rotated matrix that will be filled with zeros. The coordinates ro and co are the center of the rotation computed in irotate.

The bilinear interpolation is performed in the following way:

ta 4.1c cf



P = (1 - cf)*(1 - rf)*P1 + (1 -rf)*cf*P2 + (1 - cf)*rf*P3 + cf*rf*P4

for all points P in the matrix.

AUTHOR

Cristina Moura

SEE ALSO

rotation(2),lblint(2),cint(2)

NAME

ceiling — round up to integer value

SYNOPSIS

```
{$library math.pl }

function ceiling( x:atype ): rtype; extern;
```

TYPES

atype: an arbitrary shaped array of element type real
rtype: an array with similar dimensions to atype of element type integer or subrange

DESCRIPTION

Ceiling converts a real array to an integer array rounding each element to the smallest integer not less than the element.

NAME

cint — Cubic interpolation procedure for a matrix

SYNOPSIS

{$library cint.pl}

procedure cint(ro,co: real; rp,cp: pli; rf,cf: plr; msk: plb; b,d: plr; var result: plr); extern mxrowl, mxrowh, mxcoll, mxcolh;

TYPES

plr = array [mxrowl..mxrowh,mxcoll..mxcolh] of btype;
pli = array [mxrowl..mxrowh,mxcoll..mxcolh] of itype;
plb = array [mxrowl..mxrowh,mxcoll..mxcolh] of boolean;
Where btype is any type and itype is an integer or subrange base type

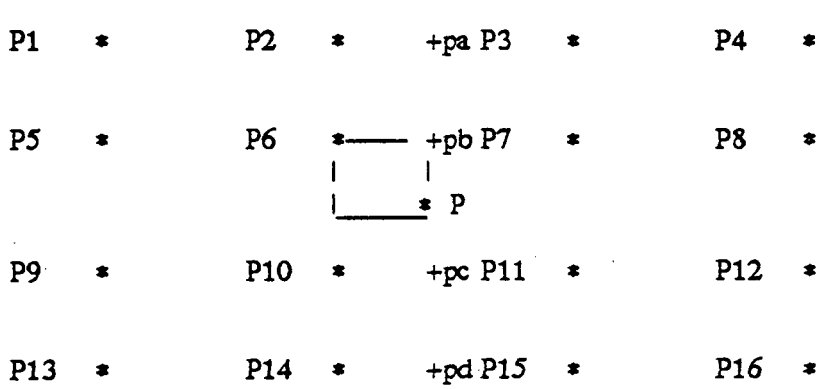
EXTERN CONSTANTS

mxrowl = the smallest row number of the input matrix.
mxrowh = the largest row number of the input matrix
mxcoll = the smallest column number of the input matrix.
mxcolh = the largest column number of the input matrix.

DESCRIPTION

Cint is a procedure that can be used in conjunction with irotate (see rotation(2)). It performs a local search to find fifteen out of sixteen points located around the point whose value we want to interpolate. The sixteenth point is immediately defined from the matrices rp and cp (row and column coordinates) and matrix b (value of the point). Matrix d is a horizontally shifted version of matrix b. Matrices rf and cf contain the coefficients used to perform the interpolation. The false values in matrix msk indicate the positions that will be filled with zeros in the rotated matrix. The coordinates ro and co are those of the center of the rotation computed in irotate.

The cubic interpolation for sixteen points is done by, first, performing a cubicinterpolation for each one of the four rows and, then, based on the points obtained, performing a fifth cubic interpolation.



We already have the value and position of point P6 through matrices rp,cp and b. The first four cubic interpolations are performed to obtain points pa, pb, pc and pd. The fifth one yields the value of point P.

AUTHOR

Cristina Moura

SEE ALSO

rotation(2),blint(2),lblint(2)

NAME

compn — near neigbor comparison function

SYNOPSIS

```
{$library mx.pl }

compn (m:mtype; w:wtype ):mtype; extern size;
```

TYPES

```
mtype = parallel array [lo1..hi1, lo2..hi2] of boolean;
wtype = array [0..size, 0..size] of 0..2;
```

DESCRIPTION

Compn compares the local neighborhood of each element of the boolean input matrix m with the window w. If a match occures then the result element is true, otherwise it is false. A zero in w matches with false in m, a one in w matches with true in m and a two in w is a don't care.

AUTHOR

A. P. Reeves

NAME

conv,convg — matrix convolution functions

SYNOPSIS

```
{$library convolve }

function conv(matrix:mtype; kernel:ktype ):rtype; extern size;

function convg(matrix:mgtype; kernel:kgtype ):rgtype; extern kl1, kh1, kl2, kh2,
mshift;
```

TYPES

```
rtype = parallel array [lo1..hi2, lo1..hi2] of rbtype;
mtype = parallel array [lo1..hi1, lo2..hi2] of mbtype;
ktype = array [0..size-1, 0..size-1] of kbtype;

mgtype = parallel array[ idxtype ] of mbtype;
rgtype = parallel array[ idxtype ] of rbtype;
kgtype = array[kl1..kl2, kl2..kh2] of kbtype;
```

where rtype, mbtype and kbtype are base types; rbtype must be conformable with the product of a kbtype with a mbtype.

EXTERN CONSTANT

size is a constant specifying the size of the kernel.

DESCRIPTION

Conv is a convolution function which convolves a small matrix with a large matrix. The small matrix is typically stored in the host computer. The result matrix has the same dimensions as the large matrix. For the highest large matrix index values the small matrix will overlap the large matrix edge. Zero values are used for large matrix elements beyond this edge. If it is desired that the result matrix is centered on the input matrix; i.e. a result element is computed from the same spatially located element in the input matrix and its near neighbors, then this may be achieved by preshifting the input matrix or postshifting the result matrix.

Convg is an extended version of *conv*. For this function the kernel index range is not constrained to start at zero and is explicitly specified by the extern parameters. Furthermore, the shift function used by convg is also specified by an extern parameter. For example, lshift or crshift could be specified which would indicate that the function is to operate on large (blocked or crinkled) matrices. If any function other than standard shift function is to be used then it must be formally declared before convg is declared.

