Dan Ames

NCC 2-387

72 P.

IN-16844

Report on the Feasibility of Hypercube Concurrent Processing Systems in Computational Fluid Dynamics

John Bruno

March 18, 1986

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS Technical Report 86.7

(NASA-TM-89398) REPORT ON THE FEASIBILITY OF HYPERCUBE CONCURRENT PROCESSING SYSTEMS IN COMPUTATIONAL FLUID DYNAMICS (NASA) 72 p
N86-29158
CSCL 20D
Unclas
G3/34 43341

Research Institute for Advanced Computer Science

Report on the Feasibility of Hypercube Concurrent Processing Systems in Computational Fluid Dynamics

John Bruno

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS Technical Report 86.7
March 18, 1986

We have studied the feasibility of using hypercube-connected concurrent processor systems for problems in Computational Fluid Dynamics. We considered both explicit and implicit numerical methods and we evaluated several alternative implementations of these methods on concurrent processor systems.

Work reported herein was supported in part by
Contract NAS2-11530 and by Cooperative Agreement NCC 2-387
from the National Aeronautics and Space Administration (NASA)
to the Universities Space Research Association (USRA).

TABLE OF CONTENTS

Report on the Feasibility of Hypercube Concurrent Processing Systems in Computational Fluid Dynamics

John Bruno

Research Insitute for Advanced Computer Science

March 18, 1986

1. Summary

Problem Statement

In this Task we evaluate the feasibility of using the hypercube-connected concurrent processor systems for problems in Computational Fluid Dynamics(CFD). We have found that concurrent processor systems can be a cost effective approach to CFD.

Accomplishments

We have designed and implemented a Lax-Wendroff explicit method for the Navier-Stokes equations. Our code runs on the Intel iPSC concurrent processor system. Tests of this code show that it is reasonably efficient. This is so because the computation dominates the communication as the size of the problem domain increases. On a 32 node hypercube we obtained a sustained processing rate (including the cost of communication) of over 475,000 floating point operations per second. Comparison of the identical code on a VAX 11/780 (one node) shows that the cube achieves a floating point operation rate that is about 15 times faster than the VAX.

We have also designed and implemented the Beam and Warming implicit factored method for Berger's equation. This code has been run on the Intel iPSC hypercube. Preliminary tests show that the efficiency of this code is poor. The reason for this stems from the fact that a substantial amount of communication is required throughout the computation. The efficiency would no doubt improve with the Navier-Stokes equations since the amount of computation relative to communication increases. Additionally, there is little chance to overlap the communication with the computation. Our implementation indicates that improvements in the communication architecture would improve the efficiency of our implementation.

On the Intel iPSC there is a large overhead for communication between nodes and that large numbers of small messages can seriously impact the efficiency of a computation[KO]. There are a number of issues concerning the Intel communication architecture.

1. There is no separate I/O processor in the node. The communication coprocessors in the Intel node are not programmable.

2. Too much data copying is required for I/O.

3. Processes in the same node cannot share address space. The above factors make it impossible to reduce the communication penalty by overlapping communication with computation.

The 512K bytes of main memory is inadequate for CFD. A version of the Intel iSPC is available with 4.5 Mbytes of main memory per node and this configuration would be significantly more useful in CFD experiments.

Projections

1. It is important to continue the evaluation of the CFD codes that have been developed especially on a hypercube with substantially more node memory. The Navier Stokes equations should be implemented using the Beam and Warming implicit factored method.

2. For the sake of completeness, a spectral method should be implemented. Much would be learned from this project.

3. Codes developed at Ames should be ported to other concurrent processor systems such as the JPL Mark III concurrent processor system. Since this system supports overlapped I/O, the efficiency of our implementations should improve.

4. A study into the feasibility of using the hypercube concurrent processor system for grid generation should be undertaken. Careful consideration should be given to graphical input and output. With the grid generation component in place, the previous codes should be expanded to include metric terms. This will help to improve efficiency.

5. Once the basic methods for CFD have been implemented and tested, zonal methods should be investigated.

2. Introduction

In this report we present the results of an investigation into the feasibility of using hypercube concurrent processor systems for problems in Computational Fluid Dynamics(CFD). Specifically, we are interested in the numerical simulation of the Navier-Stokes equations for situations in which few simplifying assumptions can be made and for high Reynolds number. If we insist on a *direct* simulation of the Navier-Stokes equations for any practical problem, an enormous number of grid points is required to resolve the wide range of scales of motion. An alternative to numerically resolving all scales of motion is to *model* those scales not resolvable by the computational grid in terms of the resolvable scales. This technique is referred to as Large Eddy Simulation(LES) and is also computationally intensive. In either case we are faced with problems whose computational demands in time and space are beyond the capacity of today's supercomputers. Thus we are motivated to consider multiprocessor architectures with the hope that they may be able to cope with the demands of CFD.

Fine-Grain Parallelism

When computational requirements outpace current technology, we often turn to (parallel) concurrent architectures. There has been a great deal of effort directed at speeding up single-processor systems using fine-grain parallelism. For example, pipeline techniques have been applied to instruction processing and to the construction of arithmetic units. Multiple memory banks are used to obtain adequate memory bandwidth to keep up with today's central processors. The supercomputers offered by manufacturers such as CRAY and CDC represent the ultimate in this direction.

An important feature of these systems is that programmers are shielded from the architectural complexities. The architectural features are usually hidden by the compiler technology which attempts to generate code that takes advantage of the architecture. For the most part, scientists have been able to maintain the abstraction provided by some programming language, such as FORTRAN. Often, better results are obtained if the high level code is written with the compiler optimizations in mind (such as writing DO loops which are easy for the compiler to "vectorize") and additionally, one could write low-level code which makes more effective use of the hardware. Many subroutine libraries have been created using this latter approach. However, the vast majority of the users of these systems are not overly concerned with their inherent complexities. This is not true for the following classes of concurrent computer systems.

Shared Memory

Multiprocessor systems with shared memory have been designed to help meet the challenge, but these systems are most often used to provide higher throughput rather than true parallel computation for a single instance of a problem. The reason for this is that we are not yet able to hide this kind of parallelism from the programmer. The programmer must explicitly design and implement algorithms which take advantage of the multiple processors since compiler technology has not been designed which can effectively shield the programmer form the hardware environment. (An important research effort in the area of automatic program restructuring is the Cedar Project at the University of Illinois. Its purpose is to provide software which will automatically detect instances where concurrent processors could be used effectively. Generally speaking, this type of compiler technology is unavailable and we are forced to design parallel algorithms which take explicit account of the concurrent hardware or find parallel implementations for known sequential algorithms.)

Shared-memory systems employ mechanisms which permit the processors to access shared memory. These mechanisms can be as simple as common bus or as complicated as a crossbar switch, and generally we expect to find something in between. Bus architectures suffer from too little bandwidth, for which they usually compensate with mechanisms such as processor caches. On the other hand, a full crossbar switch is prohibitive in hardware and expense. Other approaches trade interconnection complexity with ease and cost of construction.

Message Passing

Another important class of concurrent processor systems is the one in which there is no shared memory. The processors communicate with each other by sending messages(control and data) through a network. There is a wide range of possible configurations depending on processor and network complexity.

Regardless of the particular level of node complexity and the choice of interconnection network, the programmer is faced with the task of designing and implementing algorithms which take the system architecture into account. In general, the problem domain must be partitioned among the processors and the programmer must specify the details of the corresponding data communication.

In this report we study the feasibility of using concurrent processor systems with no shared memory for problems in CFD. We shall consider implementations of explicit and implicit numerical methods for the Navier-Stokes equations.

In the next section we describe some of the features of concurrent processor systems. After this we present the equations of interest, the Navier-Stokes equations, and give two numerical methods for their simulation. Finally we describe implementations of these methods on an Intel iPSC 32-node hypercube.

3. Processor

We define a concurrent processor system as a collection of *nodes*, where each node consists of a processor, arithmetic unit(s), local memory, and an *interconnection network* which provides the means through which nodes communicate with each other. Each node executes its own instruction stream and may have access to external storage media and I/O devices. Since there is no shared memory, the only way for processors to communicate with each other is by sending messages through the network. There are many kinds of interconnection networks and we breifly describe a few of the more important ones. Throughout th rest of the paper we will let N denote the number of nodes in the system. The nodes are numbered beginning with 0 and ranging up to $N-1$.

3.1. Interconnection Architectures

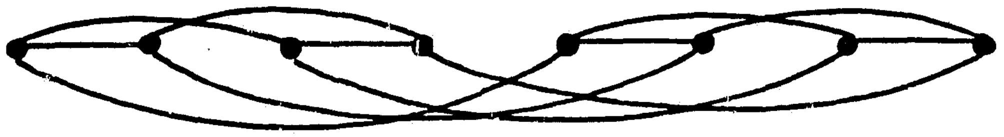
Ring: We say the nodes are connected in a ring if there is a communiation link from node i to node $i+1(mod N)$ for $i=0,1,\ldots,N-1$. The ring architecture has the advantage that it is very simple and obviously scalable†. Interestingly enough there are many problems for which the ring architecture is sufficient. Unfortunately, there are also many problems for which the ring architecture is not rich enough in the sense that the communication overhead caused by large distances between the nodes is more than we can tolerate. For example, imple-

† A design is scalable if it can be adjusted up or down in size without loss of efficiency or functionality.

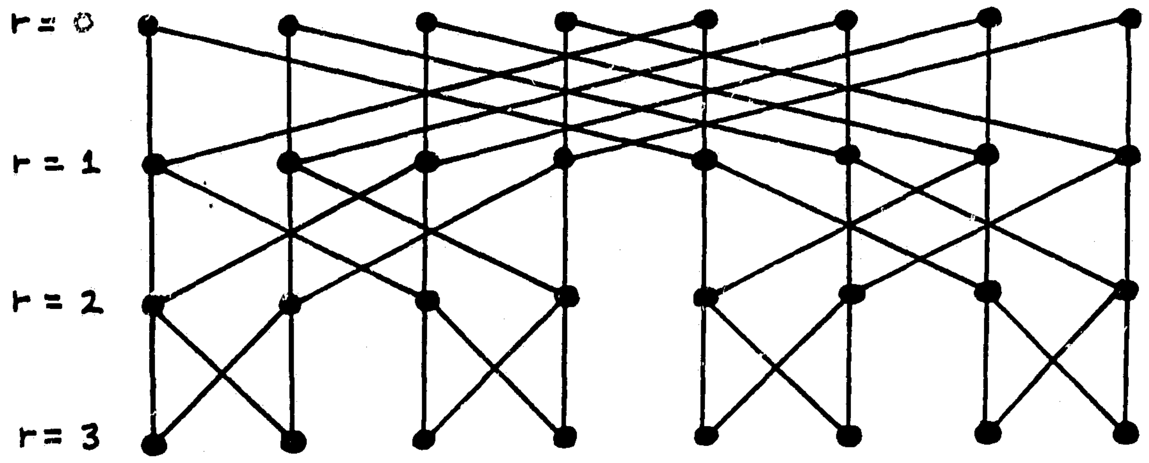
mentations of the FFT algorithm on a ring connected systems suffer from excessive communication overhead.

