NASA Contractor Report 178034

ICASE REPORT NO. 85-61

NASA-CR-178034
19860009181

ICASE

ADAPTING A NAVIER-STOKES CODE TO THE ICL-DAP

Chester E. Grosch

Contract Nos. NAS1-17070 and NAS1-18107

December 1985

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

ADAPTING A NAVIER-STOKES CODE TO THE ICL-DAP

Chester E. Grosch
Old Dominion University
Norfolk, VA 23508

British Maritime Technology
Teddington TW11 OJJ, England

Institute for Computer Applications in Science and Engineering
Hampton, VA 23665-5225

Abstract

This is a report of the results of an experiment: to adapt a Navier-Stokes code, originally developed on a serial computer, to concurrent processing on the ICL Distributed Array Processor (DAP). In this paper the algorithm used in solving the Navier-Stokes equations is briefly described. The architecture of the DAP and DAP Fortran is also described. The modifications of the algorithm so as to fit the DAP are given and discussed. Finally, performance results are given and conclusions are drawn.

Research was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-17070 and NAS1-18107 while the author was in residence at ICASE, NASA Langley Research Center, Hampton, VA 23665-5225.

1. INTRODUCTION

The objective of the research reported here was to adapt a Navier-Stokes code, originally developed on a serial computer, to concurrent processing on the ICL DAP and to measure the performance of the concurrent version of the code.

Despite the fact that the most powerful existing computers can perform several hundred MFLOPS (million floating point operations per second), they are clearly inadequate for many important applications. In part, this inadequacy is due to the fact that it is generally rather difficult to fully use the vector capabilities of these "supercomputers". In practice, average processing rates for many codes are in the range of ten to twenty MFLOPS (see, for example, Dongarra, 1984). Even if the average processing efficiency could be increased so as to more nearly reflect the potential processing power of the vector processors, the processing rates would still be inadequate for many fluid dynamic applications.

Computational fluid dynamics requires both very large amounts of fast primary storage, even larger amounts of slow, secondary storage and high average processing rates. Different individuals and groups have independently estimated that current needs for some fluid dynamic calculations are, on average, 10^3 MFLOPS processing rates, 32×10^6 words of fast primary storage, and 2.56×10^8 words of slow secondary storage. Even higher average processing rates and larger primary and secondary storage will be needed in the near future.

It appears that future supercomputers will be multiprocessors, in fact some have already appeared. There are a number of important, unresolved questions concerning these multiprocessor computers. Among these issues are: should they consist of a few, rather powerful processors or many, very much less powerful processors, or something in between? Should the new computers be SIMD or MIMD? There is a natural expectation that the multiprocessors with a few, powerful processors will have an MIMD architecture and that the others will have

SIMD architectures. It seems sterile at this point to argue what, taking into account total processing power, cost, ease of programming, etc., is the "best" combination of number of processors and power per processor. Rather, carrying out experiments with existing multiprocessor computers would appear to be of greater value.

One approach to concurrent processing for computational fluid dynamics could be to devlop highly parallel algorithmic modules or kernals to perform certain computational tasks. Complete computational fluid dynamic algorithms would then be built by combining these modules. This approach has the advantage that "good", highly parallel algorithms well suited to the architecture would be used. It has the disadvantage that much of the algorithmic and code development effort of the past would be unusable and existing production codes would have to be abandoned. It would, concievably, be many years before new production codes would be available.

An alternate approach could be to translate effective algorithms, now embodied in codes for single processor computers, onto the muiltiprocessor computer. This has the advantage of using existing, tested algorithms. The disadvantage is that, because the algorithm may not "fit" the multiprocessor architecture, this may be very difficult or even impossible. If such translation is possible, the algorithm may not execute effectively on the multiprocessor, that is the computational cost may be very high.

There have been a substantial number of theoretical studies of the performance of algorithms on parallel computers but far fewer actual experimental studies (see Ortega and Voigt, [1985] for a comprehensive, up to date review). Even if one agrees that measured performance on an actual parallel processor is the true measure of computing power, there can be disagreements as to what performance is to be measured. The time to perform a

single arithmetic operation, together with the number of processors, can be used to calculate an upper bound on performance. Such a measure is widely held to be unrealistic because it does not include any of the omnipresent overhead.

A different approach is to use, as a test problem, the evaluation of a slightly more complex expression such as a vector dyad, the sum or product of two vectors or a vector triad, the sum of a vector and the product of two vectors, etc. (see Hockney and Jesshope [1981] and Hockney [1985] for examples). While certainly more realistic than using the single operation time, the use of these test problems can be criticized because they fail to measure the overhead associated with a complex algorithm and its embodiment in a specific program in a particular programming language. Thus, these types of tests can be said to give only a measure of the maximum performance of a microsegment of a code. The average performance of a complex scientific code is probably quite different, and less than that of some microsegment. Another approach is to run a specific algorithm on a particular computer. Despite the fact that this is a very specific experiment this approach has some advantages, namely it allows an objective measurement of the performance of a complete algorithm; albeit a specific one, expressed in a specific parallel language, and executed on a specific parallel processor; and it can also yield subjective evidence as to how well an algorithm fits the architecture, how difficult it was to program in the parallel language, and so on.

The work reported here was just such an experiment. A Navier-Stokes code, embodying a compact, finite difference form of a vorticity, velocity formulation of the Navier-Stokes equations (Gatski, Grosch, and Rose [1982]) was reprogrammed in DAP Fortran and run on the ICL DAP. Although the numerical scheme and the DAP have been described elsewhere, they are briefly described in Sections 2 and 3 for the sake of completeness. The implementation is described in Section 4. The results are given in Section 5. Finally, Section 6 contains

some concluding remarks.

2. THE ALGORITHM

The basic algorithm used here to solve the Navier-Stokes equations was first described, along with the results of some test cases, by Gatski, Grosch, and Rose [1982]. It has since been used by McInville, Gatski, and Hassan [1984] to study the instability and subsequent nonlinear evolution of a shear layer; by Gatski and Grosch [1984 a] to study the flow past an open cavity in a boundary; and finally is being used to study the separating flow past a backward facing step and the impulsive start of elliptic cylinders [Gatski, and Grosch, 1984 b]. As mentioned above it is briefly described here for the sake of completeness.

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

$$u_x + v_y = 0 , \tag{2.1}$$

$$v_x - u_y = \zeta , \tag{2.2}$$

$$\zeta_t + u\zeta_x + v\zeta_y = Re^{-1} (\zeta_{xx} + \zeta_{yy}), \tag{2.3}$$

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

The finite difference scheme used to approximate these equations is based on a compact difference method described by Rose [1981] and Philips and Rose [1982]. These schemes involve only variables within and on the boundaries of a single cell and are second order accurate, with the accuracy independent of the local cell Reynolds number. Apart from the independence of the accuracy on the magnitude of the cell Reynolds number, these compact schemes have certain other advantages in that it is quite simple and straightforward to use variable grids and to impose boundary conditions.