AUTHOR

Gary Ross and A. P. Reeves

SEE ALSO

lshift(2), crshift(2)

NAME

crshift,crrotate — shift a large crinkled array on a parallel computer

SYNOPSIS

```
{$library lshift.pl }

function crshift( a:larray; r,c: integer ):larray;
        extern n, m, prow, pcol, labool;

function crrotate( a:larray; r,c: integer ):larray;
        extern n, m, prow, pcol, labool;
```

TYPES

```
larray = parallel array [ 0..n, 0..m, 0..prow, 0..pcol ] of btype;
labool = parallel array [ 0..n, 0..m, 0..prow, 0..pcol ] of boolean;
```

where btype is any base type.

DESCRIPTION

Crshift is a large matrix shift function for arrays stored with the crinkled format for parallel processors, such as the MPP, which have a fixed range of parallelism for the two parallel dimensions. *Crrotate* is similar to crshift except that a rotate operation rather than a shift operation is performed. The second and third arguments to crshift specify the amount the matrix is to be shifted in each dimension in a similar way to the standard shift function.

The large matrix is stored in a four dimensional array structure; the last two dimensions of this array specify the block size which can be directly processed in parallel by the hardware. The range of the first dimension of this array specifies the number of blocks in each column of the large matrix and the second dimension specifies the number of blocks in each row. Therefore, the dimensions of the large matrix are ((n + 1) * (prow + 1)) by ((m + 1) * (pcol + 1)).

With the crinkled storage scheme, adjacent elements of the large matrix are stored in different blocks; single element shifting is slightly more efficient with this format when compared to storing adjacent elements in blocks. The crinkled storage scheme is illustrated with the following example. Consider that we have a large matrix ,mx, which is conceptually specified as follows:

```
mx: array[0..x,0..y] of btype;
```

and which is stored in the array

```
a: larray;
```

the mapping of element i,j of the large matrix into the array a is specified by

```
mx[i,j] = a[i mod cd1, j mod cd2, i div cd1, j div cd2]
```

where

```
cd1 = n + 1
cd2 = m + 1
```

NAME

crshiftg,crrotateg — shift a large crinkled array on a parallel computer

SYNOPSIS

```
{$library gshift.pl }

function crshiftg( a:larray; r,c: integer ):larray;
        extern nl, nh, ml, mh, prowl, prowh, pcoll, pcolh, labool;

function crrotateg( a:larray; r,c: integer ):larray;
        extern nl, nh, ml, mh, prowl, prowh, pcoll, pcolh, labool;
```

TYPES

```
larray = parallel array [ nl_nh, ml_mh, prowl_prowh, pcoll_pcolh ] of btype;
labool = parallel array [ nl_nh, ml_mh, prowl_prowh, pcoll_pcolh ] of boolean;
```

where btype is any base type.

DESCRIPTION

Crshiftg and *Crrotateg* are extended versions of the functions *Crshift* and *Crrotate*. With these extended functions it is possible to have values other than zero for the first index value of array index ranges.

Crshiftg is a large matrix shift function for parallel processors, such as the MPP, which have a fixed range of parallelism for the two parallel dimensions when the crinkled format is used for large matrix storage. *Crrotateg* is similar to crshiftg except that a rotate operation rather than a shift operation is performed. The second and third arguments to crshiftg specify the amount the matrix is to be shifted in each dimension in a similar way to the standard shift function.

The large matrix is stored in a four dimensional array structure; the last two dimensions of this array specify the block size which can be directly processed in parallel by the hardware. The range of the first dimension of this array specifies the number of blocks in each column of the large matrix and the second dimension specifies the number of blocks in each row. Therefore, the dimensions of the large matrix are ((nh-nl+1) * (prowh-prowl+1)) by ((mh-ml+1) * (pcolh-pcoll+1)).

With the crinkled storage scheme blocks do not contain adjacent large matrix elements. This method is slightly faster for single element shifts than storing adjacent elements in blocks. The crinkled storage format is illustrated with the following example. Consider that we have a large matrix ,mx, which is conceptually specified as follows:

```
mx: array[0_x,0_y] of btype;
```

and which is stored in the array

```
a: larray;
```

the mapping of element i,j of the large matrix into the array a is specified by

```
mx[i,j] = a[i mod cd1, j mod cd2, i div cd1, j div cd2]
```

where

```
cd1 = nh - nl + 1
cd2 = mh - ml + 1
```

NAME

iconv, rconv, bconv — matrix convolution functions

SYNOPSIS

```
{$library convn.pl }

function iconv4(matrix:itype; k:kernel ):itype; extern;
function rconv2(matrix:rtype; k:kernel ):rtype; extern;
function bconv5(matrix:btype; k:kernel ):btype; extern;
```

TYPES

itype = parallel array [0..x, 0..y] of integer;
rtype = parallel array [0..x, 0..y] of real;
btype = parallel array [0..x, 0..y] of boolean;
kernel = parallel array [0..size-1, 0..size-1] of (real, integer or boolean);

NOTES

1) The type of data specified by kernel must be compatible with the type specified for itype.
2) The function name specifies the size of the kernel and the type of matrices involved (see description).

DESCRIPTION

The functions iconv, rconv, and bconv are all convolution functions. Selection of the proper function is determined by the data type involved in the convolution. Iconv is used if integer values are used, rconv is used with real valued matrices and bconv is used with boolean matrices. In addition to specifying the type of matrices involved, the size of the convolution kernel must be included in the function name. Fucntions exist to do convolution using kernels of size 2x2 to size 5x5. For example to do an integer valued convolution with a kernel of size 4x4 the function iconv4 would be used.