Mesh: The 2-dimensional mesh architecture represents an attempt to minimize the communication distance between the nodes. In this case we envision the nodes set out in a two-dimensional array with horizontal and vertical communication links between adjacent nodes.

Hypercube: Assume that $N = 2^k$. Let $bin(i,k)$ denote the k-bit binary representation of the integer i where $0 \leqslant i < 2^k$. The *hypercube* interconnection architecture provides a direct communication link between node i and node j if $bin(i,k)$ and $bin(j,k)$ differ in exactly one bit position. It follows that each node has a direct communication link with k other nodes and that the maximum number of communication links that is needed for any pair of nodes to communicate is k. For example, consider nodes 0 and $2^k - 1$. Clearly, $bin(0,k)$ and $bin(2^k - 1,k)$ differ in all n bit positions and therefore any message sent from node 0 to node $2^k - 1$ must pass through $k - 1$ intermediate nodes and use at least k different communication links. The hypercube network contains a total of $\frac{1}{2} N \log_2 N$ bidirectional communication links.


HYPERCUBE

Butterfly: The butterfly network is frequently associated with the FFT algorithm. A butterfly network consists of $k+1$ ranks of *network* nodes. We denote the i^{th} node on the r^{th} rank by p_{ri} for $0 \leqslant i < N$ and $0 \leqslant r \leqslant k$. Then node p_{ri} on rank $r > 0$ is connected to two nodes on rank $r-1$, the two nodes $p_{r-1\ j}$ such that either $j = i$ or the binary representation of j differs from i in only the r^{th} place from the left.



BUTTERFLY

Usually, the $r=0$ and the $r=k$ ranks are identified and correspond to the processor nodes and the remaining ranks contain network switching nodes.

The butterfly network is closely related to the hypercube network. If we coalesce all the nodes in the same column, then the network reduces to the hypercube network.

Shuffle-Exchange: The *exchange* interconnection consists of links between nodes i and $i+1$ if i is even. The *shuffle* interconnection provides a link from processor i to $2i$ $(mod\ \ N-1)$. As a special case, if $i=N-1$ we connect $N-1$ to itself.

3.2. Concurrent Processor Systems

In ths section we give a brief description of some of the concurrent processor systems which either have been implemented or are currently being designed or implemented.

Cosmic Cube

The "Cosmic Cube" is an experimental hypercube concurrent processor system built at The California Institute of Technology and consists of 64 nodes[Se85]. Each node contains Intel 8086 and 8087 co-processors and 136Kbytes of main memory.

Intel iPSC

Intel is marketing hypercube concurrent processor systems in 32, 64, or 128 node configurations. Each node contains Intel 80286/80287 co-processors and 512Kbytes of main memory. The communication links between nodes are bidirectional and have a transfer rate of 10 Mbits per second. We will have more to say about this system since it is available at NASA Ames and has been used in conjuntion with this report.

JPL Mark III

The hypercube research project at JPL is directed toward the design and implementation of a high-performance hypercube concurrent processor system[JPL85]. Each node contains Motorola 68020/68881 co-processors, a Motorola 68020 I/O processor and up to 4 Mbytes of main memory. The communication links are capable of transferring data at 100 Mbits per second. A 32-node prototype is scheduled for completion in February 1986 and a 256-node version will be available in January 1987. Beyond this, there are plans to include a Weitek scalar floating point unit (.25-2.0 Mflops) in each node and later to add a Weitek vector floating point unit (5-10Mflops). The long-range objective at JPL is to construct a 1024-node system which, when fully configured, would have a rated performance of 5-10 Gflops and a memory capacity of 4 Gbytes.

NCUBE

The NCUBE is a new machine with a proprietary CPU and small local memory. The current version can be configured with up to 1024 nodes each with a maximum of 128K bytes of main memory. The processors are interconnected in a hypercube configuration.

Lawrence Livermore Laboratory

The Parallel Processing Project at LLL is concentrating on shared memory configurations with vector processing nodes. They are studying the performance of various switch designs for processors which generate memory references typical of today's vector machines [BR185, BR 285].

The BBN Butterfly

The BBN Butterfly is a shared memory concurrent processor system with a Butterfly interconnection network. Each node consists of a M68020 processor, a M68881 floating point co-processor and up to 4 Mbytes of main memory. There is a separate processor in each node called the Processor Node Controller(PNC). The PNC initiates all messages transmitted over the butterfly switch and is involved in every memory reference made by the M68020. The PNC uses a memory management unit to translate the virtual addresses used by the M68020 into physical memory addresses. Physical addresses may correspond to locations in some other node and it is the responsibility of the PNC to initiate the data transfers through the switch. This translation is transparent to the M68020, and thus the PNC provides a shared memory view. The butterfly switch has a processor-to-processor bandwidth of 32 Mbits per second. A 128-node system has been built.

Los Alamos

Researchers at Los Alamos National Laboratory are designing a 1024 node concurrent processor system. The nodes will be capable of performing from 10 to 20 million non-pipelined floating point operations per second and are hypercube-connected. The nodes consist of two AMD 29325s, a NS32032, and at least 16Mbytes of main memory organized into 16 banks. Each node will also contain a disk with about 1/2 billion bytes of storage.

Princeton Navier-Stokes Machine

Daniel M. Nosenchuck and Michael G. Littman at Princeton University are developing a concurrent processor system(called NSC) to numerically simulate the full Navier-Stokes equations with no modeling. Each node has 8 Mwords of 32-bit interleaved memory and is capable of an average sustained computaion speed of 100 million floating point operations per second. The nodes are mesh connected and a 128 node prototype is under development.

3.3. The Intel iPSC

In this section we give a more detailed description of the architecture of the Intel iPSC concurrent processor system.

Cube:

The Intel iPSC concurrent processor system consists of either 32, 64, or 128 nodes. Each node has an Intel 80286 central processing unit and an Intel 80287 numeric processing unit supporting 32-, 64-, and 80-bit floating point operations(IEEE 754). Each node has 512 Kbytes of main memory(can be upgraded to 4.5 Mbytes) per node. The communications between nodes is over a 10 Mbit per second point-to-point serial channel(Intel's 82586 communication processor). Each node has 8 communication channels: seven for communicating with neighboring nodes and one for communicating with the cube manager.

Cube Manager:

The cube manager is an Intel 286/310 system which consists of an 80286 central processing unit, an 80287 numeric processing unit, and up to 4 Mbytes of main memory. It comes with either a 40 Mbyte winchester disk or a 320 Mbyte disk. There is a global ethernet channel for communicating with the cube nodes. An ethernet TCP/IP network subsytem is available which is used to provide access to the cube manager from other hosts on a LAN.

Software:

The cube manager comes with the XENIX 3.0 operating system, FORTRAN 77 compiler and a C compiler.

Intel provides a multiprocessing operating system which is resident in each of the nodes and provides the following services to each of the node processes:

System Call	Meaning
copen	Creates a channel for node process communication.
cclose	Destroys the communication channel created by *copen*.
send	Initiates transmission of a message to another process.
sendw	Initates transmission of a message to another process and returns only when the channel is available for reuse.
recv	Initiates the receipt of a message.
recvw	Initiates the receipt of a message and blocks until the message is received.
status	Informs the calling process of the availability of a channel.
probe	Determine if a message of the specified type has been received on a given channel.
mynode	Returns the node number of the calling process.
mypid	Returns the process id of the calling process.
cubedim	Returns the dimension of the cube.
clock	Returns the elapsed time(in milliseconds) since the node was initialized.
flick	Relinquishes the CPU.

Processes are downloaded into the nodes from the cube manager. We can load more than one process into each node and provide user-assigned process id's for each. Additionally, as part of loading processes into nodes, the user can optionally specify the maximum number of open channels and the maximum stack size per process.

Software for the cube is developed on the cube manager and then downloaded to the nodes. Typically, there is a cube manager process which communicates with the node processes during course of a computation. At the very least the cube manager process starts the computation by transmitting data to the nodes and collects data from the node processes a the end of a computation.

Communication Architecture:

The following is a list of observations concerning the communication architecture of the Intel system.

Little opportunity to overlap communication with computation. Each node has a single CPU which is required to set up all communication between nodes. This effectively eliminates much of the opportunity to overlap communication with computation. The extent to which the overlapping occurs is provided by the separate 82586 communication coprocessors. We would prefer a separate I/O processor which is capable of independently executing its own instruction stream.

Too much copying required. Communication often requires an inordinate amount of data moves. If the data to be transferred are not contiguous, then they have to be copied into contiguous locations before transfer. All data received are first placed in system buffers and then copied to contiguous locations in the user's space and, finally, if the data belong in non-contiguous locations the user has to copy the data once more. There should be some way in which constant stride data can be moved from one place to another without the intervention of the node processor. It would be useful to have a simple DMA device which could do memory-to-memory transfers.

Processes in the same node do not share address space. Processes within the same node should be able to share address space with each other. For example, if we were to have a separate communications processor, then the I/O process would most naturally be required to transfer problem data to and from the node. The most efficient way to accomplish this would be to allow the I/O process to share the address space with other processes within the same node. If it does not share address space then the only way for the communication process to pass data to other processes within the node is by message passing. This would defeat the advantage of using a separate I/O processor. Another reason to share address space within a node is to permit node processes to share code.

Performance:

The paper by Kolawa and Otto[KO] give a number of interesting performance results for the Intel iPSC. This paper determines the speed of the basic operations used in the Intel iPSC hypercube. Because of its pertinence to our report, we include the Kolawa and Otto paper as an appendix.

4. The Equations

The unsteady, three-dimensional Navier-Stokes equations in Cartesian coordinates (x, y, z, t) are taken as the basic set of equations[Lo82]. The Cartesian space represents both the physical domain and the the computational domain. It is known that the physical domain can be transformed into other curvilinear coordinates thus making it possible to treat a wide variety of geometries using one basic set of equations over a simple computational domain. These transformations introduce additional metric terms into the basic equations. We have chosen not to include these terms in order to simplify our presentation. Obviously, the elimination of the metric terms reduces the computational burden and so later in the report we will determine the effect of the metric terms on our performance estimates.