The compact difference approximation to equations (2.1) to (2.3) can be briefly described. Consider a rectangular cell, centered at (x_i,y_j), with

length Δx and height Δy, see Figure 1, and let $U^n_{ij} \equiv u(i\Delta x, j\Delta y, n\Delta t)$, for i, j, and n integers. Define the difference and average operators by:

$$\delta_x U^n_{i,j} \equiv (U^n_{i+1/2,\,j} - U_{i-1/2,\,j}) / \Delta x, \quad (2.4a)$$

$$\mu_x U^n_{i,j} \equiv (U^n_{i+1/2,\,j} + U^n_{i-1/2,\,j}) / 2. \quad (2.4b)$$

Then equations (2.1) to (2.2), hereafter the velocity equations, are replaced by:

$$\delta_x U^n_{i,j} + \delta_y V^n_{i,j} = 0 , \quad (2.5)$$

$$\delta_x V^n_{i,j} - \delta_y U^n_{i,j} = \zeta^{n,-1/2}_{i,j} \quad (2.6)$$

$$\mu_x U^n_{i,j} = \mu_y U^n_{i,j} , \quad (2.7)$$

$$\mu_x V^n_{i,j} = \mu_y V^n_{i,j} \quad (2.8)$$

The vorticity transport equation (2.33) is approximated by the system:

$$[\delta_t + (\mu_x U^n_{i,j})\delta x + (\mu_y V^n_{i,j})\delta_y]\zeta^n_{i,j} = Re^{-1}[\delta_x \phi^n_{i,j} + \delta_y \psi^n_{i,j}], \quad (2.9)$$

$$\delta_x \zeta^n_{i,j} = (\mu_x - 1/2\Delta x\, q(\theta_x)\delta_x)\phi^n_{i,j} , \quad (2.10)$$

$$\delta_y \zeta^n_{i,j} = (\mu_y - 1/2\Delta y\, q(\theta_y)\delta_y)\psi^n_{i,j} , \quad (2.11)$$

$$\mu_t \zeta^n_{i,j} = \mu_x \zeta^n_{i,j} = \mu_y \zeta^n_{i,j} . \quad (2.12)$$

Here, θ_x and θ_y are local cell Reynolds numbers given by

$$\theta_x = U^n_{i,j} Re\, \Delta x/2 , \quad (2.13)$$

$$\theta_y = V^n_{i,j} Re\, \Delta y/2 , \quad (2.14)$$

and $$q(\theta) \equiv (\coth \theta) - (1/\theta) . \quad (2.15)$$

As was shown by Gatski, et al [1982], the parameter $q(\theta)$ serves to reduce the truncation error. Of course $q(\theta)$ is not computed using the definition, equation (2.15), as that would be prohibitively expensive because of the cost of calculating $\coth(\theta)$. Instead the approximation

$$q(\theta) = \theta/3 \quad \text{for } \theta \text{ small}, \quad (2.16a)$$

$$= (\mathrm{sgn}\theta) - (1/\theta) \quad \text{for } \theta \text{ large}, \quad (2.16b)$$

is used, with $\mathrm{sgn}\theta \equiv \theta/|\theta|$.

The solution procedure for these finite difference equation is

Step (1) Assume that $\zeta_{i,j}^{n-1/2}$ is known. Then, with one component of $\vec{u}$ prescribed on the boundary, solve equations (2.5) to (2.8) by relaxation.

Step (2) Determine the vorticity boundary conditions. At inflow boundaries ζ is prescribed. At solid boundaries, an increment of vorticity is created so that the tangential velocity component will equal the speed of the boundary. At outflow and freestream boundaries the vorticity is determined by the flux condition,

$$\frac{\partial \zeta}{\partial t} + (\vec{u} \cdot \nabla) \zeta = 0. \qquad (2.17)$$

Step (3) Solve equations (2.9) to (2.11) using an ADI scheme, described in detail by Gatski, et al [1982]. Then use equation (2.12) to advance ζ in time, yielding $\zeta_{i,j}^{n+1/2}$.

Further details, as needed, will be given in Section 4, below, wherein the implementation on the DAP is described.

3. THE ICL DAP AND DAP FORTRAN

A brief description of the DAP architecture is given here in order to clarify the way of the Navier-Stokes code, described in Section 2, was adapted to the DAP architecture. A detailed description of the DAP is given by Hockney and Jesshope [1981] and in the ICL references which they cite.

The DAP is an SIMD computer with a processor array, a control unit, and an access control unit, see Figure 2. The complete processor is embedded within an ICL 2900 system and can be used as a conventional storage unit by the 2900. Access to the DAP is through the DAP access control unit. The 2900 initiates the execution of a DAP program by passing control to the DAP master control unit. Once this is done, the DAP retains control; the 2900 cannot interrupt program execution.

The DAP control unit is similar to the control unit of a conventional serial computer. There is an instruction counter, instruction register, instruction buffer, modifier register; and eight control registers. The instruction buffer can hold up to 60 instructions, each of which is 32 bits long. The modifier register acts as a base register for memory references and the control registers are used for data and/or instruction modifications. The DAP processes data in a bit serial mode. A 200 ns fetch cycle, in which two instructions are fetched, and a 200 ns execute cycle are required for an operation on a single bit. More complex operations such as integer addition, floating point addition, etc. are microprogrammed using the primitive hardware operations. This is facilitated by the use of a hardware loop capability, through the instruction buffer, which permits the rate of instruction execution to be reduced, asymptotically, to one per cycle. The execution time for typical high level instruction such as a floating point addition, is quite long, requiring from about ten to several hundred microseconds. The computational

power of the DAP results from the fact that the arithmetic is performed in a processor array, not a single processor.

The DAP processor array consists of 4096 single bit processors arranged in a 64 by 64 array. Each processor has a one-bit full adder, three one bit registers, an input multiplexer, an output multiplexer, as well as a 4096 bit memory. Thus the processor array has a total of 2 megabytes of memory. This memory is used to store both data and DAP code, so that in practice the theoretical maximum of 128 words, each of 32 bits, per processor is usually reduced to about 100 words of 32 bit data per processor. The DAP has three basic data modes; matrices which are 64 by 64 arrays, vectors having 64 or more elements, and scalers.

The storage of instructions and data can be easily visualized by considering the DAP memory as a three dimensional array of 64 rows, 64 columns, and 4096 planes. The 32 bit instructions are stored in planes with two per row. Matrices are stored on end, so to speak, with each plane containing one bit of the words. Thus matrices of 32 bit floating point words require 32 planes but logical matrices only a single plane. Vectors with 64 elements are, on the other hand, stored in one plane with one element per row. Scalers are treated similarly to vectors.

The DAP array has both global and local communication paths. The global paths are one bit wide row and column highways (buses). The single bit column highways interface with the DAP access controller and thus provide the data paths connecting the DAP to the 2900 host. The column highways are also connected to the instruction register and master control registers. Instructions are fetched from the DAP memory along the column highways and are sent to the instruction register. Data and instructions can also be moved along the column highways to the master control registers where they can be modified. In contrast the one bit wide row highways are only connected to the master

control registers and are only used to transmit data.

Apart from the data paths internal to the processor, each processor in the array is connected by four, one bit wide data paths to the four nearest neighbor processors. Thus instructions may refer to data elements in nearest neighbor processors. The edge connections can be defined by software in two ways; planer or cyclic. With a planer connection, data passed out at an edge is lost and input at an edge is defined to be zero. The cyclic connection defines, as nearest neighbors, the first and last elements of rows or columns or both.

ICL provides a DAP assembly language and a DAP Fortran, see ICL Manual, DAP: Fortran Language [1981]. DAP Fortran is an extension of standard Fortran and incorporates matrix and vector instructions. There is usually very little advantage to using the assembly language instead of DAP Fortran, as the Fortran routines are highly optimized and the overhead associated with them is only of the order of 10%. Some typical Fortran statements, their execution times, and processing rates are listed in Table 1.

The processing rate is found by dividing the number of operations, 4096 for matrices and 64 for vectors, by the execution time. Although this is quite straightforward for a simple arithmetic operation, such as the addition of two matrices, there is a difficulty when we consider data transfer operations. Traditional complexity analysis, that is operation counting, has ignored the cost (time) of data movement. While this may be justified for conventional architectures, it is certainly not always true for processor arrays such as the DAP. For example, Grosch [1979] has shown that the data transfer cost (time) can be the major portion of the cost in the implementation of relaxation and direct Poisson solvers on processor arrays with architectures similar to the DAP. Because of the importance, and possible dominance, of the data transfer cost, all data transfers are counted as floating point operations and the time

to complete them is included in the total operation time. Thus, the first Fortran statement, a matrix transfer, is counted as 4K floating point operations with an execution time of 17 μs, yielding a processing rate of 241 MFLOPS.

Logical matrices may be used as masks in arithmetic operations. The third Fortran statement in Table 1 is an example of this capability. The index, L, is a 64 by 64 logical matrix of one bit elements. The results of the right hand side arithmetic operation, here an addition, are stored in those elements of Z for which the corresponding elements of L are true.

