IN-62-CR

217920

39P

Final Report on the Feasibility of Using the Massively Parallel Processor for Large Eddy Simulations and Other Computational Fluid Dynamics Applications

John Bruno

June 1984

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS TR 84.2

RIACS

Research Institute for Advanced Computer Science

Table of Contents

Page

Final Report on the Feasibility of Using the Massively Parallel Processor for Large Eddy Simulations and Other Computational Fluid Dynamics Applications

John Bruno

Research Institute for Advanced Computer Science
NASA Ames Research Center

1. Summary

This report presents the results of an investigation into the feasibility of using the MPP for direct and large eddy simulations of the Navier-Stokes equations. A major part of this study has been devoted to the "implementation" of two of the standard numerical algorithms for CFD. These implementations were not run on the Massively Parallel Processor(MPP) since the machine delivered to NASA Goddard does not have sufficient capacity. Instead we designed detailed implementation plans and from these we derived estimates of the time and space requirements of our algorithms on a suitably configured MPP. In addition we considered other issues related to the practical implementation of these algorithms on an MPP-like architecture, namely, adaptive grid generation, zonal boundary conditions, the table lookup problem, and the software interface.

Our performance estimates show that the architectural components of the MPP, the Staging Memory and the Array Unit, appear to be well suited to the numerical algorithms of CFD. This combined with the prospect of building a faster and larger MPP-like machine holds the promise of achieving sustained gigaflop rates that are required for the numerical simulations in CFD.

Below we summarize the major results and conclusions contained in this report.

- ***The MPP delivered to NASA Goddard is not adequately configured for handling direct or large eddy numerical simulations of the Navier-Stokes equations.***

The Processing Element(PE) local memory is 1K bits which is too small for the problems considered thus far. The current Staging Memory(SM) capacity is 2 Mbytes. This is barely adequate for a 32 by 32 by 128 computational domain using an explicit algorithm. Of course, much larger domains are routinely used in practice. Also, the implementation of the implicit method requires substantially more SM capacity or a high-speed channel to disk. Currently, the only available path to disk is through the host computer.

- ***As a result of the above we have concentrated our effort on evaluating the feasibility of applying the MPP's architectural components to problems in CFD.***

The major architectural components of the MPP are the Array Unit(AU) and the SM. The AU consists of a 128 by 128 array of PEs which responds to a single instruction stream. Each PE is a bit-serial processor with local memory. The SM is a large-capacity memory system which is capable of high-speed data transfers and data reformatting operations. Our goal is to assess the advantages and disadvantages of these architectural components in CFD applications.

● *We are encouraged by the performance estimates of the direct simulation of the Navier-Stokes equations using a fully explicit Lax-Wendroff method and the Beam and Warming implicit factored method.*

We have designed an implementation of a direct numerical simulation of the Navier-Stokes equations using a fully explicit Lax-Wendroff method. For a 32 by 32 by 128 computational domain we estimate that our implementation requires .272 seconds per time step and achieves a *sustained* floating-point operation rate of 243 megaflops.

We have also considered a 128^3 computational domain. In this case we estimate 3.26 seconds per time step and a sustained floating-point operation rate of 328 megaflops. Even though there is a 16-fold increase in the problem size in going to the 128^3 domain there is less than a 16-fold increase in the computation time. This is because there is no additional cost to do the boundary computations as the problem is scaled up to 128^3.

In the above we have assumed that the PE local memory size is at least 6400 bits, the SM has all 32 banks and has a capacity of at least 80 Mbytes, and the basic cycle time is 100 nanoseconds. This configuration could be achieved by upgrading the current machine.

The above estimates compare favorably to essentially the same algorithm which was coded and run on the CRAY XMP by Alan Wray of NASA Ames. Several runs of this code were made recently and it was found to take approximately one second of CPU time per time step for the 32 by 32 by 128 domain and 16 seconds per time step for the 128^3 domain. These runs used only one of the CRAY XMP processors and all arithmetic was done with 64 bits of precision.

We also considered the Beam and Warming implicit factored method and found that sustained megaflop rates comparable to the explicit case could be maintained when the "cross-section" of the computational domain is comparable to the PE array. The implementation we recommend requires a substantial amount of memory for intermediate results for the block tridiagonal solver. Here we have assumed the existence of either a second SM with 256Mbytes storage capacity, a 128Kbit local memory for each PE, or a high-speed channel to disk.

● *A variety issues arise when we look at whether an MPP-like machine can be successfully used in practical CFD applications and consequently there are several topics which deserve a thorough evaluation.*

In this study we have paid much attention to numerical algorithms and feel that the MPP holds promise in this area. (There are many numerical schemes we have not considered with the most notable exception being the spectral methods.) However, the numerical algorithms are not the whole story.

Adaptive Grids: Adaptive grid generation couples the grid system to the physical solution in order to dynamically alter the location of the grid points as the solution evolves. This is an area of intense research and holds the promise of being able to greatly enhance the effectiveness of numerical methods. It is important to consider the MPP architecture in light of the additional computations implied by the mathematics of adaptive grids.

Zonal Boundaries: Complex geometries often require that the physical space be partitioned into zones. This introduces new boundaries between zones called zonal boundaries and an additional set of boundary conditions. It is important to consider the effect of the computations at the zonal boundaries on the effectiveness of the MPP architecture.

Table Lookup: Equation of state computations often involve table lookup processes. Table lookup is not suited to the MPP architecture as it stands and so this represents an important area for further investigation. Such an investigation should look to new architectural structures and algorithms to combat this important problem.

Software Interface: All too often exotic hardware systems are constructed and delivered without adequate software to make the system usable. The MPP is an example of such a system which at this writing has no completely working high-level programming language. In fact, it is not totally clear that the provision of a high-level programming language will be sufficient to get scientists and engineers to use the machine productively. In the case of CFD we can envision a much more sophisticated software interface which responds to a nonprocedural language for specifying the physical problem and the appropriate numerical method(s) with all the translation to data layouts and codes being handled by the software. Such a comprehensive software environment is only feasible if we are willing to sufficiently restrict the domain of discourse and to make the software part of the total system design effort from the very beginning.

● *A bigger, faster, and more flexible MPP could be built.*

Discussions with the MPP architect and hardware engineers suggests that a faster MPP can be constructed. A ten times faster PE has already been designed. PE local memory capacity and SM capacity can also be increased. Additionally, a large-capacity high-bandwidth disk subsystem has been designed. In short there doesn't seem to be any reason why we couldn't build an MPP-like machine which is capable of sustained gigaflop performance.

● *Summary and Speculation.*

The MPP architecture seems suited to the numerical algorithms of CFD. The Array Unit with its 128 by 128 array of processing elements can be used effectively for both explicit and implicit numerical algorithms. Even more importantly, the Staging Memory has the flexibility to provide the different "views" of the problem data which are required by the numerical algorithms. There are other issues in practical CFD such as adaptive grid generation, zonal boundaries, etc., which deserve further study and there are additional numerical methods, such as the spectral methods, which should be measured against this architecture. It also appears that the capacity, speed and flexibility of the MPP architectural components can be substantially improved thereby offering the prospect of taking us into the gigaflop range for sustained computational rates.

Wind tunnels are systems which are designed to carry out a specific class of experiments. Even though a wind tunnel is a very expensive system to build, run, and maintain, no one seems too scandalized to learn that a wind tunnel cannot be used to accelerate electrons. It is tempting to speculate about designing a special-purpose hardware and software system to do CFD. It seems as though many of the numerical methods in CFD have matured to the point where we could select a limited but useful class of algorithms to implement. In doing so we enhance the chances of building a system which far outpaces anything that could be accomplished by a general-purpose system.

2. Introduction

In this report we present the results of an investigation into the feasibility of using the Massively Parallel Processor(MPP) 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 todays supercomputers. Thus we are motivated to consider the MPP with the hope that its (massively) parallel architecture may be able to cope with the demands of CFD.

The MPP is a high-speed digital processor designed to solve two-dimensional problems such as those encountered in the processing of satellite imagery. The MPP was designed and built by Goodyear Aerospace for NASA Goddard and was installed and accepted at Goddard in May, 1983. It encorporates two major architectural components, namely, an Array Unit(AU) array and a Staging Memory(SM). The AU is a 128 by 128 array of bit-serial Processing Elements(PE) which responds to a single instruction stream. Each PE has 1K bits of local memory and a basic cycle time of 100 nanoseconds. The SM is a large memory system which stores the problem data and is designed to transfer data to and from the AU and perform certain data reformatting tasks. The SM encorporates the idea of a multidimensional access memory[Batc77] in the designs of the input/output substagers and in the main stager. The reader who is not familiar with the MPP is referred to[Batc80] and[Aero83] for a full description of its theory and operation.

The MPP currently installed at NASA Goddard is configured with 2 megabytes of SM and 1K bits of local storage per PE. This configuration is not adequate for CFD. There are three areas where more capacity is needed.

1) The SM is not fully configured. The current SM configuration contains 2 megabytes of storage instead of 64 megabytes which is the maximum that could be achieved using 64Kbit dynamic RAM chips. In addition, the SM is currently configured with 4 memory banks instead of the 32, the maximum number. This affects the transfer rate of the SM which currently is 20 megabytes per second rather than 160 megabytes per second.

2) The PE local memory is too small. Currently each PE contains 1K bits of local memory. The PEs can handle up to 64K bits of local memory but the upgrade could cost as much as 2.2 million dollars and is not likely to be done in the near future.

3) The MPP has no direct access to secondary storage. All accesses to secondary storage must go through a host machine and therefore the bandwidth is severely limited. This problem could be mitigated with an additional 3.3 million dollars which would give the MPP direct access to secondary storage with a transfer rate of 25 million bytes per second and a capacity of 4,992 Mbytes. This system could achieve a maximum configuration of 39,936 Mbytes of storage and a transfer rate of 100Mbytes per second.

In view of the above we shall not attempt to evaluate the MPP as installed at Goddard but instead assume a fully configured machine(possibly more than fully configured). For example, we shall assume that the PE memories are sufficiently large to hold all the data values necessary to obtain an efficient algorithm. Also we shall assume that there is enough staging memory to hold the problem data and in some cases we will make the assumption that there is direct access to secondary storage with a sufficiently high data transfer rate. The point is that we do not want to be thrown off the track by the configuration limitations. Our goal is to determine whether the MPP's architectural features are suitable for the numerical simulations described above.

The remainder of this report is organized as follows. We first present the Navier-Stokes equations. Next we design an implementation of a direct numerical simulation of the Navier-Stokes equations using an explicit Lax-Wendroff method. Using this design we derive performance estimates with respect to our implementation. These results are then compared to the performance of similar codes which were run on a Cray XMP. We then design and analyze an implementation of the Beam and Warming implicit factored numerical method. Following this we consider a number of additional factors which arise in the application of these numerical methods to practical fluid dynamics problems. Lastly, we discuss possible hardware improvements and extensions to the MPP architecture.

3. The Navier-Stokes Equations

The unsteady, three-dimensional Navier-Stokes equations in Cartesian coordinates (x,y,z,t) are taken as the basic set of equations[Loma82]. 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 make a more direct comparison of the estimated performance of our implementation with the actual preformance of an already existing code[Wray84]. 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:

$$\frac{\partial Q}{\partial t} + \frac{\partial}{\partial x}(E - E_v) + \frac{\partial}{\partial y}(F - F_v) + \frac{\partial}{\partial z}(G - G_v) = 0 \qquad (3.1)$$

where

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

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

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

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

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

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

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

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

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

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

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

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 $(1-\gamma)(e - 0.5\rho(u^2+v^2+w^2))$ 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 .

4. Numerical Simulation Using an Explicit Scheme

In this section we work through an example implementation in order to evaluate the effectiveness of the MPP architecture in computing three-dimensional flows. The numerical method we use is a Lax-Wendroff method[Ames69]. This is an explicit method which seems suited to the MPP architecture. The numerical method is given below as well as a mapping of the computation onto the MPP architecture. We shall assume that the Staging Memory is fully populated with 32 banks and that the PE memories are large enough to hold all the necessary intermediate values.