AUTHOR

Gary Ross conv(2)

NAME

irotate,nrotate — Rotation matrix generators

SYNOPSIS

{$library rotation.pl}

procedure irotate (var rp:pli; var cp:pli; var msk:plb; var rf:plr; var cf:plr; ro,co,theta:btype; id1, id2:pli); extern mxrowl, mxrowh, mxcoll, mxcolh;

procedure nrotate (var rp:pli; var cp:pli; var msk:plb; ro,co,theta:btype; id1, id2:pli); extern mxrowl, mxrowh, mxcoll, mxcolh;

TYPES

plr = array [mxrowl..mxrowh,mxcoll..mxcolh] of btype;
pli = array [mxrowl..mxrowh,mxcoll..mxcolh] of itype;
plb = array [mxrowl..mxrowh,mxcoll..mxcolh] of boolean;
Where btype is any type and itype is an integer or subrange base type

EXTERN CONSTANTS

mxrowl = the smallest row number of the input matrix
mxrowh = the largest row number of the input matrix
mxcoll = the smallest column number of the input matrix
mxcolh = the largest column number of the input matrix

DESCRIPTION

Irotate generates permutation matrices for realizing a given rotation transformation. Irotate is a version of the rotation procedure to be used in conjunction with the interpolation procedures blint or cint (see blint(2) and cint(2)). The rotation is specified by the coordinates of its center, ro and co (row and column), and by its angle theta. It returns two matrices containing the truncated row and column coordinates for rotation of the original matrix, respectively rp and cp, and a mask msk whose false elements indicate the positions in the rotated matrix that will be filled with zeros. It also returns two matrices containing the differences between the original rotated point and its truncated value, both for the row and column coordinates, respectively rf and cf. Id1 and id2 are two index identifying matrices as created by twodid (see mat(2))

Nrotate is a version of the rotation procedure to be used in near neighbor operations. It returns the same arguments as *irotate* except for rf and cf.

AUTHOR

A. P. Reeves

SEE ALSO

mat(2), blint(2), cint(2)

NAME

lblint — Bilinear interpolation procedure for a large matrix

SYNOPSIS

{$library lblint.pl}

procedure lblint(ro,co: btype; rp,cp: LINT; rf,cf: LARRAY; msk: LBOOL; b,d: LINT; var lresult: LARRAY); extern n1,n,m1,m,NL,NH,ML,MH,nar;

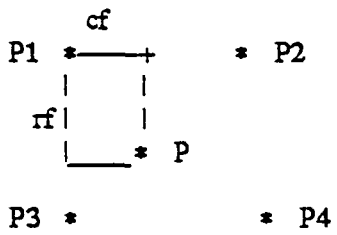
TYPES

LARRAY = array [NL..NH,ML..MH,PROWL..PROWH,PCOLL..PCOLH] of btype;
LINT = array [NL..NH,ML..MH,PROWL..PROWH,PCOLL..PCOLH] of itype;
LBOOL = array [NL..NH,ML..MH,PROWL..PROWH,PCOLL..PCOLH] of boolean;
nar = array [NL..NH,ML..MH] of boolean;
Where btype is any type and itype is an integer or subrange base type

DESCRIPTION

Lblint is a modified version of blint that can be used in conjunction with lirotate (see lrotation(2)). It performs a local search to find three out of four vertices of a square that contains the point whose value we want to interpolate. The fourth point is the top left corner of the square and is immediately defined from the matrices rp and cp (row and column coordinates) and matrix b (value of the point). Matrix d is a horizontally shifted version of matrix b. Matrices rf and cf contain the coefficients used to perform the interpolation.

The bilinear interpolation is performed in the following way:



$$P = (1 - cf)*(1 - rf)*P1 + (1 - rf)*cf*P2 + (1 - cf)*rf*P3 + cf*rf*P4$$

for all points P in the matrix.

The large matrices are stored in a four dimensional array. The last two dimensions of this array specify the block size which can be directly processed in parallel by the hardware. The range of the first dimension of this array specifies the number of blocks in each row of the large matrix and the second dimension specifies the number of blocks in each column. Therefore, the dimensions of the large matrix are
((NH - NL + 1)*(PROWL - PROWH + 1)) by ((MH - ML + 1)*(PCOLH - PCOLL + 1)).

The arguments n1, n, m1 and m specify also the dimensions of the large matrix. That is (n - n1 + 1) by (m - m1 + 1).

The function lrotateg ($library large.pl) has to be declared before lblint can be used.

AUTHOR

Cristina Moura

SEE ALSO

lrotation(2),blint(2),cint(2)

NAME

lcint — Cubic interpolation procedure for a large matrix

SYNOPSIS

{$library lcint.pl}

procedure lcint(ro,co: btype; rp,cp: LINT; rf,cf: LARRAY; msk: LBOOL; b,d: LINT; var lresult: LARRAY); extern mxrowl, mxrowh, mxcoll, mxcolh, NL, NH, ML, MH, nar;

TYPES

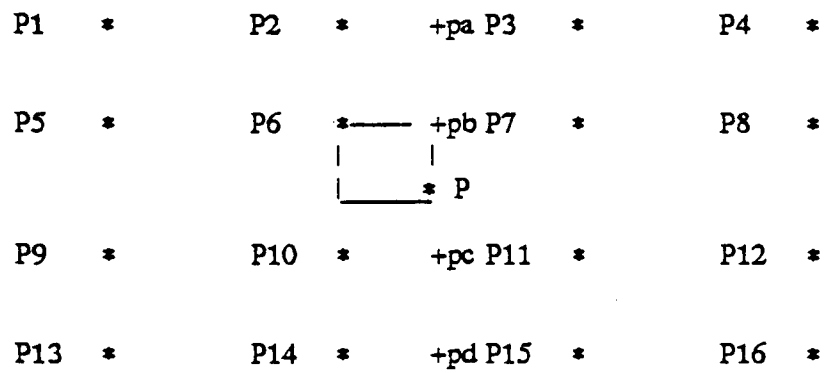
LARRAY = array [NL..NH,ML..MH,PROWL..PROWH,PCOLL..PCOLH] of btype;
LINT = array [NL..NH,ML..MH,PROWL..PROWH,PCOLL..PCOLH] of itype;
LINT = array [NL..NH,ML..MH,PROWL..PROWH,PCOLL..PCOLH] of boolean;
nar = array [NL..NH,ML..MH] of boolean;

Where btype is any type and itype is an integer or subrange base type

DESCRIPTION

Lcint is a modified version of cint (see cint(2)) that can be used in conjunction with lirotate (see lrotation(2)). It performs a local search to find fifteen out of sixteen points located around the point whose value we want to interpolate. The sixteenth point is immediately defined from the matrices rp and cp (row and column coordinates) and matrix b (value of the point). Matrix d is a horizontally shifted version of matrix b. Matrices rf and cf contain the coefficients used to perform the interpolation.

The cubic interpolation for sixteen points is done by, first, performing a cubicinterpolation for each one of the four rows and, then, based on the points obtained, performing a fifth cubic interpolation.