DAP Fortran provides an extensive set of built-in functions, which belong to one of three types: computational, aggregate, and error management. The computational fuctions include most of the standard Fortran functions, ABS, SQRT, EXP, SIN, and so forth. The execution times and processing rates for some of these are listed in Table 1. These timings show the advantages of bit level processing. For example, a SQRT is calculated in only about 111% of an addition time because a bit level Newton interation scheme is used. Similarly, the EXP is evaluated in little more than one divide time. Although only examples of matrix functions are given in Table 1, the arguments of the computational functions may be of any mode (matrix, vector, or scalers) and the result will be of the same mode.

The aggregate functions perform nonnumeric operations, that is data manipulations, on vectors or matrices. Although generally not considered in standard complexity analysis, efficient implementation of the aggregate functions can be the "sine qua non" for efficient use of processor arrays. The implementation of this class of functions on the DAP is very efficient, for example the TRAN function, which transposes a matrix, is completed in about four addition times.

Mixed mode expressions, containing scalers and vectors or scalers and matrices, are permitted in DAP Fortran. The scalers are expanded to either

vectors or matrices in context. The vector-matrix combination is invalid because of the ambiquity in interpretation; the vector could be expanded either as a matrix of column vectors or of row vectors. The programmer must, in order to have a vector-matrix combination, use either MATC which generates a matrix of column vectors or MATR which generates a matrix of row vectors.

The MERGE function sets the elements of matrix Z equal to the corresponding elements of X, for those elements of L which are true, and to the corresponding elements of Y, for those elements of L which are false. This function can be used to implement conditional, data dependent calculations. For example, a branch instruction with two branches can be implemented by (1) calculating the results of the first branch and storing them in matrix X; (2) calculating the results of the second branch and storing them in Y; (3) generating the mask using the data test with, say, true denoting the first branch and false the second branch; (4) finally, calling MERGE.

By convention the first index of a matrix, the I of X (I,J), labels the rows and the second labels the columns, see Figure 3. The DAP convention labels the first row of the array as the North and the first column as the West. These conventions are used in the shifting functions, SHEP (X,N) for example. This function shifts (SH) the X matrix N columns to the East (E), ie the first column of X becomes the N+1 column. The P denotes a planer shift so that the last N columns of X are shifted out of the array and the first N columns are filled with zeros. If N is not given it will be taken to be one, and if N is greater than or equal to 64, the shift will be modulo 64. The SHEC function performs a cyclic shift in the East-West direction on the matrix. Single row or column shifts can also be accomplished by using the + or - notation. For example X(,-) is equivalent to SHEP(X). When using the $\pm$ notation the shifts are planer by default, but a GEOMETRY declaration can be used in any subroutine to declare the

East-West or North-South shifts, or both, to be cyclic.

The MAXV function returns the maximum element of X, with an option to test only those elements of X for which the corresponding elements of L are true. The use of bit level processing yields high efficiency for this function. Finally, the SUM function calculates the sum of all of the elements of X. It is noteworthy that this requires an execution time which is slightly less than two addition times, rather than the 12 ($4096 = 2^{12}$) that might have been expected.

Figure 4 shows, in block form the structure of a typical program. There can be more than one DAP entry subroutine in a Fortran program, as is shown in this figure. The call to the DAP entry subroutine must not have any arguments; all data is transferred via COMMON blocks. Immediately after entering the DAP entry routine and, again, before returning to the calling routine, the data in COMMON must be converted. This is because the data storage of standard Fortran is different from that of the DAP, as described above. The conversion routines, for example CONVFME (see Table 1), are extremely expensive in time and so the number of transfers between standard and DAP Fortran should be minimized. In particular, this affects the way in which input and output must be handled. Because of the difference in data storage format, there is no input or output from the DAP. The program must return from the DAP segment, at the cost of a conversion, perform the input and/or output and return to the DAP, at the cost of another conversion. Because of this overhead cost of conversion, input and output calls must be minimized. The data can be stored and output, say, done after final return from the DAP.

4. IMPLEMENTATION

The data structure of this algorithm, as presented in section 2, does not quite fit the DAP architecture. This is because the difference scheme is a compact one with the dependent variables defined on the cell edges rather than at the corners of a cell as in more familiar schemes. For an N by N array of cells there are N by N+1 cell sides and N+1 by N cell tops. So with the velocity, for example, defined on the cell edges the data arrays do not map directly onto an N by N array of processors.

On the other hand, there are certain advantages to using the compact difference scheme on the DAP. First, the application of boundary conditions is quite simple. There is no need to introduce "ghost" points outside of physical boundaries. Second, all derivatives are evaluated using variables which can be seen to be nearest neighbors on the computational grid (see Section 2). Thus, the amount of long range communication is minimized.

The adaptation of this algorithm to the DAP architecture can be simplified by the introduction of box variables to represent the velocity field. The center of a cell is at (i,j). The box variables, $\vec{P}$, are defined at the corners of the cells, points $(i\pm 1/2, j\pm 1/2)$. They are related to the velocity $\vec{U}$ by

$$\vec{U}_{i,j\pm 1/2} = (\vec{P}_{i+1/2,j\pm 1/2} + \vec{P}_{i-1/2,j\pm 1/2})/2 \tag{4.1}$$

$$\vec{U}_{i\pm 1/2,j} = (\vec{P}_{i\pm 1/2,j+1/2} + \vec{P}_{i\pm 1/2,j-1/2})/2 \tag{4.2}$$

It is obvious that equations (2.7) and (2.8) are satisfied identically for any set of box variables. For the cell (i,j), equations (2.5) and (2.6) are

$$A\vec{P} = Z \tag{4.3}$$

where $\vec{P}_{ij} = (P_{ij}, Q_{ij})^T$ (4.4)

and

$$P = \begin{pmatrix} \vec{P}_{i+1/2,\ j-1/2} \\ \vec{P}_{i+1/2,\ j+1/2} \\ \vec{P}_{i-1/2,\ j+1/2} \\ \vec{P}_{i-1/2,\ j-1/2} \end{pmatrix} \qquad Z = \begin{pmatrix} 0 \\ 2(\Delta y)_i \zeta_{i,j}^{n-1/2} \end{pmatrix} \qquad (4.5a,b)$$

$$A = \begin{pmatrix} \lambda_{i,j} & -1 & \lambda_{i,j} & 1 & -\lambda_{i,j} & 1 & -\lambda_{i,j} & -1 \\ 1 & \lambda_{i,j} & -1 & \lambda_{i,j} & -1 & -\lambda_{i,j} & 1 & -\lambda_{i,j} \end{pmatrix}, \qquad (4.6)$$

$$\lambda_{i,j} \equiv (\Delta y)i/(\Delta x)_j , \qquad (4.7)$$

is the aspect ratio of cell (i,j).

The box variables lie at the vertices of the computational grid. The storage pattern used on the DAP is to store $\vec{P}_{i-1/2,j+1/2}$, $\zeta_{i,j}$, and $\lambda_{i,j}$ in memory of processor (i,j). Thus with a 64 x 64 array of processors there is an array of 63 x 63 cells and for each cell we have equation (4.3).

The set of equations is solved by an iteration scheme which was originally proposed by Kaczmarz [1937] and generalized by Tanabe [1971]. If $P^{(k)}$ is the value after the k'th iteration, then the residual after the k'th iteration, $R^{(k)}$, is given by

$$R^{(k)} = AP^{(k)} - Z. \qquad (4.8)$$

The next iteration is

$$P^{(k+1)} = P^{(k)} - \omega A^T(AA^T)^{-1}R^{(k)}, \qquad (4.9)$$

which, if $\omega \equiv 1$, would give

$$R^{(k+1)} \equiv 0. \qquad (4.10)$$

For $\omega \neq 1$, this is an SOR scheme. On a serial computer the array of computational cells is swept over, applying equation (4.9) to each, until the maximum residual is reduced to the desired level.

