INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN FLUIDS, VOL. 12, 947–963 (1991)

DEVELOPMENT OF A PARALLEL COMPUTATIONAL FLUID DYNAMICS ALGORITHM ON A HYPERCUBE COMPUTER

MARK E. BRAATEN

GE Research and Development Center, PO Box 8, Schenectady, NY 12301, U.S.A.

SUMMARY

One of the main factors limiting the widespread use of computational fluid dynamics codes for engineering design is their very large requirements both in terms of computer memory and CPU time. Distributed memory parallel computers offer both the potential for a dramatic improvement in cost/performance over conventional supercomputers and the scalability to large numbers of processors that is required if performance beyond that of current supercomputers is to be achieved. As part of an evaluation to explore the potential of such machines for computational fluid mechanics applications, a concurrent algorithm for the solution of the Navier–Stokes equations has been developed and demonstrated on a hypercube parallel computer. The algorithm is based on a domain decomposition of a well-established serial pressure correction algorithm.

The algorithm is demonstrated on both a 32-node scalar and eight-node vector Intel iPSC/2 for complicated two-dimensional laminar and turbulent flow problems with different grid sizes and numbers of processors. Speed-ups relative to a single processor of 12·9 with 16 processors and 20·2 with 32 processors are achieved on a scalar iPSC/2, demonstrating the parallel efficiency of the algorithm. Measured performance on a 32-node scalar iPSC/2 exceeds one-sixth that of a Cray X-MP running the original serial algorithm. The performance of the algorithm on an eight-node vector iPSC/2 exceeds that of the larger scalar hypercube and is about one-fifth that of the Cray X-MP. With cost/performance more than 10 times better than the Cray, these results dramatically show the cost effectiveness of vector hypercubes for this class of fluid mechanics algorithm.

KEY WORDS Computational fluid dynamics Parallel computing Parallel processing Domain decomposition

1. INTRODUCTION

One of the main factors limiting the widespread use of computational fluid dynamics codes for engineering design in their very large requirements both in terms of computer memory and CPU time. Even when present-day supercomputers are adequate to solve the problems, the high cost of such machines and their limited availability make their use impractical for all but a few government laboratories and the largest industrial companies. Many important fluid dynamics problems, including the simulation of the flow about a complete aircraft, the study of the effects of unsteady phenomena on turbine stage efficiency, the modelling of combustion instabilities and the direct simulation of turbulence, will require computer resources of the order of hundreds of hours of time on modern supercomputers and hundreds of millions of words of computer storage.

Single-processor machines are approaching fundamental limitations on performance owing to limits on signal transmission speed, switching delays and other factors.[1] Parallel processing offers

0271–2091/91/100947–17$08.50

Received 26 July 1989
Revised 27 July 1990

the best near-term hope for greatly increased computational speeds. Recent advances in microelectronics, producing such items as 1 Mbit DRAM chips and modern 32 bit microprocessors such as the Intel 80386 and Motorola 68020 have made a whole generation of high-performance/low-cost parallel distributed memory computers available, such as the Intel iPSC, NCUBE and Meiko Computing Surface. Although such machines currently lack powerful enough processors to provide performance significantly faster than existing supercomputers, their distributed memory architectures are scalable up to the level of hundreds of processors, and some algorithms have been demonstrated to retain high parallel efficiencies up to this level.[2] Since more and more powerful microprocessors are being developed at a remarkable rate, it seems only a matter of time before such machines offer performance significantly exceeding existing supercomputers for some applications. Fox *et al.*[3] project, on the basis of reasonable extrapolations of existing technology, that a distributed memory machine with 10^5 processors, 10^{13} bits of memory and 10^{14} flops performance may be possible within a few years.

There is a great deal of interest in using parallel computation for computational fluid dynamics problems, as evidenced by a number of recent conference sessions devoted to the area.[4,5] It is beyond the scope of this paper to review all of the literature—this will be attempted in Reference 6.

In an earlier paper[7] a parallel version of a well-known pressure correction algorithm for the solution of incompressible viscous fluid flows was described by the present author and demonstrated on a 16-node Intel iPSC/1 scalar hypercube. In this work that algorithm has been extended and implemented on a second-generation 32-node Intel iPSC/2 scalar hypercube and on an eight-node Intel iPSC/2VX vector hypercube. On both new machines the ratio of the computational speed of the processors to the communication speed of the machine is significantly increased compared to the older iPSC/1, which puts even a higher premium on achieving efficient communication. In addition, the vector machine demands close attention to issues of vectorization to ensure that the potential speed of the vector processors is realized.

2. OVERVIEW OF PARALLEL ALGORITHM

The numerical algorithm developed here for execution on a distributed memory parallel processor is a direct extension of the serial incompressible flow algorithm described in References 8 and 9. An initial parallel implementation of this algorithm was described in detail in Reference 7. Only a brief outline of the original implementation is given here, with the emphasis instead on new developments and issues of importance to vector concurrent computation.

The algorithm developed here is capable of solving steady two-dimensional (planar or axisymmetric) laminar and turbulent flows. For clarity, a planar situation is described. Laminar flows are described by the equations for conservation of mass, x-momentum and y-momentum, along with appropriate boundary conditions. For turbulent flows the standard k–ε turbulence model is used, along with the wall function treatment for the near-wall regions.[10]

The governing conservation equations can be written in Cartesian co-ordinates for a general dependent variable ϕ in the following form:

$$\frac{\partial}{\partial x}(\rho u\phi)+\frac{\partial}{\partial y}(\rho v\phi)=\frac{\partial}{\partial x}\left(\Gamma\frac{\partial\phi}{\partial x}\right)+\frac{\partial}{\partial y}\left(\Gamma\frac{\partial\phi}{\partial y}\right)+R(x,y). \tag{1}$$

Here Γ is the effective diffusion coefficient and R is the source term for the variable ϕ, which can represent velocity components, temperature, turbulence variables, etc. The equations are transformed by introducing new independent variable ξ and η. Equation (1) changes according to the

general transformation $\xi = \xi(x, y)$, $\eta = \eta(x, y)$ and can be rewritten in (ξ, η) co-ordinates as follows:

$$\frac{1}{J}\frac{\partial}{\partial \xi}(\rho U \phi) + \frac{1}{J}\frac{\partial}{\partial \eta}(\rho V \phi) = \frac{1}{J}\frac{\partial}{\partial \xi}\left(\frac{\Gamma}{J}(q_1 \phi_\xi - q_2 \phi_\eta)\right) + \frac{1}{J}\frac{\partial}{\partial \eta}\left(\frac{\Gamma}{J}(-q_2 \phi_\xi + q_3 \phi_\eta)\right) + S(\xi, \eta). \tag{2}$$

Here U and V are the contravariant velocity components, q_1, q_2 and q_3 are metric terms arising from the co-ordinate transformation, J is the Jacobian of the transformation and $S(\xi, \eta)$ is the source term in (ξ, η) co-ordinates. The reader is referred to References 8 and 9 for complete details.

