NASA Contractor Report 181614

ICASE REPORT NO. 88-5

ICASE

IMPLEMENTATION AND ANALYSIS OF A NAVIER-STOKES ALGORITHM ON PARALLEL COMPUTERS

Raad A. Fatoohi

Chester E. Grosch

Contract No. NAS1-18107
January 1988

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

National Aeronautics and Space Administration

Langley Research Center
Hampton, Virginia 23665

IMPLEMENTATION AND ANALYSIS OF A NAVIER-STOKES ALGORITHM ON PARALLEL COMPUTERS

Raad A. Fatoohi and Chester E. Grosch

Abstract

This paper presents the results of the implementation of a Navier-Stokes algorithm on three parallel/vector computers. The object of this research is to determine how well, or poorly, a single numerical algorithm would map onto three different architectures. The algorithm is a compact difference scheme for the solution of the incompressible, two-dimensional, time dependent Navier-Stokes equations. The computers were chosen so as to encompass a variety of architectures. They are: the MPP, an SIMD machine with 16K bit serial processors; Flex/32, an MIMD machine with 20 processors; and Cray/2. The implementation of the algorithm is discussed in relation to these architectures and measures of the performance on each machine are given. The basic comparison is among SIMD instruction parallelism on the MPP, MIMD process parallelism on the Flex/32, and vectorization of a serial code on the Cray/2. Simple performance models are used to describe the performance. These models highlight the bottlenecks and limiting factors for this algorithm on these architectures. Finally conclusions are presented.

The first author is currently with Sterling Software, Inc., Palo Alto, CA 94303, under contract to the Numerical Aerodynamic Simulation Systems Division at NASA Ames Research Center, Moffett Field, CA 94035. The second author is with Old Dominion University, Norfolk, VA 23508. This research was performed under NASA Contract No. NAS1-18107 while the authors were in residence at ICASE, NASA Langley Research Center, Hampton, VA 23665.

I. Introduction

Over the past few years a significant number of parallel computers have been built. Some of these have been one of a kind research engines, others are offered commercially. Both SIMD and MIMD architectures are included. A major problem now facing the computing community is to understand how to use these various machines most effectively. Theoretical studies of this question are valuable [11], [13]. However, we believe that comparative studies, wherein the same algorithm is implemented on a number of different architectures, provide an equally valid way to this understanding. These studies, carried out for a wide variety of algorithms and architectures, can highlight those features of the architectures and algorithms which make them suitable for high performance parallel processing. They can exhibit the detailed features of an architecture and/or algorithm which can be bottlenecks and which may be overlooked in theoretical studies. The success of this approach depends on choosing "significant" algorithms for implementation and carrying out the implementation over a wide spectrum of architectures. If the algorithm is trivial or embarrassingly parallel it will fit any architecture very well. We need to use algorithms which solve hard problems which are attacked in the scientific and engineering community.

In this paper we present the results of the implementation of an algorithm for the numerical solution of the Navier-Stokes equations, a set of nonlinear partial differential equations. In detail, the algorithm is a compact difference scheme for the numerical solution of the incompressible, two dimensional, time dependent Navier-Stokes equations. The implementation of the algorithm requires the setting of initial conditions, boundary conditions at each time step, time stepping the field, and checking for convergence at each time step. Equally important to the choice of algorithm is the choice of parallel computers. We have chosen to work on a set of machines which encompass a variety of architectures. They are: the MPP, an SIMD machine with 16K bit serial processors; Flex/32, an MIMD machine with 20 processors; and Cray/2. In this paper we briefly describe the architecture of each of these computers. We also describe the programming languages which we used to implement the Navier-Stokes algorithm because the efficiency, or lack of it, of the

software can be as important as the computer architecture in determining the success or failure of the implementation. The basic comparison which we make is among SIMD instruction parallelism on the MPP, MIMD process parallelism on the Flex/32, and vectorization of a serial code on the Cray/2. The implementation is discussed in relation to these architectures and measures of the performance of the algorithm on each machine are given. In order to understand the performances on the various machines simple performance models are developed to describe how this algorithm, and others, behave on these computers. These models highlight the bottlenecks and limiting factors for algorithms of this class on these architectures. In the last section of this paper we present a number of conclusions.

II. The numerical algorithm

The Navier-Stokes equations for the two-dimensional, time dependent flow of a viscous incompressible fluid may be written, in dimensionless variables, as:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0, \tag{2.1}$$

$$\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y} = \zeta, \tag{2.2}$$

$$\frac{\partial \zeta}{\partial t} + \frac{\partial}{\partial x}(u\,\zeta) + \frac{\partial}{\partial y}(v\,\zeta) = \frac{1}{\mathrm{Re}}\nabla^2\,\zeta, \tag{2.3}$$

where $\vec{u} = (u,v)$ is the velocity, ζ is the vorticity and Re is the Reynolds number.

The numerical algorithm used to solve equations (2.1) to (2.3) was first described by Gatski, et al. [7]. This algorithm is based on the compact differencing schemes which require the use of only the values of the dependent variables in and on the boundaries of a single computational cell. Grosch [9] adapted the Navier-Stokes code to ICL-DAP. Fatoohi and Grosch [4] solved equations (2.1) and (2.2), the Cauchy-Riemann equations, on parallel computers. The algorithm is briefly described here.

Consider the problem of approximating the solution of equations (2.1) to (2.3) in the square domain $0 \le x \le 1$, $0 \le y \le 1$ with the boundary conditions $u = 1$ and $v = 0$ at $y = 1$ and $u = v = 0$ elsewhere. Subdivide the computational domain into rectangular cells. Fig. 1 shows a typical cell and the location of the variables on that cell where $U_{i,j} \equiv u\,(i\Delta x, j\Delta y)$, for $i = 1, 2, \cdots, N$; $j = 1, 2, \cdots, M$. Define the centered difference and average operators on a cell by:

$$\delta_x U_{i,j} \equiv (\, U_{i+1/2,j} - U_{i-1/2,j}\,) \,/\, \Delta x, \tag{2.4}$$

$$\mu_x U_{i,j} \equiv (\, U_{i+1/2,j} + U_{i-1/2,j}\,) \,/\, 2. \tag{2.5}$$

Suppose that $\zeta_{i+1/2,j+1/2}$ is prescribed. Then equations (2.1) to (2.2) are approximated by,

$$\delta_x U_{i+1/2,j+1/2} + \delta_y V_{i+1/2,j+1/2} = 0, \tag{2.6}$$

$$\delta_x V_{i+1/2,j+1/2} - \delta_y U_{i+1/2,j+1/2} = \zeta_{i+1/2,j+1/2}, \tag{2.7}$$

$$\mu_x U_{i+1/2,j+1/2} - \mu_y U_{i+1/2,j+1/2} = 0, \tag{2.8}$$

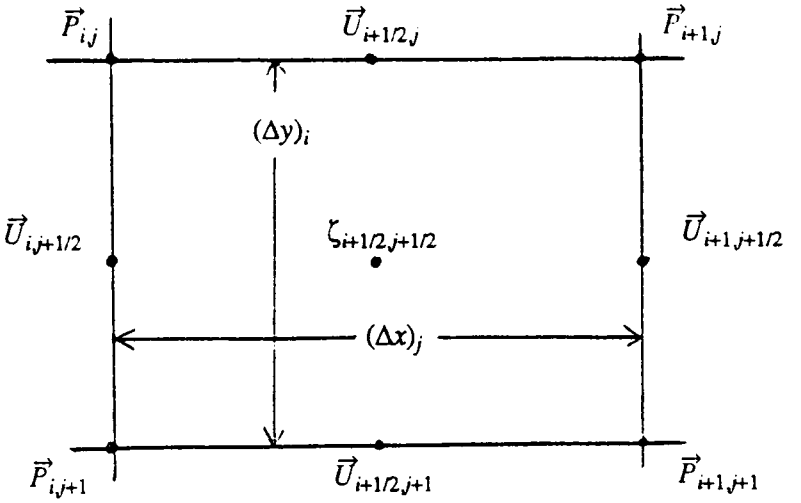

Fig. 1. Typical computational cell and the data associated with it.