The key to the adaptation of this relaxation scheme to the DAP is the realization that each $\vec{P}$ is updated four times in a sequential sweep over the array of cells. For example, see Figure 5, if the sweep is across the columns

and then down the rows, $\vec{P}_{i+1/2\ ,j+1/2}$ is changed during the relaxation of cell (i,j) first, then of cell (i,j+1), then cell (i+i,j), and finally cell (i+1,j+1). In each of these cases $\vec{P}_{i+1/2\ ,j+1/2}$ lies at a different corner of the cell being relaxed. It is therefore clear that the cell iteration scheme for the box variables is a four "color" scheme.

The DAP scheme is therefore to relax all of the $\vec{P}$'s four times; in the first pass a particular $\vec{P}$ is treated as lying in the lower right-hand corner of the cell and is labelled 1; in the next pass, labelled 2, as lying in the lower left-hand corner; in the third, labelled 3, as lying in the upper left-hand corner; and finally in the last pass, labelled 4, as being in the upper right-hand corner of the cell.

In detail, the DAP algorithm is implemented by

(1) Computing the residuals, $R^{(k)}$, for all cells using equation (4.8).

(2) Computing the correction to $P^{(k)}$ for all cells, as given in equation (4.9). Note that the coefficients in the matrices A and

$$B \equiv A^T (AA^T)^{-1} \tag{4.11}$$

are precomputed once and for all and are stored.

(3) Correct and restore the P's.

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 logical positions in a computational cell. This is easily done using logical masks which also mask out boundary values. The overhead caused by these data transfer is about 7% of the total.

Figure 6 shows the kernel of the DAP relaxation routine. The subroutine call statement, COMMON statements, declaration of temporary arrays, shifts of the data into temporary arrays, etc. are not shown in this figure. The TP's

TQ's , and R's are temporary matrices of real numbers. The P,Q, matrices are the first and second components of the box variables, and ZT is defined by equation (4.5). Finally the ZLT matrix contains the values of $\lambda_{i,j}$ and CT is a matrix of coefficients obtained from equation (4.11). The acceleration parameter, ω, is included in CT.

The residuals for all cells are computed in the first block of six DAP Fortran statements. These are stored in R1 and R2. The next set of four statements computes the terms in the correction, see equation (4.9), to the current values. The box variables are finally updated in the next eight statements.

Figure 7 show the DAP Fortran code used to compute the maximum residual. The residual matrices are zeroed, and the absolute values of the components of the residual vector are computed, as above. The matrix MR masks off the meaningless values generated at the bottom and right hand sides of the array. The R1 and R2 matrices are merged to form a matrix of maximum values. The MAXV function is then used to extract the maximum residual.

Apart from the branch instructions in the DO loops and GO TO, there are only three scaler operations in this subroutine. The first of these sets an error flag to zero, the next is used to test the maximum residual against the convergence tolerance. If the iteration does not converge in a specified number of interations, the last of the scaler operations sets the error flag to unity.

The calculation of the velocity field is the first major piece of the algorithm; the time stepping of the vorticity is the other. The first step in the vorticity calculation is to set the vorticity boundary conditions, as defined in Section 2. This relatively expensive because it involves only vector operations to generate the vectors of boundary values. A typical segment of the code is shown in Figure 8.

Here equation (2.6) is used to calculate the vorticity at a solid boundary at y = 0. It has been assumed that the heights, Δy, of the first two cells bordering the boundary are equal. This gives

$$\zeta \text{ (Boundary)} = (1/2\ u(x,2\ \Delta y)-2u(x,\Delta y)/\ \Delta y, \qquad (4.12)$$

The fact that the velocity is the average of two adajacent box variables has also been used to simplify the coefficients in this piece of code. In this figure U2, U3, and ZBOT are 64 element vectors. ZBOT is the vector of boundary values of the vorticity. Note that U2 and U3 are formed by the reduction operation of extracting a row from a matrix. In contrast, VEC(DY(63)) forms a vector by expanding the scaler DY(63). Similar code segments involving vectors are used to calculate the boundary values of the vorticity on the other three boundaries.

Once the boundary values of the vorticity have been calculated, one can proceed to the solution of the advection diffusion equation for the vorticity, equations (2.9) to (2.15). As was discussed by Gatski, Grosch, and Rose [1982] and Gatski and Grosch [1985], $\zeta^{n+1/2}$ is eliminated between equations (2.9) and (2.12). This gives an implicit system for ζ^n, which is then solved by an ADI method. In the first pass one solves for the set $\{\zeta^n_{i+1/2,j},\ \phi^n_{i+1/2,j}\}, i = 0,1,\ldots$ for all j, using the values of $\zeta^{n\ -1/2}_{i,j}$. In the second pass the set $\{\zeta^n_{i,j+1/2},\ \psi^n_{i,j+1/2}\}$, j = 0,1... for all i, is solved for. Using these values and $\zeta^{n\ -1/2}_{i,j}$, we solve for the next approximation to $\{\zeta^n_{i+1/2,j},\ \phi^n_{i1/2,j}\}$, and so on. In the first pass the set $\{\zeta^n_{i+1/2,j}\}$ is the solution of a tridiagonal system, ie,

$$a_{i,j}\ \zeta^n_{i-1/2,j} + b_{i,j}\ \zeta^n_{i+1/2,j} + c_{i,j}\ \zeta^n_{i+3/2,j} = q_{i,j} \qquad (4.13)$$

for i = 1,2..., and fixed j. Taking j = 1,2... gives a set of N tridiagonal equations, each for N variables. Next we solve for the set $\{\phi_{i+1/2,j}\}$. The next pass requires the solution of a similar set of tridiagonal systems for

$\{\zeta^n_{i,j+1/2}\}$, then solving for $\{\psi^n_{i,j+1/2}\}$. One then repeats this sequence.

It can be seen that the coefficients $\{a_{ij}, b_{ij}, c_{ij}\}$, for both passes are functions of the velocity and the parameters $(\Delta t/\Delta x)$, $(\Delta t/Re\ \Delta x^2)$, and the cell Reynolds numbers. The forcing terms, the $\{q_{ij}\}$, must be recomputed after each pass. For example, when solving for the $\{\zeta^n_{i+1/2,j}\}$, the q's are functions of the $\{\zeta^n_{i,j+1/2}, \psi^n_{i,j+1/2}\}$. Once the vorticity at time level $n\Delta t$ is computed using this ADI scheme, the time advance of the vorticity can be the fact that the time average equals the space average, completed by using see equation (2.12).

Each pass of the implicit ADI step requires that the coefficients and forcing terms of the tridiagonal systems be computed. This can be done concurrently for all N systems, ie all systems in rows or all systems in columns. A portion of the DAP Fortran code to accomplish this is shown in Figure 9. The average velocity in the cells is computed in the first four statements. The next seven statements calculate various terms needed to compute the a's, b's, and c's. They are combined in the next block of four statements to form α^-, β^-, α^+, and β^+. The next three statements compute the remaining temporaries required to set the forcing terms, the q_{ij} of equation (4.13), for all of the equations. Note that two of these statements evaluate the advective and diffusive terms for the orthogonal direction. These terms are then combined to form the q_{ij} matrix. But note that two additional vector operations are required to set the boundary conditions. Finally, the α's and β's are combined to give the coefficient matrices for the tridiagonal systems.

The block of code given in Figure 9 constitutes one pass in one direction and, as such, is about one quarter of the total amount of code to implement the ADI scheme for equations (2.9) through (2.15). This segment of DAP Fortran, to compute the vorticity at the tops and bottoms of cells at time level n, is not completely optimized. There are several uses of the MATC and MATR functions to

expand vectors to matrices. These matrices could have been precomputed and stored, but it was decided to minimize the storage used at the cost of a few expansions and matrix arithmetic operations. It is also possible, by judicious redefining of some of the temporary matrices, to save two or three matrix additions in this code segment. However it was decided to forego this in the interest of clarity of presentation.

There are two matrix functions in the DAP Fortran code shown in Figure 10. The first of these is QTHETD. The function implements the $q(\theta)$ function as defined in equations (2.16 a,b). It is programmed using the MERGE for evaluating a two branch function as described in Section 3. The second matrix function is TRIIED, the tridagonal solver.