Discretization of equation (2) leads to the following general form of the conservation equation for the variable ϕ:

$$a_P \phi_P = \sum_{i = E, W, N, S} a_i \phi_i + (S_\phi)_P. \tag{3}$$

The subscripts P, E, W, N and S refer to the grid point at the centre of the control volume and the four neighbouring grid points respectively. The term $(S_\phi)_P$ includes the original source term in the equation plus the additional terms that cannot be approximated by the values of ϕ at the five grid points. A staggered grid is used to compute the velocity components, as in the standard practice in finite volume procedures.[11] For simplicity, the combined convection and diffusion fluxes across the control volume surfaces are computed using the so-called hybrid scheme.[11]

The coupled system of momentum and continuity equations is solved by a pressure correction method similar to that described in Reference 11. The momentum equations are first solved, for a given pressure distribution p^*, to yield a tentative velocity field u^*, v^*. An equation for updating the pressure is obtained via manipulation of the discrete forms of the momentum and continuity equations. This equation is solved, the pressure is updated and the velocity components are corrected to satisfy continuity, completing one global iteration. Owing to the non-linearity of the problem, a number of a global iterations are required to obtain a converged solution.

Effective implementation of this algorithm on a distributed memory parallel computer requires the satisfactory resolution of three major computational issues, namely (i) load balancing, (ii) minimization of communications costs and (iii) the development of an efficient concurrent algorithm. The basis of the parallel implementation adopted here is the domain decomposition method.[12] The solution domain is divided up into a number of overlapping subdomains and each subdomain is assigned to a different processor. Overlapping is necessary so that each interior grid point is treated as an interior point in at least one subdomain. In parallel, the coefficients of the equation under consideration are calculated in each subdomain and an iterative solution is obtained in each region to some reasonable level of convergence. The boundary values are then exchanged with the neighboring subdomains and the solution is iterated further. When some suitable global convergence criterion is satisfied, the solutions on each subdomain can be assembled into the complete solution for the entire domain.

Domain decomposition has many advantages for this situation. Since the work per grid point is roughly equal, assigning the same number of grid points to each subdomain assures reasonable load balancing. A geometrical decomposition has the advantage that metric information arising from the co-ordinate transformation needs only be stored in the local memory of the processor that is assigned to that region of the domain and does not need to be communicated between processors. Since this metric information makes up most of the required storage for the code, the total memory of the ensemble of processors can be used effectively.

In this work the decomposition of the domain is done by strips. A one-dimensional stripwise decomposition leads to the smallest number of messages passed, making it attractive for machines such as the hypercube where high message latency is the dominant cost for short messages. The

drawback of a stripwise decomposition is that the number of processors that can be used is limited to the maximum number of grid cells in any one direction; consequently it cannot be scaled to as many processors as a multidimensional decomposition. Nearest-neighbour communication is ensured by mapping the hypercube to the required linear array through the use of binary reflected Gray codes.[13] Each processor is then a nearest neighbour to the two processors that are handling the two adjacent subdomains.

The parallel algorithm adopts the same equation-by-equation solution procedure as the original serial algorithm, solving the governing equations in the following order: x-momentum, y-momentum and pressure correction (followed by turbulent energy and turbulent dissipation for turbulent flows). This equation-by-equation procedure was selected for several reasons. First, it reduces to the original algorithm for a single processor. Thus comparison of the speed-up obtained with p processors over p subdomains is made relative to the original algorithm on a single processor over the entire solution domain, which reflects a comparison of the parallel algorithm with the best serial algorithm in the sense of S'_p as defined by Ortega and Voigt.[14] Secondly, if converged solutions are obtained for each equation, then the overall rate of convergence of the algorithm will be the same as for the serial algorithm if each equation is also solved to convergence at each stage. Although this is not the usual practice (since the coefficients are only tentative, it is more efficient to solve only loosely to convergence at each stage), this similarity ensures that the parallel algorithm will show comparable convergence behaviour to the serial algorithm, which has proved to converge well for a large variety of problems.

At each stage the linearized equations for each variable are solved by a suitable iterative procedure such as a line-by-line TDMA.[11] In the earlier work[7] it was found that the subdomain solution of the pressure correction equation on multiple processors converged more slowly than the original procedure on a single processor, leading to an increase in the required number of iterations for convergence. This was found to be the critical factor in reducing the parallel efficiency of the parallel algorithm. The use of a block correction scheme to accelerate the convergence of the pressure correction equation was found to make the overall convergence rate of the algorithm nearly independent of the number of processors, at the cost of requiring global communication between the processors during the block correction step. As a result of the need for block correction, and its slower convergence, the solution of the pressure correction equation takes up about 50% of the total computational effort.

3. NEW DEVELOPMENTS FOR VECTOR CONCURRENT COMPUTATION

The ultimate goal of this effort is to develop a parallel vectorizable CFD algorithm that will scale efficiently to distributed memory machines with large numbers of vector processors. Consequently, high levels of vectorization and parallel efficiency are of paramount concern.

The Intel iPSC2/VX machine was chosen for this study because it is a commercially available vector hypercube. The machine has an attached vector processor rated at 20 Mflops single precision paired with each 80386/30387-based scalar processor. The machine can be configured with up to 128 vector processors, although only eight were available in the machine used in this study. A source code precompiler called VAST-2[15] vectorizes DO loops in FORTRAN programmes and creates code for the vector processor. The vector code produced is in the form of calls to a subroutine library of vector arithmetic operations that execute on the vector processor. VAST-2 was found to be easy to use and to generate good, although not optimal, vector code.

