INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN FLUIDS, VOL. 10, 889–905 (1990)

SOLUTION OF VISCOUS FLUID FLOWS ON A DISTRIBUTED MEMORY CONCURRENT COMPUTER

MARK E. BRAATEN

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

SUMMARY

A concurrent algorithm for the solution of the Navier–Stokes equations expressed in curvilinear co-ordinates has been developed for execution on a distributed memory parallel computer. This algorithm offers the ultimate promise of near-supercomputer performance on relatively low-cost parallel computers. The new algorithm is based on an existing serial pressure-correction-based algorithm, and uses domain decomposition to partition the problem onto the processors. The algorithm is demonstrated on an Intel iPSC for a complicated two-dimensional laminar flow problem, for various grid sizes and numbers of processors. Initial results based on straightforward domain decomposition showed that the speed-up per iteration approached 100% parallel efficiency as the grid size was increased, but that the convergence rate of the algorithm deteriorated relative to the original serial algorithm as the number of processors was increased, limiting the speed-up achieved. This degradation in convergence rate was traced to a poorer solution of the pressure correction equation in the concurrent procedure. The addition of a global block correction procedure, implemented via efficient global communications routines, remedied this problem, making the convergence rate of the concurrent procedure equivalent to the serial algorithm. The maximum speed-up achieved with the revised concurrent algorithm was a factor of 12·3 with 16 processors, representing a parallel efficiency of 77%.

KEY WORDS Computational fluid dynamics Parallel computing Parallel processing Domain decomposition

INTRODUCTION

A major obstacle to the increased use of computational fluid dynamics in engineering design continues to be the long run times and high cost of the computer simulations. As an example, a run of a 3D finite volume combustor code developed earlier[1] requires 1–2 h of CPU time on a Cray-XMP supercomputer for a grid with 75 000 grid points. Such a calculation would require hundreds of hours on a minicomputer or engineering workstation, making it impractical for routine design purposes. More exotic codes such as those used for the direct simulation of turbulence require even greater computational resources.

Parallel processing offers the promise of greatly reducing the execution times for CFD codes by using many processors to attack the problem simultaneously. Recent advances in VLSI technology have led to the development of a class of relatively low-cost parallel computers from companies such as Intel, Ametek, NCUBE and Alliant, which use a moderate number of inexpensive processors. Such machines have been demonstrated to offer near-Cray performance for some applications.[2,3] The individual processors range from 16-bit microcomputers in machines such as the Intel iPSC/1 and Ametek System 14, to fast vector processors in machines such as the Alliant FX/8 and the Intel iPSC/VX. Other parallel supercomputers with small numbers of supercomputer processors are available or under development (such as the ETA[10] and Cray 3) which offer higher ultimate performance, but their high cost and limited availability make them less attractive for routine use for the foreseeable future.

0271–2091/90/080889–17$08.50

Received March 1989
© 1990 by John Wiley & Sons, Ltd.

If low-cost processors are to be used, tens or hundreds may be required to provide the level of performance required for CFD codes. Concurrent computers of this type can either be of a shared memory or distributed memory architecture. Memory conflicts can limit the number of processors that can be effectively used in a shared memory system.[4] Distributed memory systems do not suffer from this limitation, and applications have been demonstrated on up to 1024 processors while retaining high parallel efficiencies.[2] The disadvantage of distributed memory architectures is that the problem to be solved must be explicitly partitioned by the programmer onto the various processors in such a way that load balancing is maintained and communication between processors is minimized and well ordered. For some problems it may not be easy or even possible to find a satisfactory means of doing this partitioning. Fortunately, for finite volume fluid mechanics algorithms, a simple and natural geometrical partitioning of the problem can lead to good load balancing and reasonably minimal communications costs. Consequently, distributed memory architectures appear very well suited for the solution of CFD problems, as has been noted by others,[4-6] and are the focus of attention here. Since the field of parallel processing is developing so rapidly, with machines appearing (and disappearing) frequently, the concurrent algorithm developed here was designed to be applicable to the general class of distributed memory computers rather than to a particular machine.