$$\mu_x V_{i+1/2,j+1/2} - \mu_y V_{i+1/2,j+1/2} = 0. \tag{2.9}$$

The adaptation of this algorithm to different parallel architectures can be simplified by the introduction of box variables to represent $\vec{U}$. The center of a cell is at $(i+1/2,j+1/2)$. The box variables, $\vec{P}$, are defined at the corners of the cells, as shown in Fig. 1. They are related to $\vec{U}$ by:

$$\vec{U}_{i,j+1/2} = (\vec{P}_{i,j+1} + \vec{P}_{i,j}) / 2, \tag{2.10}$$

$$\vec{U}_{i+1/2,j} = (\vec{P}_{i+1,j} + \vec{P}_{i,j}) / 2, \tag{2.11}$$

and similarly for $\vec{U}_{i+1,j+1/2}$ and $\vec{U}_{i+1/2,j+1}$.

It is easy to see that equations (2.8) and (2.9) are satisfied identically for any set of box variables. For the cell $(i+1/2,j+1/2)$, equations (2.6) and (2.7) become,

$$AF = Z, \tag{2.12}$$

where

$$A = \begin{bmatrix} -\lambda_{i,j} & -1 & \lambda_{i,j} & -1 & \lambda_{i,j} & 1 & -\lambda_{i,j} & 1 \\ 1 & -\lambda_{i,j} & 1 & \lambda_{i,j} & -1 & \lambda_{i,j} & -1 & -\lambda_{i,j} \end{bmatrix},$$

$$F = (\vec{P}_{i,j} , \vec{P}_{i+1,j} , \vec{P}_{i+1,j+1} , \vec{P}_{i,j+1})^T,$$

$$Z = (0 , 2(\Delta y)_i \zeta_{i+1/2,j+1/2})^T,$$

$$\vec{P}_{i,j} = (P_{i,j} , Q_{i,j}),$$

$$\lambda_{i,j} \equiv (\Delta y)_i / (\Delta x)_j.$$

Equation (2.12) is solved by an iteration scheme which was originally proposed by Kaczmarz [12]. If $F^{(k)}$ is the value after the k'th iteration, then the residual after the k'th iteration, $R^{(k)}$, is given by:

$$R^{(k)} = AF^{(k)} - Z. \tag{2.13}$$

The next iteration is

$$F^{(k+1)} = F^{(k)} - \omega A^T (AA^T)^{-1} R^{(k)}, \tag{2.14}$$

where ω is an acceleration parameter. This relaxation scheme is equivalent to an SOR method.

The compact difference approximation to equation (2.3) results in an implicit set of equations which are solved by an ADI method. This method consists of two half steps to advance the solu-

tion one full step in time. Let Δt be the full time step and apply the forward difference operator to equation (2.3) for the time derivative, giving

$$\zeta_{ij}^{n+1/2} - \zeta_{ij}^{n} = \frac{\Delta t}{2}\left[\frac{1}{\mathrm{Re}}\delta_x^2\,\zeta_{ij}^{n+1/2} - \delta_x\,(U_{ij}\,\zeta_{ij}^{n+1/2}) + \frac{1}{\mathrm{Re}}\delta_y^2\,\zeta_{ij}^{n} - \delta_y\,(V_{ij}\,\zeta_{ij}^{n})\right], \tag{2.15}$$

$$\zeta_{ij}^{n+1} - \zeta_{ij}^{n+1/2} = \frac{\Delta t}{2}\left[\frac{1}{\mathrm{Re}}\delta_x^2\,\zeta_{ij}^{n+1/2} - \delta_x\,(U_{ij}\,\zeta_{ij}^{n+1/2}) + \frac{1}{\mathrm{Re}}\delta_y^2\,\zeta_{ij}^{n+1} - \delta_y\,(V_{ij}\,\zeta_{ij}^{n+1})\right]. \tag{2.16}$$

Applying the centered difference and centered second difference operators for the space derivatives, we get

$$\beta_{ij}^{(x)}\,\zeta_{i-1j}^{n+1/2} - (1 + 2\alpha_j^{(x)})\,\zeta_{ij}^{n+1/2} + \gamma_{ij}^{(x)}\,\zeta_{i+1j}^{n+1/2} = F_{ij}, \tag{2.17}$$

$$\beta_{ij}^{(y)}\,\zeta_{ij-1}^{n+1} - (1 + 2\alpha_i^{(y)})\,\zeta_{ij}^{n+1} + \gamma_{ij}^{(y)}\,\zeta_{ij+1}^{n+1} = G_{ij}, \tag{2.18}$$

where

$$F_{ij} = -\beta_{ij}^{(y)}\,\zeta_{ij-1}^{n} - (1 - 2\alpha_i^{(y)})\,\zeta_{ij}^{n} - \gamma_{ij}^{(y)}\,\zeta_{ij+1}^{n}, \tag{2.19}$$

$$G_{ij} = -\beta_{ij}^{(x)}\,\zeta_{i-1j}^{n+1/2} - (1 - 2\alpha_j^{(x)})\,\zeta_{ij}^{n+1/2} - \gamma_{ij}^{(x)}\,\zeta_{i+1j}^{n+1/2}, \tag{2.20}$$

$$\alpha_j^{(x)} = \frac{\Delta t}{2(\Delta x)_j^2\,\mathrm{Re}}, \tag{2.21}$$

$$\alpha_i^{(y)} = \frac{\Delta t}{2(\Delta y)_i^2\,\mathrm{Re}}, \tag{2.22}$$

$$\beta_{ij}^{(x)} = \alpha_j^{(x)} + \frac{\Delta t}{4(\Delta x)_j}U_{i-1j}, \tag{2.23}$$

$$\beta_{ij}^{(y)} = \alpha_i^{(y)} + \frac{\Delta t}{4(\Delta y)_i}V_{ij-1}, \tag{2.24}$$

$$\gamma_{ij}^{(x)} = \alpha_j^{(x)} - \frac{\Delta t}{4(\Delta x)_j}U_{i+1j}, \tag{2.25}$$

$$\gamma_{ij}^{(y)} = \alpha_i^{(y)} - \frac{\Delta t}{4(\Delta y)_i}V_{ij+1}. \tag{2.26}$$

for $i = 1, 2, \cdots, N$; $j = 1, 2, \cdots, M$. The velocity field is not defined at the corners of the cells in this scheme; however, it can be computed as follows:

$$\vec{U}_{ij} = \frac{1}{2}(\vec{U}_{i+1/2j} + \vec{U}_{i-1/2j}). \tag{2.27}$$

Using equation (2.11), to get

$$\vec{U}_{i,j} = \frac{1}{4}(\vec{P}_{i+1,j} + 2\vec{P}_{i,j} + \vec{P}_{i-1,j}). \tag{2.28}$$

Equation (2.17) represents a set of M independent tridiagonal systems (one for each vertical line of the domain) each of size N. Similarly, equation (2.18) represents a set of N independent tridiagonal systems (one for each horizontal line of the domain) each of size M.