4.1. The Computational Domain

The *computational domain* D is the set of spatial points over which we attempt to numerically evaluate the dependent variables ρ, ρu , ρv , ρw , and e . The points in D are called *grid points*. The grid points are equally spaced in each of the coordinate directions and there are $Imax$ grid points in the x direction, $Jmax$ in the y direction, and $Kmax$ in the z direction. The spacings between grid points in the x , y , and z directions are given by $|\Delta x|$, $|\Delta y|$, and $|\Delta z|$, respectively, where Δx , Δy , and Δz denote vectors in their respective coordinate directions. Formally we define the computational domain D as:

$$D = \{ r \mid r = i\,\Delta x + j\,\Delta y + k\,\Delta z \ , \textit{where}$$

$$i\,,j\,,k \ \textit{are nonnegative integers}\,, \text{and } i < Imax\ ,\ j < Jmax\ ,\ k < Kmax \ \} .$$

We will find it convenient in describing the computational method to define additional sets of points. We partition D into two sets I and B . The points in I and B are called *interior* grid points and *boundary* grid points, respectively. We also define the set C of "*center*" points. These points are important since many intermediate values are computed which we conceptually associate with the points in C . In order to formally define the above sets we first define the quantities r^+ and r^- where r is any point in in space.

$$r^+ = r + \frac{\Delta x \ + \Delta y \ + \Delta z}{2}$$

$$r^- = r - \frac{\Delta x \ + \Delta y \ + \Delta z}{2}$$

The sets C , I , and B are defined as follows:

$$C = \{ r^+ \mid r \in D \} \bigcap \{ r^- \mid r \in D \},$$

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

and

$$B = D - I.$$

In Figure 1 we give a two-dimensional example of the sets we have just defined.

4.2. The Numerical Method

The *Lax-Wendroff* method is and explicit procedure for advancing Q in time. The values for time t in the numerical method are restricted to $n\ \Delta t$ where $n = 0, \frac{1}{2}, 1\frac{1}{2}, 2, \cdots$ and Δt is the ***time-step.*** Variables in the numerical method are denoted as follows:

$$\rho^n(r) = \rho(x, y, z, t),$$
$$\rho u^n(r) = \rho u(x, y, z, t),$$
$$etc.$$

where $r = (x, y, z)$ and $t = n\ \Delta t$.
Other quantities are denoted similarly, namely,

$$Q^n(r) = [\rho^n(r)\ \rho u^n(r)\ \rho v^n(r)\ \rho w^n(r)\ e^n(r)]^t,$$
$$E^n(r) = E(Q^n(r)),$$
$$etc.$$

The method we use is an explicit procedure for computing $Q^{n+1}(r)$ from $Q^n(r)$ for all $r \in D$. In the process we need to compute intermediate values which we conceptually associate with points in C and $t = (n+\frac{1}{2})\Delta t$. Before giving a formal description of the algorithm we introduce some additional notation for describing the differencing and averaging processes.
Let

$$xface(r) = \{r, r+\Delta y, r+\Delta z, r+\Delta y+\Delta z\},$$
$$yface(r) = \{r, r+\Delta x, r+\Delta z, r+\Delta x+\Delta z\},$$
$$zface(r) = \{r, r+\Delta y, r+\Delta x, r+\Delta y+\Delta x\},$$

and

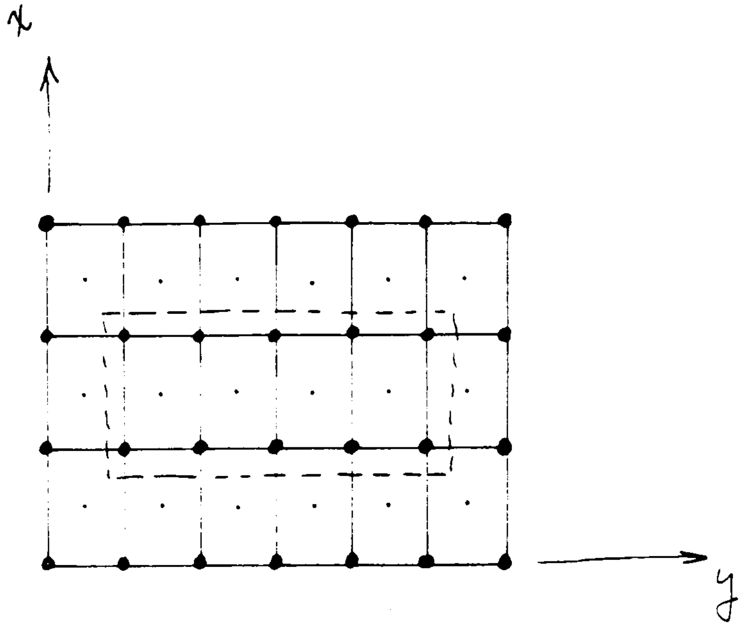
$$\boldsymbol{cube}(r) = \boldsymbol{xface}(r) \bigcup \boldsymbol{xface}(r+\Delta x).$$

If in some quantity, say $\rho^n(r)$, we replace r by a set of points, say $xface(r)$, then the notation denotes the sum of the values of ρ^n over all points in $xface(r)$. For example,

$$\rho^n(xface(r)) = \sum_{s \in xface(r)} \rho^n(s)$$

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

$$\delta_x \rho^n(r) = \frac{\rho^n(xface(r+\Delta x)) - \rho^n(xface(r))}{4|\Delta x|}$$
$$\delta_y F^n(r) = \frac{F^n(yface(r+\Delta y)) - F^n(yface(r))}{4|\Delta y|}$$
$$\delta_z e_I^n(r) = \frac{e_I^n(zface(r+\Delta z)) - e_I^n(zface(r))}{4|\Delta z|}$$



$I_{max} = 4$ ● – grid points

$J_{max} = 7$ · – center points

grid POINTS WITHIN THE DASHED BOUNDARY ARE INTERIOR POINTS. ALL OTHER GRID POINTS ARE BOUNDARY POINTS

Figure 1 COMPUTATIONAL DOMAIN

It should be clear from the above examples how the numerical difference operator works. We are now in a position to give the numerical method in detail.

Lax-Wendroff Method

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

Step 1: For all $r \in C$ compute

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

Step 2: For all $r \in C$ compute

$$E^{n+1/2}(r) = E(Q^{n+1/2}(r)) ,$$
$$F^{n+1/2}(r) = F(Q^{n+1/2}(r)) ,$$

and

$$G^{n+1/2}(r) = G(Q^{n+1/2}(r)) .$$

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

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

and

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

Step 4: For all $r \in I$ compute

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

Step 5:
For all $r \in B$ compute $Q^{n+1}(r)$ from $Q^n(r)$ according to the appropriate boundary conditions.

4.3. Mapping the Computation onto the MPP

There are two aspects to mapping a computation onto the MPP: 1) mapping the problem data into the Array Unit and 2) mapping the problem data into the Staging Memory. In this section we describe how we have chosen to handle each of these issues.

4.3.1. The Array Unit

The possible mappings of the computational steps and problem data into the AU are suggested by the inherent parallelism in the computational steps and the data dependencies among the computed values. *The primary objective is to maximize the average number of simultaneously active processing elements throughout the computation.*

The computational method described in the previous section exhibits a substantial amount of inherent parallelism. The computations associated with each grid point in D are independent of the computations associated with all other grid points, and thus it is conceivable that all grid points could be done in parallel. Furthermore, the computational steps for each point in I are identical(except for the values that are manipulated). This suggests that grid points should be mapped across processing elements. The computational steps for boundary points are not completely uniform and have to be treated in a special manner. Here again, any computation which is uniform across grid points is likely to perform best if these points are mapped across the processing elements.

The dependencies between values associated with different grid points suggest the actual mapping of grid points to processing elements. Since the processing elements are connected in a 2-dimensional mesh, we are motivated to assign "neighboring" grid points to "neighboring" processing elements. If we were to do otherwise we would pay a heavier penalty for data transfers between processing elements.

To make matters concrete we take $Imax = Jmax = 32$ and $Kmax = 128$. Our first step is to create a indexing scheme so that we can refer to the data associated with each grid point in the computational domain. Different schemes tend to anticipate alternative mappings of the problem data into the PE array. For example, Figure 2 depicts the computational domain where we have partitioned D into 128 xy -planes of grid points. Each xy -plane contains 32 rows and 32 columns of grid points and each grid point has a 32-bit data value associated with it. Actually there are five dependent variables associated with each grid point (namely, ρ, ρu , ρv , ρw , and e) but for simplicity we shall consider indexing schemes with one data value per grid point. In our implementation we will have five identical storage areas in the SM, one for each of the dependent variables.

In what follows we shall use the indexing scheme of Figure 2. Let

$$P_k = \{r \in D \mid r = (x, y, k\,|\Delta z|)\} .$$

Clearly $D = \bigcup_{k=0}^{Kmax-1} P_k$. The P_k are called *planes*. Let PE_{ab} denote the processing element in the a^{th} row and the b^{th} column of the PE array(the coordinate orientation is shown in Figure 3a) where $0 \leqslant a, b \leqslant 127$. If grid point r is mapped to PE_{ab} then the data values associated with r are stored in the local memory of PE_{ab} . Let $Board_k$ denote the set of grid points in the 16 planes P_k , P_{k+8}, P_{k+16}, $\cdots$, P_{k+120}. The grid points of $Board_k$ are mapped onto the PE array as shown in Figure 3a. It is not too difficult to see that grid point $(i\,|\Delta x|, j\,|\Delta y|, k\,|\Delta z|)$ is mapped to PE_{ab} where

$$a = 32\left\lfloor \frac{k}{32} \right\rfloor + i ,$$

and

$$b = 32\left\lfloor \frac{k \ mod\ 32}{8} \right\rfloor + j .$$

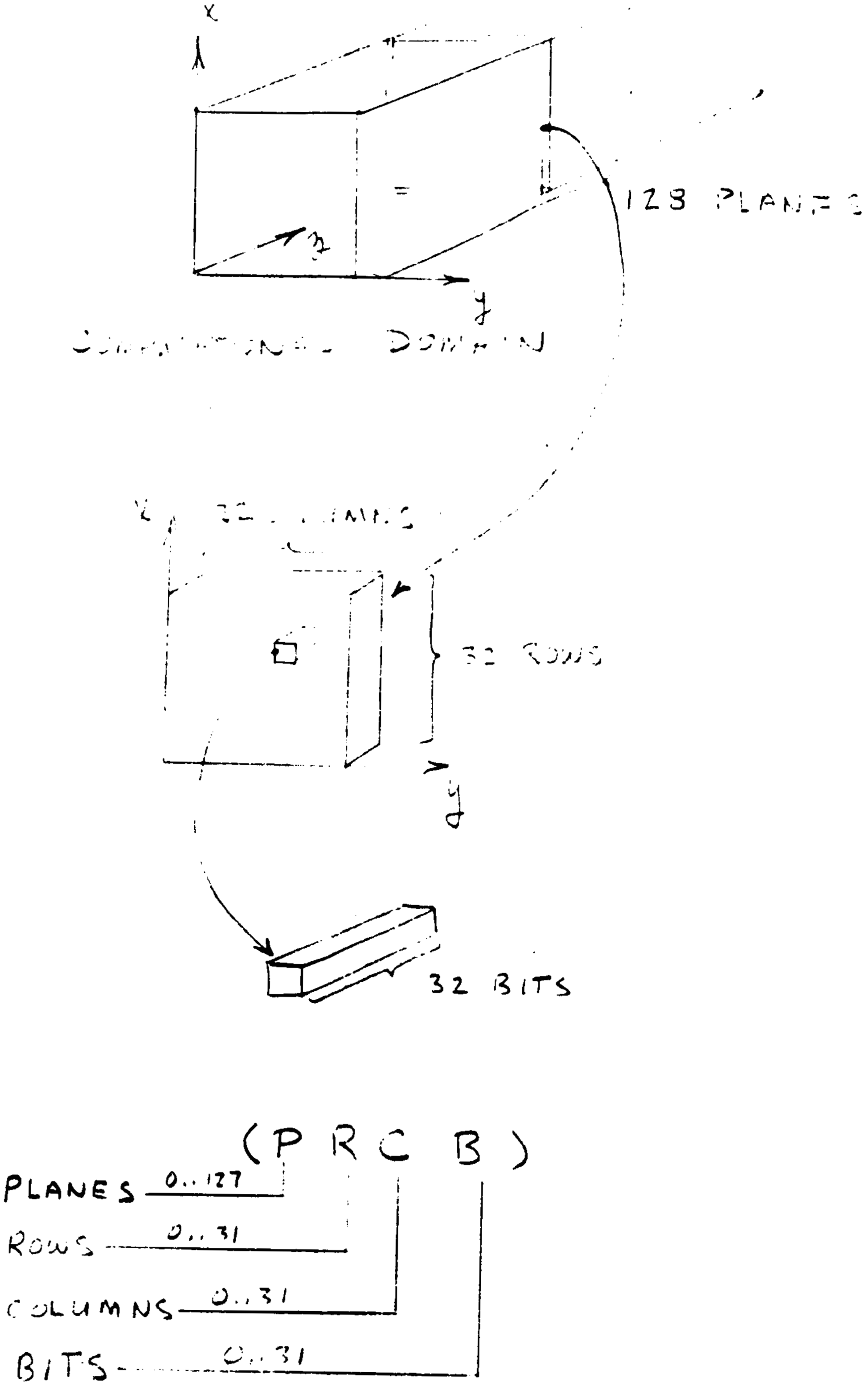
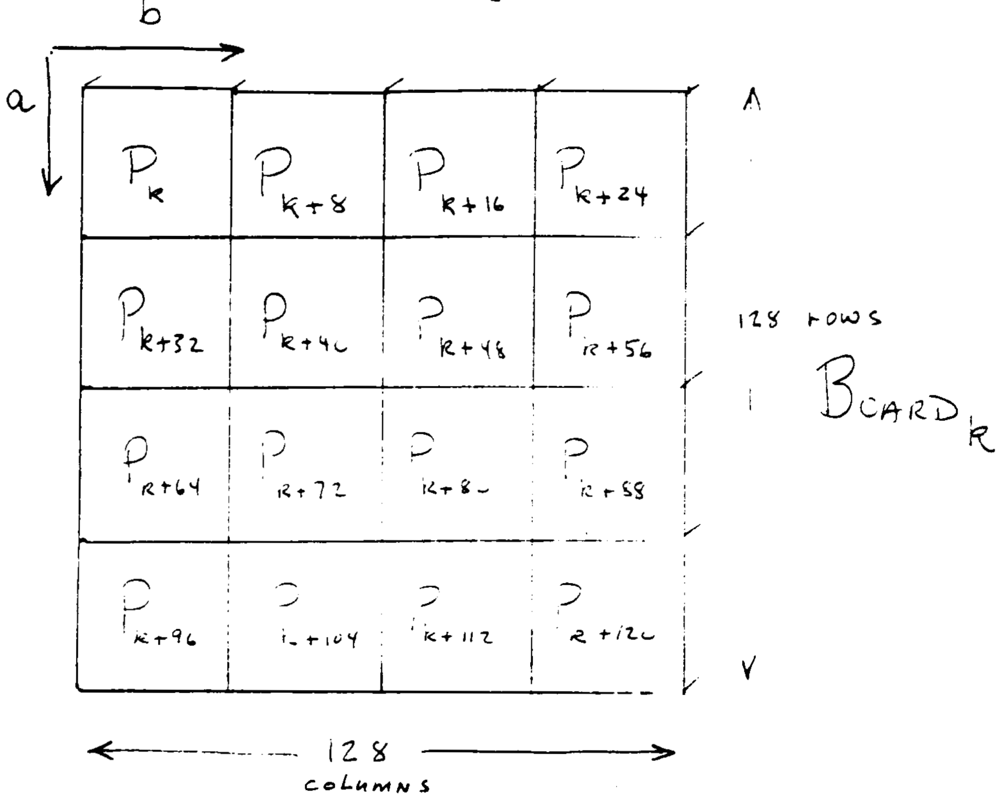