Figure 1 shows typical benchmarked performance of the vector processor on a SAXPY loop (SAXPY computes $\mathbf{aX}+\mathbf{Y}$ in single precision) for different vector lengths. Several characteristics of the vector performance are of significant interest.

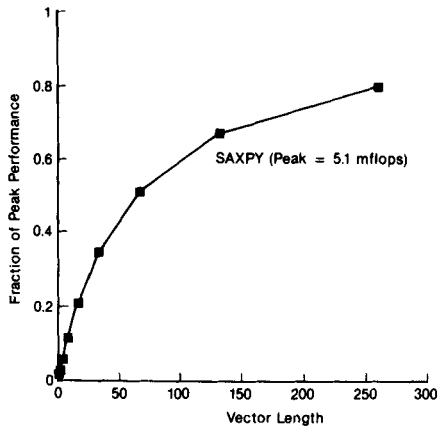

Figure 1. Typical vector processor performance on Intel iPSC/2VX

1. Peak measured performance of the vector processor was 5·1 Mflops, compared to 0·17 Mflops for the 80386/80387 scalar processor. This factor of 30 in floating point speed available from the vector processor demonstrates the importance of having a vectorizable algorithm for this machine. However, this significant increase in floating point performance will place even more of a burden on the communication system, making efficient communication even more critical if high parallel efficiency is to be achieved with large numbers of vector processors.
2. The vector length required to reach one-half of the peak performance (the so-called half-length) is of the order of 50 words, which is significantly longer than that for more sophisticated vector processors such as those used in the Cray, where the half-length is typically 10–20 words. The consequence of this is that fairly long vector lengths are needed to get good performance out of the vector processors.
3. The performance curve shows two different regions of performance. For short vectors the performance is linearly proportional to the vector length, while for long vectors the vector performance is virtually independent of the vector length as the asymptotic rate is approached. In the domain decomposition technique used here, doubling the number of processors essentially halves the number of grid points assigned to each processor. If the resulting vector lengths are also halved and become short enough so that the vector processor performance is also halved, then no speed-up will occur. To ensure the scalability of the algorithm to large numbers of vector processors, it is necessary to ensure that the vector lengths are either independent of the number of processors or long enough so that halving the vector length does not significantly degrade the vector performance.

The original serial 2D fluids code was written using two-dimensional array storage for the various coefficients and variables. A typical variable ϕ was stored as PHI (I, J). A typical code segment to compute ϕ over the interior control volumes in a subdomain appeared in the form:

```
      DO 10 J = 2, NJ - 1
      DO 20 I = 2, NI - 1
      PHI(I, J) = stuff
   20 CONTINUE
   10 CONTINUE
```

Here NI is the number of grid points in the x-direction for the subdomain of interest and NJ is the number of grid points in the y-direction. On a scalar machine it is conventional practice to put the loop which increments the leading index of the array as the innermost loop, since this leads to the fastest memory access. However, note that for the stripwise decomposition used here, NJ is independent of the number of processors, but NI is inversely proportional to the number of processors and consequently decreases as the number of processors is increased. Since on a vector machine only the innermost DO loop in a nested set can be vectorized, this leads to shorter and shorter vector lengths as the number of processors is increased. Inverting the order of the loops, to place J in the inner loop, makes the vector length independent of the number of processors and will lead to longer vector lengths for large number of processors.

The use of one-dimensional data structures can lead to even longer vector lengths, since a single loop can then run over all of the grid points in the subdomain. With one-dimensional data storage the same loop appears as

```
      DO 10 IJ = IJS, IJE
      PHI(IJ)  = stuff
   10 CONTINUE
```

The composite index IJ can be interpreted as IJ = (I − 1)*NJ + J so that IJS = NJ + 2 and IJE = NI*NJ − NJ − 1. The only difference between this loop and the previous loop is that meaningless calculations are done for some boundary points along the top and bottom surfaces. If these values are not used in the code, there is no difficulty; if they are incorrect or lead to floating point exceptions that cause later problems, they can be overwritten in a small subsequent loop. Note that the possibility of floating point exceptions requires that the code execution does not terminate in the event of an exception, in accordance with the IEEE floating point standard.

The line-by-line TDMA used in the original work is a good choice on a scalar machine, but since the TDMA algorithm does not vectorize well, is not particularly attractive on a vector machine. Point solvers such as Jacobi or Gauss–Seidel have proven to be attractive alternatives on a vector machine owing to their excellent vectorizability. Consider the following point symmetric Gauss–Seidel algorithm consisting of a forward and backward pass. The forward pass appears as

```
      DO 10 I = 2, NI − 1
      DO 10 J = 2, NJ − 1
```

$$\phi_{ij}=\frac{1}{a_{\mathrm{P}}}\left(a_{\mathrm{E}}\phi^{*}_{i+1,j}+a_{\mathrm{W}}\phi_{i-1,j}+a_{\mathrm{N}}\phi^{*}_{i,j+1}+a_{\mathrm{S}}\phi_{i,j-1}+(S_{\phi})_{\mathrm{P}}\right) \tag{4}$$

```
   10 CONTINUE
```

Here the starred quantities refer to values that are taken from the previous iteration; unstarred quantities refer to the recently updated values. Note that the distinction between whether an updated value or an old value is used comes from the path that is followed to update ϕ. In the backward pass the visiting order of the points is reversed.

On a vector machine, if vectorization is forced by putting a compiler directive on the innermost loop, all of the values in a particular column are calculated simultaneously and the forward pass appears as

$$\phi_{ij}=\frac{1}{a_{\mathrm{P}}}\left(a_{\mathrm{E}}\phi^{*}_{i+1,j}+a_{\mathrm{W}}\phi_{i-1,j}+a_{\mathrm{N}}\phi^{*}_{i,j+1}+a_{\mathrm{S}}\phi^{*}_{i,j-1}+(S_{\phi})_{\mathrm{P}}\right) \tag{5}$$

The difference between this and the scalar calculation is that old values of $\phi_{i,j-1}$ are used in the vectorized calculation rather than the recently updated values.