The DAP Fortran routine TRIIED is given in Figure 9. There are 64 sets of 64 equations for the vorticity values on the tops of the cells. These equations are solved by the cyclic elimination algorithm, which is the cyclic reduction algorithm (see Hockney and Jesshope, [1981] applied to all of the equations. This eliminates the back substitution phase of the reduction algorithm.

This subroutine is reasonbly efficient but does contain some hidden defects which are inherent in the algorithm. The shift parameter, K, takes on the successive values 2^k for $k = 0,1,\ldots,5$, so that in each pass 2^k rows of data are shifted off the array and, more importantly, 2^k rows of zeros are shifted onto the array. Thus, although all of the processors are active all of the time, some of these processors are not doing useful work because they are multiplying by zero or adding a zero. This must be taken into account in any fair calculation of processing rate.

The final calculation in a time step is the computation of the total energy, vorticity and enstrophy (the squared vorticity), as diagnostic measures. This is a simple, straight-forward computation using the SUM function. Details will not be given.

5. Results

The numerical algorithm, the DAP architecture and DAP Fortran, and the adaptation of the algorithm to this architecture were presented in previous sections. Timing and performance results are given in this section.

The host program is a standard Fortran program in which COMMON blocks containing box variables, P and Q, the vorticity, the grid sizes, etc. are defined. The host program also handles the input and output, initialization of variables and the calling of the DAP Entry routine. There is only a small amount of computation in the host program and therefore the amount of time spent in this section is insignificant. This is evident once it is realized that the DAP Fortran section is run for a hundred, perhaps several hundred, time steps in a typical case.

In the DAP Entry routine all of the variables which were passed through COMMON are converted from Fortran to DAP format. This is a fairly costly operation: a matrix conversion takes 2252 μs, and a vector or scaler conversion about 50 μs. In total, 7 matrices, 28 vectors, and 23 scalers are converted on entry to the DAP program. This takes 18.3 ms which is a little over 100 addition times. This is a significant overhead only if it is repeated every time step. Because tens, or hundreds, of time steps are taken for each call to the DAP Entry routine, this overhead, and that of conversion upon leaving, is quite unimportant. The only other initialization task in this entry routine is to generate the one bit logical masks. The total time for this is only about 150 μs; less than one addition time. Again this is negligible.

From this discussion it should be clear that the only substantial costs are those associated with the application of the numerical algorithm for each time step. The total overhead in the host Fortran program and in the data conversions in the DAP Entry routine can be shown to be less than the cost of three iterations in the relaxation routine. In the calling sequence in the

Entry routine there are only four tests on flag variables and seven scaler arithmetic operations per time step and the time to do this is less than one fifth of that to carry out one relaxation iteration; a typical time step would require about fifty iterations.

The required sequence of calls to the various DAP Fortran routines in order to advance the Navier - Stokes algorithm one time step is: RELAXD (solves for the velocity field by relaxation), FIXZBD (calculates boundary values of the vorticity), ZCALCD (advances the vorticity in time); and FNORMD (calculates the total energy, vorticity, and enstrophy of the field). In ZCALCD there are calls to the matrix functions discussed previously, QTHETD and TRIIED. The fuction TRIIED solves tridiagonal equations distributed over columns. There are also calls in ZCALCD to the matrix function TRIJED which solves tridiagonal equations distributed over rows.

Table 2 contains a list of these subprograms; the execution time for each, that for one iteration in the case of RELAXD; and the processing rate. The processing rate is determined by taking the ratio of the number of effective arithmetic operations to the total execution time for the subprogram. In counting the number of effective arithmetic operations only those operations which truly contribute to the solution are counted. In those cases for which some of the processors are not performing useful work, as in TRIIED as described previously, those processors are not counted. Data transfer operations, vital as they are, were not counted as floating point operations. However, the time required to do these transfers was included in the total execution time.

One can see that there is a wide range in both execution time, from 1 to over 200 msec, and processing rates, 0.3 to 20 mflops, for these subprograms. The amount of time spent on data transfers is quite modest for most of the subprograms; in general 2% to 7% except for TRIIED and TRIJED.

There are two reasons why the data transfer overhead is generally so

unimportant. The first is that the basic algorithm does not contain many data transfers and these transfers are only between nearest neighbors. The local nature of the data transfers is due to the fact that the differencing scheme is a compact one. The second reason is that the DAP has very efficient data transfer hardware, yielding rates in the neighborhood of 200 x 10^6 words per second. In contrast to most of the subroutines, 27% of the execution time is used by data transfers in the tridiagonal solvers. These subroutines are mostly data transfer operations, in fact only 18% of the operations are arithmetic operations, but these transfer operations take up only about a fourth of the time because of the efficient implementation of data movements on the DAP.

The subroutine FIXZBD, used to calculate the boundary values of the vorticity, is very inefficient with a processing rate of only 0.3 mflops. This is because there are only vector operations in this subroutine and the vector processing rate is 1/64 of that for matrix operations. Fortunately, the execution time of FIXZBD is a small fraction, approximately 0.5%, of the total execution time if a time step requires only one iteration and about 0.04% of the total execution time if there are 100 iterations per time step. It is obvious that: FIXZBD uses only vector operations; has a very low efficiency; and has a negligible effect on the overall efficiency of the code. It would seem that this conclusion concerning the setting of boundary conditions is independent of the structure of the algorithm.

Subroutine ZCALCD, within which the vorticity is advanced in time, consists of two principal operational parts: the generation of the coefficients for the tridiagonal systems and the solution of these tridiagonal systems. Within this subroutine there are a number of vector and scaler operations, but the time required to evaluate them is only about 14% of the total. Generation of the coefficients for the tridiagonal systems takes about 42% of the execution time so that even if the tridiagonal solvers ran in zero time the execution time

would only decrease by a factor of two and the processing rate double. The effective processing rate of 10 megaflops for this routine is due to, first the vector and scaler operations required to generate the coefficients of the tridiagonal systems and, second, the large number of data transfers required in the tridiagonal solvers. The average processing rate is approximately half the maximum possible rate. Given the necessity of treating boundaries and solving the tridiagonal systems, there doesn't seem to be any more efficient way to handle these problems than that adopted here.

The overall processing rate depends on the number of iterations per time step. It is clear that if the physics of the flow is such that the field is evolving rapidly then, with a modest size time step, there will be a large change in the velocity per time step. This will require a substantial number of iterations in RELAXD per time step. The alternative is, of course, to choose a small enough Δt so that the number of iterations is small. Considering that the relaxation routine in the most efficient of all of the routines, extra iterations for a larger time step would appear to be a reasonable trade off.

This is apparent when one considers some reasonable scenarios. If there is one iteration per time step, the execution time per step is 0.26 seconds and the processing rate is 11.3 mflops. If Δt were increased and there were 100 iterations per time step then the execution time would increase to 3.25 seconds per time step, but the processing rate would also increase to 19.5 mflops.

There is one further observation of interest. The DAP Fortran code reveals blocks of DAP instructions whose structure suggests a way in which this algorithm might be adapted to an MIMD machine. On an MIMD machine, there would appear to be a basic requirement for a "good" program: that there be reasonably large blocks of code which do not require sychronization or have likely memory conflicts. The size of the code blocks is to be intrepreted to mean the product of, say, the numbers of arithmetic operations per grid point and the number of

grid points in the field.

Consider, first, the relaxation routine. This can be partitioned in at least three ways; geometrically, by color, and in combination. One could split the array of cells into subarrays and relax these without synchronization. This would be a form of chaotic block relaxation. This would probably converge under some conditions, and deserves some study. Color partitioning among four processors would be a simple extension of the algorithm which was implemented on the DAP. A combined scheme might be quite effective.

If we consider the DAP Fortran code for the ADI scheme, Figure 8, we can see that there are blocks of code which contain independent instructions. For example in Figure 9, the calculation of U and V are independent. If the next block of code is rearranged so that QT is computed first, then the other six statements are all independent of each other. The next block of four statements are also independent, and so on. Finally the tridiagonal systems are all independent.