The three-dimensional Navier-Stokes equations are given by:

$$\frac{\partial Q}{\partial t} + \frac{\partial}{\partial x}(E - E_v) + \frac{\partial}{\partial y}(F - F_v) + \frac{\partial}{\partial z}(G - G_v) = 0 \qquad (4.1)$$

where

$$Q = [\rho \ \ \rho u \ \ \rho v \ \ \rho w \ \ e\,]^t \ ,$$

$$E = [\rho u \ \ \rho uu + p \ \ \rho vu \ \ \rho wu \ \ (e + p)u\,]^t \ , \qquad E_v = \frac{1}{Re}[0 \ \tau_{xx} \ \ \tau_{yx} \ \ \tau_{zx} \ \ \beta_x\,]^t \ ,$$

$$F = [\rho v \ \ \rho uv \ \ \rho vv + p \ \ \rho wv \ \ (e + p)v\,]^t \ , \qquad F_v = \frac{1}{Re}[0 \ \tau_{xy} \ \ \tau_{yy} \ \ \tau_{zy} \ \ \beta_y\,]^t \ ,$$

$$G = [\rho w \ \ \rho uw \ \ \rho vw \ \ \rho ww + p \ \ (e + p)w\,]^t \ , \qquad G_v = \frac{1}{Re}[0 \ \tau_{xz} \ \ \tau_{yz} \ \ \tau_{zz} \ \ \beta_z\,]^t \ ,$$

$$\tau_{xx} = \lambda\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}\right) + 2\mu\frac{\partial u}{\partial x} \ , \qquad \tau_{xy} = \tau_{yx} = \mu\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right) \ ,$$

$$\tau_{yy} = \lambda(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}) + 2\mu\frac{\partial v}{\partial y} , \qquad \tau_{xz} = \tau_{zx} = \mu(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}) ,$$

$$\tau_{zz} = \lambda(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}) + 2\mu\frac{\partial w}{\partial z} , \qquad \tau_{yz} = \tau_{zy} = \mu(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y}) ,$$

$$\beta_x = \frac{\gamma\kappa}{Pr}\frac{\partial e_I}{\partial x} + u\,\tau_{xx} + v\,\tau_{xy} + w\,\tau_{xz} ,$$

$$\beta_y = \frac{\gamma\kappa}{Pr}\frac{\partial e_I}{\partial y} + u\,\tau_{yx} + v\,\tau_{yy} + w\,\tau_{yz} ,$$

$$\beta_z = \frac{\gamma\kappa}{Pr}\frac{\partial e_I}{\partial z} + u\,\tau_{zx} + v\,\tau_{zy} + w\,\tau_{zz} ,$$

$$e_I = \frac{e}{\rho} - \frac{1}{2}(u^2 + v^2 + w^2) .$$

The velocity components u ,v , and w are made dimensionless by a_∞, the freestream speed of sound, the density ρ is made dimensionless by ρ_∞ and the total energy e by $\rho_\infty a_\infty^2$. The pressure p is given by $(\gamma-1)(e-0.5\rho(u^2+v^2+w^2))$ where γ is the ratio of specific heats. Also, κ is the coefficient of thermal conductivity, μ is the dynamic viscosity, and λ from Stokes' hypothesis is $-2/3\mu$. The Reynolds number is Re and the Prantl number is Pr .

5. Numerical Simulation Methods

In this section we describe two methods for numerically simulating the Navier-Stokes equations. The first method is an explicit procedure of the Lax-Wendroff type[Am69] and the second is the implicit factored method developed by Beam and Warming[BW78]. Before presenting these methods we provide some useful notation and definitions.

The *computational domain* **D** is the set of spatial points over which we attempt to numerically simulate the evolution of the dependent variables ρ, ρu, ρv, ρw, and e. The points in **D** are called *grid points* and they form a uniformly spaced "grid" over **D**. Let Δx Δy, and Δz denote the grid spacings in each to the coordinate directions, respectively. Let there be I grid points in the x-direction, J grid points in the y-direction, and K grid points in the z-direction.

Formally, we define **D** as

$$\mathbf{D} = \{ (x,y,z) \mid x = i\,\Delta x ,\ y = j\,\Delta y ,\ z = k\,\Delta z \ \textit{where}\ i,\ j, \text{ and } k \text{ are positive integers}$$
$$\textit{and}\ i \leqslant I,\ j \leqslant J, \text{ and } k \leqslant K \} .$$

The grid points in **D** are represented by triples of indices i, j, and k. That is, (i,j,k) denotes the grid point (x,y,z) with $x = i\,\Delta x$, $y = j\,\Delta y$, and $z = k\,\Delta z$.

The dependent variables are simulated over a discrete set of points uniformly spaced in time. The difference between successive simulation times is called the *time step* and is denoted by Δt.

The notation ρ_{ijk}^{n} refers to the quantity ρ at spatial location (x, y, z) and time t where $x = i\,\Delta x$, $y = j\,\Delta y$, $z = k\,\Delta z$, and $t = n\,\Delta t$. Extending this notation we have,

$$Q_{ijk}^{n} = [\rho_{ijk}^{n}\ \ \rho u_{ijk}^{n}\ \ \rho v_{ijk}^{n}\ \ \rho w_{ijk}^{n}\ \ e_{ijk}^{n}]^{t} \quad ,$$

and

$$E_{ijk}^{n} = E(Q_{ijk}^{n}) \quad ,$$

and so on.

5.1. Lax-Wendroff Explicit Method

The Lax-Wendroff method is an explicit procedure for computing Q_{ijk}^{n+1} from Q_{ijk}^{n} for all grid points (i, j, k). Part of this procedure involves computing intermediate values which are associated with the spatial indices $(i+1/2, j+1/2, k+1/2)$ and the time parameter $n+1/2$.

We introduce some notation for describing differencing and averaging processes. Let

$$\begin{aligned}
xface\,(i,j,k) &= \{(i,j,k), (i,j+1,k), (i,j,k+1), (i,j+1,k+1)\}\ , \\
yface\,(i,j,k) &= \{(i,j,k), (i+1,j,k), (i,j,k+1), (i+1,j,k+1)\}\ , \\
zface\,(i,j,k) &= \{(i,j,k), (i+1,j,k), (i,j+1,k), (i+1,j+1,k)\}\ ,
\end{aligned}$$

and

$$cube\,(i,j,k) = xface\,(i,j,k) \bigcup xface\,(i+1,j,k)\ .$$

If in some quantity, say ρ_{ijk}^n, we replace the triple i,j,k by a set of points, say $xface(i,j,k)$, then the notation denotes the *sum* of the values of ρ^n over all points in $xface(i,j,k)$. For example, if r denotes the triple (i,j,k) then

$$\rho_{xface(r)}^n = \sum_{s \, \epsilon \, xface(r)} \rho_s^n \; .$$

Finally, we define, by example, the numerical differencing operators δ_x, δ_y, and δ_z.

$$\delta_x \rho_{ijk}^n = \frac{\rho_{xface(i+1,j,k)}^n - \rho_{xface(i,j,k)}^n}{4\Delta x}$$

$$\delta_y F_{ijk}^n = \frac{F_{yface(i,j+1,k)}^n - F_{yface(i,j,k)}^n}{4\Delta y}$$

$$\delta_z (e_I)_{ijk}^n = \frac{(e_I)_{zface(i,j,k+1)}^n - (e_I)_{zface(i,j,k)}^n}{4\Delta z}$$

It should be clear from the above examples how the numerical difference operator works.

Let (i,j,k) denote a triple of indices (not necessarily integer valued). Then we define $(i,j,k)^+$ to be $(i+1/2, j+1/2, k+1/2)$ and $(i,j,k)^-$ to be $(i-1/2, j-1/2, k-1/2)$.

Finally, we define the sets **C** and **I** as

$$\mathbf{C} = \{(i,j,k)^+ \mid (i,j,k)\epsilon\mathbf{D}\} \bigcap \{(i,j,k)^- \mid (i,j,k)\epsilon\mathbf{D}\} \; ,$$

$$\mathbf{I} = \{r \mid r\,\epsilon\mathbf{D} \text{ and } r^+\epsilon\mathbf{C} \text{ and } r^-\epsilon\mathbf{C}\} \; ,$$

and

$$\mathbf{B} = \mathbf{D} - \mathbf{I} \quad .$$

The grid points in **B** are called *boundary* grid points, those in **I** are called *interior* grid points, and those in **C** are called *central* grid points.

Lax-Wendroff Method

Input: Q_r^n is given for all $r \in \mathbf{D}$ and some nonnegative integer n.
Output: Q_r^{n+1} for all $r \in \mathbf{D}$
Method:

Step 1: For all $r \in \mathbf{C}$ compute

$$Q_r^{n+1/2} = \frac{Q^n_{cube\,(r^-)}}{8} - \frac{\Delta t}{2}(\delta_x E^n_{r^-} + \delta_y F^n_{r^-} + \delta_z G^n_{r^-}) .$$

Step 2: For all $r \in \mathbf{C}$ compute

$$E_r^{n+1/2} = E\,(Q_r^{n+1/2}) ,$$

$$F_r^{n+1/2} = F\,(Q_r^{n+1/2}) \text{ , and}$$

$$G_r^{n+1/2} = G\,(Q_r^{n+1/2}) .$$

Step 3: For all $r \in \mathbf{C}$ compute $(E_v)_r^n$, $(F_v)_r^n$, and $(G_v)_r^n$. Some examples of these computations are:

$$(\tau_{xx})_r^n = \lambda(\delta_x u^n_{r^-} + \delta_y v^n_{r^-} + \delta_z w^n_{r^-}) + 2\mu\delta_x u^n_{r^-} ,$$

and

$$(\beta_x)_r^n = \frac{\gamma\kappa}{Pr}\delta_x (e_I)^n_{r^-} + \frac{1}{8}(u^n_{cube\,(r^-)}(\tau_{xx})_r^n + v^n_{cube\,(r^-)}(\tau_{xy})_r^n + w^n_{cube\,(r^-)}(\tau_{xz})_r^n) .$$

Step 4: For all $r \in \mathbf{I}$ compute

$$Q_r^{n+1} = Q_r^n - \Delta t\,(\delta_x (E^{n+1/2}_{r^-} - (E_v)^n_{r^-}) + \delta_y (F^{n+1/2}_{r^-} - (F_v)^n_{r^-}) + \delta_z (G^{n+1/2}_{r^-} - (G_v)^n_{r^-})) .$$

Step 5:

For all $r \in \mathbf{B}$ compute Q_r^{n+1} from Q_r^n according to the appropriate boundary conditions.

5.2. Beam and Warming Implicit Factored Method (Berger's Equation)

Implicit methods have been proposed for the numerical solution of various forms of the Navier-Stokes equations. Implicit methods are more complex than explicit methods since the former usually require the solution of a large number of systems of equations. However, implicit methods have improved stability properties over explicit methods thereby permitting a larger Δt . They have the drawback that they require significantly more computation than explicit methods.

The numerical method we shall consider is based on the work of Beam and Warming. The formulation of the method by Beam and Warming actually includes a number of different methods depending on the choice of certain parameters. We will not give the complete details of the method since they can be found in [BW78] and [Pu84].

As before, our objective is to determine Q^{n+1} given Q^k where $k \leqslant n$ (Notice that we admit the possibility that Q^{n+1} may depend on more than just the immediately preceding time-step). The temporal scheme for advancing time is given by

$$\Delta Q^n = \frac{\theta\,\Delta t}{1+\xi}\frac{\partial}{\partial t}\Delta Q^n + \frac{\Delta t}{1+\xi}\frac{\partial}{\partial t}Q^n + \frac{\xi}{1+\xi}\Delta Q^{n-1}$$
$$+ O\left[(\theta - 1/2 - \xi)\Delta t^2 + \Delta t^3\right],$$

where $Q^n = Q(n\,\Delta t)$ and $\Delta Q^n = Q^{n+1} - Q^n$. The choice of θ and ξ reproduces many two and three-level, explicit and implicit schemes.