In this work, the development of a concurrent algorithm for the solution of the Navier–Stokes equations expressed in curvilinear co-ordinates is described. First, the original serial algorithm is reviewed briefly, and then the basic development of the concurrent algorithm is described. A theoretical analysis of the potential speed-up available from this algorithm is followed by a demonstration of the algorithm on an Intel hypercube for a complicated 2D laminar flow problem. Degradation in the convergence rate of this initial algorithm as the number of processors was increased led to the addition of a global block correction scheme for the solution of the pressure correction equation to overcome this problem. This addition makes the convergence rate of the concurrent procedure equivalent to that of the serial algorithm and significantly improves the parallel efficiency.

2. REVIEW OF ORIGINAL 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 7 and 8. Only a brief outline of the original algorithm is given here; the reader is referred to the original references for the details. For simplicity, the discussion is limited to incompressible laminar flows, but the algorithm is extendable to more general flows, involving turbulence, compressibility and chemical reaction.

The governing conservation equations typically can be written in Cartesian co-ordinates for the dependent variable ϕ in the following form, following the notation of Reference 7:

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

where Γ is the effective diffusion coefficient and R is the source term. When new independent variables ξ and η are introduced, equation (1) changes according to the general transformation $\xi=\xi(x,y)$, $\eta=\eta(x,y)$. Equation (1) can be rewritten in (ξ,η) co-ordinates as follows:

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

where 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 the $\xi-\eta$ co-ordinates.

A staggered grid system is adopted, following the standard practice for incompressible finite volume algorithms. The scalar variables (p, ρ) are located at the centre of the control volumes, and the Cartesian and contravariant velocity components are located on the faces of the control volumes. 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 successive line relaxation method is used to solve the resulting finite difference equations for each variable ϕ.

The momentum and continuity equations, along with appropriate boundary conditions, make up the complete description of a laminar flow. The solution of these coupled equations makes up the kernel of any computational fluid mechanics algorithm. Laminar flows are commonly used to test the performance of numerical algorithms since the effects of the pressure–velocity coupling, which usually controls the convergence of the algorithm, are most clearly evident for such flows.[9]

The method used to solve the coupled system of momentum and continuity equations is a pressure correction method similar to that described in Reference 10. Following the notation of Reference 7, the momentum equations can be written as

$$A_P^u u_P = \sum_{i=E,W,N,S} A_i^u u_i + D^u + (B^u p_\xi + C^u p_\eta), \tag{4}$$

$$A_P^v v_P = \sum_{i=E,W,N,S} A_i^v v_i + D^v + (B^v p_\xi + C^v p_\eta). \tag{5}$$

The D represent coefficients arising from the viscous cross-derivative terms, and the B and C represent the projected areas acted on by the pressure gradients in the ξ- and η-direction respectively. The momentum equations can be solved, for a given pressure distribution p^*, to yield a tentative velocity field u^*, v^*. Since u^*, v^* do not necessarily satisfy the continuity equation, they and the guessed pressure field p^* must be updated. The corrected velocities and pressure are obtained from

$$p = p^* + p', \qquad u = u^* + u', \qquad v = v^* + v'. \tag{6}$$

Through manipulation of the momentum equations, and the formulae defining the contravariant velocities U, V in terms of u, v and the various metric derivatives, the velocity corrections U', V' can be expressed in terms of p' through the relations

$$U' = (B^u y_\eta - B^v x_\eta) p'_\xi, \qquad V' = (C^v x_\xi + C^u y_\xi) p'_\eta. \tag{7}$$

These expressions are then substituted into the discrete continuity equation

$$(\rho U)_e - (\rho U)_w + (\rho V)_n - (\rho V)_s = 0, \tag{8}$$

leading to the final incompressible form of the pressure correction equation:

$$(\rho U^*)_e + (\alpha\rho)_e(p'_E - p'_P) - (\rho U^*)_w - (\alpha\rho)_w(p'_P - p'_W) + (\rho V^*)_n + (\beta\rho)_n(p'_N - p'_P) - (\rho V^*)_s - (\beta\rho)_s(p'_P - p'_S) = 0, \quad (9)$$

where α and β are the coefficients derived by combining equations (7) and (8). This equation is solved, and the pressure and the velocity components are updated, completing one global iteration. Owing to the non-linearity of the problem, a number of global iterations are required to obtain a converged solution.