We already have the value and position of point P6 through matrices rp,cp and b. The first four cubic interpolations are performed to obtain points pa, pb, pc and pd. The fifth one yields the value of point P.

The large matrices are stored in a four dimensional array. The last two dimensions of this array specify the block size which can be directly processed in parallel by the hardware. The range of the first dimension of this array specifies the number of blocks in each row of the large matrix and the second dimension specifies the number of blocks in each column. Therefore, the dimensions of the large matrix are
((NH - NL + 1)*(PROWL - PROWH + 1)) by ((MH - ML + 1)*(PCOLH - PCOLL + 1)).

The arguments mxrowl,mxrowh, mxcoll and mxcolh specify also the dimensions of the large matrix. That is (mxrowh - mxrowl + 1) by (mxcolh - mxcoll + 1).

The function lrotateg ($library large.pl) has to be declared before lcint is used.

AUTHOR

Cristina Moura

SEE ALSO

lrotation(2),blint(2),lblint(2),cint(2)

NAME

lirotate,lnrotate — Rotation matrix generators for large matrices

SYNOPSIS

```
{$library lrotation.pl}

procedure lirotate (var rp:LINT; var cp:LINT; var msk:LBOOL; var rf:LARRAY;
var cf:LARRAY; ro, co, theta:btype; id1, id2:LINT);
extern NL, NH, ML, MH, PROWL, PROWH, PCOLL, PCOLH;

procedure lnrotate (var rp:LINT; var cp:LINT; var msk:LBOOL; ro, co, theta:btype;
id1, id2:LINT);
extern NL, NH, ML, MH, PROWL, PROWH, PCOLL, PCOLH;
```

TYPES

```
LARRAY = array [NL..NH,ML..MH,PROWL..PROWH,PCOLL..PCOLH] of btype;
LINT = array [NL..NH,ML..MH,PROWL..PROWH,PCOLL..PCOLH] of itype;
LBOOL = array [NL..NH,ML..MH,PROWL..PROWH,PCOLL..PCOLH] of boolean;
```

Where btype is any type and itype is an integer or subrange base type

DESCRIPTION

Lirotate is a large matrix version of irotate for interpolation (see rotation(2)).

It returns two matrices, rp and cp, containing the truncated, rf and cf, row and column coordinates for rotation of the original matrix, as well as two matrices, rf and cf, containing the differences between the original rotated points and their truncated values.

Lnrotate is a large matrix version of nrotate for near neighbor (see rotation(2)).

It returns two matrices, rp and cp, containing the row and column coordinates for rotation of the original matrix, rounded to its near neighbor coordinates.

For both lintrotat and lnearotat the large matrix is stored in a four dimensional array structure; the last two dimensions of this array specify the block size which can be directly processed in parallel by the hardware. The range of the first dimension of this array specifies the number of blocks in each row of the large matrix and the second dimension specifies the number of blocks in each column. Therefore, the dimensions of the large matrix are ((NH - NL + 1) * (PROWH - PROWL + 1)) by ((MH - ML + 1) * (PCOLH - PCOLH + 1)).

SEE ALSO

mat(2),rotation(2)

NAME

lperm2 - permute data in a large array on a parallel computer

SYNOPSIS

```
{$library perm2.pl }

function lperm2( mx: larray; r,c: lint; msk: lbool ): larray;
        extern nl, nh, ml, mh, prowl, prowh, pcoll, pcolh, btype;
```

TYPES

```
larray = parallel array [ nl_nh, ml_mh, prowl_prowh, pcoll_pcolh ] of btype;
lint = parallel array [ nl_nh, ml_mh, prowl_prowh, pcoll_pcolh ] of integer;
lbool = parallel array [ nl_nh, ml_mh, prowl_prowh, pcoll_pcolh ] of boolean;
pi = parallel array [ prowl_prowh, pcoll_pcolh ] of integer;
```

where btype is any base type.

VARS

```
id1,id2:pi;
```

Id1 and id2 are globally declared index identifying variables as created by twodid (see mat(2)). These should be initialized before using lperm2.

DESCRIPTION

Lperm2 is a large matrix permutation function for parallel processors, such as the MPP, which has a fixed range of parallelism for the two parallel dimensions. It is similar in concept to perm2(2) but operates on large matrices stored in the blocked format. The second and third arguments to lperm2 specify the conceptual large matrix index coordinates for the data to be permuted. The fouth argument specifies the permutation conditions - true where permutation is possible and false where permutation is out of range. In the false condition, zero will be put in place of data in the resulting matrix.

The large matrix is, in fact, stored in a four dimensional array structure; the last two dimensions of this array specify the block size which can be directly processed in parallel by the hardware. The range of the first dimension of this array specifies the number of blocks in each column of the large matrix and the second dimension specifies the number of blocks in each row. Therefore, the dimensions of the large matrix are ((nh-nl+1) * (prowh-prowl+1)) by ((mh-ml+1) * (pcolh-pcoll+1)).

SEE ALSO

mat(2), perm2(2)

NAME

lshift,lrotate — shift a large array on a parallel computer

SYNOPSIS

```
{$library lshift.pl }

function lshift( a:larray; r,c: integer ):larray;
        extern n, m, prow, pcol, labool;

function lrotate( a:larray; r,c: integer ):larray;
        extern n, m, prow, pcol, labool;
```

TYPES

```
larray = parallel array [ 0..n, 0..m, 0..prow, 0..pcol ] of btype;
labool = parallel array [ 0..n, 0..m, 0..prow, 0..pcol ] of boolean;
```

where btype is any base type.

DESCRIPTION

Lshift is a large matrix shift function for parallel processors, such as the MPP, which have a fixed range of parallelism for the two parallel dimensions. *Lrotate* is similar to lshift except that a rotate operation rather than a shift operation is performed. The second and third arguments to lshift specify the amount the matrix is to be shifted in each dimension in a similar way to the standard shift function.

The large matrix is, in fact, stored in a four dimensional array structure; the last two dimensions of this array specify the block size which can be directly processed in parallel by the hardware. The range of the first dimension of this array specifies the number of blocks in each column of the large matrix and the second dimension specifies the number of blocks in each row. Therefore, the dimensions of the large matrix are ((n + 1) * (prow + 1)) by ((m + 1) * (pcol + 1)).