The Navier-Stokes equations are solved for $\partial Q/\partial t$ and then substituted into the temporal scheme given above. This results in a nonlinear set of equations for ΔQ^n. A linear set of equations is obtained by the use of Taylor series expansions of various terms. For example, E^{n+1} is replaced by

$$E^{n+1} = E^n + \left(\frac{\partial E}{\partial Q}\right)^n (Q^{n+1} - Q^n) + O(\Delta t^2)$$

We have implemented the Beam and Warming implicit factored method for an equation called *Berger's* equation rather than the full Navier-Stokes equations. It was felt that the time spent in developing the code for the full Navier-Stokes equations would be excessive and that the basic issue concerning the performance of an implicit method could be resolved with the simpler equation. Furthermore, the implicit method for the Navier-Stokes equations leads to a large number of block tridiagonal equations whose simultaneous solution requires a large amount of intermediate storage which is not available on the Intel hypercube. Burger's equation has only a single dependent variable and the implicit method gives rise to a large number of (scalar) tridiagonal equations. Both of these factors mitigate the storage requirements and permit the use of reasonably sized domains.

From a performance evaluation point-of-view, it appears that nothing is lost by using Berger's equation. The reason is that the *computational* requirements for the Navier-Stokes equations are much greater than for Berger's equation but the *communication* requirements for Navier-Stokes are greater to a much lesser extent. This implies that efficient implementations for Berger's equation should be more difficult to find than for the Navier-Stokes equations.

Berger's equation is given by

$$\frac{\partial Q}{\partial t} + \frac{\partial}{\partial x}(E - E_v) + \frac{\partial}{\partial y}(F - F_v) + \frac{\partial}{\partial z}(G - G_v) = 0$$

where $Q = u$, and

$$E = u^2, \qquad E_v = \nu\frac{\partial u}{\partial x},$$

$$F = u^2, \qquad F_v = \nu\frac{\partial u}{\partial y},$$

$$G = u^2, \qquad G_v = \nu\frac{\partial u}{\partial z}.$$

Setting $\xi = 0$ and $\theta = 1/2$ in the temporal scheme yields the scheme used in our implementation:

$$\Delta Q^n = \frac{\Delta t}{2}\left(\frac{\partial Q^{n+1}}{\partial t} + \frac{\partial Q^n}{\partial t}\right).$$

After linearization and factorization the temporal scheme can be written as

$$L_x L_y L_z \Delta Q^n = H \tag{5.1}$$

where L_x , L_y , and L_z are *operators* given by

$$L_x = \{1 + \Delta t\,[(\frac{\partial}{\partial x})u^n - \frac{\nu}{2}\frac{\partial^2}{\partial x^2}]\} ,$$

$$L_y = \{1 + \Delta t\,[(\frac{\partial}{\partial y})u^n - \frac{\nu}{2}\frac{\partial^2}{\partial y^2}]\} ,$$

$$L_z = \{1 + \Delta t\,[(\frac{\partial}{\partial z})u^n - \frac{\nu}{2}\frac{\partial^2}{\partial z^2}]\} ,$$

and

$$H = -\Delta t\,[(\frac{\partial u^2}{\partial x} - \nu\frac{\partial^2 u}{\partial x^2}) + (\frac{\partial u^2}{\partial y} - \nu\frac{\partial^2 u}{\partial y^2}) + (\frac{\partial u^2}{\partial z} - \nu\frac{\partial^2 u}{\partial z^2})]$$

Equation (5.1) holds pointwise in the spatial coordinates and relates the dependent variable at the various time steps. The important point about the operators, L_x , L_y , and L_z is that they each involve spatial derivatives in a single coordinate direction.

Letting $X = L_y L_z \Delta Q^n$ and $Y = L_z \Delta Q^n$ we can rewrite equation (5.1) as a sequence of equations which corresponds to the actual implementation sequence.

$$L_x X = H , \tag{5.2a}$$

$$L_y Y = X \ , \tag{5.2b}$$

and

$$L_z \Delta Q^n = Y \ . \tag{5.2c}$$

The idea is to first solve equation (5.2a) for X and then we use X in (5.2b) and solve for Y. Finally, we use Y in (5.2c) and solve for ΔQ^n.

We obtain the basis for a numerical algorithm by approximating the spatial derivatives with finite-difference quotients. We assume a computational domain **D** as defined at the beginning of section 5. When we substitute finite difference quotients (three-point central-difference) for the spatial derivatives in equation (5.2a) we get a system of difference equations of the form

$$Cx_{i-1} X_{i-1} + Ax_i X_i + Bx_{i+1} X_{i+1} = H_i \ , \tag{5.3a}$$

where

$$Ax_i = 1 + \frac{\Delta t \, \nu}{\Delta x^2} \ ,$$

$$Bx_i = -\frac{\Delta t}{2\Delta x}\left(u_i + \frac{\nu}{\Delta x}\right) \ ,$$

and

$$Cx_i = \frac{\Delta t}{2\Delta x}\left(u_i - \frac{\nu}{\Delta x}\right) \ ,$$

for $1 < i < I$.

In equation (5.3a) we have suppressed the j and k indices. The dependent variable X and the coefficients are defined for each grid point and therefore we should write them as X_{ijk} , Ax_{ijk} , etc. However, we drop the j and k indices since we are assuming that the suppressed indices are identical throughout (5.3a). This will be the usual assumption for suppressed indices. Thus, according to (5.3a) we get one system of equations for each pair j , k corresponding to interior grid points.

We obtain similar results by approximating the spatial derivatives in (5.2b) and (5.2c) with finite-difference approximations, namely,

$$Cy_{j-1} Y_{j-1} + Ay_j Y_j + By_{j+1} Y_{j+1} = X_j \ , \tag{5.3b}$$

for each i , k corresponding to an interior grid point, and

$$Cz_{k-1} \Delta Q^n_{k-1} + Az_k \Delta Q^n_k + Bz_{k+1} \Delta Q^n_{k+1} = Y_k \ , \tag{5.3c}$$

for each i , j corresponding to an interior grid point.

Boundary conditions enter the picture when the terms in equation (5.3) depend on values associated with boundary points. Just as in the case of the explicit method, the boundary conditions are problem specific and therefore it is difficult to say anything general about them. Usually equations (5.3) result in a set of tridiagonal equations. However, if the boundary conditions are periodic in the x-direction then equations (5.3a) result in a periodic tridiagonal system of equations for which solution algorithms are available [Te75].

We show by example how the boundary conditions can affect the form of equations (5.3). Let $I = 7$. Then equation (5.3a) for a fixed j and k, in matrix notation, is

$$\begin{bmatrix} Cx_1 & Ax_2 & Bx_3 & 0 & 0 & 0 & 0 \\ 0 & Cx_2 & Ax_3 & Bx_4 & 0 & 0 & 0 \\ 0 & 0 & Cx_3 & Ax_4 & Bx_5 & 0 & 0 \\ 0 & 0 & 0 & Cx_4 & Ax_5 & Bx_6 & 0 \\ 0 & 0 & 0 & 0 & Cx_5 & Ax_6 & Bx_7 \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \\ X_7 \end{bmatrix} = \begin{bmatrix} H_2 \\ H_3 \\ H_4 \\ H_5 \\ H_6 \end{bmatrix} . \tag{5.4}$$

We have 5 equations and 7 unknowns and thus we need additional conditions to completely specify the system of equations. These additional conditions are obtained from the boundary conditions of the particular problem at hand. For example, if the boundary conditions are periodic in the x-direction, then $Q_1^n = Q_6^n$ and $Q_2^n = Q_7^n$. This implies that $X_1 = X_6$ and $X_2 = X_7$. Substituting into (5.4) we get the following periodic tridiagonal system of equations.

$$\begin{bmatrix} Ax_2 & Bx_3 & 0 & 0 & Cx_1 \\ Cx_2 & Ax_3 & Bx_4 & 0 & 0 \\ 0 & Cx_3 & Ax_4 & Bx_5 & 0 \\ 0 & 0 & Cx_4 & Ax_5 & Bx_6 \\ Bx_7 & 0 & 0 & Cx_5 & Ax_6 \end{bmatrix} \begin{bmatrix} X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \end{bmatrix} = \begin{bmatrix} H_2 \\ H_3 \\ H_4 \\ H_5 \\ H_6 \end{bmatrix} .$$

Next suppose that Q_1^n is fixed at some freestream value and $Q_7^n = Q_6^n$. It follows that $X_1 = 0$ and $X_7 = X_6$. Substituting into (5.4) we get the usual tridiagonal system of equations.

$$\begin{bmatrix} Ax_2 & Bx_3 & 0 & 0 & 0 \\ Cx_2 & Ax_3 & Bx_4 & 0 & 0 \\ 0 & Cx_3 & Ax_4 & Bx_5 & 0 \\ 0 & 0 & Cx_4 & Ax_5 & Bx_6 \\ 0 & 0 & 0 & Cx_5 & (Ax_6 + Bx_7) \end{bmatrix} \begin{bmatrix} X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \end{bmatrix} = \begin{bmatrix} H_2 \\ H_3 \\ H_4 \\ H_5 \\ H_6 \end{bmatrix} .$$

The above examples were intended to show how boundary conditions affect equations (5.3). Boundary conditions also come into play in computing, H, the right-hand-side of equation (5.3a). The approximation of the spatial derivatives in H in (5.1) with finite-difference quotients is affected by the boundary conditions since some of the terms in H contain spatial derivatives in each of the coordinate directions. Therefore, when the indices of H_{ijk} are adjacent to a boundary grid point, then the boundary conditions are taken into account.

We are now in a position to state the numerical algorithm.

Beam and Warming Implicit Factored Method

Input: Q_{ijk}^{n} for all $(i, j, k) \epsilon \mathbf{D}$.
Output: Q_{ijk}^{n+1} for all $(i, j, k) \epsilon \mathbf{D}$.
Method:

Step 1: (Compute X_{ijk} for all interior grid points) For each $1 < j < J$ and $1 < k < K$ solve equation (5.3a) for $1 < i < I$.

Step 2: (Compute Y_{ijk} for all interior grid points) For each $1 < i < I$ and $1 < k < K$ solve equation (5.3b) for $1 < j < J$.

Step 3: (Compute ΔQ_{ijk}^{n} for all interior grid points) For each $1 < i < I$ and $1 < j < J$ solve equation (5.3c) for $1 < k < K$.

Step 4: (Update the dependent variables) Set

$$Q_{ijk}^{n+1} = Q_{ijk}^{n} + \Delta Q_{ijk}^{n}$$

for all $(i, j, k) \epsilon \mathbf{I}$.

Step 5: Update all values of Q_{ijk}^{n+1} for $(i, j, k) \epsilon \mathbf{B}$.

6. Requirements for CFD

6.1. Storage Requirements for CFD

We characterize the storage requirements for CFD in terms of the size of the computational domain **D** and the amount of storage required per grid point. The amount of storage per grid point depends on a number of factors. The dependent variable Q occupies 5 floating point words per grid point for each time step and even with an explicit method it is sometimes convenient to store the value of Q^n while computing Q^{n+1}. If we have transformed the original problem from a physical domain into the computational domain, then there are additional metric terms which are associated with each grid point. Also, when using an implicit numerical scheme, intermediate results are usually generated by algorithms for solving block tridiagonal systems. For example, forward substitution increases the storage requirements by 30 additional floating point values per node. Whether we must store all these intermediate values simultaneously depends on the particular implementation strategy.