If the code was rewritten with a single loop running over all of the interior points and a compiler directive was used to force vectorization, then the forward pass would appear as

$$\phi_{ij}=\frac{1}{a_P}(a_E\phi^*_{i+1,j}+a_W\phi^*_{i-1,j}+a_N\phi^*_{i,j+1}+a_S\phi^*_{i,j-1}+(S_\phi)_P). \tag{6}$$

The iterative method then reverts to a point Jacobi scheme. For convection-dominated problems, point Jacobi will converge much more slowly than a corresponding Gauss–Seidel scheme (note for a 1D convection-dominated problem that Gauss–Seidel converges immediately). For a diffusion-dominated problem the convergence rate of point Jacobi is roughly half that of Gauss–Seidel. Consequently the use of the Jacobi scheme is not attractive since the reduction in convergence rate more than offsets the increased floating point speed of the vectorized loop. The iterative procedure represented by equation (5) appears to be the best compromise for the vector machine.

As mentioned earlier, the increased speed of the vector processors makes efficient communication even more critical than before. In the earlier work the global communication required by the block correction scheme was done via spanning trees. Each processor calculated its contribution to the block correction coefficients and these contributions were then passed to a root processor via a spanning tree. The root processor then computed the corrections while the other processors sat idle, and then broadcast the results of all processors via another spanning tree. In this work, the global communication required by the block correction was done by a global exchange as described in Reference 16. The contributions of each processor are globally exchanged with all of the other processors, so that each processor memory ultimately contains all of the coefficients. Each processor can then independently compute the corrections, eliminating the need for the final broadcast step. This reduces the time required for communication, which dominates the time required by the block correction, by almost a factor of two.

Owing to the staggered nature of the grid used in the calculation, it is sometimes necessary to pass the values from two adjacent columns of grid cells to a neighbouring processor rather than just one. The use of higher-order differencing schemes will also require the passing of multiple columns of values. The use of a 1D data structure proves helpful here, since such a message can be passed simply by prescribing the starting address of the message to be PHI(IJS), the length of the message to be 2*NJ and using appropriate values of IJS at both the sending and receiving ends. This avoids the time-consuming bother of packing two columns of a two-dimensional array into a one-dimensional message vector and then having to unpack it upon receipt. The time required for packing and unpacking messages can be substantial for large numbers of processors, so this is an important time saving.

4. ESTIMATION OF PARALLEL EFFICIENCY

In this work the parallel efficiency of the algorithm is determined for problems of fixed size while the number of processors is varied. The parallel efficiency ε is defined as

$$\varepsilon=T_1/pT_p, \tag{7}$$

where T_1 is the total execution time on a single processor, p is the number of processors and T_p is the total execution time on p processors. Direct measurement of the efficiency requires timing of the problem on a single processor, which is not always possible for large problems because the

node memory on a single processor is very limited. On the Intel iPSC2/VX used in this study, each scalar processor had only 4 Mbytes of memory and each vector processor had only 1 Mbyte of memory. When the vector processors are used, the default procedure is to load to programme data into the 1 Mbyte vector memory since it is directly addressable from both the scalar and vector CPUs. The vector processor requires all vector operands to reside in the vector memory, so the additional 4 Mbytes of scalar memory cannot be conveniently used except for a small amount of data which is not required by the vector processor. Data can be moved back and forth between the scalar and vector memories, but doing so requires extensive recoding and significantly reduces the available performance from the vector processor. For the code developed here, the available memory limited the size of problem that could be solved on a single processor to about 2500 grid points on a single vector processor or 10000 grid points on a single scalar processor.

Consequently the larger test problems studied here could not be run on a single vector processor, so the parallel efficiency of such cases had to be estimated. Gustafson *et al.*[2] outlined a procedure for estimating parallel efficiency in such situations based on the notion of a hypothetical processor node with direct access to all of the memory in the machine. Their procedure requires accurate measurements of the time each processor spends communicating data and sitting idle during the calculation. For a complicated application code such as considered here, timing all of the communication and idle periods is a substantial task. Estimating the efficiency on the basis of the ratio of Mflop rates it is also difficult because of problems in counting the number of floating point operations in a complicated algorithm with numbers of blocks of conditional code.

To avoid the need for timing all of the communication and idle periods for each processor, the following simpler procedure was adopted. The time required to solve the problem on one processor can be expressed as

$$T_1 = n_{\text{calc}}\, t_{\text{calc}}, \tag{8}$$

where n_{calc} represents the number of floating point operations performed and t_{calc} is the average time per calculation. The time required by p processors to solve the same problem, assuming perfect load balancing, can be expressed as

$$T_p = \frac{n_{\text{calc}}\, t_{\text{calc}}}{p} + t_{\text{comm}}, \tag{9}$$

where t_{comm} represents the time spent communicating data and sitting idle. The communication time can be expressed as

$$t_{\text{comm}} = t_{\text{comml}} + t_{\text{commg}}, \tag{10}$$

where t_{comml} represents the time spent exchanging local data with neighbouring processors and t_{commg} represents the time spent performing global data exchanges.

The local communication time is independent of the number of processors (for more than one processor) and can be expressed as

$$t_{\text{comml}} = t_{\text{local}} H(p-1), \tag{11}$$

where $H(\phi)$ represents the unit step function. The time for global exchange of data scales logarithmically with the number of processors for the hypercube topology, so the global communication time can be expressed as

$$t_{\text{commg}} = t_{\text{global}} \log_2 p. \tag{12}$$

The following expression for the efficiency is obtained after combining the previous equations and

consolidating the various constants

$$\varepsilon = \frac{1}{1+\beta pH(p-1)+\gamma p \log_2 p} \tag{13}$$

Equations (7) and (13) contain three empirical parameters, T_1, β and γ, which may be determined by timing a given problem on three different numbers of processors. Unfortunately, in this work the largest problems run on the vector processors could only be run on four or eight processors, so a simple two-parameter model was needed.