One can thus see that there are, on the average, blocks of 4 or 5 independent statements in the code. Each of these performs the same calculation in all processors. These calculations can thus be split up among the independent processors of an MIMD computers. Without considering the details of the MIMD machine it is difficult to say more, but it appears that this algorithm might be a good candidate for MIMD architectures.

6. Concluding Remarks

The adaptation of the Navier-Stokes algorithm to the DAP architecture was quite straightforward and fairly successful. Even though there are a few sections of the code where vector, rather than matrix, operations were used, these slow operations did not significantly effect the overall performance because there are so few of them. There are even fewer scaler operations. The overall performance of the algorithm ranges from 10 to 20 megaflops, depending on the number of iterations per time step. This is really about as good as one could expect, given the basic performance of the DAP. In general the data transfer overhead is only a minor part of the calculation. This is due to the high transfer rate, as compared to the arithmetic processing rate. The only routine that contains many transfers is the tridiagonal solver. It seems inherent in the algorithm that this is so. It might be desirable to change the basic Navier-Stokes algorithm and use a relaxation scheme, say, in place of ADI. Finally it must be noted that a good deal of the success in adapting this algorithm to the DAP is due to the fact that it is for a two dimensional problem and the computational array size was chosen to fit the size of the DAP array. Further work is needed to examine the effect of using a much larger computational array and treating three dimensional problems.

REFERENCES

[1] DAP: Fortran Language, Technical Publication 6918, International Computers Limited, 1981.

[2] J. J. Dongarra, Performance of various computers using standard linear equations software in fortran environment, Argonne National Labortory Technical Memo, 23, 1984.

[3] T. B. Gatski, C. E. Grosch, and M. E. Rose, A numerical study of the two-dimensional Navier-Stokes equations in vorticity-velocity variables, J. Comput. Phys., Vol. 48 (1982), pp. 1-22.

[4] T. B. Gatski and C. E. Grosch, A numerical study of the two- and three-dimensional unsteady Navier-Stokes equations in velocity-vorticity variables using compact schemes, Proc. Ninth International Conference on Numerical Methods in Fluid Dynamics, Springer-Verlag, pp. 235-239, 1985a.

[5] T. B. Gatski and C. E. Grosch, Embedded cavity drag in steady laminar flow, AIAA J., 23 (July 1985b), pp. 1028-1037.

[6] C. E. Grosch, Performance analysis of Poisson solvers on array computers, in Infotech State of the Art Report: Supercomputers, C. Jesshope and R. W. Hockney, eds., 2 (1979), Infotech International, pp. 147-181.

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

[8] R. W. Hockney, (r_∞, $h_{1/2}$, $s_{1/2}$) measurements on the 2-CPU Cray X-MP, University of Reading, Department of Computer Science, Report RCS 185, 1985.

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

[10] R. M. McInville, T. B. Gatski, and H. A. Hassan, Analysis of large vortical structures in shear layers, AIAA J., 23 (1985), pp. 1165-1171.

[11] J. M. Ortega and R. G. Voigt, Solution of partial differential equations on vector and parallel computers, SIAM Rev., 27 (1985), pp. 149-240.

[12] R. B. Philips and M. E. Rose, Compact finite difference schemes for mixed initial-boundary value problems, SIAM J. Numer. Anal., 19 (1982), pp. 698-719.

[13] M. E. Rose, A "unified" numerical treatment of the wave equation and the Cauchy-Riemann equations, SIAM J. Numer. Anal., 18 (1981), pp. 372-376.

[14] K. Tanabe, Projection method for solving a singular system of linear equations and its applications, Numer. Math., 17 (1971), pp. 203-214.

DAP Fortran Statment	Execution Time (microseconds)	Processing Rate (MFLOPS)
Z = X	17	241
Z = X+Y	175	23
Z(L) = X+Y	179	23
Z = X*Y	274	15
Z = X/Y	386	11
Y = SQRT(X)	194	21
Y = EXP(X)	414	10
Y = SIN(X)	862	5
Y = TRAN(X)	714	6
X = MATC(V)	31	132
X = MATR(V)	31	132
Z = MERGE(X,Y,L)	50	82
Y = SHEP(X,32)	227	577
Y = X(,-)	23	178
Y = SHEC(X)	23	178
Z = X(+,)+Y(,-)	221	56
Z = X(+,+)+Y(-,+)	267	77
S = MAXV(X)	56	73
S = MAXV(X,L)	57	72
S = SUM(X)	450	9
CALL CONVFME(X)	2252	2

Table 1. Examples of DAP Fortran statements, execution times, and the corresponding processing rates. Here X, Y, and Z are 64 by 64 matrices of 32 bit floating point numbers; L is a 64 by 64 single bit logical matrix, ie a mask; V is a 64 element vector of 32 bit floating point numbers; S is a 32 bit floating point scaler.

Subprogram	Execution Time (msec)	Proportion of Transfer Time (%)	Processing Rate (MFLOPS)
RELAXD	30.2*	6.9	20.2
FIXZBD	1.4	5.9	0.3
ZCALCD	222.4	3.0	10.1
FNORMD	5.4	2.8	15.1
QTHETD	1.6	2.1	5.0
TRIIED	32.0	27.2	10.1
TRIJED	32.0	27.2	10.1

*Per Iteration

Table 2. Summary of results.

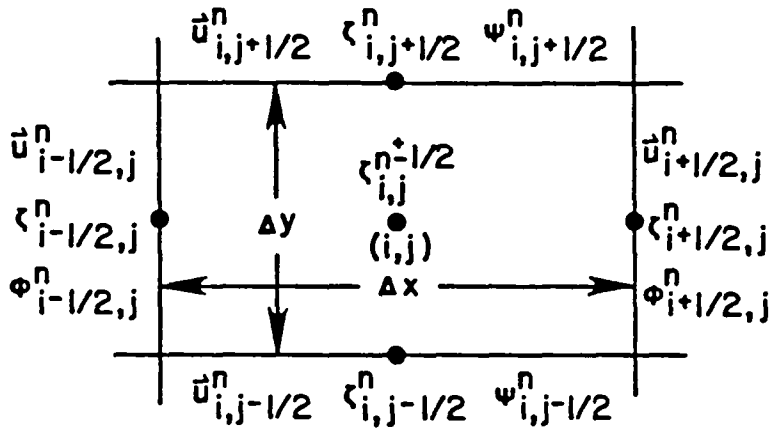

Figure 1. Typical computational cell and the data values associated with it.