The ADI method for equation (2.3) is applied to all interior points of the domain. The values of ζ on the boundaries are computed using equation (2.2). On $x = 0$, for example, we have

$$(\zeta = \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y})_{x=0} \tag{2.29}$$

but $u = 0$ at $x = 0$, so $(\partial u / \partial y)_{x=0} = 0$ and

$$(\zeta)_{x=0} = (\frac{\partial v}{\partial x})_{x=0}. \tag{2.30}$$

Assume that the length, Δx, of the first two cells bordering the boundary are equal and approximate the derivative of v, to get

$$(\frac{\partial v}{\partial x})_{x=0} \approx a_0\,(v)_{x=0} + a_1\,(v)_{x=\Delta x} + a_2\,(v)_{x=2\Delta x}. \tag{2.31}$$

Using the Taylor series for v at $x = \Delta x$, $2\Delta x$ about $x = 0$, we get

$$(\frac{\partial v}{\partial x})_{x=0} = a_0\,(v)_{x=0} + a_1\left\{(v)_{x=0} + \Delta x\,(\frac{\partial v}{\partial x})_{x=0} + \frac{1}{2}(\Delta x)^2(\frac{\partial^2 v}{\partial x^2})_{x=0} + O\left[(\Delta x)^3\right]\right\}$$
$$+ a_2\left\{(v)_{x=0} + 2\Delta x\,(\frac{\partial v}{\partial x})_{x=0} + \frac{1}{2}(2\Delta x)^2(\frac{\partial^2 v}{\partial x^2})_{x=0} + O\left[(\Delta x)^3\right]\right\}. \tag{2.32}$$

Solving equation (2.32) for the unknowns a_0, a_1, a_2, we have

$$(\zeta)_{x=0} = (\frac{1}{2\Delta x})\left[-3\,(v)_{x=0} + 4\,(v)_{x=\Delta x} - (v)_{x=2\Delta x}\right]. \tag{2.33}$$

The value of v on $x = 0$ is zero while its value on $x = \Delta x$, $2\Delta x$ is computed using equation (2.28). Similarly,

$$(\zeta)_{x=1} = (\frac{\partial v}{\partial x})_{x=1} \approx (\frac{1}{2\Delta x})\left[3\,(v)_{x=1} - 4\,(v)_{x=1-\Delta x} + (v)_{x=1-2\Delta x}\right], \tag{2.34}$$

$$(\zeta)_{y=1} = (-\frac{\partial u}{\partial y})_{y=1} \approx (\frac{1}{2\Delta y})\left[-3\,(u)_{y=1} + 4\,(u)_{y=1-\Delta y} - (u)_{y=1-2\Delta y}\right], \tag{2.35}$$

while

$$(\zeta)_{y=0} = (\frac{\partial v}{\partial x})_{y=0} \simeq (\frac{1}{2\Delta x})\Big[(v)_{y=0,\, x=\Delta x} - (v)_{y=0,\, x=2\Delta x} \Big]. \tag{2.36}$$

The key to the adaptation of the relaxation scheme for solving equations (2.1) and (2.2) to parallel computers is the realization that each $\vec{P}$ is updated four times in a sequential sweep over the array of cells. For example, $\vec{P}_{i,j}$ is changed during the relaxation of cell $(i-1/2,j-1/2)$ first, then of cell $(i-1/2,j+1/2)$, then cell $(i+1/2,j-1/2)$, and finally cell $(i+1/2,j+1/2)$. This fact is utilized on parallel computers by using the concept of reordering to achieve parallelism; operations are reordered in order to increase the percentage of the computation that can be done in parallel. The computational cells are divided into four sets of disjoint cells so that the cells of each set can be processed in parallel. It is therefore clear that the cell iteration for the box variables is a four "color" scheme. Also, different linear combinations of the residuals are used to update each $\vec{P}$ and all of the box variables are updated in each step. Thus the four steps are necessary for a complete relaxation sweep.

The main issue in implementing the ADI method for equation (2.3) on parallel computers is choosing an efficient algorithm for the solution of tridiagonal systems. The selection of the algorithm depends on the amount of hardware parallelism available on the computer, storage requirement, and some other factors. Two algorithms are considered here: Gaussian elimination and cyclic elimination. These algorithms are described in the literature, see [5], [11] for details. The Gaussian elimination algorithm is based on an LU decomposition of the tridiagonal matrix. This algorithm is inherently serial because of the recurrence relations in both stages of the algorithm. However, if one is faced with solving a set of independent tridiagonal systems, then Gaussian elimination will be the best algorithm to use on a parallel computer. This means that all systems of the set are solved in parallel. In this case we obtain both the minimum number of operations and the maximum parallelism. The cyclic elimination algorithm, also called odd-even elimination [10], is a variant of the cyclic reduction algorithm [11] applying the reduction procedure to all of the equations and eliminating the back substitution phase of the algorithm. This makes cyclic elimination most suitable for machines with a large natural parallelism, like the MPP.

The solution procedure for the Navier-Stokes equations can be summerized as follows:

step (1) Assume that ζ is zero everywhere at $t = 0$. The box variables for the relaxation process are initialized and their boundary values are computed.

step (2) The vorticity at the corners of the domain is not defined in this scheme. These values are approximated using the values of their neighboring points. The values of $\zeta_{i+1/2,j+1/2}$ are computed using the values of ζ at the corners of the cells.

step (3) The relaxation process is implemented by computing the residuals, $R^{(k)}$, using equation (2.13) for each set of cells, followed by updating the box variables using equation (2.14). This sequence must be completed four times in order to complete a sweep. Finally, the maximum residual is computed and tested against the convergence tolerance. The whole process is repeated until the iteration procedure converges.

step (4) The coefficients $\alpha_j^{(x)}$, $\alpha_i^{(y)}$, $\beta_{i,j}^{(x)}$, $\beta_{i,j}^{(y)}$, $\gamma_{i,j}^{(x)}$, $\gamma_{i,j}^{(y)}$ (equations (2.21) to (2.26)) for both passes of the ADI method are computed. This includes computing $\vec{U}_{i,j}$, equation (2.28).

step (5) The values of ζ on the boundaries, equations (2.33) to (2.36), are computed.

step (6) Solving tridiagonal equations distributed over columns, equation (2.17).

step (7) Solving tridiagonal equations distributed over rows, equation (2.18).

These steps were implemented using the following subprograms: *setbc*, step (1); *zcntr*, step (2); *relaxd*, step (3); *cof*, step (4); *zbc*, step (5); *triied*, step (6); and *trijed*, step (7). The repetition of steps (2) through (7) yields the values of the velocity and vorticity at any later time.