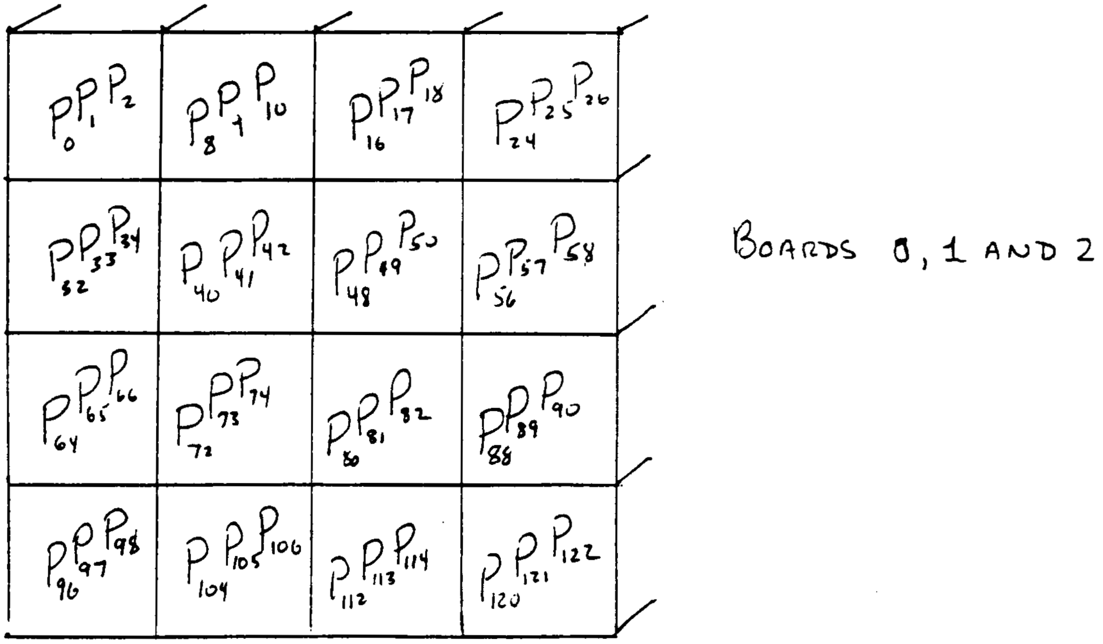
FIGURE 2 INDEXING SCHEME



P_k	P_{k+8}	P_{k+16}	P_{k+24}
P_{k+32}	P_{k+40}	P_{k+48}	P_{k+56}
P_{k+64}	P_{k+72}	P_{k+80}	P_{k+88}
P_{k+96}	P_{k+104}	P_{k+112}	P_{k+120}

PE ARRAY

(3a)



$P_0 P_1 P_2$	$P_8 P_9 P_{10}$	$P_{16} P_{17} P_{18}$	$P_{24} P_{25} P_{26}$
$P_{32} P_{33} P_{34}$	$P_{40} P_{41} P_{42}$	$P_{48} P_{49} P_{50}$	$P_{56} P_{57} P_{58}$
$P_{64} P_{65} P_{66}$	$P_{72} P_{73} P_{74}$	$P_{80} P_{81} P_{82}$	$P_{88} P_{89} P_{90}$
$P_{96} P_{97} P_{98}$	$P_{104} P_{105} P_{106}$	$P_{112} P_{113} P_{114}$	$P_{120} P_{121} P_{122}$

PE ARRAY

(3b)

FIGURE 3 MAPPING OF BOARDS INTO PE ARRAY

The importance of a *Board* is that if grid points r and r' are in the same *Board* then the data values associated with grid points r and r' are stored at identical locations in their corresponding PE memories. Also, the motivation for layout of the grid points within a *Board* is to maximize the utilization of the PEs by permitting the computation of $Q^{n+1}(r)$ to take place on 16 different planes simultaneously.

To compute $Q^{n+1}(r)$ for r in I we need the values of the variables associated with r and the "surrounding" grid points and center points. It is easy to see that the computation of $Q^{n+1}(r)$ for $r \in P_k$ depends on $Q^n(r)$ for r in planes P_{k-1}, P_k, and P_{k+1}. Considering Figure 3b which shows *Boards* 1, 2, and 3 loaded into the PE array, we see that we can simultaneously compute $Q^{n+1}(r)$ for all r in $Board_1 \bigcap I$. By "loading" $Board_3$ we are able to compute $Q^{n+1}(r)$ for r in $Board_2 \bigcap I$ and so on. As this process is repeated we "sweep" through the computational domain computing Q^{n+1} for all grid points in $Board_k$ for $k = 1, 2, \ldots, 8$. P_k is undefined for $k \geqslant 128$. There are two planes, P_0 and P_{127}, for which the computations have to be masked since these planes contain only boundary points and thus require special treatment.

The data dependencies for boundary points are relatively simple. For example, the boundary point computation associated with point $(0,y,z)$ depends only on the values associated with the point $(1,y,z)$. Our objective is to perform a maximum number of boundary point computations simultaneously. Let Row_i denote the yz-plane with $x = i\,|\Delta x|$ and $Column_j$ denote the xz-plane with $y = j\,|\Delta y|$ where $0 \leqslant i, j \leqslant 31$. In order to compute $Q^{n+1}(r)$ for all r in Row_0 we need the values of $Q^n(r)$ for all r in Row_1. Similarly, the computation of $Q^{n+1}(r)$ for r in Row_{31} depends on the values of $Q^n(r)$ for r in Row_{30}. In Figure 4 we show how we intend to store these *Rows* in the AU. With this arrangement there is no need for data transfers between PEs and we will be able to compute boundary values for all points in Row_0 simultaneously. We treat the computations of the boundary points in $Column_0$ and $Column_1$ similarly. Plane P_0 depends on plane P_1 and is handled in a manner similar to the *Row* and *Column* boundary points. Finally, on plane P_{127} the value of Q^{n+1} is set equal to the value of Q^n on plane P_{126}.

4.3.2. The Staging Memory

In this section we determine an appropriate layout for the problem data in the SM. To simplify matters we assume that there is enough SM storage to accommodate the dependent variables at two successive time steps. Therefore, while computing the dependent variables for time step $n+1$ we retain the values of the dependent variables at time step n. The primary objective in designing a layout for the data in the SM is to find one which permits data transfers to and from the AU at a sufficiently high rate so as to "hide" these data transfers behind the PE computations. There is also the consideration of data transfers to and from the host. This is less of a problem due to the assumption that the SM can store the problem data for two successive time steps. Finally, we would like to obtain a layout that "packs" the problem data in a space-efficient manner.

It follows from the various mappings of the problem data into the PEs (e.g. Figures 3 and 4) that we will need a corresponding number of access mechanisms to the problem data in the SM. We will need fast access to the problem data viewed as *Boards* , *Rows* and *Columns* The process of finding as suitable layout can be somewhat tricky. It can be done "by hand" using the road map idea[Batc81] or automated through the use of a software package called the Staging Memory Manager[Batc83]. In either case we begin by determining indexing schemes for the three views. These indexing schemes are basically permutations of the original coordinates given in Figure 2. Once these schemes are known, we then determine whether there exists a layout of the problem data in SM which permits fast access to the data according to all the required indexing schemes. This determination is done either using road maps or the Staging Memory Manager.

In the following we show how to determine the indexing schemes for our problem data. Subsequently, we shall present a layout for the problem data in the SM which is compatible with each of the indexing schemes. We will not give the steps by which the layout was determined. Although this process is important, it would require a rather lengthy explanation which is beyond the scope of this report. The interested reader should refer to[Batc81, Batc83].

In Figure 2 we gave the indexing scheme for referring to the problem data. The coordinates are P, R, C, and B where P denotes the planes $(0 \leqslant P \leqslant 127)$, and R, C, and B denote the rows, columns, and bits, respectively$(0 \leqslant R, C, B \leqslant 31)$. In what follows we will make use of the binary representation of the coordinates, for example, $P = (P_6P_5P_4P_3P_1P_0)$ where P_i denotes the i^{th} binary digit of P and

$$P = \sum_{i=0}^{6} P_i 2^i .$$

It is helpful to recall that data is loaded into the AU by bit planes. This means that to load a particular variable we have to transfer 32 bit planes corresponding to the 32 bits used to represent the data value. We assume that the S-plane is loaded column-by-column from the right-hand edge. After a column is loaded the whole S-plane is shifted to the left by one column in order to accept the next column of bits from the SM.

We shall first consider the problem of determining the indexing scheme for loading a *Board* into the PE memory. Since we load the S-plane column-by-column the R coordinate will vary the fastest. It follows from Figure 2 that when R reaches its limit (i.e., 31) we then skip ahead by 32 planes. This means that the coordinate P_5 is incremented next. Actually, to form a complete column of a *Board* we must count through (P_6P_5R). After completing a column the C coordinate is incremented. When C reaches its limit we then skip ahead by 8 planes and so (P_4P_3) is next to be incremented. After loading a complete S-plane the next coordinate to be incremented is B. The board itself is selected by the coordinates P_2, P_1, and P_0. For the case of $Board_0$ we have $P_2 = P_1 = P_0 = 0$. Finally, we arrive at a coordinate permutation for accessing the problem data by *Boards*, namely

$$(P_2P_1P_0B \;\; P_4P_3C \;\; P_6P_5R)$$

Using this indexing scheme there are only 8 distinct boards, $Board_k$ for $k = 0, \ldots, 7$. We will need $Board_8$ and $Board_9$ when we compute $Q^{n+1}(r)$ for $r \in Board_7 \bigcup Board_8$. We can obtain $Board_8$ ($Board_9$) by modifying the manner in which we load $Board_0$ ($Board_1$).