A two-parameter model can be obtained by taking suitable approximations to the logarithmic term and the term containing the step function in equation (13). The Taylor series expansion of $p \log p$, taken about $p=1$, gives the approximation

$$p \log p = p - 1 + \cdots. \tag{14}$$

The step function can be approximated in an asymptotic sense by the expression $(p-1)/p$, which has the correct behaviour at $p=1$ and as $p \to \infty$. With these approximations, equation (13) can be written as

$$\varepsilon = \frac{1}{1+\beta(p-1)+\gamma(p-1)}. \tag{15}$$

The two terms in the denominator can be combined into a single term, resulting in the final modelled equation for the efficiency

$$\varepsilon = \frac{1}{1+\alpha(p-1)}. \tag{16}$$

If a timing of t_{p_1} is obtained with p_1 processors and a timing of t_{p_2} is obtained with p_2 processors, then α is given by

$$\alpha = -\left(\frac{p_1 t_{p_1} - p_2 t_{p_2}}{p_1 t_{p_1}(p_2-1) - p_2 t_{p_2}(p_1-1)}\right) \tag{17}$$

and T_1 is given by

$$T_1 = \frac{p_1 t_{p_1}}{1+\alpha(p_1-1)}. \tag{18}$$

The accuracy of this empirical model was judged by comparing the predicted efficiencies with measured efficiencies for cases that could be run on a single processor for both scalar and vector nodes; in no case did the predicted efficiency vary from that measured by more than 5%. Sample comparisons are given in the next section. Note again that expressions with more empirical parameters such as equation (13) could be used in cases where it is possible to make more than two runs on two different numbers of processors.

5. TEST PROBLEMS

A series of demonstration calculations have been made on a 32-node scalar iPSC/2 at Cornell University and an eight-node vector iPSC/2VX loaned to the GE Research and Development Center by Intel Scientific Computers. The first test problem involves steady laminar flow in an axisymmetric afterburner configuration. This is the same test problem as was considered in the earlier work,[7] allowing comparison of the revised parallel algorithm with the original parallel algorithm on the iPSC/1. The improved computational performance of the iPSC/2 allowed larger

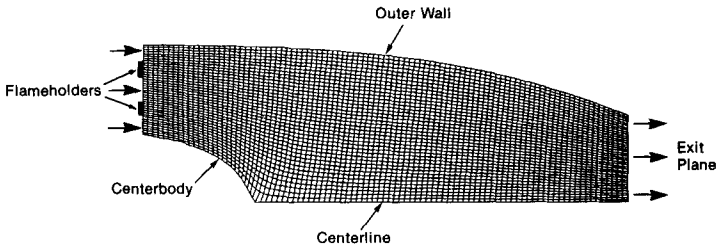

Figure 2. Body-fitted grid for axisymmetric afterburner configuration (test problems 1 and 3)

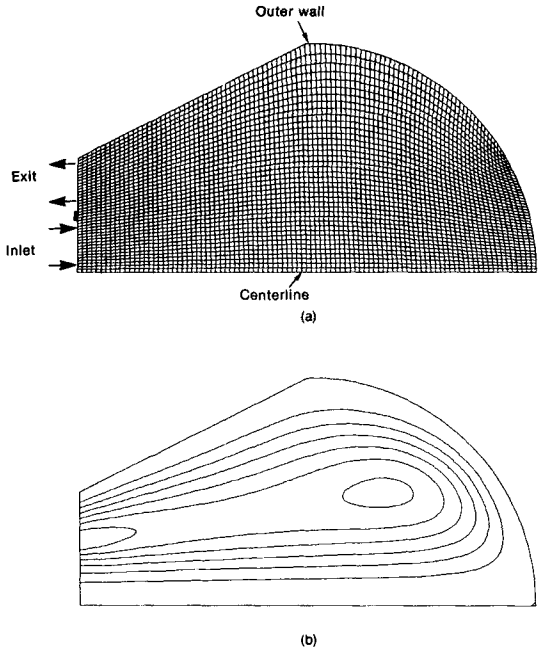

Figure 3. Test problem 2—laminar flow in conical combustor geometry: (a) body-fitted grid; (b) computed streamlines

grid sizes to be tested; three grid sizes of 64×20, 96×40 and 96×96 were considered. The 96×40 grid is pictured in Figure 2. The second test problem involves laminar flow in a conically shaped combustor geometry. This test problem was chosen because the flow is very strongly recirculating throughout the entire solution domain, as shown in Figure 3, which illustrates the 96×40 grid and the computed streamlines. The third test problem involves turbulent flow in the same afterburner configuration as in problem 1. The treatment of the turbulence requires the solution of the two additional equations from the k–ε turbulence model for the turbulent kinetic energy and the turbulence energy dissipation. For reference purposes the streamlines for both the laminar and turbulent test cases, computed on the 96×96 mesh, are shown in Figure 4.

The values for the underrelaxation factors for velocity are taken to be 0·3 and that for pressure to be 0·5, as in the earlier paper. The calculations done on the Cornell machine used the same line-by-line TDMA procedure as in the earlier work; those done on the eight-node vector machine used the point symmetric Gauss–Seidel procedure for both the scalar and vector calculations, to

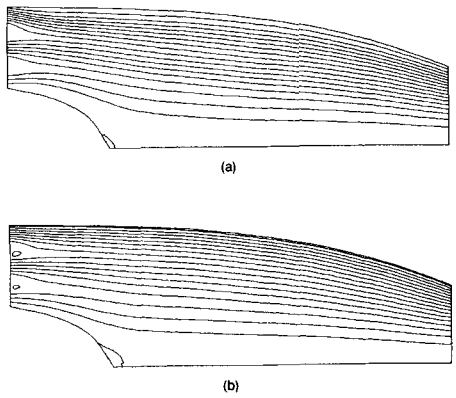

Figure 4. Computed streamlines for axisymmetric afterburner configuration: (a) test problem 1—laminar flow; (b) test problem 3—turbulent flow

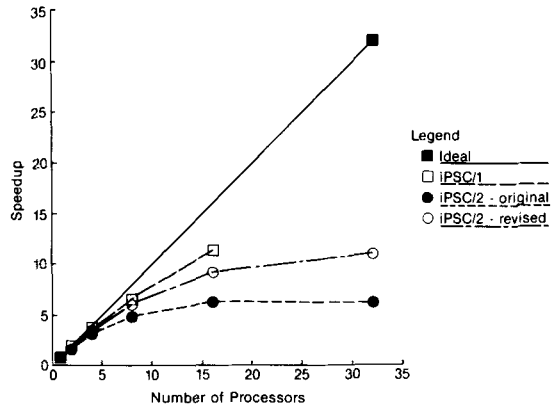

Figure 5. Comparison of parallel algorithm on iPSC/1 and iPSC/2 (64 × 20 grid)

allow for a comparison of the speed-up available from the vector processors for the same code. One iteration of the point SSOR procedure was used for the momentum and turbulence equations and 10 iterations were used for the pressure correction equation for these calculations. All solutions were taken to be converged when the mass residual fell below 10^{-3}.