III. Implementation on the MPP

The Massively Parallel Processor (MPP) is a large-scale SIMD processor built by Goodyear Aerospace Co. for NASA Goddard Space Flight Center [1]. The MPP is a back-end processor for a VAX-11/780 host, which supports its program development and I/O needs.

The MPP has three major units: the Array Unit (ARU), Array Control Unit (ACU), and staging memory. The ARU is a square array of 128 × 128 bit-serial Processing Elements (PE's). Each PE has a local 1024 bit random access memory and is connected to its four nearest neighbors with programmable edge connections. Arithmetic in each PE is performed in bit serial fashion using a serial-by-bit adder. The ACU supervises the PE array processing, performs scalar arithmetic, and shifts data across the PE array. The staging memory is a large scale data buffer that stores, permutes, and transfers data between external devices and the array. The MPP has a cycle time of 100 nsec.

The MPP's high level language is MPP Pascal [8]. It is a machine-dependent language which has evolved directly from the language Parallel Pascal which is an extended version of serial Pascal with a convenient syntax for specifying array operations. These extensions provide a parallel array data type for variables to be stored in the array memory and operations on these parallel arrays.

The Navier-Stokes algorithm, described in section II, was implemented on the MPP using 127 × 127 cells (128 × 128 grid points). The computational cells are mapped onto the array so that each corner of a cell corresponds to a processor. The storage pattern used on the MPP is to store $\vec{P}_{i,j}$, $\zeta_{i+1/2,j+1/2}$, and $\lambda_{i,j}$ in the memory of the processor (i,j). The seven subprograms required to implement this algorithm (see section II) were written in MPP Pascal. These subprograms were executed entirely on the MPP; only the input and output routines were run on the VAX.

The relaxation process, subprogram *relaxd*, was implemented on the array by computing the residuals and updating the box variables for each set of cells separately. This sequence must be completed four times in order to complete a sweep. The only difference between these sequences

is the assignment of the data in a particular processor memory to one of the four corners of a computational cell. This is done using temporary parallel arrays and boolean masks which also masks out boundary values. Each sweep is followed by the computation of the maximum residual. The whole step is repeated until the process converges. Note that in updating the box variables for each set only one forth of the processors do useful work.

The ADI method, subprograms *triied* and *trijed*, was implemented on the array by computing the forcing terms $F_{i,j}$ and $G_{i,j}$, equations (2.19) and (2.20), and solving two sets of 128 tridiagonal systems, equations (2.17) and (2.18). The tridiagonal systems were solved by the cyclic elimination algorithm for all rows and all columns. This is done in parallel on the array with a tridiagonal system of 128 equations being solved on each row or column. After solving each set of these systems, all points of the domain are updated except the boundary points.

One of the problems in solving Navier-Stokes equations on the MPP is the size of the PE memory. The relaxation subprogram uses almost all of the 1024 bit PE memory; 22 parallel arrays of floating point numbers, all but 5 of which are temporary. Although the staging memory can be used as a backup memory, this causes an I/O overhead and reduces the efficiency of the machine. This problem was solved by declaring all of the parallel arrays as global variables and using them by different procedures for more than one purpose. This means that temporary arrays were redefined in different parts of the code. Beside this hardware problem, there are problems in using MPP Pascal to perform vector operations and to extract elements of parallel arrays. Operations on vectors are performed on the MPP by expanding them to matrices and performing matrix operations. This means that vector processing rate is 1/128 of that for matrix operations. MPP Pascal does not permit extracting an element of a parallel array on the MPP. This means that scalar operations involving elements of parallel arrays need to be expanded to matrix operations or they should be performed on the VAX.

The relaxation subprogram is quite efficient; almost all of the operations are matrix operations, no vector operations, only two scalar operations per iteration, and data transfers are only

between nearest neighbors. The ADI subprograms are reasonably efficient; mostly matrix operations with few scalar operations and no vector operations. However, the cyclic elimination algorithm has some hidden defects. For each level of the elimination process, a set of data is shifted off the array and an equal set of zeros is shifted onto the array. Since all of the processors are executing the same instruction every cycle, some of these processors may not be doing useful work; here they are either multiplying by zero or adding a zero. This is a problem with many algorithms on SIMD machines.

Table I contains the execution time for each subprogram of the Navier-Stokes algorithm, that for one iteration in the case of *relaxd;* the percentage of the total time spent in that subprogram; and the processing rate. The percentage of time spent in each subprogram determines which subprograms are using the most time for a given run. It is clear, from Table I, that the majority of the time was spent in *relaxd* for this particular run. This is because the average time step requires about 270 iterations and the total time spent in the other subprograms (*zcntr, cof, zbc, triied, trijed*) is only about the time to do two iterations of *relaxd.* The number of iterations in *relaxd* per time step depends on the particular data used during a given run. A different input data set could result in a smaller number of iterations per time step and relatively less time spent in the relaxation subprogram.

Table I. Measured execution time and processing rate of the Navier-Stokes subprograms for the 128 × 128 problem on the MPP.

Subprogram	Execution time (msec)	Percentage of time spent (%)	Processing rate (MFLOPS)
setbc	0.587	0.00	84
zcntr	2.694	0.06	24
relaxd	15.265*	99.23	156
cof	1.933	0.05	136
zbc	1.833	0.04	1.1
triied	12.717	0.31	125
trijed	12.725	0.31	125
overall#	41.597	100.00	155

* per iteration.

\# for ten time steps (execution time is in seconds for this row).

The processing rates in Table 1 are determined by counting only the arithmetic operations which truly contribute to the solution. Scalar and vector operations which were implemented as matrix operations are counted as scalar and vector operations. This is the reason why the subprograms *zbc* and *zcntr* have low processing rates; *zbc* has only vector operations while *zcntr* has some scalar operations implemented as matrix operations. The subprogram *setbc* has mostly scalar and data assignment operations which reduce its processing rate. Beside these three subprograms, the processing rate ranges from 125 to 155 MFLOPS with an average rate of about 140 MFLOPS.

In order to estimate the execution time of an algorithm on the MPP, the numbers of arithmetic and data transfer operations are counted and the cost of each operation is measured. This is illustrated in the following model. Only operations on parallel arrays are considered here.

The execution time of an algorithm on the MPP, T, can be modeled as follows:

$$T = T_{cmp} + T_{cmm}, \tag{3.1}$$

$$T_{cmp} = t_c \, (N_a \, C_a + N_m \, C_m + N_d \, C_d), \tag{3.2}$$

$$T_{cmm} = t_c \, (N_{sh} \, C_{sh} + N_{st} \, C_{st}), \tag{3.3}$$

where

T_{cmp} Computation cost,

T_{cmm} Communication cost,

t_c Machine cycle time = 100 nanoseconds,

N_a Number of additions,

N_m Number of multiplications,

N_d Number of divisions,

N_{sh} Number of shift operations,

N_{st} Number of steps involved in all shift operations,

C_a Number of cycles to add two arrays of 32 bit floating point numbers,

C_m Number of cycles to multiply two arrays of 32 bit floating point numbers,

C_d Number of cycles to divide two arrays of 32 bit floating point numbers,

C_{sh} Startup cost (in cycles) of shifting an array of 32 bit floating point numbers,

C_{st} Number of cycles to perform a one step shift within a shift operation.