We define *gp_per_node* to be the number of grid points per node, *bytes_per_node* to be the amount of main memory per node devoted to storing the data, and *val_per_gp* to be the number of floating point values associated with each grid point. We assume that it take 4 bytes to store one floating point value. It is obvious that,

$$gp\_per\_node = \frac{bytes\_per\_node}{4\ val\_per\_node} .$$

6.2. Computational Requirements for CFD

In the following we develop some straight-forward relationships which give some insight into the factors which determine the computational requirements of CFD. As we shall see, an important measure of the capability of a concurrent processor system is given by the product of the number of nodes times the sustained floating point computation rate of each node. In the table below we define some of the important terms.

NAME	MEANING
N	Number of nodes.
$\|\mathbf{D}\|$	Number of grid points in the domain.
gp_per_node	Number of grid points per node(defined above).
gp_per_sec	Rate at which grid points are updated.
flop_per_gp	Number of floating point operations to update one grid point.
flop_per_sec	Sustained rate at which node can perform floating point operations.
sec_per_ts	The number of seconds to update all grid points in the domain. i.e., The number of seconds to advance the solution by one time step.

There are some obvious relationships among the above quantities, namely,

$$gp\_per\_node = |\mathbf{D}| \, / \, N \tag{1}$$

$$flop\_per\_sec = flop\_per\_gp \;\; gp\_per\_sec \tag{2}$$

$$sec\_per\_ts = gp\_per\_node \, / \, gp\_per\_sec \tag{3}$$

We use "node complexity" to refer to the computational capability of each node. For example, a node with a bit-serial CPU would have a very low node complexity while a node which consists of a CRAY CPU would have a high node complexity. The sustained floating point operation rate of a node is a reasonable measure of its complexity, and the product of the floating point operation rate of each node times the number of nodes is a useful figure of merit for a concurrent processor system.

Using the above equations we obtain the product of *flop_per_sec* and N. It is interesting that this product is determined by the domain size, the number of floating point operations per grid point(per time step), and the number of seconds allowed per time step. The quantities on the right-hand-side are measures of the computational demands of the problem and the left-hand-side is a measure of the computational capability of the concurrent processor system.

$$N \; flop\_per\_sec = \frac{|\mathbf{D}| \; flop\_per\_gp}{sec\_per\_ts}$$

For example, if we have that the number of floating point operations per grid point is 2K, and if we require that the number of seconds per time step not exceed one, and that the domain contains 256^3 grid points, then we get:

$$N \; flop\_per\_sec = 2^{35} \; .$$

Thus for a system with 1024 nodes, each node must be capable of a sustained processing rate of 32 million floating point operations per second. The *rated* performance of the proposed Los Alamos machine comes close to meeting this requirement. If we have more nodes(no machine has been proposed with more than 1024 nodes and significant floating point capabilities.) then a diminished floating point capability would suffice. On the other hand, if we want to do the job with an 8-node system, then each node would have to achieve a sustained rate of 4 billion floating point operations per second.

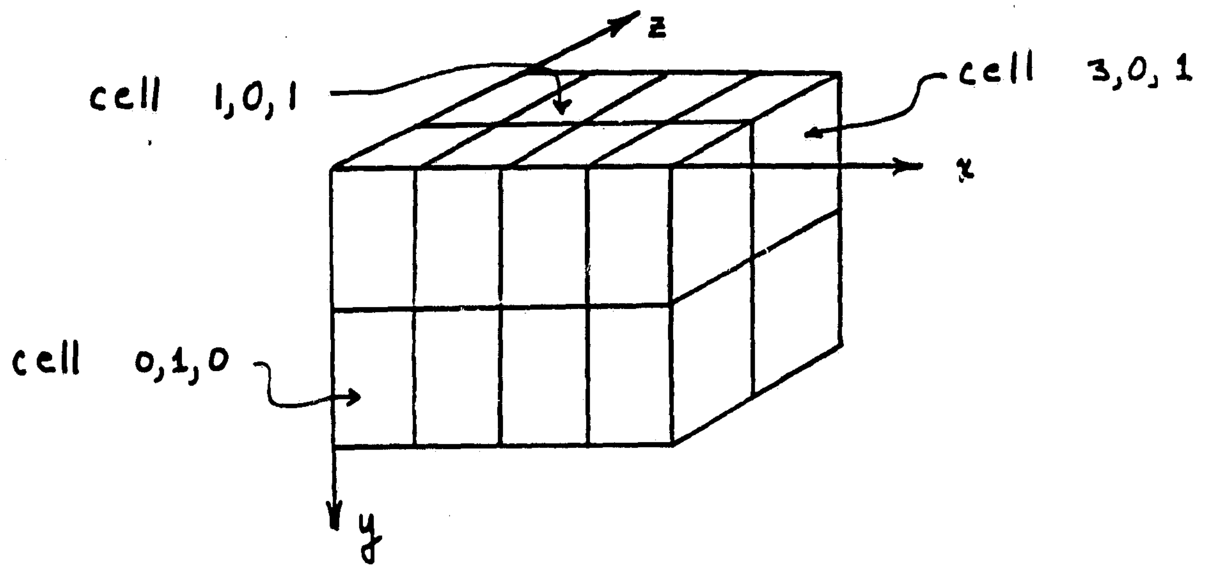
The Princeton machine will have 128 nodes and therefore each node would have to achieve a sustained rate of 256 million floating point operations per second. The predicted performance of each node is 100 million floating point operations per second.

7. Lax-Wendroff Method

7.1. Implementation

In this section we describe an implementation of the Lax-Wendroff method. The program was written in C and was run on a 32-node Intel hypercube.

The obvious way to implement the Lax-Wendroff method is to partition the computational domain **D** into subsets, which we call *cells*, and to assign each of these cells to a different node in the hypercube. Recall that the domain **D** has I grid points in the x-direction, J grid points in the y-direction, and K grid points in the z-direction. Each cell is a "box" of grid points with II grid points in the x-direction, JJ grid points in the y-direction, and KK grid points in the z-direction. The indices i, j, and k are used to refer to grid points within a cell and these indices range from 1 to II, JJ, and KK, respectively. Throughout we assume that I, J, K, and II, JJ, KK are powers of 2.



In the Figure we show a computational domain partitioned into 16 cells. Each cell is identified by the coordinates a , b , and c . In general, we have AA cells along the x-direction, BB cells along the y-direction, and CC cells along the z-direction. The cell coordinates range from 0 to $AA-1$, $BB-1$, and $CC-1$, respectively. It is easy to see that

$$AA = \frac{I}{II} , \quad BB = \frac{J}{JJ} ,$$

and

$$CC = \frac{K}{KK} .$$

We define $ABIT$, $BBIT$ and $CBIT$ as follows:

$$AA = 2^{ABIT} \;;\; BB = 2^{BBIT} \;;\; CC = 2^{CBIT}$$

In this implementation we assign each cell to a node according to the function *cell_to_node* which maps the cell coordinates into a node number. In preparation for defining this function we introduce the function $gray(r,s)$ which is defined for all integers r and s where $0 \leqslant r < 2^s$. The value of $gray(r,s)$ is a Gray code on s bits for the integer r. This function has the important property that $gray(r,s)$ and $gray(r+1 \; mod \; 2^{s},s)$ differ in exactly one bit. Finally,

$$cell\_to\_node(a,b.c) = gray(c,CBIT) \hat{\ } gray(b,BBIT) \hat{\ } gray(a,ABIT)$$

where ^ denotes concatenation. The importance of the *cell_to_node* mapping is that "adjacent" cells map into "adjacent" nodes. Nodes are *adjacent* if they have a direct communication link between them and cells are *adjacent* if they differ by one in exactly one coordinate. Clearly, adjacent cells will have to transfer data between themselves and so it is advantageous that they be mapped to adjacent nodes. Certain cells which are not adjacent are also required to exchange data, but these cases do not dominate and in the worst case the data passes through two intermediate nodes.

Since there are $AA*BB*CC$ nodes (there is a one-to-one correspondence between cells and nodes), the dimension of the hypercube must equal $ABIT + BBIT + CBIT$.

Each node must compute Q^{n+1} for all grid points in its cell. This evaluation uses the values of Q^n associated with grid points in other cells. It turns out that cell (a,b,c) will require certain values of Q^n from cells with indices $(a \pm 1, b \pm 1, c \pm 1)$. In our program the dependent variable Q is represented by five 3-dimensional arrays called d, du, dv, dw, and e, corresponding to ρ, ρu, ρv, ρw, and e, respectively. The arrays are dimensioned $II+2$ in the x-direction, $JJ+2$ in the y-direction, and $KK+2$ in the z-direction. As was mentioned earlier, the indices i, j, and k are used to refer to grid points within the cell and they range from 1 to II, 1 to JJ, and 1 to KK, respectively. These arrays are oversized to make room for values associated with grid points in "neighboring" cells. For example, if our reference cell is (a,b,c), then $d[0][1][KK+1]$ is the density associated with the grid point II, 1, 1 located in cell $(a-1,b,c+1)$. The values for variables such as $d[0][1][KK+1]$ in cell (a,b,c) are obtained by explicit communication with the node that contains the cell $(a-1,b,c+1)$.

The node program is the following:

```
init_data();
for( i = 1; i <= ITER; i++ ) {
        xfer_data();
        sweep();
}
```

The init_data() routine initializes the data arrays.

The xfer_data() routine specifies all communications between the nodes. The idea is to transfer into each node sufficient information from "neighboring" nodes so that each node can advance the solution to Q^{n+1} for all of its grid

points. Each node uses nonblocking receive system calls(recv) to establish buffers for the incoming data, then uses blocking send system calls(sendw) to transmit data to neighboring nodes, and finally waits until all the anticipated data arrives.

The sweep() routine performs the Lax-Wendroff computation for one time step. Sweep() corresponds to Steps 1 to 5 of the Lax-Wendroff method presented earlier. The implementation of this routine takes into account that the terms $Q_r^{n+1/2}$, $E_r^{n+1/2}$, $(E_v)_r^n$, etc. for $r \epsilon C$ are common to the evaluation of Q^{n+1} for eight different grid points. The grid points (i,j,k) corresponding to a fixed value of k, (called a $k-plane$) depend on terms evaluated at central grid points with $r=(i,j,k\pm1/2)$. Accordingly, we compute the values associated with central grid points for two successive k-planes of central grid points. This enables us to evaluate Q^{n+1} for all grid points on the k-plane in between the two k-planes of central grid points. After this we compute the next k-plane of central grid points, $r=(i,j,k+1+1/2)$ using the space occupied by the values associated with $r=(i,j,k-1/2)$. In this manner we "sweep" through the domain and at any time we need only store the values associated with central grid points associated with two k-planes.

7.2. Efficiency

We define the efficiency, ϵ_A , of an algorithm, A , as follows:

$$\epsilon_A = (t_1^{Opt} / N) / t_N^A ,$$

where N is the number of nodes, t_1^{Opt} is the time to solve the problem on one node using the best possible algorithm, and t_N^A is the time to solve the problem on N nodes using algorithm A . Clearly, t_1^{Opt} / N is the fastest time we could achieve using N nodes and so ϵ^A is not greater than 1.