Comparison with iPSC/1

Figure 5 shows the speed-ups obtained for the first test problem on a 64 × 20 grid on both the scalar iPSC/1 and scalar iPSC/2 machines. The curve marked iPSC/2-original represents the results of the original porting of the code developed in Reference 7 to the iPSC/2, making only those changes necessary to accommodate the new syntax of the communications calls on the iPSC/2 machine. With 16 processors the speed-up achieved on the newer machine was only a factor of 6·4, compared to the earlier factor of 12·3 on the iPSC/1. These results are evidence that the effective ratio t_{comm}/t_{calc} is higher in the new machine than in the old for this application. This suspicion was confirmed by looking at benchmark data measured by Intel.[17] Floating point

benchmarks for the 80387 show performance four to five times better than the 80287 coprocessor. Communication benchmarks show that the nearest-neighbour communication rates for the most common messages in this code (between 128 and 1024 bytes) are not significantly different between the iPSC/1 and iPSC/2. For a 128 byte message the iPSC/2 is only 35% faster and for a 1024 byte message is 69% faster than the iPSC/1. The big gains in communication performance for the new machine come in non-nearest-neighbour communication and for very short and very long messages, which are not prevalent in this code. In an attempt to improve the parallel efficiency of the code, further streamlining of the communications calls and the elimination of all redundant calculations in the overlapping regions were done. These improvements brought the parallel efficiency back up close to what was obtained earlier on the iPSC/1, as shown by the curve labelled iPSC/2-revised.

Figure 6 shows a comparison of the measured execution times for the same problem on the two scalar machines. The code runs two to four times faster on the iPSC/2 for the same number of processors. The single-processor case runs about four times faster on the iPSC/2 than on the iPSC/1, which is consistent with the floating point benchmarks.

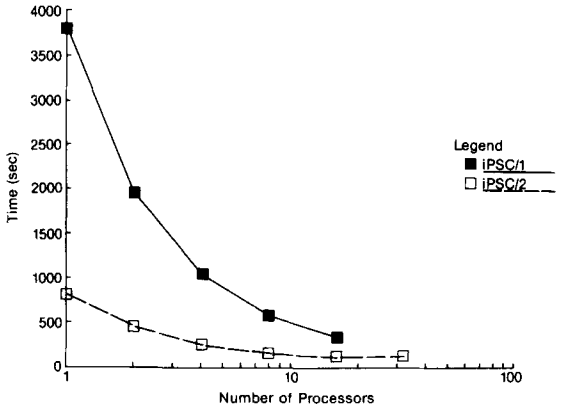

Figure 6. Comparison of timings for scalar iPSC/1 and iPSC/2 (64 × 20 grid)

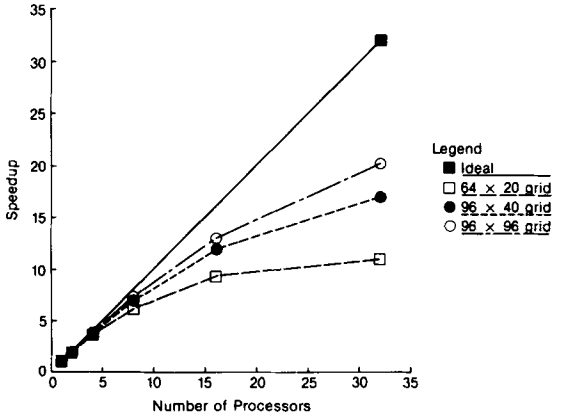

Figure 7. Speed-up of parallel algorithm of Intel iPSC/2 (test problem 1)

Effect of grid size and number of processors on scalar machine performance

Figure 7 shows the speed-ups obtained with the revised code on the 32-node scalar iPSC/2 for the first test problem as a function of grid size and the number of processors. As expected, the parallel efficiency increases as the grid is made larger. The maximum speed-up obtained was a factor of 20·2 with 32 processors, which is equivalent to a parallel efficiency of 63%.

Figure 8 shows a comparison of execution times on a number of different computers for the first test problem computed on the 96 × 40 grid. The original serial code was run on VAX minicomputers; the Cray code was a version significantly modified to vectorize efficiently (this represents the version used within GE for production purposes). With one processor the iPSC/2 is equivalent to one processor in a VAX 11/782 (equivalent to a VAX 11/780). With four processors the iPSC/2 is equivalent to a VAX 8530 and with 32 processors the performance of the iPSC/2 is about one-sixth that of a single-processor Cray X-MP.

Comparison between iPSC/2 and iPSC2/VX

Figures 9 and 10 show both the measured and predicted efficiencies for problems 1 and 3 respectively computed on the smallest grid. The estimated efficiencies are within 5% of the measured values. As expected, the efficiencies are lower for the vector machine owing to its

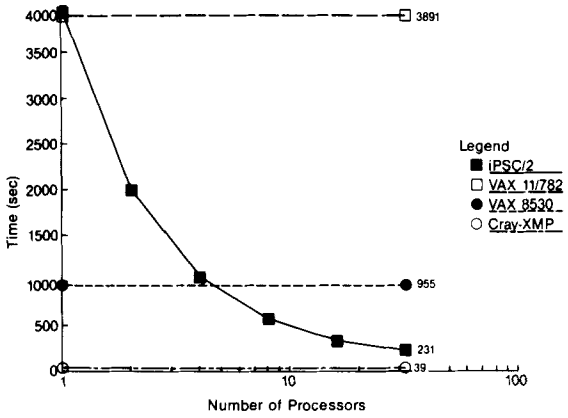

Figure 8. Timings of parallel algorithm on Intel iPSC/2—test problem 1 (96 × 40 grid)