Table II contains the measured values of C_a, C_m, C_{sh}, and C_{st}. These values were obtained by measuring the execution time of each operation using a loop of length 1000.

Table II. Measured execution times (in machine cycles) of the elementary operations on the MPP.

Addition	Multiplication	Division	One step shift	k step shift
965	811	1225	168	$136 + 32\,k$

Table III contains the operation counts per grid point for the Navier-Stokes subprograms on the MPP using the cyclic elimination algorithm for solving the tridiagonal systems. The arithmetic operations are counted in this table the way they were implemented; i.e., scalar and vector operations (in *zcntr* and *zbc)* which were implemented as matrix operations are considered here as matrix operations. Table IV contains the estimated computation and communication times of the Navier-Stokes subprograms using equations (3.2) and (3.3) and Tables II and III. The cost of scalar operations is not included in this model; this explains the differences between the estimated and measured times for *setbc* and *cof.* Beside these two subprograms, the difference between the total estimated times and measured times ranges between 3% to 8% of the measured times. The amount of time spent on data transfers is quite modest for these subprograms; from 6% for *relaxd* to 25% for *triied* and *trijed.* This is because the basic algorithm does not contain many data transfers and these transfers are only between nearest neighbors except for the tridiagonal solvers.

Table III. Operation counts per grid point for the Navier-Stokes subprograms on the MPP, using the cyclic elimination algorithm for solving the tridiagonal systems.

Subprogram	Addition	Multiplication	Division	Shift	Steps shifted
setbc	1	1	1	-	-
zcntr	15	9	-	19	28
*relaxd**	119	26	-	42	84
cof	8	8	-	8	8
zbc	5	7	4	8	11
triied	30	45	22	44	764
trijed	30	45	22	44	764

* per iteration.

Table IV. Estimated times (in milliseconds) of the Navier-Stokes subprograms on the MPP.

Subprogram	Computation time	Communication time	Total estimated time	Measured time
setbc	0.300	-	0.300	0.587
zcntr	2.177	0.348	2.525	2.694
relaxd	13.592	0.840	14.432	15.265
cof	1.421	0.134	1.555	1.933
zbc	1.540	0.144	1.684	1.833
triied	9.239	3.043	12.283	12.717
trijed	9.239	3.043	12.283	12.725

IV. Implementation on the Flex/32

The Flex/32 is an MIMD shared memory multiprocessor based on 32 bit National Semiconductor 32032 microprocessor and 32081 coprocessor [6]. The results presented here were obtained using the 20 processor machine that is now installed at NASA Langley Research Center.

The machine has ten local buses; each connects two processors. These local buses are connected together and to the common memory by a common bus. The 2.25 Mbytes of the common memory is accessible to all processors. Each processor contains 4 Mbytes of local memory. Each processor has a cycle time of 100 nsec.

The UNIX operating system is resident in processors 1 and 2. These processors are also used for software development and for loading and booting the other processors. Processors 3 through 20 run the Multicomputing Multitasking Operating System and are available for parallel processing.

The Flex/32 system software has a special concurrent version of Fortran 77. Concurrent Fortran comprises the standard Fortran 77 language and extensions that support concurrent processing. Among the constructs available for implementing parallel programs are: "shared", to identify variables that are shared between processors; "process", to define and start the execution of a process on a specified processor; "lock", to lock a shared variable if it is not locked by any other process; and "unlock", to release a locked variable.

The Navier-Stokes algorithm, described in section II, was implemented on the Flex/32 using 64×64 grid points (63×63 cells) and 128×128 grid points (127×127 cells). The main program as well as the seven subprograms of the algorithm were written in Concurrent Fortran.

The parallel implementation of the Navier-Stokes algorithm is done by assigning a strip of the computational domain to a process and performing all the steps of the algorithm by each process. The main program performs only the input and output operations and creates and spawns the processes on specified processors. In our implementation, we used 1, 2, 4, 8, and 16 processors of the machine. The domain is decomposed first vertically for the first six subprograms of the algorithm (*setbc, zcntr, relaxd, cof, zbc,* and *triied*) and then horizontally for the subprogram *trijed.*

For the first pass of the ADI method, subprogram *triied*, a set of tridiagonal equations corresponding to the vertical lines of the domain are solved while for the second pass of the method, subprogram *trijed*, a set of tridiagonal equations corresponding to the horizontal lines of the domain are solved.

Data is stored in the common memory, in the local memory of each processor, or in both of them. For the relaxation subprogram, data corresponding to $\vec{P}_{i,j}$, $\zeta_{i+1/2,j+1/2}$, and $\lambda_{i,j}$ for each strip are stored in the local memory while the interface points and the error flags are stored in the common memory. A copy of the matrix $Q_{i,j}$ is also stored in the common memory to be used in computing the matrix $V_{i,j}$, equation (2.28). The vorticity at the corners of the cells ($\zeta_{i,j}$), the velocity field ($\vec{U}_{i,j}$), and the coefficients of the tridiagonal equations ($\alpha_{i,j}^{(x)}$, $\alpha_{i,j}^{(y)}$, $\beta_{i,j}^{(x)}$, $\beta_{i,j}^{(y)}$, $\gamma_{i,j}^{(x)}$, and $\gamma_{i,j}^{(y)}$) are all stored in the common memory; this is required in order to implement the ADI method. The forcing terms and the temporary matrices for the tridiagonal solvers are stored in the local memory of each processor.

The relaxation scheme for each strip was implemented locally. After relaxing each set of cells, each process exchanges the values of the interface points with its two neighbors through the common memory. A set of flags are used here to ensure that the updated values of the interface points are used for the next set of cells. The tridiagonal equations were solved using the Gaussian elimination algorithm for both passes of the ADI method.

In order to satisfy data dependencies between segments of the code running many processes, a counter is used. This counter, which is a shared variable with a lock assigned to it, can be incremented by any process and be reset by only one process. It is implemented as a "barrier" where all processes pause when they reach it. A set of flags are also used, as described above, for synchronization in the relaxation subprogram.

Table V contains the speedups and efficiencies as functions of the number of processors for the 64×64 and 128×128 problems for two time steps. The measured execution times for ten time steps and processing rates for these problems using 16 processors are listed in Table VI. The

efficiency of the algorithm ranges from about 94%, for the 64 × 64 problem using 16 processors, to about 99%, for the 128 × 128 using two processors.

Table V. Speedup and efficiency of the Navier-Stokes algorithm on the Flex/32.

Number of processors	64 × 64 points		128 × 128 points	
	speedup	efficiency	speedup	efficiency
1	1.000	1.000	1.000	1.000
2	1.959	0.980	1.976	0.988
4	3.893	0.973	3.941	0.985
8	7.715	0.964	7.850	0.981
16	15.027	0.939	15.483	0.968

Table VI. Measured execution times for ten time steps and processing rates for the Navier-Stokes algorithm using 16 processors of the Flex/32.

Problem size (grid points)	Execution time (sec)	Processing rate (MFLOPS)
64 × 64	268.7	1.09
128 × 128	2587.1	1.13

The following performance model is based on estimating the values of various overheads resulting from running an algorithm on more than one processor. Also, the time to do the real computation on each processor is estimated.