Each block of the array structure contains consecutive elements from the large matrix. For example, consider that we have a large matrix ,mx, which is conceptually specified as follows:

```
mx: array[0..x,0..y] of btype;
```

and which is stored in the array

```
a: larray;
```

the mapping of element i,j of the large matrix into the array a is specified by

```
mx[i,j] = a[i div cd1, j div cd2, i mod cd1, j mod cd2]
```

where

```
cd1 = prow + 1
cd2 = pcol + 1
```

SEE ALSO

lshiftg(2)

NAME

lshiftg,lrotateg — shift a large array on a parallel computer

SYNOPSIS

```
{$library gshift.pl }

function lshiftg( a:larray; r,c: integer ):larray;
        extern nl, nh, ml, mh, prowl, prowh, pcoll, pcolh, labool;

function lrotateg( a:larray; r,c: integer ):larray;
        extern nl, nh, ml, mh, prowl, prowh, pcoll, pcolh, labool;
```

TYPES

```
larray = parallel array [ nl_nh, ml_mh, prowl_prowh, pcoll_pcolh ] of btype;
labool = parallel array [ nl_nh, ml_mh, prowl_prowh, pcoll_pcolh ] of boolean;
```

where btype is any base type.

DESCRIPTION

Lshiftg and *lrotateg* are extended versions of the *lshift* and *lrotate* functions. For these extended functions the initial index values of the array ranges are no longer constrained to be zero.

Lshiftg is a large matrix shift function for parallel processors, such as the MPP, which have a fixed range of parallelism for the two parallel dimensions. *Lrotateg* is similar to lshiftg except that a rotate operation rather than a shift operation is performed. The second and third arguments to lshiftg specify the amount the matrix is to be shifted in each dimension in a similar way to the standard shift function.

The large matrix is, in fact, stored in a four dimensional array structure; the last two dimensions of this array specify the block size which can be directly processed in parallel by the hardware. The range of the first dimension of this array specifies the number of blocks in each column of the large matrix and the second dimension specifies the number of blocks in each row. Therefore, the dimensions of the large matrix are ((nh-nl+1) * (prowh-prowl+1)) by ((mh-ml+1) * (pcolh-pcoll+1)).

Each block of the array structure contains consecutive elements from the large matrix. For example, consider that we have a large matrix ,mx, which is conceptually specified as follows:

```
mx: array[0_x,0_y] of btype;
```

and which is stored in the array

```
a: larray;
```

the mapping of element i,j of the large matrix into the array a is specified by

```
mx[i,j] = a[i div cd1, j div cd2, i mod cd1, j mod cd2]
```

where

```
cd1 = prowh - prowl + 1
cd2 = pcolh - pcoll + 1
```

SEE ALSO

lshift(2)

NAME

matrand,randinit — matrix random number generator

SYNOPSIS

```
{$library matrand }

function matrand; extern
  dim,m,mm1,mrand,mbool,pai,rang,coef,ncoef,state,window;

procedure randinit; extern
  dim,m,mm1,mrand,mbool,pai,rang,coef,ncoef,state,window;
```

TYPES and VARS

```
const
  dim = 128; (* size of matrix or mpp dimensions *)
  m = 40;    (* number of bits in the shift register *)
  mm1 = 39;  (* m - 1 *)
  mrand = 14; (* number of bits in generated random number *)
type
  index = 1..dim;
  mbool = parallel array[index,index] of boolean;
  pai   = parallel array[index,index] of integer;
  rang = 0..mm1;
var
  coef:array[1..m] of rang; (* list of feedback bits *)
  ncoef:integer;(* number of feedback bits *)
  state: parallel array[rang,index,index] of boolean;
          (* array of shift registers *)
  window:rang; (* index of last random bit of shift register *)
```

DESCRIPTION

Matrand is a function that returns an array of pseudo-random integers. *Randinit* initializes the random number register used by matrand. This register contains a random seed for every element of the random matrix. Initial seed random numbers are generated serially by randinit.

All constants, types and variables used by matrand are passed in the extern statement so that multiple random number generators with different parameters may be used in the same program.

AUTHORS

B. Finnerty, A.P.Reeves

BUGS

Randinit calles srand for a single positive integer random number. Srand uses overflow integer arithmetic. It could be replaced with a conventional Pascal random number generator.

NAME

maxm — masked max reduction functions

SYNOPSIS

```
{ $library reduce.pl }

function maxm1( arg : vecr, mask: vec ): btype; extern;
function maxm2( arg : matr, mask: mat ): btype; extern;
function maxm3( arg : arr3r, mask: arr3 ): btype; extern;
function maxm4( arg : arr4r, mask: arr4 ): btype; extern;
function maxm5( arg : arr5r, mask: arr5 ): btype; extern;
```

TYPES

vec: a vector of type boolean
mat: a matrix of type boolean
arr3: a three dimensional array of type boolean
arr4: a four dimensional array of type boolean
arr5: a five dimensional array of type boolean
vecr: a vector of btype
matr: a matrix of btype
arr3r: a three dimensional array of btype
arr4r: a four dimensional array of btype
arr5r: a five dimensional array of btype
where btype is a numeric base type

DESCRIPTION

These functions reduce the first argument where there are true elements in the boolean mask second argument. All dimensions are automatically reduced and the final result is always a scalar value. The range of the dimensions of arg and mask must match. The following functions are available:

maxm1
: max reduction for vector arguments

maxm2
: max reduction for matrix arguments

maxm3
: max reduction for three dimensional arrays

maxm4
: max reduction for four dimensional arrays

maxm5
: max reduction for five dimensional arrays

NAME

minm — masked min reduction functions

SYNOPSIS

```
{ $library reduce.pl }

function minm1( arg : vecr, mask: vec ): btype; extern;
function minm2( arg : matr, mask: mat ): btype; extern;
function minm3( arg : arr3r, mask: arr3 ): btype; extern;
function minm4( arg : arr4r, mask: arr4 ): btype; extern;
function minm5( arg : arr5r, mask: arr5 ): btype; extern;
```

TYPES

vec: a vector of type boolean
mat: a matrix of type boolean
arr3: a three dimensional array of type boolean
arr4: a four dimensional array of type boolean
arr5: a five dimensional array of type boolean
vecr: a vector of btype
matr: a matrix of btype
arr3r: a three dimensional array of btype
arr4r: a four dimensional array of btype
arr5r: a five dimensional array of btype
where btype is a numeric base type