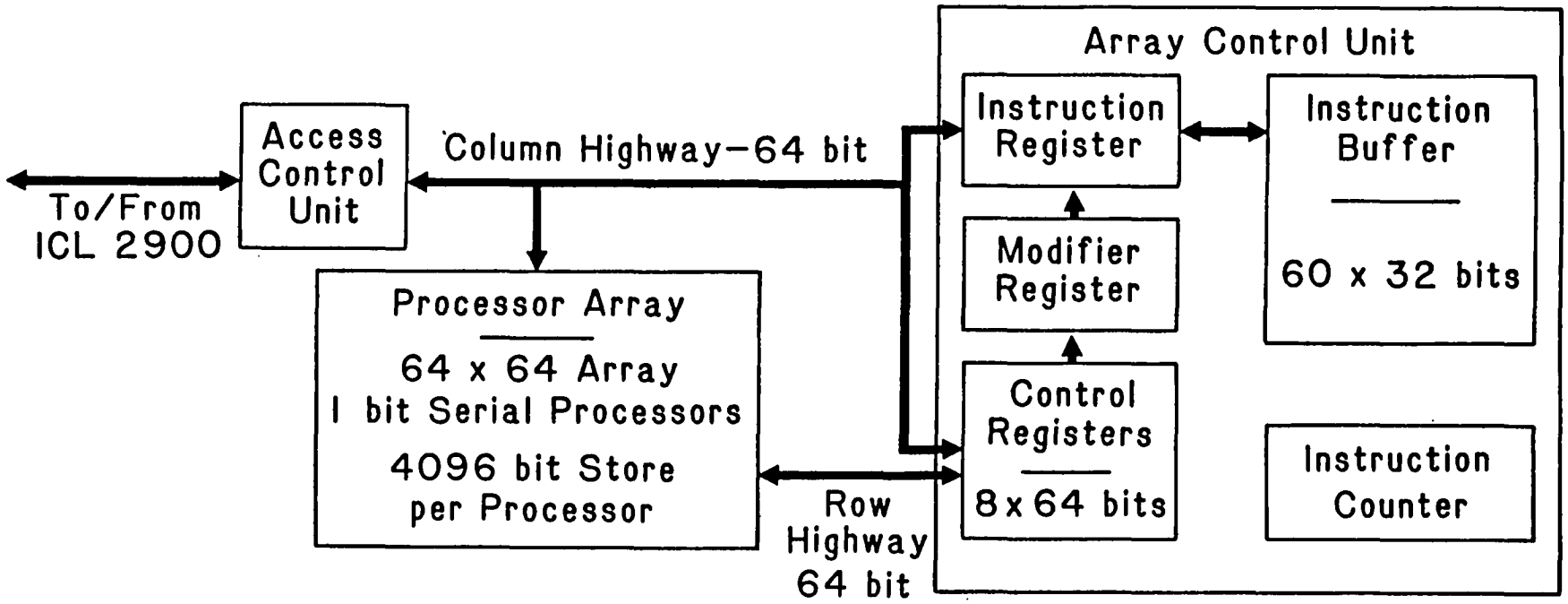

Figure 2. Schematic diagram of the DAP architecture.

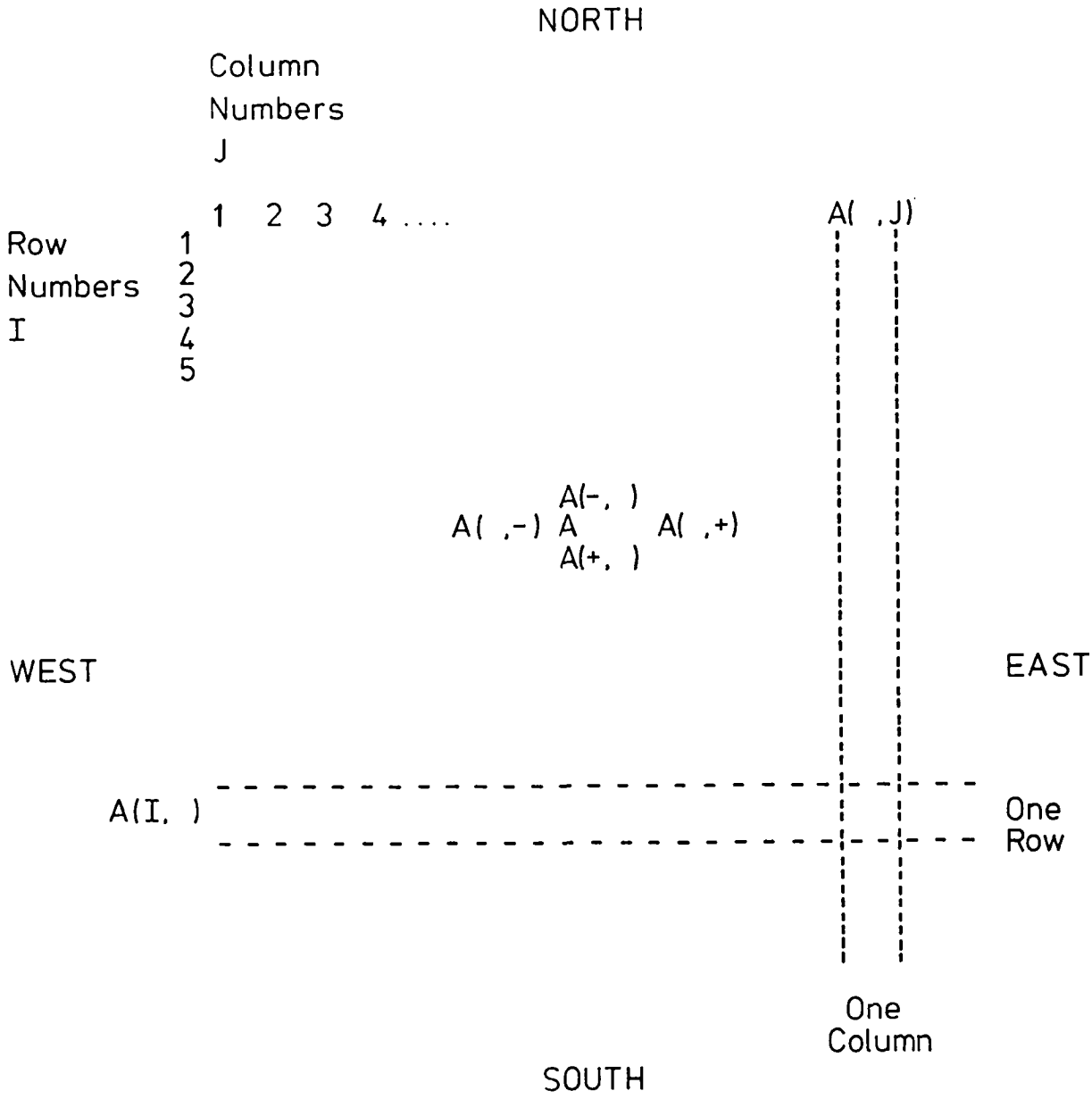

Figure 3. Matrix storage and indexing conventions on the DAP.

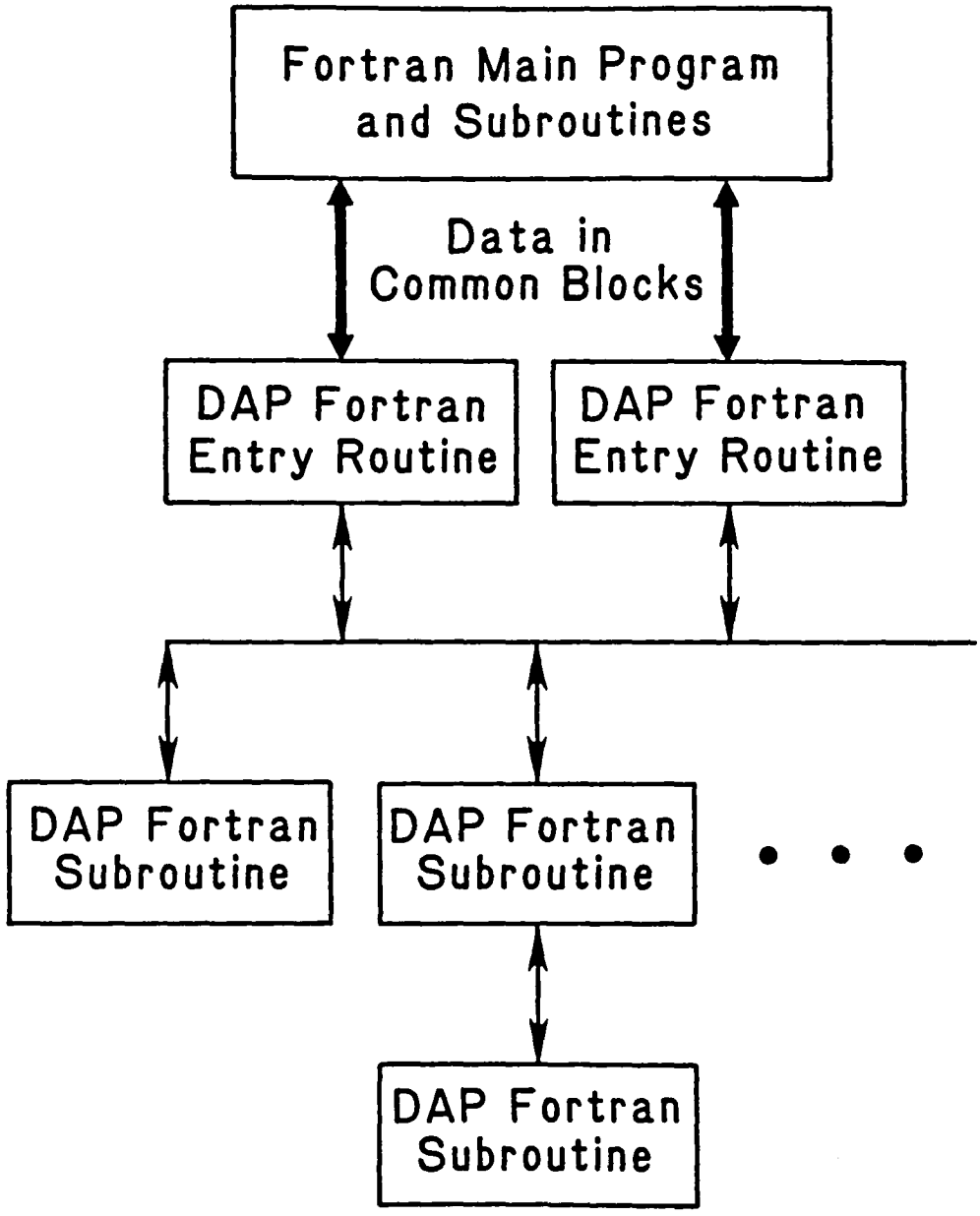

Figure 4. Structure of a typical Fortran Program for the DAP.