The execution time of an algorithm on p processors of the Flex/32, T_p, can be modeled as follows:

$$T_p = T_{cmp} + T_{ovr}, \tag{4.1}$$

where T_{cmp} is the computation time and T_{ovr} is the overhead time. Let f_{ld} be a load distribution factor where $f_{ld} = 1$ if the load is distributed evenly between the processors and $f_{ld} > 1$ if at least one processor has less work to do than the other processors. Then the computation time on p processors can be computed by

$$T_{cmp} = f_{ld}\, T_1 / p, \tag{4.2}$$

where T_1 is the computation time using a single processor.

The overhead time can be modeled by:

$$T_{ovr} = T_{spn} + T_{cmo} + T_{syn}, \tag{4.3}$$

where

T_{spn} Spawning time of p processes,

T_{cmo} Total common memory overhead time,

T_{syn} Total synchronization time.

These times can be estimated as follows:

$$T_{spn} = p\ t_{spn}, \tag{4.4}$$

$$T_{syn} = p\ k_{lck}\ t_{lck}, \tag{4.5}$$

$$T_{cmo} = T_{cma} + T_{clm} + T_{cld}, \tag{4.6}$$

$$T_{cma} = n\ k_{cma} f_{bc}(p)\ t_{cma}, \tag{4.7}$$

$$T_{clm} = n\ k_{clm}\ (f_{bc}(p)\ t_{cma} + t_{lma}), \tag{4.8}$$

$$T_{cld} = n\ k_{cld}\ (f_{bc}(p)\ t_{cma} - t_{lma}), \tag{4.9}$$

where

T_{cma} Total common memory access time,

T_{clm} Total time required for copying shared vectors to local memory,

T_{cld} Total time difference between storing vectors in common and local memories; i.e., Overhead time of storing vectors in common memory instead of local memory,

t_{spn} Time to spawn one process; a reasonable value is 13 msec,

t_{lck} Total time to lock and unlock a shared variable; a reasonable value is 47 μsec,

t_{cma} Time to access an element of a vector in common memory; a reasonable value is 6 μsec,

t_{lma} Time to access an element of a vector in local memory; a reasonable value is 5 μsec,

k_{lck} Number of times a shared variable is locked and unlocked for each process,

k_{cma} Number of times a shared vector is referenced,

k_{clm} Number of times a shared vector is copied to local memory,

k_{cld} Number of times a vector is stored in common memory instead of local memory,

$f_{bc}(p)$ Bus contention factor. This contention results from having more than one processor trying to access the common memory at the same time; it is a function of p.

The values of t_{spn}, t_{lck}, t_{cma}, and t_{lma} are estimated based on timing experiments performed by Bokhari [2] and Crockett [3]. It is assumed that all common memory access operations are performed on vectors of length n.

The performance of the Navier-Stokes algorithm is heavily influenced by the performance of the relaxation subprogram; about 98% of the total time was spent in this subprogram for the two time step run. Since the number of cells is not divisible by the number of processors used, the last processor has less work to do than the other processors. Therefore, the load distribution factor, equation (4.2), can be computed by

$$f_{ld} = \left\lceil \frac{n-1}{p} \right\rceil \left(\frac{p}{n-1} \right). \tag{4.10}$$

Using the performance model, equations (4.1) through (4.10), the overhead time represents at most 5% of the execution time of the algorithm; this includes the impact of the load distribution factor on the computation time. The overhead time of the relaxation subprogram dominates the total overhead time. The values of k_{lck} and k_{cma} for each iteration of the relaxation process are 1 and 8. The spawning time has a minor impact on the overhead time because the processes are spawned only once during the lifetime of the program. The synchronization time is insignificant because the routines that provide the locking mechanism are very efficient and overlap with the memory access. The bus contention factor is very small even for a large number of processors. The common memory access time, T_{cma}, dominates the overhead time. The other components of the common memory overhead time, T_{clm} and T_{cld}, have a negligible impact on the total overhead time because these operations are performed only once during every time step.

V. Implementation on the Cray/2

The Cray/2 is an MIMD supercomputer with four Central Processing Units (CPU), a foreground processor which controls I/O and a central memory. The central memory has 256 million 64 bit words organized in four quadrants of 32 banks each. Each CPU has access to one quadrant during each clock cycle. Each CPU has an internal structure very similar to Cray/1 with the addition of 16K words of local memory available for storage of vector and scalar data. The clock cycle is 4.1 nsec.

The Navier-Stokes algorithm, described in section II, was implemented on one processor of the Cray/2 using 64×64 and 128×128 grid points. The codes were written and run through the CFT/2 compiler. The reordered form of the relaxation scheme, the four color scheme, was implemented on the Cray/2 with no major modifications. The reordering process removes any recursion because each of the four sets (colors) contains disjoint cells. The two sets of the tridiagonal systems were solved by the Gaussian elimination algorithm for all systems of each set in parallel. This was done by changing all variables of the algorithm into vectors running across the tridiagonal systems. The inner loops of all of the seven subprograms of the Navier-Stokes algorithm were fully vectorized.

The use of the main memory can be reduced by using scalar temporaries, instead of array temporaries, within inner DO loops. When scalar temporaries are used the CFT/2 compiler stores these variables in the local, rather than the main memory. This reduces memory conflicts and speeds up the calculation. The residuals in the relaxation process and the forcing terms in the ADI method are stored in the local memory.

Table VII contains the execution time for each subprogram of the algorithm, the percentage of the total time spent in that subprogram, and the processing rate for the 64×64 and 128×128 problems. As described in section III, most of the time was spent in *relaxd*. The average time step requires about 110 iterations for the 64×64 problem and about 270 iterations for the 128×128 problem. The subprogram *setbc* has a low processing rate because it has mostly memory access

and scalar operations; however, this subprogram is called only once during the lifetime of the program. Beside this subprogram, the processing rate ranges from 57 to 97 MFLOPS with an average rate of about 70 MFLOPS for the subprograms of both problems.

Table VII. Measured execution time and processing rate of the Navier-Stokes subprograms on the Cray/2.

Subprogram	64 × 64 grid points			128 × 128 grid points		
	Exec. time (msec)	Perc. of time (%)	Proc. rate (MFLOPS)	Exec. time (msec)	Perc. of time (%)	Proc. rate (MFLOPS)
setbc	0.480	0.02	25	1.651	0.01	29
zcntr	0.252	0.08	63	1.059	0.03	61
relaxd	2.719*	99.02	96	11.001*	99.60	97
cof	0.720	0.24	85	3.036	0.10	84
zbc	0.015	0.01	66	0.034	0.00	59
triied	1.007	0.33	57	4.014	0.13	59
trijed	0.928	0.30	62	3.870	0.13	62
overall#	3.048	100.00	96	30.286	100.00	97

* per iteration.

\# for ten time steps (execution times are in seconds for this row).

In order to estimate the cost of arithmetic and memory access operations on the Cray/2, the following timing values are used:

Clock Period (CP) = 4.1 nanoseconds,

Length of data path between the main memory and the registers, L_m = 56 CPs,

Length of each floating point functional unit, L_f = 23 CPs,

Data transfer rate with stride of 1 through main memory, R_1 = 1 CP/word,

Data transfer rate with stride of 2 through main memory, R_2 = 2 CPs/word.