We can estimate the theoretical efficiency of our implementation of the Lax-Wendroff method. Each cell contains $v(cell)$ grid points and $\partial(cell)$ grid points on the boundary of the cell. We estimate the values of t_1^{Opt} and t_N^A for one time step as follows:

$$t_1^{Opt} = N \;\; v(cell) t_{calc} \, d$$

and

$$t_N^A = v(cell) t_{calc} \, d + \partial(cell) t_{comm} \, c ,$$

where t_{calc} is the time for a floating point operation, d is the number of floating point operations required per grid point, t_{comm} is the time required to transfer one floating point number between adjacent nodes, and c is the number of floating point numbers that are transferred per boundary grid point. We are glossing over a few details which do not change the qualitative nature of this efficiency estimate. For example, not all floating point operations require the same amount time, and the number of floating point numbers that are transferred between cells depends on the *cells* , and not all transfers are between adjacent nodes.

Using the above equations and after some simplifications we get

$$\epsilon_A \geqslant 1 - \frac{\partial(cell)}{v(cell)} \frac{t_{comm}}{t_{calc}} \frac{c}{d} .$$

Next we estimate $\partial(cell)$ and $v(cell)$ in terms of II, JJ, and KK. The result, after simplification, is

$$\epsilon_A \geqslant 1 - 2\left(\frac{1}{II} + \frac{1}{JJ} + \frac{1}{KK}\right) \frac{t_{comm}}{t_{calc}} \frac{c}{d} .$$

It is easy to see that the efficiency grows as $1 - O(|\mathbf{D}|^{-1/3})$. Also, it is expected that $d \gg c$ and so this tends to improve the efficiency. In the case of the Intel cube the ratio t_{comm} / t_{calc} is 149 [KO], but this is offset by the other terms.

7.3. Performance

The code has been instrumented to count the total number of floating point operations performed and to determine the amount of time devoted to the computation and the communication. We worked with a cell size of $II = 8$, $JJ = 10$ and $KK = 10$. Therefore each cell contained 800 grid points and since there are 32 nodes, **D** contains 25,600 grid points. The total time for a run in which ITER was 10 took 309 seconds and each node performed 4,594,680 floating point operations. This amounts to a total of 475,936 floating point operations per second.

Out of the 309 seconds, 288 seconds were devoted to computation and 21 seconds were devoted to communication. A crude estimate for the efficiency is:

$$\epsilon_{LAX} = \frac{288}{21 + 288} = 0.932 \ .$$

The same code, specialized to a single node, was run on a VAX11/780 and we found that the VAX maintained a sustained rate of approximately 32,000 floating point operations per second. Thus the 32-node hypercube is almost 15 times faster than the VAX.

The Lax-Wendroff code has not been extensively tested and there is a problem with the way in which the C deals with NaNs generated by the 80287 coprocessor. The NaN problem is expected to be cleared up shortly. The next phase will be to test this code extensively for a variety of boundary conditions and domain sizes.

8. The Beam and Warming Implicit Factored Method

8.1. Implementation

We have implemented the Beam and Warming implicit factored method for an equation called Berger's equation rather than the full Navier-Stokes equation. It was felt that the time spent in developing the code for the full Navier-Stokes equations would be excessive and that the basic issue concerning the performance of an implicit method could be resolved with the simpler equation. Furthermore, the implicit method for the Navier-Stokes equations leads to a large number of block tridiagonal equations whose simultaneous solution requires a large amount of intermediate storage which is not available on the Intel hypercube. Burger's equation has only a single dependent variable and the implicit method gives rise to a large number of (scalar) tridiagonal equations. Both of these factors mitigate the storage requirements and permit the use of reasonably sized domains.

From a performance-evaluation point-of-view, it appears that nothing is lost by using Berger's equation. The reason is that the computational requirements for the Navier-Stokes equations are much greater than Berger's equation but the communication requirements for Navier-Stokes are greater to a much lesser extent. This implies that efficient implementations for Berger's equation should be more difficult to find than for the Navier-Stokes equations.

The following is Berger's equation:

$$\frac{\partial u}{\partial t} + \frac{\partial u^2}{\partial x} + \frac{\partial u^2}{\partial y} + \frac{\partial u^2}{\partial z} - \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right) = 0$$

The most important aspect of the implementation is the mapping of the domain **D** to the nodes. Consider Step 1 of the Beam and Warming method where we solve the equation

$$Cx_{i-1}X_{i-1} + Ax_i X_i + Bx_{i+1}X_{i+1} = H_i,$$

for all j and k. Remember that the j and k subscripts are suppressed and Cx_i and Bx_i depend on the value of u associated with grid point ijk.

We use Gaussian elimination to solve the tridiagonal systems. The algorithm consists of a *forward sweep* in which we eliminate variables and a *backward sweep* where we back-substitute to find the solution. The efficiency of our implementation of Gaussian elimination depends on the mapping of grid points to nodes. A desirable mapping would map all grid points with the same j, k coordinates into the the same node. If this were so, then the above equation, for a particular choice of j and k, could be solved completely within the node without communication with neighboring nodes, except for the case of H_i.

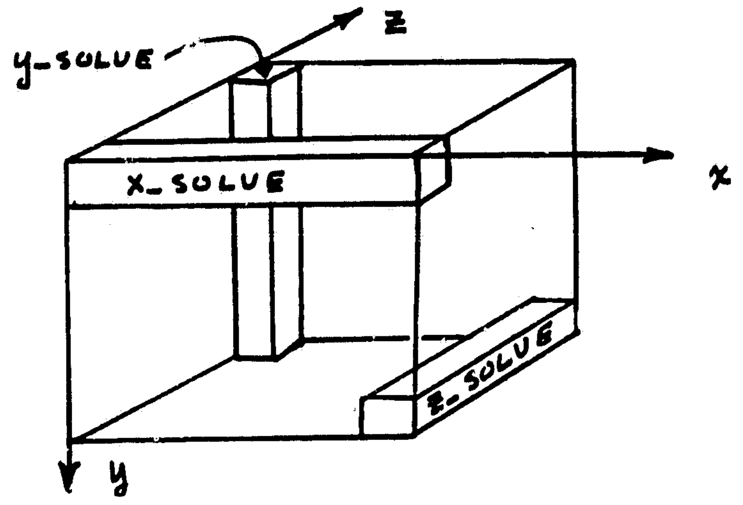
Next consider Step 2 of the Beam and Warming method where we solve the following equation,

$$Cy_{j-1} Y_{j-1} + Ay_j\, Y_j + By_{j+1} Y_{j+1} = X_j \ ,$$

for all i and k. In this case the i and k subscripts are suppressed and the X_js are determined by Step 1. A desirable mapping for Step 2 would map all grid points with the same i and k coordinates to the same node. If this were the case then the above equation, for a particular choice of i and k, could be solved completely within a node.

Finally, if we consider Step 3 of the Beam and Warming method we find yet another preferred mapping, one that maps grid points with the same i and j coordinates into the same node.

In the Figure we show the preferred mappings of the grid points into the nodes. *X_solve* corresponds to Step 1 and shows how the preferred mapping would map grid points lying along a line in the x-coordinate direction into the same node. If the whole domain were partitioned and mapped in this manner, then all the nodes could compute in parallel, each one solving the set of tridiagonal systems corresponding to the x-coordinate lines it contains. The Figure also shows the preferred mappings for Steps 2 and 3, indicated as *y_solve* and *z_solve* , respectively.



The preferred mappings for each of the three Steps are not compatible and because of the communication costs it seems undesirable to change the mapping in between Steps. We are motivated to look for mappings which can be maintained throughout the computation and which are efficient at each Step.

As before, we partition the computational domain **D** into *cells* and assign each cell to a node. However, we do *not* insist that each cell be mapped into a distinct node. Each cell is identified by its coordinates a , b , and c where,

$$0 \leqslant a < AA = \frac{I}{II}, \quad 0 \leqslant b < BB = \frac{J}{JJ}, \quad 0 \leqslant c < CC = \frac{K}{KK},$$

and

$$AA = 2^{ABIT}, \quad BB = 2^{BBIT}, \quad CC = 2^{CBIT} .$$

Before discusing the effect of various mappings, we describe the our implementation in terms of cells and *cell processes.* We assign a *process*, called a *cell process*, to each cell. This means that if a mapping assigns r cells to a particular node, then the node will contain r processes, one corresponding to each cell. Each cell process determines its own cell coordinates a, b, c, from its node identifier, its process identifier, and the *cell_to_node* mapping. It turns out that our implementation is relatively easy to specify in term of the *cell processes.*

Each *cell process* is given by:

```
init_data();
for(i = 1; i <= ITER; i++){
    xfer_data();
    x_solve();
    y_solve();
    z_solve();
}
```

The init_data() routine initializes the value of u in each cell.

The xfer_data() routine transfers the values of u between cells. Cell a,b,c needs values of u from cells $a \pm 1, b, c$, $a, b \pm 1, c$, and $a, b, c \pm 1$. The transmitted values are those which are required to evaluate H in equation (5.1).

The x_solve() routine consists of forward and backward sweeps along the x-direction for each value of j and k where $1 \leqslant j \leqslant JJ$ and $1 \leqslant k \leqslant KK$. In general, each cell process does only a portion of the forward and backward sweeps along the x-direction for each j and k. The forward and backward sweeps along a "line" in the x_direction will span AA different cells. Cell processes a, b, c with $a = 1$ can begin their forward sweeps immediately. When all the sweeps reach $i = II$, the intermediate results are sent to cell processes with coordinates $2, b, c$. In general, a cell process a, b, c with $a \neq 1$, must wait until it receives intermediate results from cell process $a-1, b, c$ before proceeding with its forward sweeps. If $a \neq AA-1$, then when all the forward sweeps reach $i = II$, intermediate results are sent to cell process $a+1, b, c$.

Similarly, when a cell process a, b, c with $a = AA-1$ finishes all its forward sweeps, it begins back substitutions for all j and k. When all the back substitutions reach $i = 1$, the intermediate results are sent to the cell process with coordinates $AA-2, b, c$. In general, all cell processes a, b, c with $a \neq AA-1$, after completing their forward sweeps, wait until they receive intermediate results from cell processes $a+1, b, c$ before they perform their portions of the back substitutions and send their intermediate results to cell process $a-1, b, c$.

Since each node contians more that one cell process, all waiting must be structured to relinquish the node CPU. Typically, a cell process waits for data to arrive on a channel. The wait code is:

```
while(status(channel))flick();
```

Note that cell processes finish sweeps for all j and k before sending intermediate results. Another alternative would be to send intermediate results for each j and k. This strategy leads to much more communication overhead and would be intolerable on the Intel iPSC. In a system with an efficient and independent I/O processor this method might be effective in eliminating all waiting for intermediate results.

8.2. Cell-To-Node Mappings

In this section we discuss the gross effect of the cell-to-node mapping on the efficiency of our implementation. A reasonably complete discussion of these issues can be found in [CSS85] and [JSS85] where various implementations of the Alternating Direction method are presented and analyzed for the two-dimensional case.

Suppose we have a 3-dimensional hypercube and $AA = BB = CC = 2$. In the Figure 8.1a we show an adjacency preserving 1-1 mapping of the cells into the nodes of the hypercube. It is easy to see that during the the course of the computation that at least half of the nodes will be idle at every point in time.