Next consider the view of Figure 4 for accessing the problem data by *Rows*. The coordinate P varys the fastest as we load an S-plane column. When P reaches its limit we next increment the column coordinate C. At this point a 32 column and 128 row portion of the S-plane should be transferred to the PE memories. Thus we only have data to fill one quarter of the S-plane before transferring the data to PE memory. The next coordinate to vary is B. Finally, we use R to select the "row" to be loaded. The resultant coordinate permutation for accessing the data by *Rows* is

$$(R \;\; B \;\; C \;\; P).$$

The coordinate permutation for indexing the data by *Columns* is given by

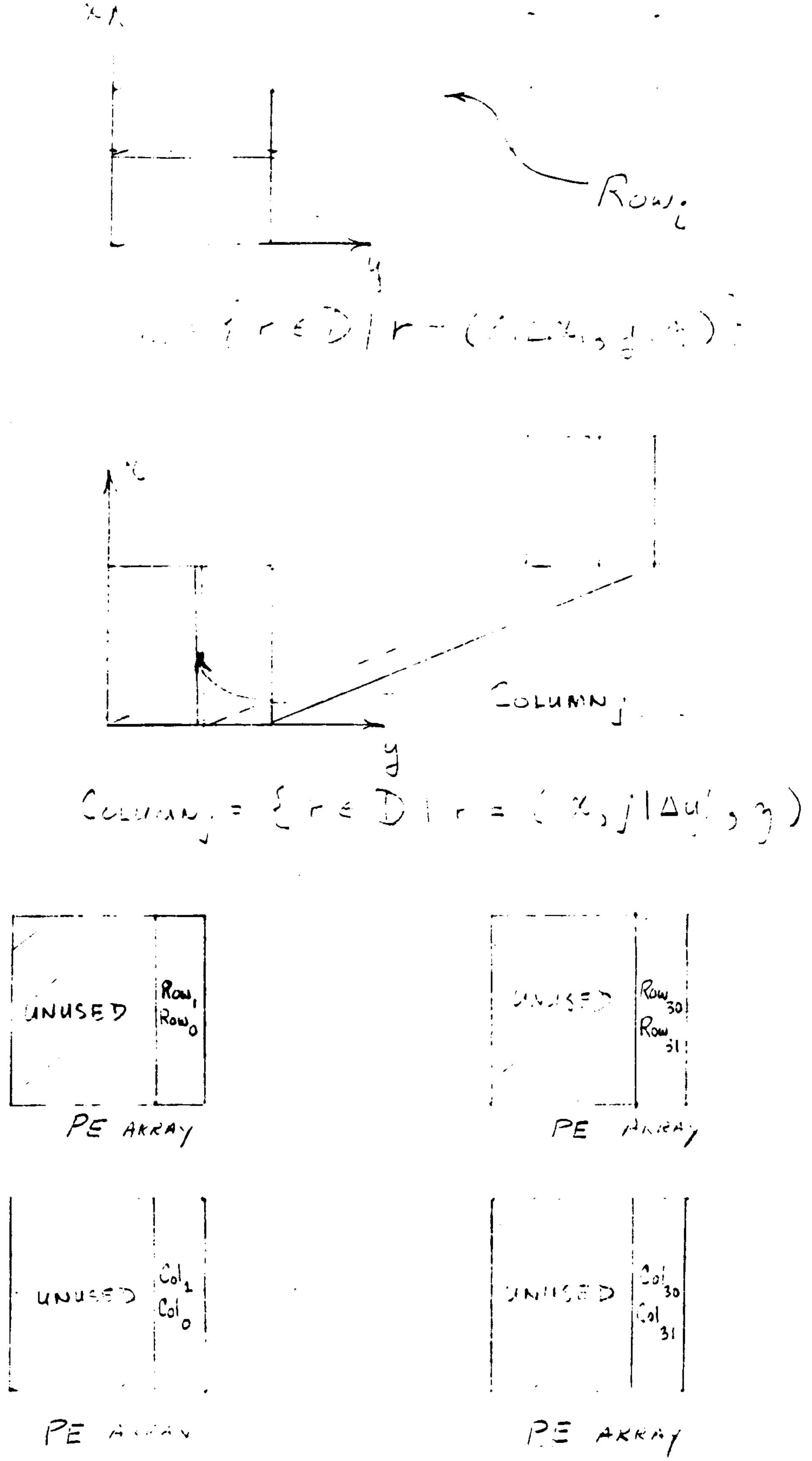


FIGURE 4 MAPPING OF ROWS AND COLUMNS INTO PE ARRA

$$(C\ B\ R\ P).$$

A layout of the problem data which is compatible with the three different views was found using the road-map approach. Specifically, each main stager word contains all the bits which are selected by the coordinates $(P_6P_5P_4P_3B_1B_0)$ with all the remaining coordinates held constant. Thus each main stager word contains 4 bits of some variable located at the same row and column position from each of 16 equally spaced planes. The remaining coordinates R and C select the row and column, $(P_2P_1P_0)$ selects one of the eight possible sets of 16 planes(*Board*), and $(B_4B_3B_2)$ selects a particular group of four bits.

4.4. The Algorithm

In this section we give the remainder of the implementation of the Lax-Wendroff method on the MPP. The previous section dealt with the static details of data mapping and layout while in this section we give an algorithm based on these structures. We have taken the general description of the Lax-Wendroff method given in Section 4.3 and converted it into a sequence of computational steps to be performed on the MPP. There are 7 types of steps which we shall specify in enough detail to make an accurate estimate of the number of machine cycles required to perform each of them. Additionally, we give the number of PE memory locations required to store problem data and intermediate values, and an upper bound on the number of locations needed for temporary values used during the calculations. In what follows we shall use the variables *div* , *mult* , *add* , *sub*, and *move* to denote the number of machine cycles required for the corresponding operations. For example, the numerical difference operator on a simple variable requires *mult* + 6*add* + *sub* + 8*move* cycles, one location to store the result, and no more than three locations for temporaries. The numerical difference operator uses a value from each grid point in *cube* (r). Two of these grid points are 2 grid points away from r and four of them are 1 grid point away. Thus we get 8*move* cycles. Later we will set *move* equal to 96 cycles which represents using 3 cycles per bit: one to get form PE memory to the P-plane, one to shift, and one to get from the P-plane to PE memory. Using 2**move* for points which are two grids points away is an over estimate since we are counting the load/store twice.

S1(k): Load $Q^n(r)$ for $r \in Board_k$.

In this step the problem data (ρ, ρu , ρv , ρw , and e) associated with grid points in $Board_k$ is loaded from the SM into the AU. Each variable requires the loading of 32 bit planes and there are 5 variables. It takes 128 cycles to load one bit plane into the S-plane and an additional cycle to store the S-plane into the PE memories. Therefore, this step takes 129*32*5 = 20,640 cycles and requires 5 locations in each PE memory.

Time: 20640 cycles.
Space: 5 words

S2(k): Compute $E^n(r)$, $F^n(r)$, and $G^n(r)$ for $r \in Board_k \cap I$.

In this step we compute the 15 entries in E^n , F^n , and G^n . We also compute and save the values u , v , w , and e_I since they will be used at a later stage in the computation. Therefore, not counting temporary values used in these computations, we will require 19 additional locations per PE. It is not too difficult to tally the total number of operations needed to compute these values.

Time: 15*mult* + 4*div* + 6*add* + *sub*.
Space: 19 locations + 10 locations for temporaries.

S3(k): Compute $Q^{n+1/2}(r)$ for $r \in \{r^- \mid r \in Board_k\} \bigcap C$.

The computation of $Q^{n+1/2}$ depends on the computation of E^n, F^n, and G^n on $Board_k$ and $Board_{k-1}$. We will need an additional 5 locations to store the results.

Time: 25*mult* + 135*add* + 20*sub* + 160*move*.
Space: 5 locations + 10 locations for temporaries.

S4(k): Compute $E^{n+1/2}(r)$, $F^{n+1/2}(r)$, and $G^{n+1/2}(r)$ for $r \in \{r^- \mid r \in Board_k\} \bigcap C$.

This computation is similar to *S2(k)* except that e_I is not computed and u, v, and w are not saved in this case.

Time: 14*mult* + 4*div* + 6*add* + *sub*.
Space: 15 locations + 16 locations for temporaries.

S5(k): Compute $E_v^n(r)$, $F_v^n(r)$, and $G_v^n(r)$ for $r \in \{r^- \mid r \in Board_k\} \bigcap C$.

Time: 46*mult* + 89*add* + 12*sub* + 99*move*.
Space: 12 locations + 16 locations for temporaries.

S6(k): Compute $Q^{n+1}(r)$ for $r \in Board_{k-1} \bigcap I$.

Time: 20*mult* + 100*add* + 35*sub* + 120*move*.
Space: 5 locations + 20 locations for temporaries.

S7(k): Store $Q^{n+1}(r)$ for $r \in Board_{k-1} \bigcap I$.

Time: 20640 cycles.

The following table summarizes the above time and space estimates.

Summary								
Step	mult	div	add	sub	move	cycles	locations	temporary locations
S1						20640	5	
S2	15	4	6	1		14826	19	10
S3	25		135	20	160	83350	5	10
S4	14	4	6	1		14264	15	16
S5	46		89	12	99	70504	12	16
S6	20		100	35	120	69740	5	20
S7						20640		

We next give the computation of Q^{n+1} for points in I in terms of the steps given above. In Figure 5 we give a parallel flowchart for the MPP program. The computation begins at the top of the diagram and uses directed edges to denote the precedence among the steps. A computation "box" may not begin execution until all the boxes preceding it have completed execution. The column of boxes along the left side of the figure corresponds to the transfer of the newly computed values, Q^{n+1}, from the AU to the SM. The column of boxes along the right side of the figure corresponds to the transfer of Q^n form the SM to the AU. The column of boxes down the middle of the figure correspond to the computational steps. The flow chart clearly shows the possibilities for overlapping I/O with the computations. The only steps that have no potential for overlap are *S1(0)* and *S7(9)* and it turns out that *S1(1)* is only partially hidden by step *S2(0)*. The remainder of the I/O steps are completely overlapped by the computational steps. Data transfers in and out of the AU essentially do not interfere with ongoing computation in the AU. During a data transfer the I/O process will steal one cycle

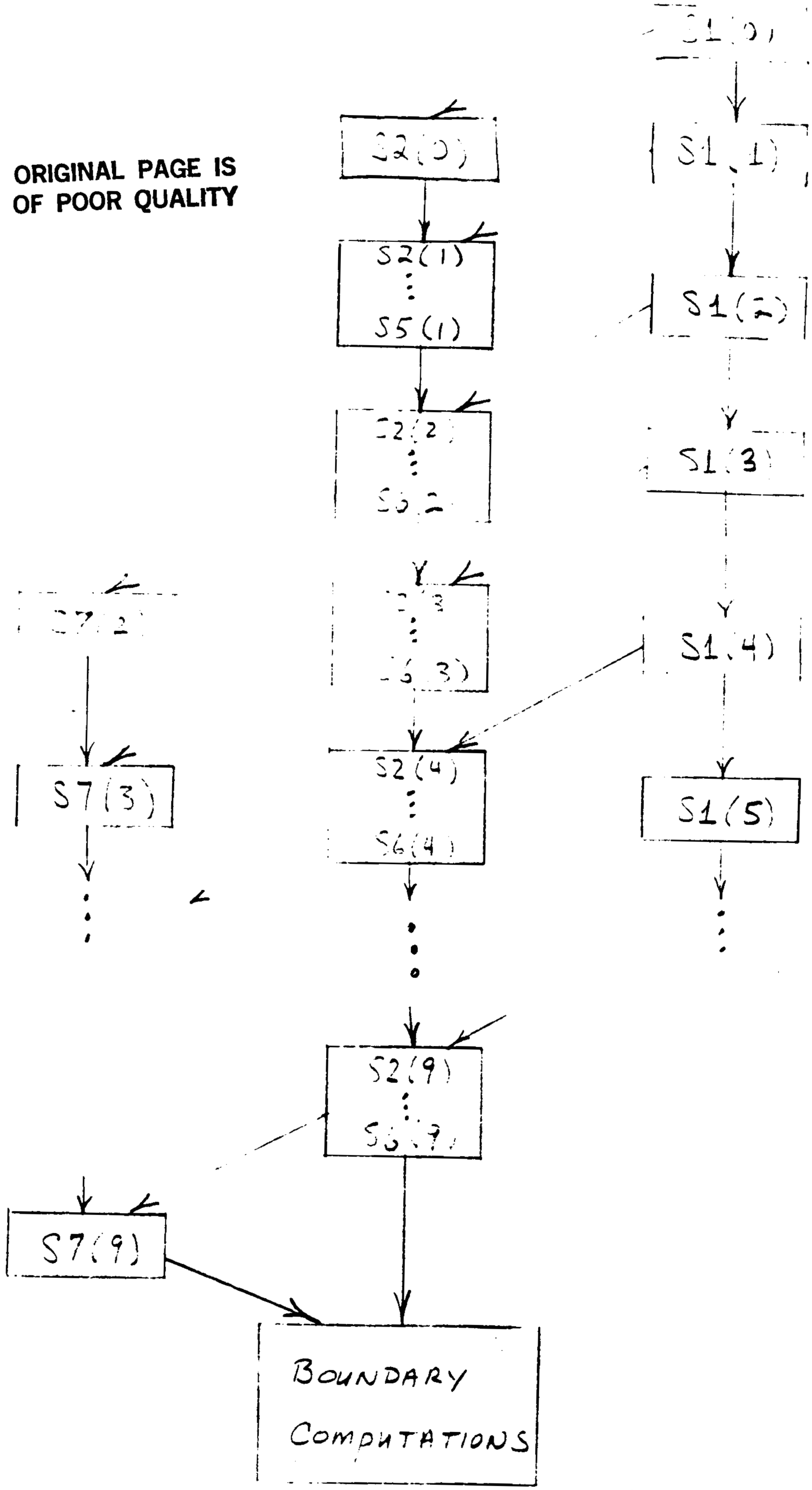


FIGURE 5. THE ALGORITHM FOR ONE TIME STEP

out of every 129 cycles which is negligible for our purposes.