A lower bound on the values of R_1 and R_2 is assumed here. Competition for memory banks from other processors causes a lower transfer rate and hence increased values of R_1 and R_2. The actual values are difficult to estimate.

Based on the fact that Cray vector operations are "stripmined" in sections of 64 elements, the time required to perform arithmetic and memory access operations on vectors of length L_{vcr} can be

modeled as follows:

$$T_{f1} = \left(\left\lceil \frac{L_{vcr}}{64} \right\rceil L_f + L_{vcr}\right) N_{f1}\ CP, \tag{5.1}$$

$$T_{f2} = \left(\left\lceil \frac{L_{vcr}}{128} \right\rceil L_f + \frac{L_{vcr}}{2}\right) N_{f2}\ CP, \tag{5.2}$$

$$T_{m1} = \left(\left\lceil \frac{L_{vcr}}{64} \right\rceil L_m + R_1\, L_{vcr}\right) N_{m1}\ CP, \tag{5.3}$$

$$T_{m2} = \left(\left\lceil \frac{L_{vcr}}{64} \right\rceil L_m + R_2\, \frac{L_{vcr}}{2}\right) N_{m2}\ CP, \tag{5.4}$$

where

T_{f1} Time to perform floating point operations on vectors with stride of 1,

T_{f2} Time to perform floating point operations on vectors with stride of 2,

T_{m1} Time to perform main memory access operations on vectors with stride of 1,

T_{m2} Time to perform main memory access operations on vectors with stride of 2,

N_{f1} Number of floating point operations on vectors with stride of 1,

N_{f2} Number of floating point operations on vectors with stride of 2,

N_{m1} Number of main memory access operations on vectors with stride of 1,

N_{m2} Number of main memory access operations on vectors with stride of 2.

Table VIII contains the operation counts per grid point for the Navier-Stokes subprograms using the Gaussian elimination algorithm for solving the tridiagonal systems. These operations are performed on all grid points of the domain except for *zbc* where they are performed on vectors. Tables IX and X contain the estimated times of the Navier-Stokes subprograms for the 64×64 and 128×128 problems. These times are obtained using equations (5.1) to (5.4) and Table VIII. The multiplication time includes the time required for division where it is assumed that each division takes three times the multiplication time. The main memory access time for each subprogram represents about 50% to 70% of the total estimated time and the measured time. This shows that

the Cray/2 is a memory bandwidth bound machine; the CPU clock period is 4.1 nsec while the main memory cycle time is approximately 240 nsec. The impact of memory access time for *relaxd* is also contributed to the use of a memory stride of 2 which causes more than a 50% slowdown in data transfer rate; this is a consequence of the four CPU's taking turns accessing the four quadrants of the main memory. The difference between the total estimated values and the measured values can be contributed to several reasons. Among these reasons are: the memory access and arithmetic operations can overlap, specially for large routines; the time to perform scalar operations is not included in this model; and there is up to 20% offset on the results depending on the memory traffic and the number of the active processes on the system.

Table VIII. Operation counts per grid point for the Navier-Stokes subprograms on the Cray/2, using the Gaussian elimination algorithm for solving the tridiagonal systems.

Subprogram	Addition	Multiplication	Division	Memory access
setbc	1	1	1	9
zcntr	3	1	-	5
*relaxd**	46	20	-	35
cof	8	8	-	16
zbc#	5	7	4	20
triied	6	7	2	17
trijed	6	7	2	17

* per iteration.

\# vector operations.

Table IX. Estimated and measured execution times (in milliseconds) of the Navier-Stokes subprograms for the 64 × 64 problem on one processor of the Cray/2.

Subprogram	Memory access time	Addition time	Multiplication time	Total estimated time	Measured time
setbc	0.277	0.022	0.089	0.388	0.480
zcntr	0.154	0.067	0.022	0.243	0.252
relaxd	1.783	1.206	0.551	3.540	2.719
cof	0.480	0.173	0.173	0.826	0.720
zbc	0.010	0.002	0.006	0.018	0.015
triied	0.510	0.130	0.281	0.921	1.007
trijed	0.510	0.130	0.281	0.921	0.928

Table X. Estimated and measured execution times (in milliseconds) of the Navier-Stokes subprograms for the 128 × 128 problem on one processor of the Cray/2.

Subprogram	Memory access time	Addition time	Multiplication time	Total estimated time	Measured time
setbc	1.120	0.090	0.360	1.570	1.651
zcntr	0.622	0.270	0.090	0.982	1.059
relaxd	7.218	4.144	1.802	13.164	11.001
cof	1.967	0.711	0.711	3.389	3.036
zbc	0.019	0.004	0.013	0.036	0.034
triied	2.090	0.533	1.155	3.778	4.014
trijed	2.090	0.533	1.155	3.778	3.870

VI. Comparisons and Concluding Remarks

There are a number of measures that one can use to compare the performance of these parallel computers using a particular algorithm. One is the processing rate and another is the execution time (see Tables I, VI, and VII). However it must be borne in mind that both of these measures depend on the architectures of the computers, the overhead required to adapt the algorithm to the architecture, and the technology, that is, the intrinsic processing power of each of the computers.

If we consider a single problem, a ten time step run of the Navier-Stokes algorithm on a 128×128 grid, then the processing rate is a maximum for the MPP, 155 MFLOPS, compared to 97 MFLOPS for the Cray/2, and only 1.13 MFLOPS on 16 processors of the Flex/32. The low processing rate of the algorithm on the 16 processors of the Flex/32 is simply due to the fact that the National Semiconductor 32032 microprocessor and 32081 coprocessor are not very powerful. Although the algorithm has higher performance rate on the MPP than on the Cray/2, it takes less time to solve the problem on the Cray/2 than on the MPP. This is due to the algorithm overhead involved in adapting the method to the MPP. As shown in Tables III and VIII, each iteration of the relaxation process has 145 arithmetic operations per grid point on the MPP compared to 66 operations per grid point on the Cray/2. Also, the cyclic elimination algorithm, used to solve tridiagonal systems on the MPP, has 97 arithmetic operations per grid point while the Gaussian elimination algorithm, used on the Cray/2, has only 15 operations per grid point.

The implementation of the algorithm on the Flex/32 has the same number of arithmetic operations per grid point as on the Cray/2; there is only a reordering of the calculations and no additional arithmetic operations in the overhead. The algorithmic overhead for the Flex/32 version is the cost of exchanging the values of the interface points and setting the synchronization counters for the relaxation scheme and accessing the common memory for the ADI method. This means that the code on each processor is the serial code plus the overhead code. When the code is run on one processor, it is just the serial code with the overhead portion removed.

Another measure of performance is the number of machine cycles required to solve a problem. This measure reduces the impact of technology on the performance of the machine. For the 128×128 problem, for example, the ten time step run requires about 416 billion cycles on the MPP, 7387 billion cycles on the Cray/2, and 25871 billion cycles on 16 processors of the Flex/32. This means that the MPP outperformed the Cray/2, by a factor of 18, and the latter outperformed the Flex/32, by a factor of 3.5, in this measure. This also means that one processor of the Cray/2 outperformed 16 processors of the Flex/32 even if we assume that both machines have the same clock cycle. The problem with the Flex/32 is that, although each processor has a cycle time of 100 nsec, the memories (local and common) have access times of about 1 μsec.