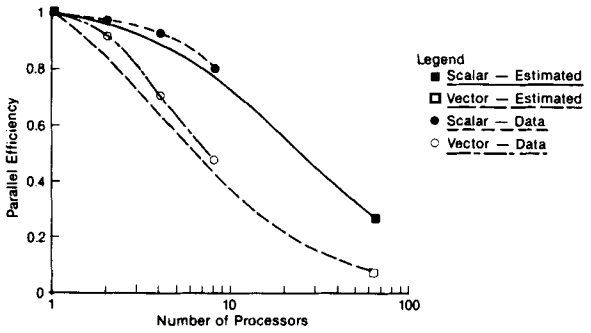

Figure 9. Parallel efficiency for problem 1 (64 × 20 grid)

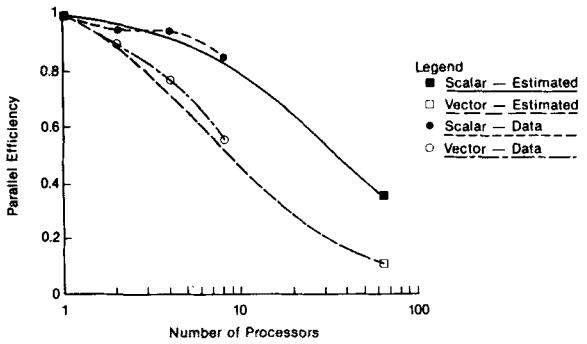

Figure 10. Parallel efficiency for problem 3 (64 × 20 grid)

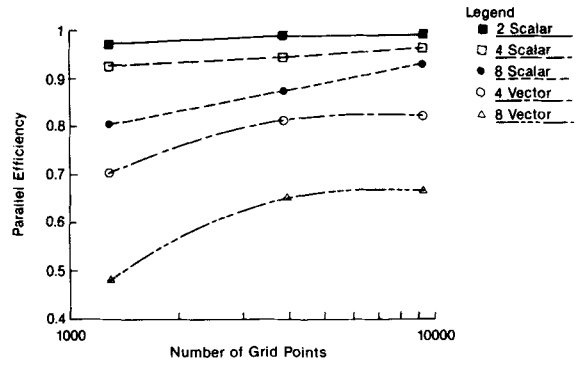

Figure 11. Effect of grid size on parallel efficiency for problem 1

increased ratio of computational speed to communication speed. For this problem, 64 processors is a practical limit owing to the stripwise decomposition. For a 64-processor machine the estimated efficiencies are projected to be about 30% with scalar processors and about 10% with vector processors. These low efficiencies reflect the small size of the problem, since each of the 64 processors would only calculate the solution for a single column of 20 grid cells in this instance.

Figure 11 shows the effect of grid size on the parallel efficiency for problem 1 for both scalar and vector processors. As expected, the efficiency increases as the grid size is increased, and once again the efficiency with the vector processors is less. For the largest probem (96 × 96 grid) the parallel efficiency with eight vector processors is 0·67, roughly equivalent to that found earlier for 32 scalar processors. The execution times for the scalar and vector hypercube are compared with a VAX minicomputer and a single-processor Cray X-MP in Figure 12. With eight processors the use of the vector processors led to a speed-up of 3·4 relative to the scalar processors and the iPSC/2VX gave about one-fifth the performance of the Cray X-MP.

Figure 13 compares the parallel efficiency between the two laminar flow problems. The parallel efficiency for the second test problem, which features more strongly recirculating flow, is lower than for the first test problem and falls off more rapidly as the number of processors is increased. Figure 14 plots the ratio of the number of iterations required for convergence for a given number of processors versus what was required for single scalar processor. Examination of this figure shows that the convergence rate of the parallel algorithm shows a 25% degradation going from

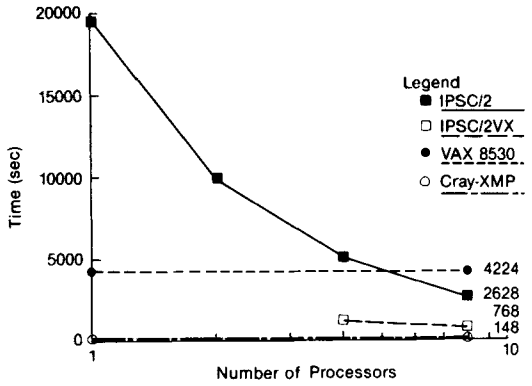

Figure 12. Timings of parallel algorithm—problem 1, laminar flow (96 × 96 grid)

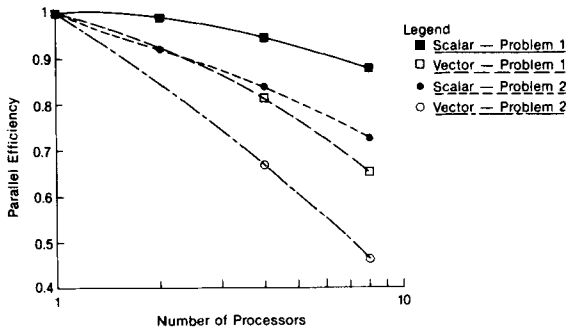

Figure 13. Comparison of parallel efficiency between problems 1 and 2

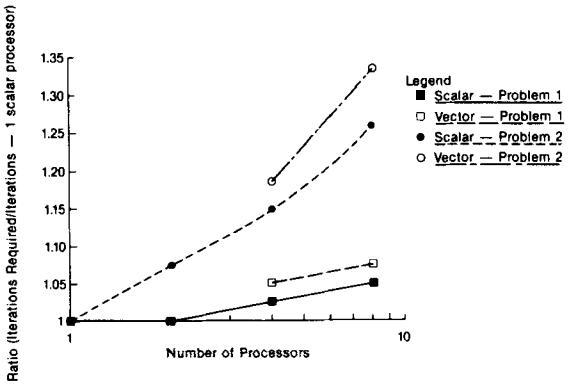

Figure 14. Comparison of convergence rate between problems 1 and 2

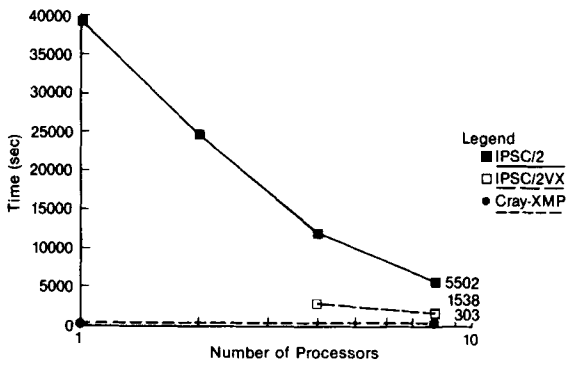

Figure 15. Timings of parallel algorithm—problem 3, turbulent flow (96 × 96 grid)