The boundary computations are highly problem-specific and so we will only give cycle time estimates for the computation and I/O. Since there are six boundary planes to consider we will require 12 I/O transfers. We shall assume that each of these transfers costs the same as the transfer of a *Board* , i.e. 20640 cycles for each plane. We have gone through a boundary computation and for one particular case we estimate about 200,000 cycles were required for the total computation over all six boundary planes. Additionally, we shall not assume any I/O overlap for the boundary computations. Thus we get a conservative estimate of 450,000 cycles for the boundary computations.

4.5. Performance Estimates

In this section we give running time estimates of the algorithm presented in the previous sections. We first give the cycle counts for various operations on 32 bit floating-point data[Aero83].

MPP Cycle Counts	
Operation	Cycles
mult	562
div	990
add/sub	348
move	96

We are now in a position to estimate the total number of cycles required to compute Q^{n+1} from Q^n for all points in D. The primary computational unit in Figure 5 is the sequence $S2(k),S3(k),...,S6(k)$. Using the timing values form the previous section and the values for *mult* , *div* , *add / sub* , and *move* given above, we find that this sequence of steps takes 252,684 cycles. This sequence is repeated 8 times. There are some initial steps and a final one to add to this total. Step $S1(0)$ appears twice at the beginning, followed by steps $S2(1),S3(1),S4(1),S5(1)$. Lastly there is step $S7(9)$ to account for. The total for these step is 244,864 cycles. The boundary computations add another 450,000 cycles. Putting everything together we get:

$$\begin{aligned} \textit{Total Cycles} &= 8^{*}(252{,}684) + 244{,}864 + 450{,}000 \\ &= 2{,}716{,}336\,. \end{aligned}$$

Another quantity of interest is the number of floating-point operations per second(flops) we can achieve. The primary computational unit, i.e. $S2(k),S3(k),...,S6(k)$. uses 533 floating-point operations. The initial steps and the final one use 386 floating-point operations. During the computation for points in I all the PEs corresponding to boundary points are masked. There are nearly 2K PEs corresponding to boundary points out of the 16K PEs. Therefore we are achieving a 14K-fold duplication during the computation over the interior grid points. We will ignore the floating-point operations of the boundary computation. Putting everything together we get:

$$\begin{aligned} \textit{Total Floating-Point Operations} &= 14K^{*}(8^{*}533 + 386) \\ &= 66{,}662{,}400\,. \end{aligned}$$

The cycle time of the MPP is 100 nanoseconds and using this we can translate the above quantities into more interesting forms. For example, the total time to advance the solution one time step is approximately .272 seconds and the rate at which we are doing floating-point operations is 66,662,400/ .272 which is approximately 243 megaflops(millions of floating-point operations per second).

Since we need to hold 3 *Boards* simultaneously, we will require approximately 200 PE memory locations. We need 5 Mbytes(Mbytes = 2^{20}) of SM to hold the problem data corresponding to two successive time steps. The following table summarizes these results.

Summary		
Quantity	Value	Dimension
Domain	32 by 32 by 128	
Cycles/Time Advance	2,716,336	cycles
Time/Time Advance	.272	seconds
Floating Point Operation Rate	243	megaflops
PE Storage	6,400	bits
SM Storage	5	Mbytes

The above estimates compare favorably to essentially the same algorithm which was coded and run on a CRAY XMP by Alan Wray of NASA Ames. Several runs of this code were made recently and it was found to take approximately one second of CPU time per time step for a 32 by 32 by 128 domain. These runs used only one of the XMP's processors and all arithmetic was done in 64-bit precision.

The above analysis can be applied to a larger problem, say 128^3. The data transfer time for a *Board* remains the same but each *Board* now corresponds to a single xy -plane. In order to "sweep" through the entire computational domain we will execute the primary computational unit, $S2(k)S3(k)....S6(k)$, a total of 126 times. There are 6 boundary planes and the total number of cycles does not increase form the previous case. The major difference between the two domains is that the 128^3 domain makes more efficient use of the PEs. For example, since each *Board* corresponds to a single xy -plane, there are fewer PEs to be masked during the execution of the primary computational unit. In fact only 510 are masked as compared to 2016 in the previous case.

$$\begin{aligned} \textit{Total Cycles} &= 126^*(252{,}684) + 244{,}864 + 450{,}000 \\ &= 2{,}716{,}336\,. \end{aligned}$$

$$\begin{aligned} \textit{Total Floating-Point Operations} &= 15.5K^*(126^*\,533 + 386) \\ &= 66{,}662{,}400\,. \end{aligned}$$

Summary		
Quantity	Value	Dimension
Domain	128 by 128 by 128	
Cycles/Time Advance	32,533,048	cycles
Time/Time Advance	3.26	seconds
Floating Point Operation Rate	328	megaflops
PE Storage	6,400	bits
SM Storage	80	Mbytes

We report that the code which was run on the CRAY XMP took 16 seconds per time step for the 128^3 domain.

We mentioned earlier that we have not included metric terms in the above computations in order to compare our estimates with an implementation of the same numerical method which was run on the CRAY XMP. It is clear that the inclusion of these terms will require more storage and more computational cycles. These terms are included by associating with each grid point its corresponding coordinates in physical space and from this we would then compute the additional terms[Loma82]. The computational steps will again overlap essentially all of the I/O so the performance in terms of sustained floating-point computation rate will increase.

5. Numerical Simulation Using an Implicit Scheme

Implicit methods have been proposed for the numerical solution of various forms of the Navier-Stokes equations[Beam78, Pull80, Loma82, MacC82, Beam76]. Implicit methods are more complex than explicit methods since the former require the solution of a large number of block-tridiagonal systems of equations. However, implicit methods have improved stability properties over explicit methods thereby permitting a larger Δt . They have the drawback that there is significantly more computation to be done due to the resultant block-tridiagonal systems of equations that must be solved. In this section we shall consider the implementation of the implicit method described in[Beam78].

Below we present the numerical method and a mapping of the computation onto the MPP architecture. As before, we shall assume that the Staging Memory is fully populated with 32 banks and that the PE memories are large enough to hold all the necessary intermediate values. In addition, we shall assume the existence of a large-capacity store for the intermediate results of the block tridiagonal solver.

5.1. The Numerical Method

The numerical method we shall consider is based on the work of Beam and Warming[Beam78]. 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 [Beam78].

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

$$\Delta Q^n = \frac{\theta\, \Delta t}{1+\xi}\frac{\partial}{\partial t}\Delta Q^n + \frac{\Delta t}{1+\xi}\frac{\partial}{\partial t}Q^n + \frac{\xi}{1+\xi}\Delta Q^{n-1}$$

$$+ O[(\theta - \tfrac{1}{2} - \xi)\Delta t^2 + \Delta t^3],$$

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 in section 3 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

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

Care must be taken with the spatial cross-derivative terms, since they are a potential obstacle to obtaining a spatially factored form of the equations.

After linearization and factorization the temporal scheme can be written as

$$LMN\,\Delta Q^n = H + O((\theta - \tfrac{1}{2} - \xi)\Delta t^2, (\theta - \bar{\theta})\Delta t^2, \Delta t^3), \qquad (5.1)$$

where L, M, and N are *operators and* $\bar{\theta}$ is the coefficient of the spatial cross-derivative terms. Equation (5.1) holds pointwise in the spatial coordinates and relates the dependent variables at the various time steps. The important point about the operators, L, M, and N is that they each involve spatial derivatives in a single coordinate direction. For example the operator L is

$$L = \{1 + \frac{\theta\,\Delta t}{1 + \xi}[\frac{\partial}{\partial x}(A - P + R_x)^n - \frac{\partial^2}{\partial x^2}(R)^n]\},$$

where A, P, and R are Jacobians, and $R_x = \partial R / \partial x$. Similarly, M and N involve spatial derivatives in the y- and z-directions, respectively.

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

$$LX = RHS(5.1), \qquad (5.2a)$$

$$MY = X, \qquad (5.2b)$$

and

$$N\,\Delta Q^n = Y. \qquad (5.2c)$$

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

We obtain the basis of a numerical algorithm by approximating the spatial derivatives with the finite-difference quotients. We assume a computational domain D as defined in section 4.1. The coordinates of a grid point will be given by the indices i, j, and k where $r = i\,\Delta x + j\,\Delta y + k\,\Delta z$. When we substitute the 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

$$LU_{i-1}X_{i-1} + LV_i\,X_i + LW_{i+1}X_{i+1} = H_i, \qquad (5.3a)$$

where LU_{i-1}, LV_i and LW_{i+1} are 5 by 5 matrices and $0 < i < Imax - 1$. In equation (5.3a) we have suppressed the j and k indices. The dependent variable X and the coefficient matrices are defined for each grid point and therefore we should write them as X_{ijk}, LU_{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,

$$MU_{j-1}Y_{j-1} + MV_jY_j + LW_{j+1}Y_{j+1} = X_j \ , \tag{5.3b}$$

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

$$NU_{k-1}\Delta Q_{k-1}^n + NV_k\,\Delta Q_k^n + NW_{k+1}\Delta Q_{k+1}^n = Y_k \ , \tag{5.3c}$$

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 block tridiagonal systems of equations. However, if the boundary conditions are periodic in the x-direction then equations (5.3a) result in a periodic block-tridiagonal system of equations for which solution algorithms are available[Temp75].

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

$$\begin{bmatrix} LU_0 & LV_1 & LW_2 & 0 & 0 & 0 & 0 \\ 0 & LU_1 & LV_2 & LW_3 & 0 & 0 & 0 \\ 0 & 0 & LU_2 & LV_3 & LW_4 & 0 & 0 \\ 0 & 0 & 0 & LU_3 & LV_4 & LW_5 & 0 \\ 0 & 0 & 0 & 0 & LU_4 & LV_5 & LW_6 \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \end{bmatrix} = \begin{bmatrix} H_1 \\ H_2 \\ H_3 \\ H_4 \\ H_5 \end{bmatrix} . \tag{5.4}$$

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 $Q_0^n = Q_5^n$ and $Q_1^n = Q_6^n$. This implies that $X_0 = X_5$ and $X_1 = X_6$. Substituting into (5.4) we get the following periodic block-tridiagonal system of equations.

$$\begin{bmatrix} LV_1 & LW_2 & 0 & 0 & LU_0 \\ LU_1 & LV_2 & LW_3 & 0 & 0 \\ 0 & LU_2 & LV_3 & LW_4 & 0 \\ 0 & 0 & LU_3 & LV_4 & LW_5 \\ LW_6 & 0 & 0 & LU_4 & LV_5 \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \end{bmatrix} = \begin{bmatrix} H_1 \\ H_2 \\ H_3 \\ H_4 \\ H_5 \end{bmatrix} .$$

Next suppose that Q_0^n is fixed at-some freestream value and $Q_6^n = Q_5^n$. It follows that $X_0 = 0$ and $X_6 = X_5$. Substituting into (5.4) we get the usual block-tridiagonal system of equations.

$$\begin{bmatrix} LV_1 & LW_2 & 0 & 0 & 0 \\ LU_1 & LV_2 & LW_3 & 0 & 0 \\ 0 & LU_2 & LV_3 & LW_4 & 0 \\ 0 & 0 & LU_3 & LV_4 & LW_5 \\ 0 & 0 & 0 & LU_4 & (LV_5 + LW_6) \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \end{bmatrix} = \begin{bmatrix} H_1 \\ H_2 \\ H_3 \\ H_4 \\ H_5 \end{bmatrix}.$$

The above examples were intended to show how boundary conditions affect equations (5.3). Boundary conditions also come into play in computing, H_i, 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 as well as spatial cross-derivatives. Therefore when the indices of H_{ijk} are adjacent to a boundary grid point, then the boundary conditions will have to be taken into account.