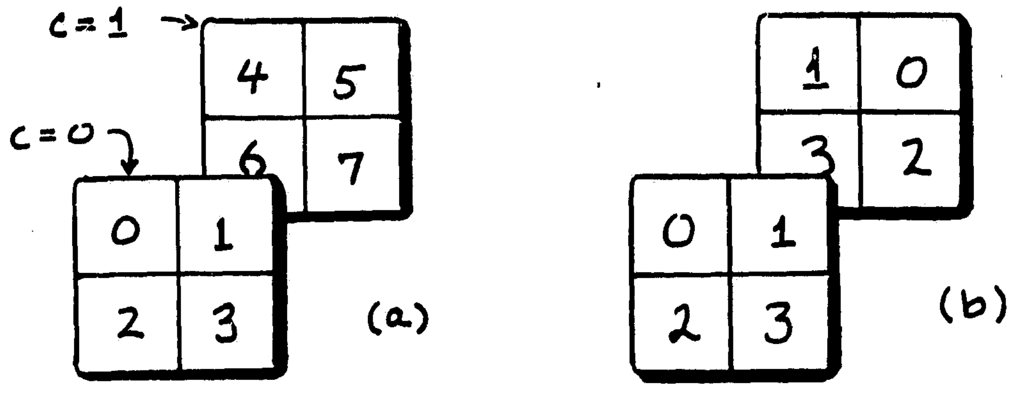


FIGURE 8.1

Next consider a 2-dimensional hypercube and a computational domain partitioned into 8 cells. The cell-to-node mapping in this case is shown in Figure 8.1b. Since each node contains a cell with $a = 1$, during the *x_solve* phase *all* nodes will have work to do and are initially active. Furthermore, all the nodes contain a cell with $a = 2$ so that all the nodes will remain active after the cells with $a = 1$ have finished their portion of the forward sweep. Of course, all the nodes will be idle during the time that the partial results from the forward sweep are transmitted from the cells with $a = 1$ to the cells with $a = 2$. The the situation is the same for the backward sweep.

The *y_solve* phase is not as favorable since only half of the nodes contain cells with $b = 1$. Therefore at least half of the nodes are idle throughout the *y_solve* phase.

The *z_solve* phase is similar to the *x_phase* , since all the nodes contain cells with $c = 1$ and $c = 2$.

These examples demonstrate the nature of the relationship between the cell-to-node mapping and the potential efficiency of our implementation. It follows that the best case would be to have every node contain exactly one cell for each different value for a , b , and c and have adjacent cells map into adjacent nodes or the same node. This would mean that in each "solve" phase, every node would be busy except for the time during the transmission of intermediate results to neighboring cells. We have not achieved this condition in the previous example since node 0 contains *two* cells with $b = 1$. It is easy to show that it is not possible to achieve such a mapping for any hypercube with dimension less than 6. It is also clear that the dimension of the hypercube would have to be even, say $2d$, and that $AA = BB = CC = 2^d$. It is not obvious whether such cell-to-node mappings exist.

We might relax the above conditions by requiring that each node contain *at least* one cell for each different value for a , b , and c and that adjacent cells map into adjacent nodes. We can always find cell-to-node mappings which satisfy this condition. An example is shown in Figure 8.2. The difficulty with such mappings is that additional communication and cell processes are required.

In this case cells which are adjacent in the z direction are mapped into nodes which are at a distance 2 from each other. All other adjacencies are preserved.

8.3. Performance

The code for the Beam and Warming method was run using the cell-to-node mapping shown in Figure 8.3 on a 16-node hypercube with 4 cell processes per node. We have $II = JJ = KK = 5$. This amounts to a domain with 20^3 grid points. The total number of floating point operations is approximately 214,000 and the time per iteration (x_solve, y_solve, and z_solve) is about 2.5 seconds. This is a sustained rate of 5.350 floating point operations per second per node. The same code, specialized to a single node, was run on a VAX11/780 and we obtained a sustained rate of approximately 32,000 floating point operations per second. This is the same rate obtained for the Lax-Wendroff code on the VAX.

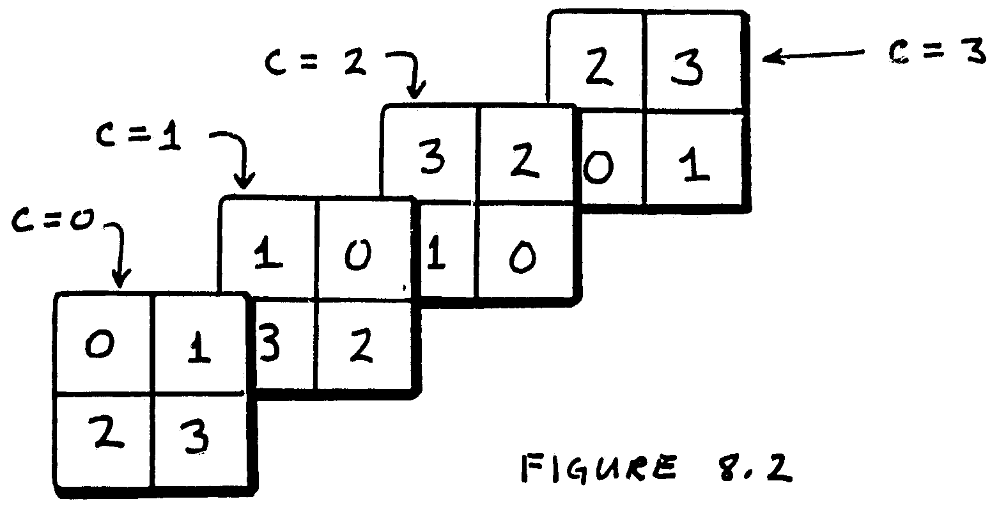


FIGURE 8.2

Another possibility is to relax the constraint that adjacent cells map into adjacent nodes. It can be shown that if we allow adjacent cells to map into nodes that are at most a distance 2 apart, then we can find a cell-to-node mapping in which each node contains exactly one cell for each value of a , b , and c . In this case it is also clear that the dimension of the hypercube must be even($2d$) and $AA = BB = CC = 2^d$. Such a mapping is shown in Figure 8.3. In this case cells which are adjacent in the z direction are mapped into nodes which are at a distance 2 from each other. All other adjacencies are preserved.

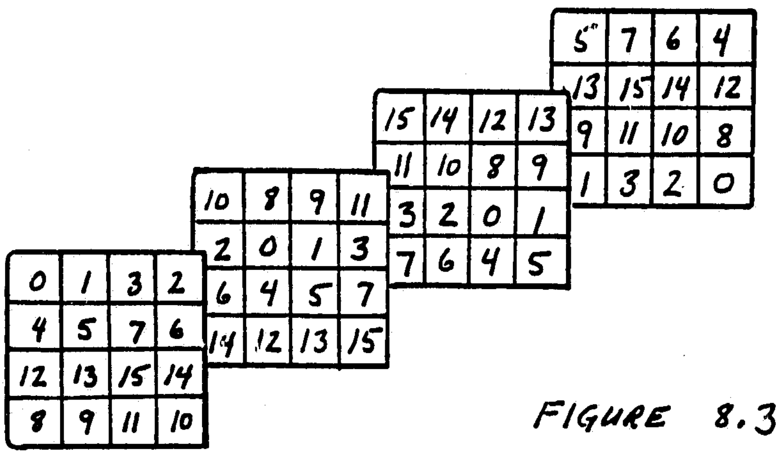


FIGURE 8.3

8.3. Performance

The code for the Beam and Warming method was run using the cell-to-node mapping shown in Figure 8.3 on a 16-node hypercube with 4 cell processes per node. We have $II = JJ = KK = 5$. This amounts to a domain with 20^3 grid points. The total number of floating point operations is approximately 214,000 and the time per iteration (x_solve, y_solve, and z_solve) is about 2.5 seconds. This is a sustained rate of 5,350 floating point operations per second per node. The same code, specialized to a single node, was run on a VAX11/780 and we obtained a sustained rate of approximately 32,000 floating point operations per second. This is the same rate obtained for the Lax-Wendroff code on the VAX. We conclude that a 32-node cube is about 5.3 times faster than a VAX on this code. The performance for this code is only 30% of the performance of the Lax-Wendroff code.

The hypercube performance is poor for the Beam and Warming code. One reason is that the tridiagonal systems do not require much floating point computation. Implementation of the Navier Stokes equations will result in systems of *block* tridiagonal equations which require significantly more floating point operations. This will tilt the balance away from communication and should result in improved performance.

More extensive tests of this code should be carried out for different cell-to-node mappings. These results should be compared to massive data rearrangement strategies.

9. References

[Am69]
W. F. Ames, *Numerical Methods for Partial Differential Equations*, Barnes and Noble, 1969.

[Br185]
Eugene D. Brooks III "A Butterfly Processor-Memory Interconnection for a Vector Processing Environment," UCRL-92325, preprint. Parallel Processing Project, Lawrence Livermore Laboratory, February 1985

[Br285]
Eugene D. Brooks III "The Share Memory Hypercube," UCRL-92479, preprint. Parallel Processing Project, Lawrence Livermore Laboratory, March 1985

[BW78]
R. M. Beam and R. F. Warming, "An Implicit Factored Scheme for the Compressible Navier-Stokes Equations", *AIAA*, vol. 16, no. 4, pp. 393-402, April 1978.

[CS85]
Tony Chan and Youcef Saad, "Multigrid Algorithms on the Hypercube Multiprocessor," Research Report, YALEU/DCS/RR-368, February 1985.

[CSS85]
Tony Chan, Youcef Saad, Martin Schultz, "Solving Elliptic Partial Differential Equations on the Hypercube Multiprocessor," Research Report, YALEU/DCS/RR-373, March 1985.

[Ho84]
M. Holt, *Numerical Methods in Fluid Dynamics*, Second Revised Edition, Springer-Verlag, 1984.

[JPL85]
Hypercube Research Project: Mark III Core Engineering Notebook, JPLD-2431, Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, 3 June 1985.

[JSS85]
S. Lennart Johnsson, Youcef Saad, and Martin H. Schultz, "Alternating Direction Methods on Multiprocessors," Research Report

YALEU/DCS/RR-382, October 1985.

[KO]A. Kolawa and S. Otto, "Performance of the Intel iPSC Hypercube," Cal Tech Report Hm205.

[Lo82]
H. Lomax and T. H. Pulliam, "A Fully Implicit, Factored Code for Computing Three-Dimensional Flows on the Illiac IV," in *Parallel Computations*, ed. G. Rodrigue, 1982.

[PT83]
R. Peyret and T. D. Taylor, *Computational Methods for Fluid Flow*, Springer-Verlag, 1983

[Pu84]
T. H. Pulliam, "Euler and Thin Layer Navier-Stokes Codes: ARC2D, ARC3D," Notes for Computational Fluid Dynamics User's Workshop, The University of Tennessee Space Institute, Tullahoma, Tennessee, March 12-16, 1984.

[SS85]
Youcef Saad and Martin H. Schultz, "Data Communication in Hypercubes," Research Report YALEU/DCS/RR-428, October 1985.

[Se85]
C. Seitz, "The Cosmic Cube," *Communications of the ACM*, V. 28, N. 1, pp. 22-33, January 1985

[Te75]
C. Temperton, "Algorithms for the Solution of Cyclic Tridiagonal Systems," *Journal of Computational Physics*, V. 19, pp. 317-323, 1975.

[Ul83]
J. D. Ullman *Computational Aspects of VLSI*, Computer Science Press, 1983.