one to eight scalar processors for the second test problem, compared to only a 5% degradation for the first problem. This increased sensitivity to the number of processors results from the strong ellipticity of the flow in the second problem, which makes the subdomain solution and block correction less effective. Similar behaviour was noted for subdomain solution procedures in Reference 18. The vector algorithm shows similar behaviour but requires higher numbers of iterations owing to the less implicit nature of the vectorized point solver. Note that the vector results are normalized relative to the number of iterations required on a single *scalar* processor, so that typically the ratio would be larger than unity even with one processor.

Figure 15 shows timings for problem 3 on both the scalar and vector iPSC/2 and a single-processor Cray X-MP. The eight-processor scalar iPSC/2 runs the problem in 5502 s, the eight-processor vector iPSC/2 in 1538 s and the Cray X-MP in 303 s. The ratio of the timings between the scalar and vector hypercube, and the vector hypercube and Cray are nearly identical to those obtained for problem 1.

6. CONCLUDING REMARKS

A concurrent algorithm for solving both laminar and turbulent flow problems has been developed and successfully demonstrated on both scalar and vector distributed memory parallel computers. For large two-dimensional flow simulations, parallel efficiencies are reasonably high, being around 65% with 32 scalar processors or eight vector processors. Results with an eight-node vector iPSC/2VX give one-fifth the performance of a single-processor Cray X-MP, which translates to more than a 10-fold improvement in cost performance.

With larger machines, faster vector processors and faster interprocessor communication, this parallel algorithm has the potential to provide performance significantly faster than that of existing supercomputers. The next logical step is to extend this algorithm to three dimensions, where the problems of interest are much more computationally demanding, and run it on a machine such as the Intel iPSC/860, which has much faster processors that the Intel iPSC/2.

ACKNOWLEDGEMENTS

Thanks are due to G. MacDonald and C. Lawson of Intel for arranging the availability and support of the Intel iPSC/2VX hypercube, to the Center for Theory and Simulation in Science and Engineering at Cornell University for access to their iPSC/2, to my colleague S. Lamson for helpful discussions and to R. Mani for believing in parallel computation.

REFERENCES

1. G. M. Johnson, 'Parallel processing in fluid mechanics, in O. Baysal (ed), *Applications of Parallel Processing in Fluid Mechanics, FED Vol. 47*, ASME, New York, 1987.
2. J. L. Gustafson, G. R. Montry and R. E. Benner, 'Development of parallel methods for a 1024-processor hypercube', *SIAM J. Sci. Stat. Comput.*, **9**, 609–638 (1988).
3. G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon and D. Walker, *Solving Problems on Concurrent Processors, Vol. 1*, Prentice-Hall, Englewood Cliffs, NJ, 1988.
4. O. Baysal (ed.), *Applications of Parallel Processing in Fluid Mechanics, FED Vol. 47*, ASME, New York, 1987.
5. A. K. Noor (ed.), *Parallel Computations and their Impact on Mechanics, AMD Vol. 86*, ASME, New York, 1987.
6. M. E. Braaten, 'Applications of parallel computing in computational fluid dynamics: a review', in H. Tyrer (ed.), *Advances in Distributed and Parallel Processing, Vol. 1: Applications*, Ablex, Norwood, NJ, 1990. (in press).
7. M. E. Braaten, 'Solution of viscous fluid flows on a distributed memory concurrent computer', *Int. j. numer. methods fluids*, **10**, 889–905 (1990).
8. W. Shyy, S. S. Tong and S. M. Correa, 'Numerical recirculating flow calculation using a body-fitted coordinate system'. *Numer. Heat Transfer.*, **8**, 99–113 (1985).
9. M. E. Braaten and W. Shyy, 'A study of recirculating flow computation using body-fitted coordinates: consistency aspects and mesh skewness', *Numer. Heat Transfer.*, **9**, 559–574 (1986).
10. B. E. Launder and D. B. Spalding, 'The numerical calculation of turbulent flows', *Compt. Methods Appl. Mech. Eng.*, **3**, 269–289 (1974).
11. S. V. Patankar, *Numerical Heat Transfer and Fluid Flow*, Hemisphere, New York, 1980.
12. Q. V. Dihn, R. Glowinski and J. Periaux, 'Solving elliptic problems by domain decomposition methods with applications', in G. Birkhoff and A. Schoenstadt (eds), *Elliptic Problem Solvers II*, Academic Press, New York, 1984.
13. T. F. Chan, 'On Gray code mapping for mesh-FFTs on binary N-cubes', *RIACS Technical Report 86.17*, NASA Ames Research Center, Moffett Field, CA, 1986.
14. J. M. Ortega and R. G. Voigt, 'Solution of partial differential equations on vector and parallel computers', *SIAM Rev.*, **27**, 149–240 (1985).
15. Intel Corporation, *VAST-2 Users Guide*, Order Number 311571-001, March 1988.
16. Y. Saad and M. H. Schultz, 'Data communication in hypercubes', *Report YALEU/DCS/RR-248*, Yale University, 1985.
17. Intel Scientific Computers, *iPSC/2 Performance Report*, January 1988.
18. M. E. Braaten, 'Development and evaluation of iterative and direct methods for the solution of the equations governing recirculating flows', *Ph.D. Thesis*, Department of Mechanical Engineering, University of Minnesota, 1985.