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

Beam and Warming Implicit Factored Method

Input: Q_{ijk}^n and Q_{ijk}^{n-1} all i, j, and k.
Output: Q_{ijk}^{n+1} for all i, j, and k.
Method:

Step 1: (Compute X_{ijk} for all interior grid points) For each $0 < j < Jmax - 1$ and $0 < k < Kmax - 1$ solve equation (5.3a) for $0 < i < Imax - 1$.

Step 2: (Compute Y_{ijk} for all interior grid points) For each $0 < i < Imax - 1$ and $0 < k < Kmax - 1$ solve equation (5.3b) for $0 < j < Jmax - 1$.

Step 3: (Compute ΔQ_{ijk}^n for all interior grid points) For each $0 < i < Imax - 1$ and $0 < j < Jmax - 1$ solve equation (5.3c) for $0 < k < Kmax - 1$.

Step 4: (Update the dependent variables) Set

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

for all i, j, and k corresponding to interior grid points.

Step 5: Update all boundary values of Q^{n+1}.

5.2. Block Tridiagonal Solver on the MPP

In this section we describe a straight-forward solution algorithm for block tridiagonal systems of equations and propose an implementation for the MPP architecture. The positive aspect of the proposed implementation is that it makes efficient use of the PEs. A drawback is that it requires a very large amount of storage for intermediate results.

Steps 1, 2, and 3 of the implicit factored method require the solution of a large number of block tridiagonal systems of equations, or a large number of periodic block tridiagonal systems in the case of periodic boundary conditions. Solution algorithms are available for the periodic block tridiagonal systems[Temp75]. We will confine our attention to the nonperiodic case.

In Step 1 we must solve one system of equations for each pair of indices j, k corresponding to interior grid points. Our approach is to assign one PE to each system of equations. Given a 128 by 128 array of processing elements we could potentially solve up to 126 by 126 equations in parallel. We do not get the full 128 by 128 since the PEs on the "edges" of the PE array are used to provide "boundary" data.

Consider a particular pair of indices $j = J$, $k = K$. The coefficient matrix for the block tridiagonal system can be constructed from the data associated with grid points along the line $j = J$ and $k = K$. The vector H_{ijk} depends on values associated with neighboring grid points in all three coordinate directions because H involves spatial derivatives and spatial cross derivatives in all three coordinate directions.

We shall consider the following system of equations which is representative of the systems which arise in all three steps of the implicit factored method.

$$\begin{bmatrix} a_1 & b_2 & & & & \\ c_1 & a_2 & b_3 & & & \\ & c_2 & a_3 & b_3 & & \\ & & \cdot & \cdot & \cdot & \\ & & & c_{n-2} & a_{n-1} & b_n \\ & & & & c_{n-1} & a_n \end{bmatrix} \begin{bmatrix} x_1 \\ \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix} = \begin{bmatrix} h_1 \\ \\ \cdot \\ \cdot \\ \cdot \\ h_n \end{bmatrix} \tag{5.5}$$

We assume that the 5 by 5 block matrices a_i, b_i and c_i are associated with grid point $r = i\,\Delta x + J\,\Delta y + K\,\Delta z$ and can be computed from data associated with grid points having indices $i-1$, i, and $i+1$ with $j = J$ and $k = K$. Furthermore we assume that h_i can be computed from values associated with grid points in the neighborhood of r. We shall employ the straight-forward algorithm for solving this system of equations. The algorithm consists of two stages, a forward sweep followed by back-substitution.

Forward Sweep

```
Initialization: $c_0 = 0$, $\beta_0 = 0$, $z_0 = 0$, and $b_{n+1} = 0$;
for $i$ := 1 to $n$ do
    $\alpha_i := a_i - c_{i-1}\beta_{i-1}$;
    $\beta_i := \alpha_i^{-1} b_{i+1}$;
    $z_i := \alpha_i^{-1}(h_i - c_{i-1} z_{i-1})$;
end Forward Sweep
```

Back Substitution

```
Initialization: $\beta_n = 0$ and $x_{n+1} = 0$;
for $i := n$ to 1 do
        $x_i := z_i - \beta_i x_{i+1}$;
end Back Substitution
```

Back substitution uses β_i and z_i which are computed during the forward sweep. We have to save these values between the forward sweep and the back substitution since the order in which they are used is the reverse of the order in which they are generated. Also notice that during the i th iteration of the forward sweep we use a_i, b_{i+1}, c_{i-1}, and h_i. As the iterations progress during the forward sweep, conceptually, we are moving in the x-direction through the computational domain along the line with $j = J$, $k = K$.

For simplicity, we assume that $Jmax = Kmax = 128$. Let P_i denote the set of grid points $r = i \Delta x + j \Delta y + k \Delta z$ for $0 \leqslant j,k \leqslant 127$. We assign the jk th PE to the evaluation of the jk th block tridiagonal system. Since the PE local memories are limited in size we bring data into the PE memory as it is needed.

In Figures 6 and 7 we give partial parallel flowcharts for the forward sweep and back substitution phases, respectively. The boxes on the left and right-hand sides of the figures correspond to data moves between the AU and the SM. The boxes down the center of the figures correspond to the PE computations and the contents of the boxes give the terms to be computed. It is clear that we can overlap the data movement with the computational steps but it is not so clear whether the computation is I/O bound or not.

We can show that the forward sweep algorithm is *not* I/O bound by obtaining a lower bound on the number of cycles required for the computational steps and comparing this number to the number of cycles required for the I/O transfer. It turns out that the forward sweep algorithm is compute bound assuming that the transfers are between the SM and the AU. The back substitution phase, on the other hand, is I/O bound. We show this by determining an upper bound on the number of cycles to compute x_i and compare this to the number of cycles required to transfer the data(β_i and z_i) into the AU. It turns out that the back substitution phase is marginally I/O bound.

The storage requirements of the above method are quite large since β_i is a 5 by 5 matrix and z_i is a 5 by 1 vector. We have to store a β value and a z value for every internal grid point. If there are $O(128^3)$ grid points then we will need approximately 240 Mbytes of intermediate storage. If we were to try to save the intermediate data in the PE memories, then each PE memory would have to be about 128K bits.

There are tricks which could be used to reduce the storage requirements at the expense of processing time to recompute the data which are not stored.

5.3. Staging Memory

In this section we consider the problem of mapping the problem data into the SM to accommodate the required data moves of the forward sweep procedure.

The implicit factored method requires a forward sweep and a back substitution in each of the coordinate directions corresponding to Steps 1, 2, and 3, respectively. In the previous section we considered the case of forward sweep and back substitution in the x-direction. In Figure

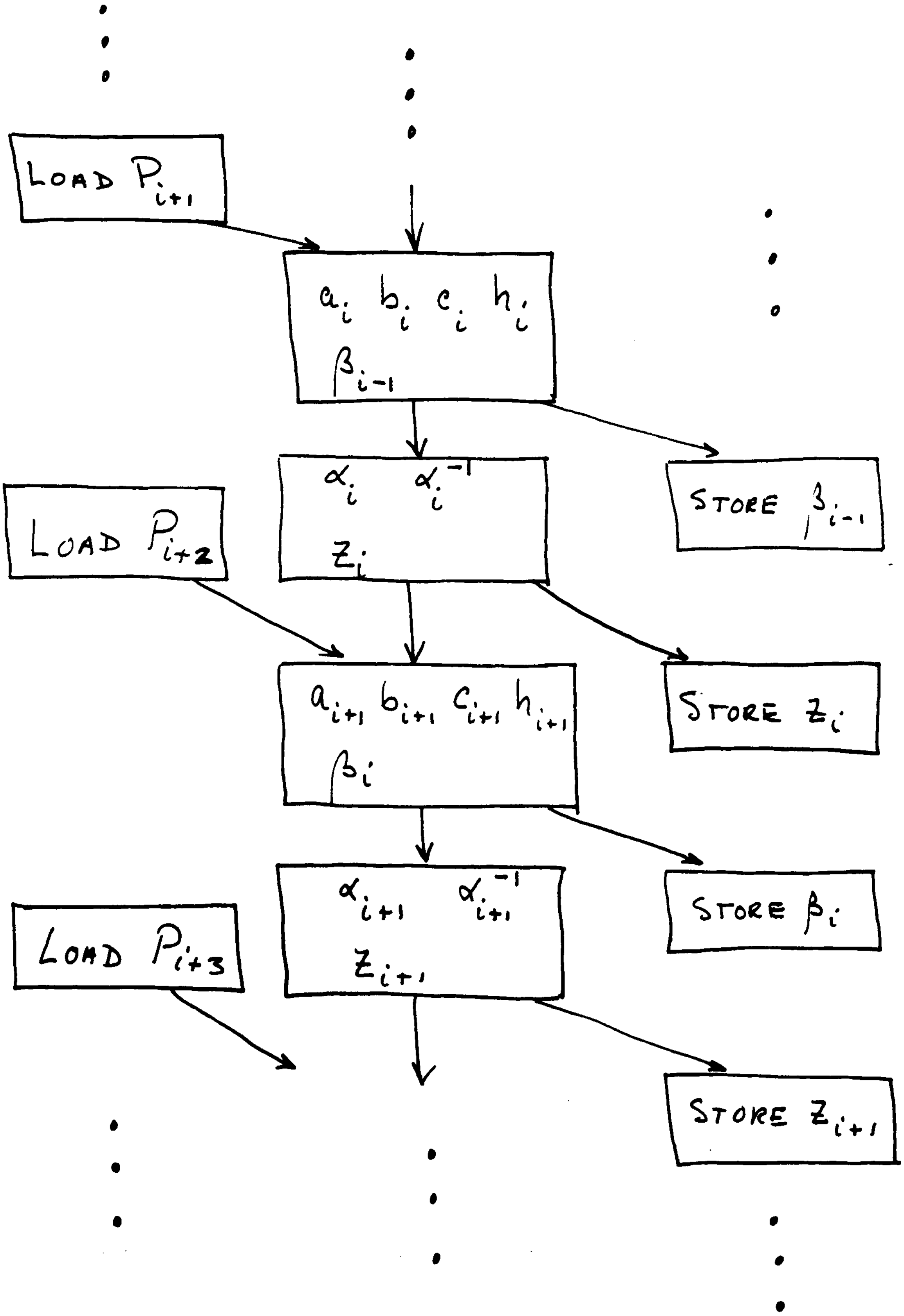


FIGURE 6 FORWARD SWEEP

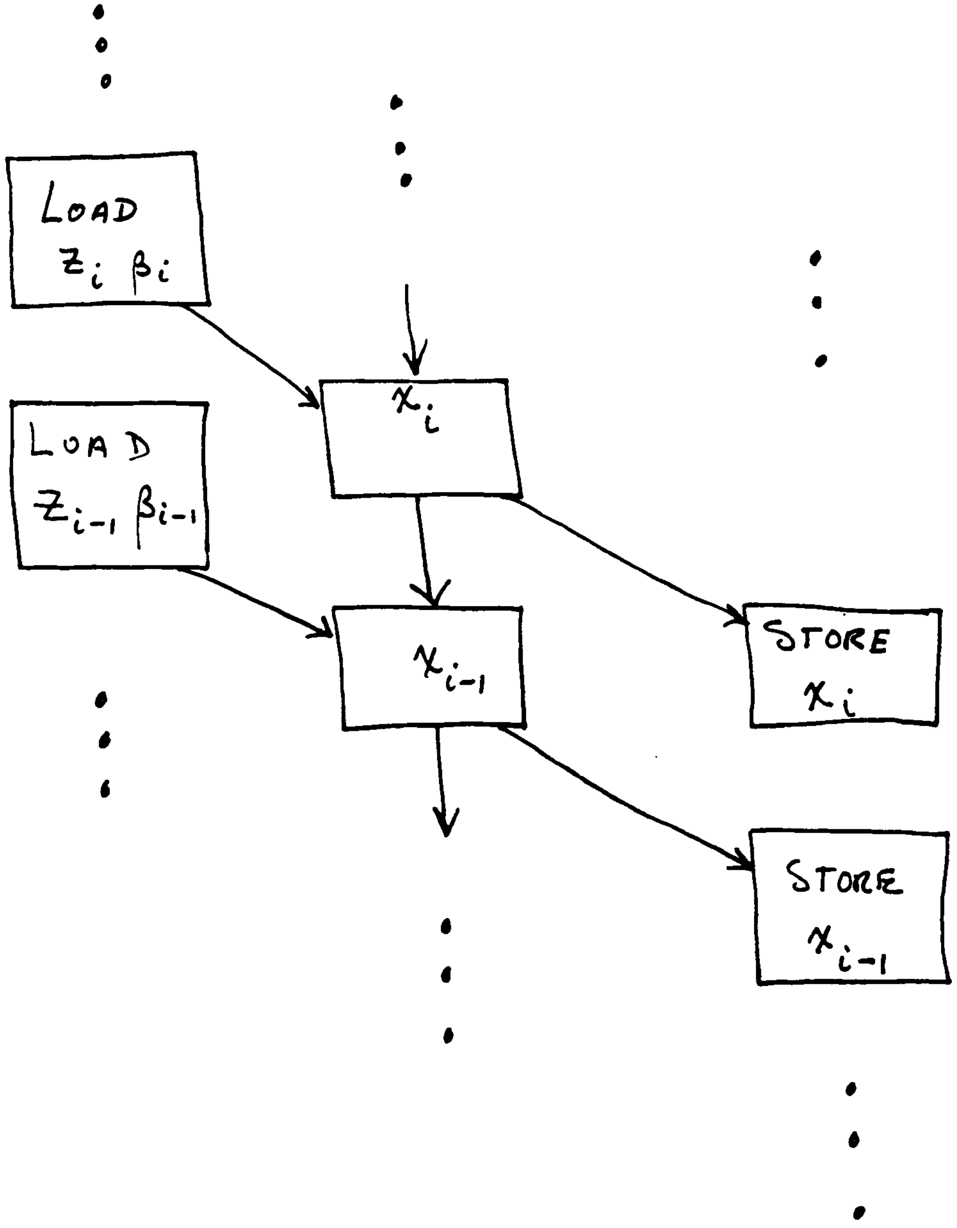


FIGURE 7 BACK SUBSTITUTION

6 we show the order in which the problem data is loaded plane-by-plane. For Steps 2 and 3 of the implicit factored method we will have to load the problem data in the remaining two coordinate directions.