3. CONCURRENT IMPLEMENTATION

Effective implementation of the algorithm described in Section 2 on a distributed memory concurrent computer requires the satisfactory resolution of three major computational issues, namely (1) load balancing, (2) minimization of communications costs and (3) the development of an efficient concurrent algorithm. The development of an efficient concurrent algorithm is the critical step in making effective use of any parallel architecture. In the attempt to keep all of the processors busy generating floating-point numbers at impressive combined Mflop rates, it is easy to lose sight of the fact that the true goal is to achieve the same solution to the physical problem in less time than with a single processor.

In this section, the initial development of the concurrent algorithm based on domain decomposition is discussed. Description of the global block correction scheme, which was found to significantly improve the performance of the basic concurrent algorithm, is deferred until Section 6.

The concurrent algorithm described in this paper was implemented on a 16-node, memory-enhanced Intel hypercube. Each processor is rated at only 0·03 Mflop; consequently, ideal performance with a 16-node system is only about 0·5 Mflop. Consequently, the emphasis here is not on the absolute performance of the code but on the speed-up obtained with multiple processors relative to a single processor. The ultimate goal is to run the code on a machine with faster processors to achieve near-supercomputer performace. Although the basic concurrent algorithm is applicable to any distributed memory parallel processor, details of the implementation motivated by the specific architecture of the iPSC are mentioned when appropriate.

A widely used technique for partitioning the solution of a partial differential equation onto a number of processors is the domain decomposition method.[11] This method represents an extension of the classical Schwarz alternating method[12] to a parallel architecture. 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 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 neighbouring 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.

Since domain decomposition appears to be a natural (and general purpose) way of partitioning problems involving the solution of PDEs by either finite volume or finite element methods for execution on a parallel computer, it was the basis of the method adopted here. The solution domain is divided up into a number of overlapping subdomains, one per processor, with the number of grid points in each subdomain approximately equal. Since the work required for each point is roughly the same, this ensures reasonable load balancing.

In a curvilinear co-ordinate code, much of the storage burden is taken up by the storage of the grid point positions and the many metric derivative terms that arise from the co-ordinate transformation. Since the grid is held fixed in the course of the calculation, it is far more computationally efficient to compute the metric terms once and for all at the beginning and store the results, rather than recompute the metric terms repeatedly throughout the calculation. A geometrical decompostion has the advantage that this metric information needs only to 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. Only the interfacial values of the solution variables, which are updated in the course of the solution, need to be passed between processors. The only storage penalty that arises from the domain decomposition method is due to the need to store the metric information and solution variables in the overlapping regions twice.

The solution domain can be divided into strips, boxes or arbitrarily shaped regions with roughly the same number of points. The key factor for minimizing the communications costs is to maximize the ratio of computation in each processor to the communication between processors. In general, the computational work in each subdomain is dependent on the number of control volumes (volume) of the subdomain, while the amount of communication between subdomains depends on the number of boundary cells (surface area) of the subdomain. Although a boxwise decomposition can lead to a smaller surface-to-volume ratio of the subdomains than stripwise decomposition, decomposition by strips leads to a smaller number of messages. With the structured grid used in a finite volume formulation, there is no need to resort to arbitrarily shaped regions, although they are useful in a finite element context. The major disadvantage of stripwise decomposition is that the number of processors that can be effectively used is limited to something of the order of n, where n is the largest number of control volumes in any given direction. For both two- and three-dimensional problems, n will seldom exceed 100; consequently, the number of processors that can be effectively used is similarly limited. In three-dimensional problems, boxwise decomposition may be a better choice since it will allow a larger number of processors, and consequently more computational power, to be applied to the problem.

The best choice for the partitioning of the problem depends on the number of processors available, the ratio of computational speed to communications speed for the machine under consideration, and the importance of message latency to the cost of message passing. The Intel iPSC/1 hypercube consists of an Intel 80286-based host machine and a computational cube consisting of up to 128 Intel 80286-based processors, connected in a hypercube arrangement.[13] All communication is slow relative to the speed of computation, and message latency dominates the cost of sending short messages. Consequently, stripwise decomposition is a good choice for this machine since it minimizes the total number of messages.

Owing to the nature of the staggered grid used, two rows of grid points must be overlapped to ensure that all interior u-velocities appear as interior points in at least one subdomain. This also causes one extra row of v-velocities and pressures to be solved redundantly in each subdomain, which increases the computational effort for the concurrent algorithm somewhat over that of the original serial algorithm.