DESCRIPTION

These functions reduce the first argument where there are true elements in the boolean mask second argument. All dimensions are automatically reduced and the final result is always a scalar value. The range of the dimensions of arg and mask must match. The following functions are available:

minm1 min reduction for vector arguments

minm2 min reduction for matrix arguments

minm3 min reduction for three dimensional arrays

minm4 min reduction for four dimensional arrays

minm5 min reduction for five dimensional arrays

NAME

mperm2 — Modified two dimensional mapping procedure

SYNOPSIS

{$library mperm2.pl }

procedure mperm2(mx:pa;var perm2:pa;var perm3:pa; r:pi; c:pi; ro,co:btype);extern lo1, hi1, lo2, hi2;

TYPES

pa = array [lo1..hi1,lo2..hi2] of b1type;
pi = array [lo1..hi1,lo2..hi2] of itype;
Where btype and b1type are any type and itype is an integer or subrange base type.

VARS

id1,id2: pi;
Id1 and id2 are two global index identifying matrices as created by twodid (see mat(2)). These must be initialized before using mperm2.

DESCRIPTION

Mperm2 is a modified version of the two dimensional mapping function perm2 (see perm2(2)) which can implement any into mapping and produces also

```
perm2[i,j] := mx[r[i,j],c[i,j]];
r1[i,j] := r[i,j];
c1[i,j] := c[i,j] + e;
perm3[i,j] := mx[r1[i,j],c1[i,j]];
```

where e can be 1 or -1 depending on the values of co
(the column value of the point of rotation) and hi2.

That is, the r and c matrices contain the row and column indices, respectively, of where each element in perm2 is to be obtained from in mx and r1 and c1 the indices for perm3.

AUTHOR

Cristina Moura

SEE ALSO

mat(2),perm2(2)

NAME

mx — input values into a square matrix

SYNOPSIS

```
{$library mx.pl }

function mx3( v00, v01, v02,
              v10, v11, v12,
              v20, v21, v22 :btype) :rtype; extern;
```

TYPES

vij = The value to go into location (i,j)
rtype = parallel array [0..x, 0..x] of btype;

where btype is any base type;

DESCRIPTION

MX inputs the values of elements of a square matrix into the matrix. It can be used to input values into arrays of size 2x2 to 5x5 . There are four different mx functions, one for each size matrix. To get the appropriate function use mx# where # is the size of the array (Note: # = x+1 from the type definitions above).

AUTHOR

Gary Ross

NAME

nearand, nearor, andnn, ornn — near neighbor logical functions

SYNOPSIS

```
{$library nn.pl }

function nearand(matrix:mtype ; k3:ktype): mtype; extern;
function nearor(matrix:mtype; k3:ktype): mtype; extern;

function andnn(matrix:mtype; mask:dirset): mtype; extern;
function ornn(matrix:mtype; mask:dirset): mtype; extern;
```

TYPES

```
mtype = parallel array [lo1..hi1, lo2..hi2] of boolean;
ktype = parallel array [0..2, 0..2] of integer;

neighbors = (NW,N,NE,W,C,E,SW,S,SE);
dirset = set of neighbors;
```

DESCRIPTION

The near neighbor logical functions logically combine (AND or OR) each matrix element with its near neighbors according to the second argument. The set of neighbors to be combined is specified in two different ways.

In nearand and nearor the set of neighbors is specified by a 3x3 kernel. Those neighbors that are to be selected will have positive (or true) values in the corresponding position in the kerenel.

In andnn and ornn the set of neighbors is specified by a direction set. The directions are NW, N, NE, W, E, SW, S, SE corresponding to compass directions, as well as C indicating the central element.

AUTHOR

Gary Ross

NAME

nn,nnet,recur — BASE assembly language functions

SYNOPSIS

```
{$library base.pl }

function nn(d:direction, b:bplane):bplane; extern; (* edge false *)
function nnet(d:direction, b:bplane):bplane; extern; (* edge true *)

procedure recur(var r:bplane, b:bplane):bplane; extern;

procedure readbp(var bp:bplane); extern;
procedure writebp(bp: bplane); extern;
```

CONST

```
dim1 = (* first dimension of the bitplane matrix *)
dim2 = (* second dimension of the bitplane matrix *)
```

TYPES

```
bplane = parrallel array [1..dim1, 1..dim2] of boolean;
directon = set of 0..8;
```

VARS

```
terminated: boolean;
```

DESCRIPTION

These functions form the primitives for the BASE binary array processor (BAP) assembly language. The may also be used for general BAP operations and for implementing other BAP assembly languages.

Nn computes an OR function over the selected near neighbors of a bitplane (boolean matrix). The near neighbors are labeled as follows:

```
1 2 3
8 0 4
7 6 5
```

Nnet is similar to nn except that near neighbor values outside the edge of the matrix are considered to be true rather than false.

recur is used to implement recursive instructions. It compares the old and new values for each iteration of the instruction and sets terminated to true when they are identical. See the example section for the use of this function.

Readbp reads a bitplane matrix from a file. The data values in the file are either 0 or 1 which are converted to false and true respectively. *Writebp* writes a boolean bitplane to a file using the 0 1 format.