The issue is whether there is a layout of the problem data in the SM which allows fast access to the data in each of the three coordinate directions. Using the roadmap idea we have found that it is possible to find a suitable layout. In our layout the main stager word (64 bits) will contain one bit from each of 64 different data values corresponding to 64 different grid points. If P, R, and C are the coordinates of our computational domain(see Figure 2), then a main stager word will contain one bit from each of the values associated with the grid points indexed by $(P_1 P_0 R_1 R_0 C_1 C_0)$ where P_i denotes the i^{th} digit in the binary representation of P, etc. Thus each main stager word contains one bit from each of the data values associated with a 4 by 4 by 4 cube of grid points. Intuitively, this layout works because the main stager word is not biased toward any one of the coordinate directions since it contains equivalent amounts of data from each direction. A similar idea can be found in[Loma82] where the notion of *pencil data bases* was introduced. In this case the computational domain was divided into cubes of grid points and strings of cubes parallel to a coordinate axis were referred to as *pencils*.

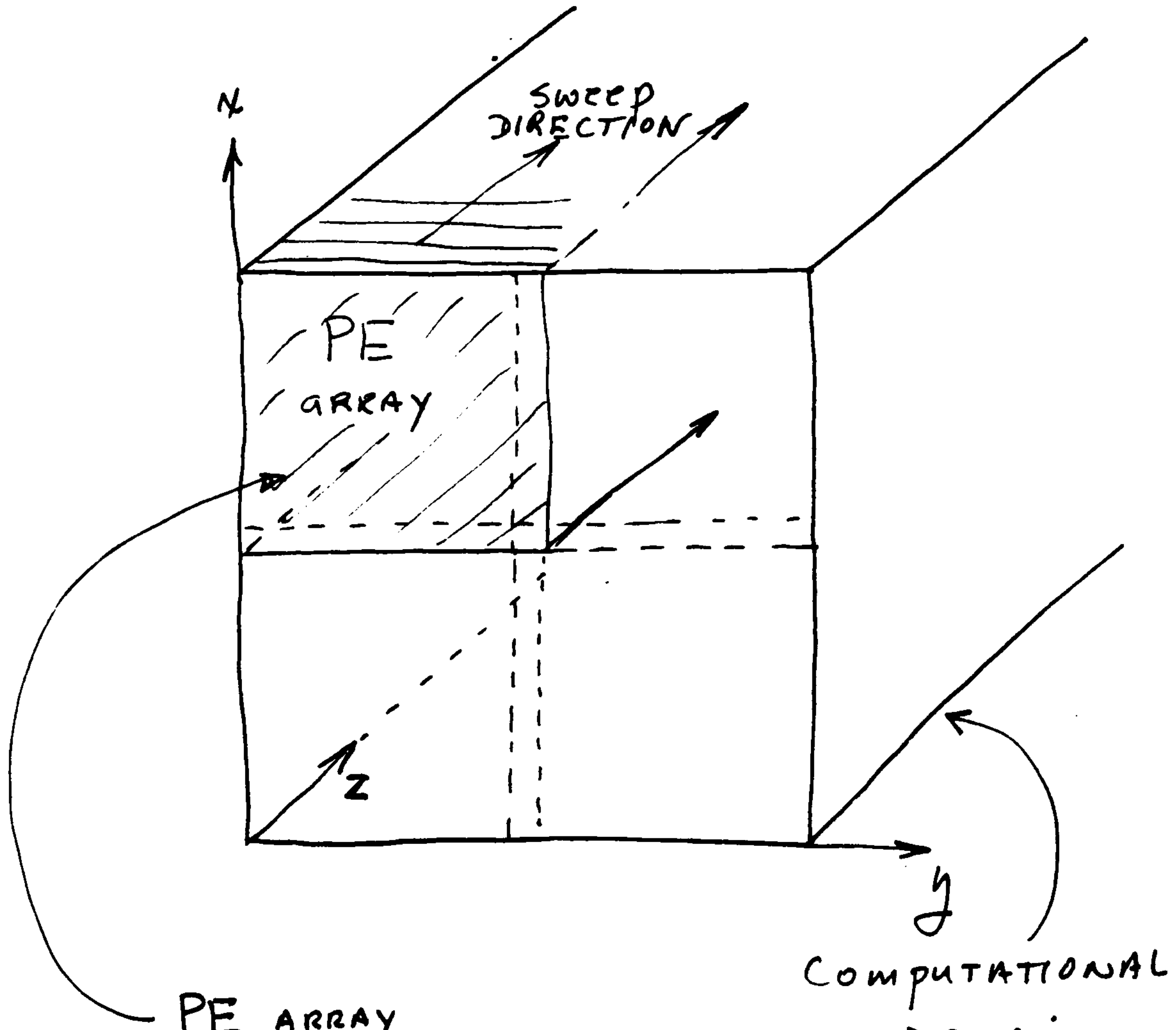
An implication of this scheme is that we must load four planes simultaneously no matter which direction we choose to go. For the computation of the previous section it means that we would have to load planes P_i, P_{i+1}, P_{i+2}, and P_{i+3} simultaneously. This causes no difficulty since the in the forward sweep all of these planes must eventually be loaded.

In the case of the explicit method we considered two problem sizes, that is, one problem for which the computational domain was "smaller" than the PE array and one for which the problem size "matched" the PE array size. In the former case we were able to maintain efficient use of the PE array by assigning PEs to 16 different cross-sections of the computational domain. This technique will not work in the implicit case since the forward sweep and back substitution algorithms are not easily partitioned--each iteration depends on data computed from the previous iterations. So in this case we would like the PE array size to "match" the problem size.

The other case to consider is when the computational domain is larger than the PE array. For example, what happens if the domain is 254 by 254 by *Kmax* and we are sweeping through the data in the z-direction? We envision the sweep being done in 4 phases where on each phase we sweep through one quadrant in the xy plane. The reason for 254 rather than the more usual 256 is that the sweeps must overlap by at least two grid points in order to carry out the computation for every interior line in the z-direction. In Figure 8 we depict these ideas.

5.4. Performance Estimates

The method used in our analysis of the explicit method was essentially "bottom up"—we made detailed estimates of the cost of the basic computational steps and then put these together to obtain an overall performance estimate. There is another approach which takes a more global view. In this approach we estimate the *utilization* of the PEs from the mapping of the problem onto the AU and the SM. If it appears that the PEs utilization is 85% on the average, then we can assume that the performance of the MPP will be approximately 85% of its peak rated performance. Looking back on our analysis of the explicit method, we could have arrived at the megaflop estimates more quickly by taking the latter approach.



PE ARRAY SWEEPING THROUGH COMPUTATIONAL DOMAIN IN THE Z-DIRECTION

COMPUTATIONAL DOMAIN

FIGURE 8

In the implicit method there a number of factors which affect the PE utilization. As we mentioned above, if the computational domain is smaller than the PE array then we will have idle PEs and thus low utilization. For example, a 32^3 computational cube allows no more than 6.23% utilization of the PEs whereas a 128^3 computational domain allows close to full utilization of the PEs. The mapping of the problem data into the SM can have an effect on the PE utilization. In this case we have found that there exists a layout of the problem data in the SM which permits almost all of the I/O transfers to be masked by PE computations. The one exception to this is the back substitution phase where the computation becomes I/O bound. This is mitigated somewhat by the fact that the disparity between the I/O and the computation is not too great. The most serious drawback of our proposed implementation of the block tridiagonal solver is the storage requirement. We estimate that we need 240 Mbytes of storage for intermediate results with a 128^3 computational domain. It is not surprising that this number is so large since we are solving 126^2 block tridiagonal linear systems simultaneously! A second SM just for intermediate results would serve well here.

The performance of the MPP on direct simulation of the Navier-Stokes equations using the implicit factored method can be quite good given a sufficiently large computational domain. If the computational domain is too small we loose utilization because the inherent parallelism is not sufficient to keep a large fraction of the PEs busy. As the computational domain grows we can approach the rated performance of the MPP so long as the intermediate storage problem can be handled. One way to do this is to have a second SM (256 Mbytes) to store intermediate results. Another way is to transfer this data to external disk storage. A disk subsystem has been designed built which could store the intermediate data and transfer it at sufficiently high data rates(100 Mbytes per second)[Hudg84].

6. Other Computational Requirements of CFD

In this section we address a number of additional issues which arise in CFD and which must be taken into account in the evaluation of the usefulness of the MPP architectural components. These issues are especially important in the application of numerical methods to problems of practical interest.

6.1. Grid Generation

The use of numerically-generated body-conforming grid systems has become well established in CFD. The objective of these techniques is to distribute the grid points throughout physical space in such a manner that they can represent the physical solution with sufficient accuracy and do this at minimal computational cost. The idea is to concentrate grid points in the regions where high resolution of the physical process is necessary.

The generation of a suitable grid system is done off-line along with the graphical generation of the physical objects to be placed in the flowfield. The effect of the grid system on the numerical computation is to introduce metric terms into the equations. These terms depend on the structure of the grid system. These metric terms imply more storage for problem data since each grid point in the computational domain must store its associated physical coordinates. They also imply more computation but, in general, they will not affect the efficiency of the computation on the MPP.

There is increasing interest in the dynamic coupling of the grid system to the physical solution. The idea is that as the solution evolves, the regions of interest in the solution field change dynamically and consequently the grid system should be able to adapt to these changes. The mathematics of adaptive grid generation controls the location of the grid points by sensing the gradients of the evolving physical solution. The requirements of smoothness, orthogonality, and accuracy are jointly optimized as the solution evolves. The optimized evolution of the grid system is determined by the solution of a partial differential equation which is coupled to the physical solution.

Adaptive grid generation is still an area of intense research and holds the promise of being able to enhance the effectiveness of numerical methods. Thus it is important to study the effect of dynamic grid generation techniques on the suitability of the MPP architectural components in CFD. At first glance, the methods of dynamic grid generation seem to fit into the framework of the computational schemes we have presented for the explicit and implicit numerical methods.

6.2. Zonal Methods

In this section we consider some of the implications of zonal methods on the effectiveness of the MPP architectural components.

In practical applications of CFD the geometrical configuration in the physical space is often too complex to be easily mapped into a box-like computational domain. In these cases the physical space may be partitioned into *zones* and within each of these zones a computational grid is generated. Another reason for partitioning the physical space into zones stems from the fact that the uniform modeling of the physics may be very wasteful of computer resources. There may also be some reason to use different numerical methods in different zones.

One effect of the partitioning of the physical space into zones is to introduce new boundaries where different zones meet. These boundaries are called *zonal boundaries* and, naturally, they introduce additional boundary conditions. Interestingly, these boundary conditions must be designed to eliminate the effect of the zonal boundaries on the solution. Schemes have been proposed for dealing with zonal boundaries[Rai84, Hess82]. Basically, these methods are designed to insure that the fluxes of the dependent variables are conserved across the zonal boundaries. The computations associated with zonal boundaries amount to the interpolation of the fluxes and dependent variables across the boundaries with the fluxes being interpolated in one direction and the dependent variables in the other. Clearly, these interpolations depend heavily on the way the grid systems in adjacent zones line up with one another at the zonal boundaries.