There are a number of characteristics of the iPSC that influence the detailed coding of the algorithm required to achieve good concurrent performance. There is a significant overhead associated with routing messages from one processor to the next, putting a premium on nearest-neighbour communication. This overhead is overcome by mapping the hypercube to the required linear array through the use of binary reflected Gray codes.[14] Gray codes are sequences of n-bit binary numbers with the properties that any two successive numbers differ in only one bit and that all binary numbers with n bits are included. Each processor is then a nearest neighbour to the two processors that are handling the two adjacent subdomains.

As the code is implemented here, the host machine is used only to read in the grid file and the input variables, monitor convergence and write out the final solution. Note that on the existing iPSC, all I/O must be done on the host machine. An efficient concurrent broadcast routine, contained in Intel's protocode library, is used to pass all of the input information from the host to each processor in the form of identical messages. In parallel, each processor then extracts the portion of the grid file that it needs and the input variables from these messages. This procedure is much more efficient than having the host machine sequentially create and send individualized messages to each node. The host machine is slow, and the cost of host-to-processor messages is high. The concurrent broadcast routine requires only one host-to-processor message, with all the other messages passed via a spanning tree from processor to processor, with some message passing occurring in parallel. Convergence is monitored by sending the mass residual to the host machine and comparing it to the prescribed convergence criterion. A partial mass residual is computed by each processor and summed together using a concurrent concentration routine that forms the total mass residual and sends it to the host, via a spanning tree, in a reverse manner to what was done in the concurrent broadcast. This convergence check is performed every fifth iteration to reduce the overhead of the node-to-host communications.

The existing serial solution algorithm solves the governing equations in the following order: x-momentum, y-momentum and finally pressure correction. The same structure is retained in the concurrent implementation. In parallel, the coefficients for the x-momentum equation are computed for each subdomain. In parallel, a few sweeps of an iterative solution procedure are performed in each subdomain. During this process, the boundary values for u in each subdomain are held fixed. Next, the boundary values for u are exchanged between subdomains, and the line-by-line solution procedure is repeated. The number of repetitions of the line-by-line solution and the exchange of the boundary values is prescribed by the user.

Upon completion of the x-momentum equation, the procedure is repeated for the y-momentum equation and finally for the pressure correction equation. 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 spirit of S'_p as defined by Ortega and Voigt.[15] Secondly, if enough repetitions of the line-by-line solver and the exchange of boundary points are performed so that a converged solution is obtained for each equation before proceeding to the next 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 it is not the usual practice to solve each equation fully to convergence at each step, since the coefficients are only tentative, this similarity suggests that the new concurrent algorithm will show comparable convergence behaviour to the serial algorithm, which has proved to converge well for a large variety of problems.

4. THEORETICAL SPEED-UP ANALYSIS

Theoretical estimates can be made of the speed-up that can be attained with p processors relative to a single processor for the basic concurrent algorithm. The speed-ups achieved will be somewhat less than linear owing to the following factors: (1) the additional computational work due to the overlapping regions, (2) the cost of the message passing and (3) the reduction in convergence rate due to the change in the solution procedure from line-by-line over a single domain to the concurrent Schwarz alternating procedure.

Expressions for the degradation factors resulting from overlapping and communication costs can be formulated by counting the number of arithmetic operations and messages passed as a function of the problem size and the number of processors. The overlapping of the subdomains causes more lines of control volumes to be solved than in the serial algorithm, increasing the computational work, and the cost of message passing represents an overhead not found in the serial algorithm.

The computational effort in each subdomain is proportional to the number of control volumes in the subdomain and can be expressed as

$$(t_{\mathrm{CPU}})_{\mathrm{p}} = k_{\mathrm{CPU}}\left(\frac{NI+p-1}{p}\right)NJ, \tag{10}$$

where k_{CPU} represents the CPU time per control volume per iteration of the algorithm, and NI and NJ are the number of computational points in each of the two co-ordinate directions. Each processor exchanges essentially the same mix of messages with its two adjacent processors at each iteration. On the Intel iPSC, the overhead for message passing is so high that messages less than 1024 bytes all take essentially the same amount of time to transfer.[11] Since most of the message traffic consists of short messages, the time per iteration that a processor spends passing messages can be expressed as

$$(t_{\mathrm{m}})_{\mathrm{p}} = k_{\mathrm{m}}(2n_{\mathrm{m}}). \tag{11}$$

Here k_{m} represents the average time required to send and receive a short message, and n_{m} represents the number of messages exchanged with an adjacent processor per iteration. In this simple analysis, the time that processors spend idle waiting for messages that have not yet been sent, owing to a lack of synchronization between processors, is included in this term.