EXAMPLES

Examples are given below of the syntax structures which are valid for the BASE instruction set. Other BAP instruction sets may be emulated.

```
const NW = 1; N = 2; NE = 3; E = 4; SE = 5; S = 6; SW = 7; W = 8;
begin

(* Boolean *)
   r := {Boolean function of three variables};
```

```
(* Simple near neighbor *)
   begin
        t := nn([{direction list}], {bplane variable});
           {the bplane variable may be preceeded by not}
        r := {Boolean expression of two variables and t}
   end
   {In most cases a single statement involving nn may be used}

(* Recursive near neighbor *)
   repeat recur(r, {simple near neighbor expression}) until terminated;

(* Global feature extraction *)
   bor := any({bplane variable}, 1, 2);
   isum := sum(ord({bplane variable}), 1, 2);

(* Examples *)
   r := a;
   r := a and b or c and d;
   r := a <> b;

   r := nnet([W], a);
   r := a and nn([N,S,E,W], a);
   r := nn([1..8], a);

   repeat recur(r, r or nn([N], r)) until terminated;
   repeat recur(r, r and not nn([1..8], not r)) until terminated;

   bor := any(a, 1, 2);
end.
(* bitplane stacks may be defined as an array of bitplanes *)
var
        bpst: array[0..7] of bplane;
(*      bpst: array[0..7, 1..dim1, 1..dim2] of Boolean *)
(*    bpst[0] is the least significant bitplane      *)
```

AUTHOR

A.P.Reeves

NAME

perm2,perm2s — General two dimensional mapping function

SYNOPSIS

```
{$library perm2.pl }

function perm2(mx:pa; r:pi; c:pi):pa;extern lo1, hi1, lo2, hi2;

function perm2s(mx:pa; r:pi; c:pi):pa; extern lo1, hi1, lo2, hi2;
```

TYPES

pa = array [lo1..hi1,lo2..hi2] of btype;
pi = array [lo1..hi1,lo2..hi2] of itype;
Where btype is any type and itype is an integer or subrange base type.

VARS

```
id1,id2: pi;
```

Id1 and id2 are two global index identifying matrices as created by twodid (see mat(2)). These must be initialized before using perm2.

DESCRIPTION

Perm2 is a general purpose two dimensional mapping function which can implement any into mapping. It is designed for a parallel computer architecture and uses a heuristic approach to reduce the execution time.

Perm2s is a version of perm2 designed for serial computers. The transformation implemented by perm2 and perm2s is as follows:

```
perm2[i,j] := mx[r[i,j],c[i,j]];
```

That is, the r and c matrices contain the row and column indices, respectively, of where each element in perm2 is to be obtained from in mx.

AUTHOR

A. P. Reeves

SEE ALSO

mat(2)

NAME

prodm — masked prod reduction functions

SYNOPSIS

```
{ $library reduce.pl }

function prodm1( arg : vecr, mask: vec ): btype; extern;
function prodm2( arg : matr, mask: mat ): btype; extern;
function prodm3( arg : arr3r, mask: arr3 ): btype; extern;
function prodm4( arg : arr4r, mask: arr4 ): btype; extern;
function prodm5( arg : arr5r, mask: arr5 ): btype; extern;
```

TYPES

vec: a vector of type boolean
mat: a matrix of type boolean
arr3: a three dimensional array of type boolean
arr4: a four dimensional array of type boolean
arr5: a five dimensional array of type boolean
vecr: a vector of btype
matr: a matrix of btype
arr3r: a three dimensional array of btype
arr4r: a four dimensional array of btype
arr5r: a five dimensional array of btype
where btype is a numeric base type

DESCRIPTION

These functions reduce the first argument where there are true elements in the boolean mask second argument. All dimensions are automatically reduced and the final result is always a scalar value. The range of the dimensions of arg and mask must match. The following functions are available:

prodm1
: prod reduction for vector arguments

prodm2
: prod reduction for matrix arguments

prodm3
: prod reduction for three dimensional arrays

prodm4
: prod reduction for four dimensional arrays

prodm5
: prod reduction for five dimensional arrays

NAME

pyramid, bpyr— pyramid convolution functions

SYNOPSIS

```
{$library pyramid.pl }

function pyramid(image:gtype; weights:wtype1):rtype;
                extern nrows ncols type;
function bpyr(image:btype; weights:wtype2):btype;
                extern nrows ncols;

function pyrmskg( id1:itype; id2:itype):btype extern 0, nrows, 0, ncols;
function pyrgen( id1:itype; id2:itype; up:boolean; dim:integer):itype;
                extern 0, nrows, 0, ncols;
```

TYPES

gtype = parallel array [0..nrows, 0..ncols] of (real or integer);
wtype1 = parallel array [0..13] of (real or integer);
btype = parallel array [0..nrows, 0..ncols] of boolean;
itype = parallel array [0..nrows, 0..ncols] of integer;
wtype2 = parallel array [0..13] of boolean;

EXTERN CONSTANTS

nrows = The largest row number of the image matrix
ncols = The largest column number of the image matrix
type = The data type of the weights vector (i.e the word integer)

VARS

id1,id2,up1,dn1,up2,dn2: itype
pyrmsk: btype;
Id1 and *id2* are two global index identifying matrices as created by twodid (see mat(2)). These must be iniatailized first. The other matrices specify transformations for managing pyramid data; see the description section for their initialization.

DESCRIPTION

The functions *pyramid* and *bpyr* are convolution functions for pyramid structure images stored in a two dimensional matrix. The successive levels of the pyramid structure are stored in successive rows of the image matrix with the lowest level image (1 pixel image) being located at position [0,0] in the input matrix. *Pyramid* is designed for use with mnumeric data while *bpyr* is suitable for boolean data.

The pyramid operations require several constant matrices; these are declared globally for efficiency. The global variables are generated with the functions *twodid* (see mat(2)), *pyrmskg* which generates the boolean pyramid constraint mask, and *pyrgen* which generates all shift matrices used for moving up and down the pyramid. A typical program code for setting up these matrices is dhown below:

```
twodid( id1, id2);
pyrmsk := pyrmskg(id1, id2);
up1 := pyrgen (id1, id2, true, 1);
dn1 := pyrgen (id1, id2, false, 1);
up2 := pyrgen (id1, id2, true, 2);
dn2 := pyrgen (id1, id2, false, 2);
```

The weight vector for the pyramid convolution is organized as follows:

```
        w[0]    parent

    w[1]  w[2]  w[3]
    w[4]  w[5]  w[6]  same plane near neighbors
    w[7]  w[8]  w[9]

      w[10]  w[11]  children
      w[12]  w[13]
```

The *gather* library function may be used to do explicit pyramid manipulation. An upward shift of all levels of the pyramid can be achieved with:

```
pyr := gather (gather ( pyr, up1, 1), up2, 2);
```

A downward shift of all levels of the pyramid can be achieved with:

```
pyr := gather (gather ( pyr, dn1, 1), dn2, 2);
```

Horizontal shifting of all levels of the pyramid can be achieved with the library function *xshift* with the mask pyrmsk as follows:

```
pyr := xshift (pyr, x, y, pyrmsk);
```

SEE ALSO

xshift(2), conv(2), xconv(2), varshift(2),mat(2)

AUTHOR

Gary Ross and A. P. Reeves

NAME

spread, gather— variable shift functions

SYNOPSIS

```
{$library shift.pl }

function spread(orig:itype; shmask:mtype; dim:integer):itype;
                extern nrows, ncols;
function gather(orig:itype; shmask:mtype; dim:integer):itype;
                extern nrows, ncols;
```

TYPES

itype = parallel array [0..nrows, 0..ncols] of (integer or real);
mtype = parallel array [0..nrows, 0..ncols] of boolean;

EXTERN CONSTANTS

nrows = the highest row number of the input matrix
ncols = the highest column number of the input matrix

VARS

id1,id2: itype;
Gather accesses *id1* and *id2* which are two global index identifying matrices as created by twodid (see mat(2)). These must be initialized before using gather; gather does not change their values.