We assume that each zone is fitted with its own grid system. Within each zone we have a box-like computational domain with boundary conditions that arise either from the original problem or the zonal boundaries. At the zonal boundaries, grid systems from different zones abut. These abutments may be classified according to how well the grid points from each zone line up[Rai84]. The interpolation of fluxes and dependent variables at a zonal boundary depends on the structure of the interface between the two grid systems at a zonal boundary. The value of some quantity associated with a grid point on one side of a zonal boundary will depend on the values associates with a certain collection of grid points from the other zone. The zonal boundary computations are treated much like the usual boundary conditions with the exception that the some values come from computations being carried out in a different zone.

Zonal methods pose some problems for the MPP architecture. Additional boundary computations are introduced which could to limit the amount of inherent parallelism in the computation. Clearly, in two dimensions the zonal boundaries are one-dimensional and thus we can expect poor performance from the MPP. In three dimensions the zonal boundaries will be two-dimensional but they may have a complex structure. For example, a "face" of one zone may be part of two or more zonal boundaries. It seems possible that in the course of creating the zones we could perform some preprocessing which would simplify the computations at zonal boundaries making them more suitable to the MPP architecture. This deserves additional study.

The zonal boundary computations requires that values in one zone be made available to computations in another zone. This will add additional requirements on the layout of the problem data in the SM.

6.3. Equations of State

In this section we consider the problem of function evaluation using table lookup. This problem comes up when we replace complicated function evaluations with table lookup followed by interpolation. This is a classic example of trading space for time, i.e., we run faster at the expense of using additional space(the table). It also arises in attempting to express the properties of materials with an equation of state(EOS).

The Navier-Stokes equations do not form a closed system of equations unless we add to them an equation which supplies an additional relationship among the dependent variables. For example, there are five conservation equations but there are six dependent variables including p , the pressure. Thus we need the additional equation,

$$p = (1-\gamma)(e - 0.5\rho(u^2 + v^2 + w^2) .$$

This equation relates the pressure to the other five dependent variables thereby giving us a closed system of equations. It is the EOS which expresses the relationship between the variables which is dependent on the properties of the material. The EOS can be simple as the above or be some complex relationship requiring table lookup. It is well known that the evaluation of such functions can be a dominant part of certain computations[Dubo82, Jord82, Jr.83]. Much of the previous work addresses this problem in the context of vector processors.

In this section we raise the problem of table lookup in the context of the MPP architecture. A typical problem in CFD is to evaluate p in terms of the temperature t and the density ρ. This evaluation is to be done for each grid point in the computational domain where each grid point r supplies its own arguments $t(r),\rho(r)$. Typically, the argument values are used to locate a cell in the table from which certain parameters are extracted. These parameters are then used in an interpolation formula to arrive at the desired function value. We shall be assuming that each PE holds a pair of arguments. The first issue is whether the spacing in the table is arbitrary or not. If the spacing is arbitrary, then we will have to search the table to determine the appropriate indices and accordingly, some form of a parallel search must be developed. If the table is ordered then the PEs can compute their corresponding indices in parallel.

Once each PE has its local index into the table the next problem is to get the appropriate values to each of the PEs. This problem is nontrivial. Even if each PE had its own copy of the table(which is highly unlikely) we would require a local memory address register for each PE which currently is not the case in the MPP. What is more likely is that the table is too large to be duplicated among all the PEs and therefore we have to devise a scheme to somehow broadcast the appropriate data to each of the PEs.

It is clear from the above that the table lookup problem is alive and well in the context of the MPP and deserves careful consideration in the future.

6.4. Software Requirements

Digital Systems such as the MPP, which are comprised of several major components each with their own control unit, are nontrivial to "program." In the case of the MPP there are two major functional units, the AU and the SM. Additionally, there is an AU control unit, an I/O control unit, and the Main Control Unit(MCU). Each of the control units has its own machine language and the SM even has an associated software package, called the Staging Memory Manager, to assist the programmer. Clearly, this is the lowest level of control and thus affords the ultimate in flexibility in using the hardware. Unfortunately, the MPP is virtually unusable at this level of detail.

A Pascal compiler for the MPP which will generate code for the AU is under development. The initial specifications for the language did not include any mechanisms to deal with the SM, but we understand that some of the aspects of programming the SM are now being encorporated into the language. At the time of this writing the Pascal compiler has not been advanced to the point where it is a useful programming tool.

It is clear that the scientist has a language in which to express the physics of the problems at hand. What is needed is a careful study of the extent to which scientific notation in a particular discipline such as CFD can be mapped into appropriate algorithms and hardware structures. If we want to devise a hardware and software system which does CFD with superior ease and speed we can probably do so by limiting the range of computations that the system can do well. In the case of CFD, where we have a limited application area in mind, we can expect to have languages and software support systems that make it possible for the scientist to specify the physical problem at a high level and then have the software system manage the details concerning the mapping of the problem onto the hardware. For example, there will probably only be a limited number of different data layouts in the SM and a limited number of data transfer methods. All these different aspects of "programming" the SM should be worked out in advance.

It takes a lot of effort to implement a numerical algorithm on a machine like the MPP. However if the methods being implemented are central to CFD, then the investment in creating the software can be amortized over the (extensive) use of the code. Thus it becomes cost effective to conceive, design, and implement the software system along with the hardware system. All too often, a manufacturer will concentrate on the construction and delivery of a hardware system with little or no concern about the software and this results in the delivery of systems which are essentially not finished. Then we wonder why the device is not as useful as we had imagined—it is just too difficult to use.

7. MPP Architecture Extensions

The material for this section comes from a meeting at Goodyear Aerospace with the system architect and design engineers for the MPP. The purpose of the meeting was to explore the various way in which the MPP architectural components could be enhanced to achieve more storage capacity and faster processing speeds.

7.1. Storage

The current design of the PE's will permit addressing of up to 64K bits per PE. However, with the current technology there would be a memory cycle-time penalty for PE memory of 8K bits or more. This is because the larger chips have a slower access time. There are several ways in which this limitation could be avoided: (1) One could wait for technology to catch up. It is not unreasonable to expect that the memory access times will decrease with time. (2) One could increase the width of the data path to PE memory to 8 bits or more. In this case we can tolerate slower access times since we are fetching more bits with each access. Due to the speed requirements the PE memory size is dependent on developments in static RAM technology.

Current limit for the SM capacity is 64M bytes using 256K bit dynamic RAM chips. The same design can be pushed to 256M bytes by using the forthcoming 1M bit chips. The SM storage size depends on the dynamic RAM technology since the access speed to the individual chips is not the critical problem. The architecture of the SM compensates for the slower dynamic RAMs and so we can look to dynamic RAM technology for increasing the capacity of the SM.

Goodyear Aerospace has designed a disk subsystem (not built) with an adequate storage capacity and transfer rate[Hudg84]. The subsystem is capable of storing up to 39936 megabytes of data and can achieve a maximum transfer rate of 100 megabytes per second.

7.2. Speed

We discussed the possibility of increasing the speed of the PE's. Basically this is done by making it more like a CPU with wider internal data paths and with local register memory. A 10x design has already been sketched. This design leads to the following estimates:

	32 bits	40 bits
FPADD	32 cycles	40 cycles
FPMult	64 cycles	72 cycles
FPDiv	96 cycles	96 cycles ?

cycle = 100 nanoseconds

This will give a machine with approximately 10x the processing rate of the current machine and leads to a sustained processing rate of 2 or 3 gigaflops on certain CFD problems. With a faster PE we would also need faster PE memory access. It was thought that this might be accomplished by interleaving the PE local memory. This would probably help a great deal since these accesses tend to be sequential.

We speculated on the possibility of building a 100x faster PE and it was not thought to be out of the question.

Another desirable change to the current machine and any future incarnation would be to replace the MCU with a fast scalar and vector processor such as the proposed new machine from Convex which promises to be about 25% of a CRAY at a cost of 500,000 dollars.

7.3. Other Issues

We discussed the issue of the ultimate size of the PE array. It appears that the design is such that the PE array can easily be expanded to larger dimensions if this is necessary (a 256 x 256 array is possible). In this design we are assuming that the data path between PE's is one bit wide. This is needed to limit interconnects. The ultimate limitation on speed is directly related to interconnects between PEs that are located on different boards.

Smaller PE arrays are also possible with a 16 x 16 array probably fitting onto a single board. Multiple independent PE arrays are also possible along with multiple independent SMs. It is feasible to think of the AU and the SM as building blocks which can be connected together in different ways. For example, we could string together alternate AUs and SMs in a pipeline if this would help.

8. Conclusions

The MPP architecture seems suited to the numerical algorithms of CFD. The Array Unit with its 128 by 128 array of processing elements can be used effectively for both explicit and implicit numerical algorithms. Even more importantly, the Staging Memory has the flexibility to provide the different "views" of the problem data which are required by the numerical algorithms. There are additional issues in practical CFD such as adaptive grid generation, zonal boundaries, etc., which deserve additional study. There are additional numerical methods, such as the spectral methods, which should be measured against this architecture. It also appears that the capacity, speed and flexibility of the MPP architectural components can be substantially improved thereby offering the prospect of taking us into the gigaflop range for sustained computational rates.

Wind tunnels are systems which are designed to carry out a specific class of experiments. Even though a wind tunnel is a very expensive system to build, run, and maintain, no one seems too scandalized to learn that a wind tunnel cannot be used to accelerate electrons. It is tempting to speculate about designing a special-purpose hardware and software system to do CFD. It seems as though many of the numerical methods in CFD have matured to the point where we could select a limited but useful class of algorithms to implement. In doing so we enhance the chances of building a system which far outpaces anything that could be accomplished by a general-purpose system.

References

Aero83.
Goodyear Aerospace, "General Description of the MPP," GER-17140, April 1983.

Ames69.
W. F. Ames, *Numerical Methods for Partial Differential Equations*, Barnes and Noble, 1969.

Batc77.
Kenneth E. Batcher, "The Multidimensional Access Memory in STARAN," *IEEE Transactions on Computers*, February 1977.

Batc80.
Kenneth E. Batcher, "Design of a Massively Parallel Processor," *IEEE Transactions on Computers*, vol. C-29, no. 9, pp. 836-840, September 1980.

Batc81.
Kenneth E. Batcher, "MPP Staging Memory," Goodyear Aerospace GER-16964, March 1981.

Batc83.
Kenneth E. Batcher, "MPP Staging Memory Manager," Goodyear Aerospace GER-17062 Rev. 2, April 1983.

Beam76.
Richard M Beam and R. F. Warming, "An Implicit Finite-Difference Algorithm for Hyperbolic Systems in Conservation-Law Form," *Journal of Computational Physics*, vol. 22, pp. 87-110, 1976.

Beam78.
Richard 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.

Dubo82.
Paul F. Dubois, "Swimming Upstream: Calculating Table Lookups and Piecewise Functions," in *Parallel Computations*, ed. G. Rodrigue, 1982.

Hess82.
K. A. Hessenius and T. H. Pulliam, "A Zonal Approach to Solution of the Euler Equations," AIAA Paper 82-0969, St. Louis, MO, 1982.

Hudg84.
W. A. Hudgins, "MPP Disk Subsystem," Goodyear Aerospace GER-17234, March 1984.

Jord82.
T. L. Jordan, "A Guide to Parallel Computation and Some Cray-1 Experiences," in *Parallel Computations*, ed. G. Rodrigue, 1982.

Jr.83.T. R. Young Jr., "Table Lookup: An Effective Tool on Vector Computers," Proceedings of the Tenth Conference on Numerical Simulation of Plasmas, San Diego, CA, January 1983.

Loma82.
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.

MacC82.
R. W. MacCormack, "A Numerical Method for Solving the Equations of Compressible Viscous Flow," *AIAA*, vol. 20, no. 9, pp. 1275-1281, September 1982.

Pull80.
Thomas H. Pulliam and Joseph L. Steger, "Implicit Finite-Difference Simulations of Three-Dimensional Compressible Flow," *AIAA*, vol. 18, no. 2, pp. 159-167, February 1980.

Rai84.M. M. Rai, "A Conservative Treatment of Zonal Boundaries for Euler Equation Calculations," AIAA paper 84-0164, AIAA 22nd Aerospace Sciences Meeting, Reno Nevada, 1984.

Temp75.
Clive Temperton, "Algorithms for the Solution of Cyclic Tridiagonal Systems," *Journal of Computational Physics*, vol. 19, pp. 317-323, 1975.

Wray84.
Alan Wray, April 1984. NASA Ames, Personal Communication