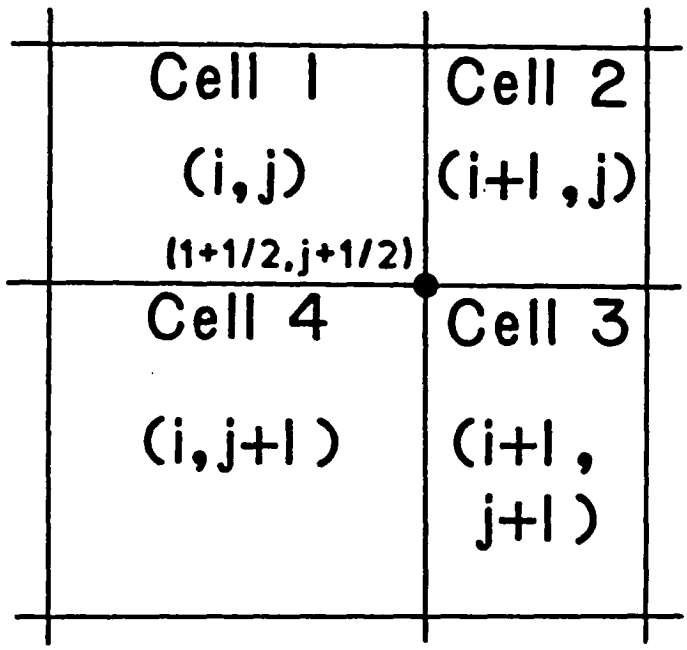

Figure 5. Computational cells bordering the grid point $(i+\frac{1}{2}, j+\frac{1}{2})$.

```
TP1 = P1 - P3
TP2 = P2 - P4
TQ1 = Q1 - Q3
TQ2 = Q2 - Q4
R1 = ZL1*(TP1+TP2)-TQ1+TQ2
R2 = TP1 - TP2+ZLT*(TQ1+TQ2)-ZT

TP1 = CT*R1
TQ1 = ZLT*TP1
TP2 = CT*R2
TQ2 = ZLT*TP2

P1 = P1-TQ1-TP2
Q1 = Q1+TP1-TQ2
P2 = P2-TQ1+TP2
Q2 = Q2-TP1-TQ2
P3 = P3+TQ1+TP2
Q3 = Q3-TP1+TQ2
P4 = P4+TQ1-TP2
Q4 = Q4+TP1+TQ2
```

Figure 6. Kernel of the relaxation subroutine in DAP Fortran.

```
R1 = 0.0
R2 = 0.0
TP1 = P(+,+)-P
TP2 = P(,+)-P(+,)
TQ1 = Q(+,+)-Q
TQ2 = Q(,+)-Q(+,)
R1(MR) = ABS(ZL*(TP1+TP2)-TQ1+TQ2)
R2(MR) = ABS(TP1-TP2+ZL*(TQ1+TQ2)-Z)
ERR = MAXV(MERGE(R1,R2,R1.GT.R2))
```

Figure 7. DAP Fortran code to compute the maximum value of the residuals.

```
U2 = P(63,)
U3 = P(62,)
U2 = U2+U2(+)
U3 = 0.25*(U3+U3(+))
ZBOT = (U3-U2)/VEC(DY(63))
```

Figure 8. DAP Fortran code to calculate the boundary values of the vorticity at a solid wall.

```
U = P+P(+,)
U = 0.25*(U+U(,+))
V = Q+Q(+,)
V = 0.25*(V+V(,+))

T1 = 0.5/MATC(RLY)
T2 = MATC(RKY)
QT = QTHETD(0.5*RE*V*MATC(DY))
CC = QT-1.0
QQ = QT+1.0
AA = 1.0+V*MATC(RLY)
BB = 1.0-V*MATC(RLY)

ALFAM = T1(-,)*(CC(-,)*AA(-,)+T2(-,))
BETAM = T1(-,)*(CC(-,)*BB(-,)-T2(-,))
ALFAP = T1*(QQ*AA+T2)
BETAP = T1*(QQ*BB-T2)

CC = 0.5/MATR(DX)
AA = ZETA-DT*CC*(U*(ZXC,+)-ZX)-(PHI(,+)-PHI)/RE)
BB = ZETA(-,)-DT*CC(-,)*(U(-,)*(ZX(-,+)-ZX(-,))-(PHI(-,+)-PHI(-,))/RE)

QQ = 2.0*((1.0-QT(-,))*T1(-,)*BB+(1.0+QT)*T1*AA)
QQ(1,) = ZTOP
QQ(64,) = ZBOT

AA = BETAP
AA(1,) = 0.0
AA(64,) = 0.0
BB = ALFAP-BETAM
BB(1,) = 1.0
BB(64,) = 1.0
CC = -ALFAM
CC(1,) = 0.0
CC(64,) = 0.0

ZY = TRIIED(AA,BB,CC,QQ)
```

Figure 9. A portion of the DAP Fortran code to carry out the ADI calculation.

```
      MATRIX FUNCTION TRIIED (A,B,C,Q)

      REAL A(,),B(,),C(,),Q(,)

      K = 1

      DO 100 L = 1,6
      A = A/B
      C = C/B
      Q = Q/B
      Q = Q-A*SHNP(Q,K)-C*SHSP(Q,K)
      B = 1.0-A*SHNP(C,K)-C*SHSP(A,K)
      A = -A*SHNP(A,K)
      C = -C*SHSP(C,K)
  100 K = K+K

      TRIIED = Q/B

      RETURN
      END
```

Figure 10. Tridiagonal equation systems solver in DAP Fortran.

1. Report No. NASA CR-178034 ICASE Report No. 85-61	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle ADAPTING A NAVIER-STOKES CODE TO THE ICL-DAP		5. Report Date December 1985 6. Performing Organization Code
7. Author(s) Chester E. Grosch		8. Performing Organization Report No. 85-61 10. Work Unit No.
9. Performing Organization Name and Address Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225		11. Contract or Grant No. NAS1-17070; NAS1-18107
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, D.C. 20546		13. Type of Report and Period Covered Contractor Report 14. Sponsoring Agency Code 505-31-83-01

15. Supplementary Notes

Langley Technical Monitor:
C. South Jr.
Final Report

Submitted to SIAM J. Sci. Stat. Comput.

16. Abstract

This is a report of the results of an experiment: to adapt a Navier-Stokes code, originally developed on a serial computer, to concurrent processing on the ICL Distributed Array Processor (DAP). In this paper the algorithm used in solving the Navier-Stokes equations is briefly described. The architecture of the DAP and DAP Fortran is also described. The modifications of the algorithm so as to fit the DAP are given and discussed. Finally, performance results are given and conclusions are drawn.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Navier-Stokes parallel computing ICL-DAP	34 - Fluid Mechanics & Heat Transfer 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	41	A03

For sale by the National Technical Information Service, Springfield, Virginia 22161