DESCRIPTION

The functions spread and gather are both two dimensional variable shift functions. Given a two dimensional input matrix, a shift mask and a direction these functions will shift each element of the input array an amount specified by the shift mask.

Shifting can only be done in one direction at a time, therefore the direction must be specified by the value of the parameter dim.

The difference between the two functions is in how the shift mask (shmask) specifies where to shift each element. For the function spread, the shift mask indicates how far the corresponding element in the input matrix should be moved. Both positive and negative values are allowed for the shift mask. For the function gather, the shift mask defines where the result at the corresponding location expects to get its value from (i.e. the row number or the column number, depending on the direction specified).

CAUTIONS

In the function spread it is possible for more than one element to be shifted to the same location. If this occurs the element that is shifted the farthest will be the final result. (Note the shift mask is converted to be all positive values and the shifts performed are actually rotates. Thus a shift value of -1 is actually shifted farther than a shift value of +1)

Just as likely in the function spread is that nothing is shifted to a given location in the result. In this instance the resulting value will be zero.

In the function gather, the shmask should consist of non-negative values only. A negative value in the shift mask indicates that the resulting value at the corresponding position in the result will be zero.

AUTHOR

Gary Ross

NAME

summ — masked sum reduction functions

SYNOPSIS

```
{ $library reduce.pl }

function summ1( arg : vecr, mask: vec ): btype; extern;
function summ2( arg : matr, mask: mat ): btype; extern;
function summ3( arg : arr3r, mask: arr3 ): btype; extern;
function summ4( arg : arr4r, mask: arr4 ): btype; extern;
function summ5( arg : arr5r, mask: arr5 ): btype; extern;
```

TYPES

vec: a vector of type boolean
mat: a matrix of type boolean
arr3: a three dimensional array of type boolean
arr4: a four dimensional array of type boolean
arr5: a five dimensional array of type boolean
vecr: a vector of btype
matr: a matrix of btype
arr3r: a three dimensional array of btype
arr4r: a four dimensional array of btype
arr5r: a five dimensional array of btype
where btype is a numeric base type

DESCRIPTION

These functions reduce the first argument where there are true elements in the boolean mask second argument. All dimensions are automatically reduced and the final result is always a scalar value. The range of the dimensions of arg and mask must match. The following functions are available:

summ1
: sum reduction for vector arguments

summ2
: sum reduction for matrix arguments

summ3
: sum reduction for three dimensional arrays

summ4
: sum reduction for four dimensional arrays

summ5
: sum reduction for five dimensional arrays

NAME

writemx,twodid,twodids — general matrix functions

SYNOPSIS

```
{$library mat.pl }

procedure writemx(mx:ptype;fmt:integer); extern lo1, hi1;

procedure twodid(var id1:ptype; var id2:ptype); extern lo1,hi1,lo2,hi2;
procedure twodids(var id1:ptype; var id2:ptype); extern lo1,hi1,lo2,hi2;
```

TYPES

ptype = array [lo1..hi1,lo2..hi2] of ntype;
Where ntype is a numeric base type: real, integer or subrange.

DESCRIPTION

Writemx is a convenient procedure for printing the contents of a small numeric matrix. The fmt parameter specifies the field width for each element to be printed. Writemx is not clever enough to know when the width limit on an output line has been exceeded.

Twodid generates two integer matrices id1 and id2 which can be used for identifying each element in a parallel array operation. The contents of these arrays are defined as follows:

id1[i,j] = i id2[i,j] = j

Twodids is a serial version of twodid.

AUTHOR

A. P. Reeves

NAME

xconv — constrained matrix convolution function

SYNOPSIS

```
{$library convolve.pl }

function xconv(image:gtype;k:kernel;mask:mtype):gtype;
                extern size;
```

TYPES

gtype = parallel array [0..x, 0..y] of (real or integer);
kernel= parallel array [0..size-1, 0..size-1] of (real or integer);
mtype = parallel array [0..x, 0..y] of boolean;

EXTERN CONSTANT

size : integer constant specifying the size of the kernel.
(i.e. for a 3 x 3 kernel size=3)

DESCRIPTION

Xconv is a version of matrix convolution where pseudo-edges are defined internally to the image matrix by use of a boolean mask matrix. The definition of edges internal to the image matrix prevents data from crossing the defined boundaries.

SEE ALSO

xshift(2), conv(2)

AUTHOR

Gary Ross

NAME

xshift — constrained shift function

SYNOPSIS

```
{$library shift }

function xshift(orig:itype; x,y:integer; mask:mtype):itype;
```

TYPES

itype = parallel array [0..x, 0..y] of (integer or real);
mtype = parallel array [0..x, 0..y] of boolean;

DESCRIPTION

Xshift is a two dimensional shift function where the boolean matrix mask defines pseudo-edges inside the array being shifted. The mask plane should be set to true only in those places where the user wishes to define an lower edge or a right edge. This definition introduces a non-symmetrical response in doing a shift. When shifting down or to the right the effect is as if an edge existed where defined in the mask. When shifting up the effect is the same as having an edge one row down from where it is defined in the mask. Also when shifting left, the effect is as if there were an edge one column to the right of where it is defined in the mask.

NOTE

It may be useful to think of the edge mask as defining an edge which is half a pixel down and half a pixel to the right of where it is defined in the edge mask.

AUTHOR

Gary Ross

LIBRARY FUNCTION SUBJECT INDEX