One simple comparison between the MPP and Cray/2 is the time to perform a single arithmetic operation using the models developed in sections III and V. Using equation (5.1), the time to perform a single floating point operation (addition or multiplication) on an array of size 128×128 elements on the Cray/2, excluding the memory access cost, is 91.3 μsec. The time to perform the same operation on the MPP using MPP Pascal, see Table II, ranges from 81.1 μsec (for multiplication) to 96.5 μsec (for addition). This shows that the processing power of a single functional unit of the Cray/2 is comparable to the processing power of the 16384 processors of the MPP. However, much of the overhead is not included in this comparison: memory access cost on the Cray/2, data transfers on the MPP, and so on.

This experiment showed that by reordering the computations we were able to implement the relaxation scheme on three different architectures with no major modifications. It also showed that two different algorithms, Gaussian elimination and cyclic elimination, were used to solve the tridiagonal equations on the three architectures; the two algorithms were chosen to exploit the parallelism available on these architectures. The algorithm exploits multiple granularities of parallelism. The algorithm vectorized quite well on the Cray/2. A fine grained parallelism, involving sets of single arithmetic operations executed in parallel, is obtained on the MPP. Parallelism at higher level, large grained, is exploited on the Flex/32 by executing several program units in parallel.

The performance model on the MPP was fairly accurate on predicting the execution times of the algorithm. The performance model on the Flex/32 showed the impact of various overheads on the performance of the algorithm. The performance model on the Cray/2 was based on predicting the execution costs of separate operations. This model is used to identify the major costs of the algorithm and reproduced the measured results with an error of at most 30%.

The ease and difficulty in using a machine is always a matter of interest. The Cray/2 is relatively easy to use as a vector machine. Existing codes that were written for serial machines can always run on vector machines. Vectorizing the unvectorized inner loops will improve the performance of the code. Unlike parallel machines, vector machines do not have the problem of "either you get it or not". The Flex/32 is not hard to use, except for the unavailability of debugging tools which is a problem for many MIMD machines (a synchronization problem could cause a program to die). On the other hand, the MPP is not a user-friendly system. The size of the PE memory is almost always an issue. MPP Pascal does not permit vector operations on the array nor does it allow extraction of an element of a parallel array. The MCU has 64 Kbytes of program memory. This memory can take up to about 1500 lines of MPP Pascal code. This means that larger codes can not run on the MPP. Finally, input/output is somewhat clumsy on the MPP. However, other machines with architectures similar to the MPP may not have the same problems that the MPP does.

There is one further observation of interest. This algorithm can be implemented concurrently on four processors of the Cray/2 (multitasking). The code will be similar to the Flex/32 version except that all of the variables should be stored in the main memory; the local memory on the Cray/2 is used only to store scalar temporaries. Adapting this algorithm to a local memory multiprocessor with a hypercube topology should be relatively easy. A high efficiency is predicted in this case because all data transfers are to nearest neighbors and their cost should be very small compared to the computation cost.

References

[1] Batcher, K. E., "Design of a Massively Parallel Processor," IEEE Trans. Computers, Vol. C-29, Sept. 1980, pp. 836-840.

[2] Bokhari, S. H., "Multiprocessing the Sieve of Eratosthenes," Computer, Vol. 20, No. 4, April 1987, pp. 50-58.

[3] Crockett, T. W., "Performance of Fortran Floating-Point Operations on the Flex/32 Multicomputer," ICASE Interim Rep. No. 4, NASA Langley Research Center, Hampton, VA, August 1987.

[4] Fatoohi, R. A. and Grosch, C. E., "Solving the Cauchy-Riemann Equations on Parallel Computers," ICASE Rep. No. 87-34, NASA Langley Research Center, Hampton, VA, May 1987.

[5] Fatoohi, R. A. and Grosch, C. E., "Implementation of an ADI Method on Parallel Computers," Journal of Scientific Computing, Vol. 2, No. 2, 1987, pp. 175-193.

[6] Flexible Computer Co., "Flex/32 Multicomputer System Overview," Publication No. 030-0000-002, 2nd ed., Dallas, TX, 1986.

[7] Gatski, T. B., Grosch, C. E., and Rose, M. E., "A Numerical Study of the Two-Dimensional Navier-Stokes Equations in Vorticity-Velocity Variables," J. Comput. Phys., Vol. 48, No. 1, 1982, pp. 1-22.

[8] Goddard Space Flight Center, "MPP Pascal Programmer's Guide," Greenbelt, MD, June 1987.

[9] Grosch, C. E., "Adapting a Navier-Stokes code to the ICL-DAP," SIAM J. Scientific & Statistical Computing, Vol. 8, No. 1, 1987, pp. s96-s117.

[10] Heller, D., "A Survey of Parallel Algorithms in Numerical Linear Algebra," SIAM Review, Vol. 20, No. 4, 1978, pp. 740-777.

[11] Hockney, R. W. and Jesshope, C. R., "Parallel Computers: Architecture, Programming and Algorithms," Adam Hilger, Bristol, England, 1981.

[12] Kaczmarz, S., "Angenaherte auflosung von systemen linearer gleichungen," Bull. Acad. Polon, Sci Lett. A, 1937, pp. 355-357.

[13] Ortega, J. M. and Voigt, R. G., "Solution of Partial Differential Equations on vector and Parallel Computers," SIAM Review, Vol. 27, No. 2, June 1985, pp. 149-240.

NASA

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-181614 ICASE Report No. 88-5		

4. Title and Subtitle	5. Report Date
IMPLEMENTATION AND ANALYSIS OF A NAVIER-STOKES ALGORITHM ON PARALLEL COMPUTERS	January 1988
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
Raad A. Fatoohi and Chester E. Grosch	88-5
	10. Work Unit No.
	505-90-21-01

9. Performing Organization Name and Address	11. Contract or Grant No.
Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225	NAS1-18107
	13. Type of Report and Period Covered
12. Sponsoring Agency Name and Address	Contractor Report
National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225	14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor:
Richard W. Barnwell

Submitted to 1988 International Conference on Parallel Processing

Final Report

16. Abstract

This paper presents the results of the implementation of a Navier-Stokes algorithm on three parallel/vector computers. The object of this research is to determine how well, or poorly, a single numerical algorithm would map onto three different architectures. The algorithm is a compact difference scheme for the solution of the incompressible, two-dimensional, time dependent Navier-Stokes equations. The computers were chosen so as to encompass a variety of architectures. They are: the MPP, an SIMD machine with 16K bit serial processors; Flex/32, an MIMD machine with 20 processors; and Cray/2. The implementation of the algorithm is discussed in relation to these architectures and measures of the performance on each machine are given. The basic comparison is among SIMD instruction parallelism on the MPP, MIMD process parallelism on the Flex/32, and vectorization of a serial code on the Cray/2. Simple performance models are used to describe the performance. These models highlight the bottlenecks and limiting factors for this algorithm on these architectures. Finally conclusions are presented.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
ADI method, parallel processing, parallel algorithms, performance analysis, SOR method	61 - Computer Programming and Software 64 - Numerical Analysis Unclassified - unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	31	A03