10. Appendix

Hm205

Performance of the Intel IPSC Hypercube

A. Kolawa, S. Otto †
Physics Dept., Caltech, Pasadena CA 91125

October 8, 1985

Introduction:

The purpose of this note is to present the speeds of the fundamental operations used in the Intel Hypercube, IPSC. It is a companion to an earlier note Hm188 describing the timing of the Mark II hypercubes constructed at JPL.

Floating Point Speed

First off, we give the floating point performance (single precision, 32 bit) of a node. This was done by employing an accurate timing routine which runs independently in every node.

Multiply

The following code was timed :

```
float a,b,c;
for (i=0; i < NumTimes; ++i) {
      a = b*c;
}
```

The loop overhead was separately measured (see below) and subtracted.

Intel IPSC : 40.4 μsec/multiply or .024 Mflops

The same measurement done for a,b,c being double gives:

Intel IPSC : 43.5 μsec/multiply or .023 Mflops.

Add:

The code:

```
float a,b,c;
for (i=0; i < NumTimes; ++i) {
        a = b + c;
    }
```

loop overhead was again subtracted. We find:

Intel IPSC : 39.5 μs/add or .025 Mflops

For a,b,c double it is:

Intel IPSC : 43 μs/add or .023 Mflops.

Loop Overhead

Just the above loop was run and timed.

Intel IPSC : 8.2 μs/loop

For more complex, realistic expressions, the apparent floating point performance increases in realistic codes (e.g., lattice gauge). To illustrate this, we give a second measure of floating point speed.

+ Research supported by the Department of Energy grants DE-AS03-ER13118,DE-FG03-85ER25009 and by the Parsons Foundation and Systems Development Foundation. S.Otto holds a Bantrell Research Fellowship at Caltech.

Floating Point Performance #2

The code executed:

```
 float a, b, c, d, e;
 for (i = 0; i< NumTimes; ++i) {
     a = b*c + b*e + d*c;
 }
```

The time to execute this was:

Intel IPSC : 119.3 μs When a,b,c,d,e were double the execution time was:
Intel IPSC : 126.8 μs

Giving as the performance figure (A floating point operation is now considered as a "*" or "+".):

Intel IPSC : 23.86 μs / flop -> .042 Mflops

For double performance we got:

Intel IPSC : 25.36 μs/flop -> .039 Mflops

Integer (16 bit) performance

Multiply:

The code:

```
short j, k, l;
for (i = 0; i< NumTimes; ++i){
     j = k*l;
}
```

giving:

Intel IPSC : 4 μs / integer multiply

Add:

```
The code:      short j, k, l;
for (i = 0; i< NumTimes, ++i){
     j = k + l;
}
```

giving:

Intel IPSC : 2 μs / integer add

Internode Communications:

Single Packet:

The objective is to measure the speed of the fundamental communications routines, wtELT/rdELT. The Hypercube was mapped to a ring and each node along the ring transferred a single, 64 bit packet one step forward in the ring. This is the sort of thing which happens in many codes: each node is both sending and receiving data. The code executed was:

```
 int data [4];
for (i = 0; i< NumTimes; ++i){
      wtELT (data, forward chan);
      rdELT (data, backward chan);
}
```

The timings per single, 64 packet transfer are:

Intel IPSC : 11920 μs / single packet transfer

This gives us a "t_{comm}" for single precision arithmetic by dividing these times by two, since t_{comm} is conventionally defined as the transfer time of a 32 bit word. Note that t_{comm} is the time both to write and read a 32 bit word - this is what normally occurs - in homogeneous applications at least. (This definition is different from that in reference 1.)

Therefore, the "t_{comm}" appropriate to the usual efficiency analyses is:

$$t_{comm} = 5960\ \mu s \qquad \text{Intel IPSC}$$

We can also relate "t_{comm}" to "t_{flop}", defined as the time to do a single floating point operation (32 bit). "t_{flop}" has also been called "t_{calc}"; "t_{comp}" in other Hm memos. For the Intel IPSC machine, we have:

$$t_{comm} = 149 * t_{flop}.$$

t_{comm} for double precision work is achieved by doubling the above and all following t_{comm} estimates.

Global Communications ("recsig, sendsig")

The global broadcast utility, recsig, was timed. If N is the number of nodes in the Hypercube, the timings are of the form:

$$\alpha + \beta\,(\log N + 1)$$

where α reresents a constant startup time, β represents the communication time through each of the log N stages of the broadcast, and the +1 is there because the corner node must first read from the IH. Results are:

Dimension of Cube	Intel
1	9600μs
2	12000
3	17000
4	22000
5	27500

These timings do not take into account the operating system overhead, waiting, etc. .

The timings fit the theoretical form given in the above quite well; the parameters are:

	α	β
Intel IPSC	7000μs	2500μs

Block Transfer. "Shift"

We measured the block transfer of data between two neighboring nodes and compare result with MarkII(5MHz) Caltech Hypercube [2].

Number of Packets in the Shift	t_{comm} per packet Intel	t_{comm} per packet MarkII(5MHz)
1	5960μs	125μs
2	3007	93
4	1510	76
8	777	68
16	390	64
32	202	62
64	110	61
128	65	60
129	102	60
132	100	60
136	97	60
144	93	60
160	86	60
192	75	60
256	62.5	60
257	80	60
260	79.1	60
272	77	60
288	75.5	60
320	69	60
384	60	60
385	73	60
416	71	60
448	67	60
512	61.6	60

The above results are plot on figure 1. They show that when message length is equal to multiplicity of 1kbyte then communication time per packet is minimal. The next byte cause the jump of communication time which then starts to decrease until the length of the message is again the multiplicity of 1kbyte. This behavior is caused by the operating system which tends to sends messages in 1kbyte pieces. It seems that asymptotic communication time per 64 bit packet is 58 ~ 60 μs.

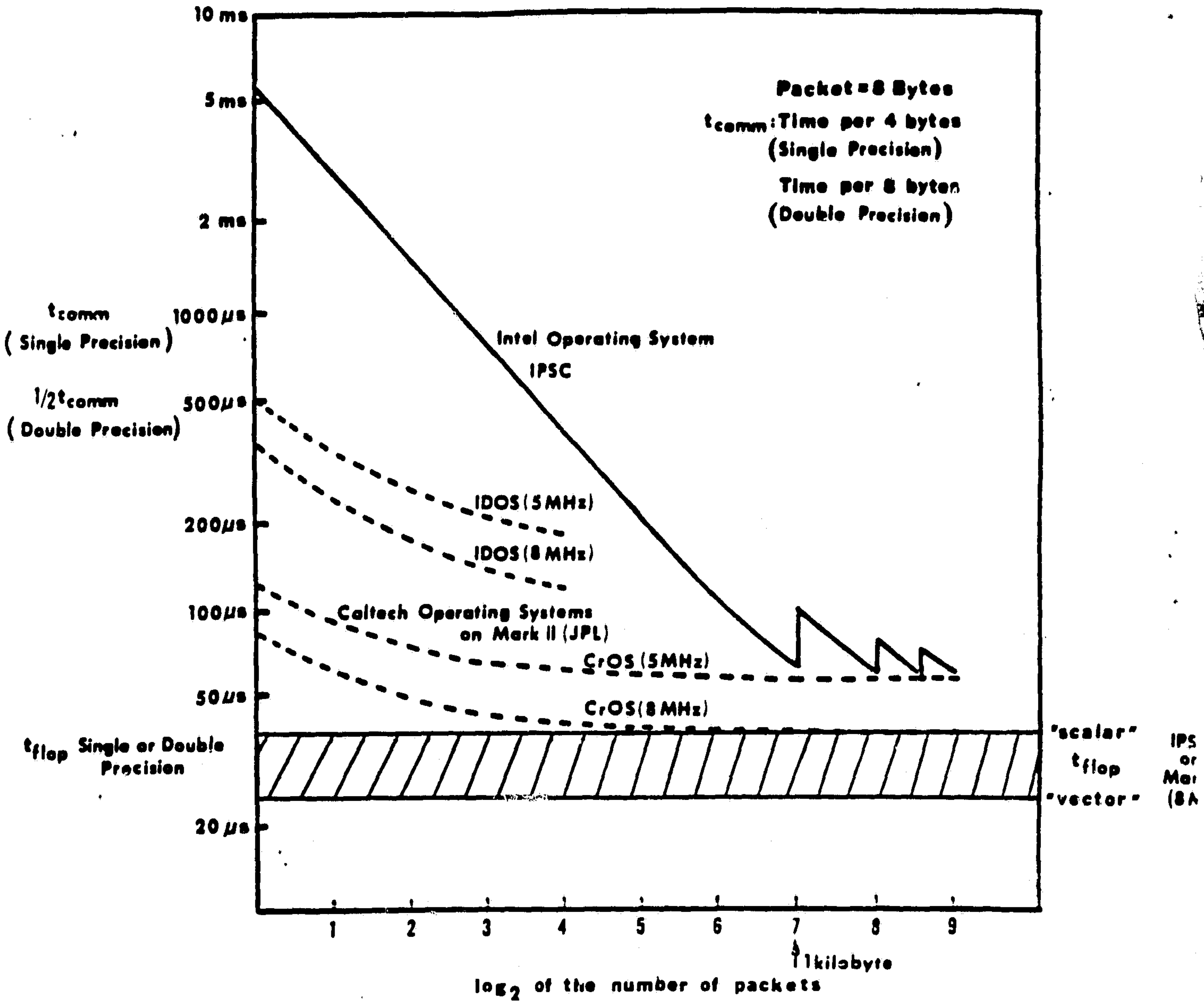
References:

1) "The Performance of the Caltech Hypercube in Scientific Calculations: A Preliminary Analysis:, G. C. Fox at Symposium of "Algorithms, Architectures, and the Future of Scientific Computation", Austin, March 1985

2) Performance of the Mark II Hypercube, S. Otto, A. Kolawa, A. Hey Caltech report Hm188

Figure Captions:

1) Plot of logarithm of communication time per packet vs logarithm of base 2 of number of 64 bits packets in the message. The solid line presents communication time per packet for Intel IPSC. The dashed lines represent communication time per packet for the Interrupt Driven Operating System (IDOS) and the Crystalline Operating System (CrOS) on Mark II 5MHz and 8MHz Caltech/JPL Hypercubes . The dashed area presents range of change of floating point performances on Intel IPSC and MarkII(8MHz) machines.The "scalar" is floating point multiply and "vector" is an operation like that in section Floating Point Performance #2.


10 ms
5 ms
2 ms
1000 μs
500 μs
200 μs
100 μs
50 μs
20 μs
t_{comm} (Single Precision)
1/2 t_{comm} (Double Precision)
t_{flop} Single or Double Precision
Packet = 8 Bytes
t_{comm}: Time per 4 bytes (Single Precision)
Time per 8 bytes (Double Precision)
Intel Operating System IPSC
IDOS (5 MHz)
IDOS (8 MHz)
Caltech Operating Systems on Mark II (JPL)
CrOS (5 MHz)
CrOS (8 MHz)
"scalar"
t_{flop}
"vector"
IPS
or
Ma
(8 M
1
2
3
4
5
6
7
8
9
1 kilobyte
$\log_2$ of the number of packets