The ratio of CPU time per iteration, including both computation and message passing, for the concurrent algorithm with p processors to the CPU time for the original serial algorithm becomes

$$\frac{t_{\mathrm{p}}}{t_{\mathrm{s}}} = \frac{k_{\mathrm{CPU}}[(NI+p-1)/P]NJ + 2k_{\mathrm{m}}n_{\mathrm{m}}}{k_{\mathrm{CPU}}NI \cdot NJ}. \tag{12}$$

The ratio of the total computational time for the concurrent algorithm to that for the serial algorithm is given by

$$\frac{T_{\mathrm{p}}}{T_{\mathrm{s}}} = \frac{i_{\mathrm{p}}t_{\mathrm{p}}}{i_{\mathrm{s}}t_{\mathrm{s}}}, \tag{13}$$

where i_{p} and i_{s} are the numbers of iterations required to obtain a solution to the same level of convergence for the parallel and serial algorithms respectively. After rearrangement, the following expression for the speed-up obtained with the concurrent algorithm with p processors over the original serial algorithm on one processor is obtained:

$$S = \left(\frac{i_{\mathrm{s}}}{i_{\mathrm{p}}}\right)\left(\frac{1}{1+(p-1)/NI + (k_{\mathrm{m}}/k_{\mathrm{CPU}})(2n_{\mathrm{m}}p/(NI \cdot NJ))}\right)p \tag{14}$$

In a general sense, the speed-up S can be expressed as

$$S = \varepsilon p = \left(\frac{i_{\mathrm{s}}}{i_{\mathrm{p}}}\right)\left(\frac{1}{1+O+C}\right)p. \tag{15}$$

Here the term ε equals the parallel efficiency, which is less than unity because of the three factors

listed above, namely the overlapping penalty O, the communication cost C and the reduction in convergence rate (i_s/i_p).

The important thing to note from the above expressions is that the penalties due to overlapping and communication become smaller and smaller as the problem size gets bigger, with everything else fixed. Hence for a large enough problem, the only degradation from a linear performance improvement will result from any reduction in convergence rate that may result. Unfortunately, this degradation cannot be predicted analytically and must be determined from computational experiments. Note that owing to the particular formulation of the concurrent algorithm adopted here, the same convergence rate as the original serial algorithm can always be achieved by increasing the number of line-by-line sweeps. However, since this increases the CPU work per iteration, the total CPU time may or may not decrease. The optimum number of sweeps for each variable for the concurrent algorithm must also be determined via numerical experiments and is not necessarily the same as for the original serial algorithm.

5. DEMONSTRATION RUNS

A series of demonstration runs of the concurrent algorithm was made on a 16-node, memory-enhanced Intel iPSC/1 hypercube at the University of Lowell, MA. The concurrent version of the algorithm was first developed and tested using Intel's hypercube simulator, running under Unix 4.2 BSD on a SUN 3/160 workstation. Although it certainly would be useful to confirm the performance of the concurrent algorithm for a much wider range of flow configurations, the similarity of the concurrent algorithm to the serial algorithm, which has been widely tested, gives confidence that the limited results presented here will be representative of the performance of the algorithm in general.

The test case selected involves steady laminar flow in the axisymmetric afterburner configuration shown in Figure 1. As described earlier, it is useful to study laminar flows since the solution of the coupled momentum and continuity equations forms the kernel of any computational fluid dynamics algorithm and can be most clearly studied in laminar flow. The solutions presented here are a first step towards a realistic afterburner simulation, which will include equations for turbulence and combustion that can be solved by the same basic procedure.

Two body-fitted grids, one with 32 × 20 nodes and the second with 64 × 20 nodes, were used in the initial calculations. Both grids are shown in Figure 2. For reference, the converged solution for the fine grid is shown in Figure 3 by means of plots of the calculated velocity vectors and the streamlines. Although the flow is laminar, the flowfield is not simple and contains wake regions behind the flame-holders and a recirculation zone near the trailing edge of the centrebody.

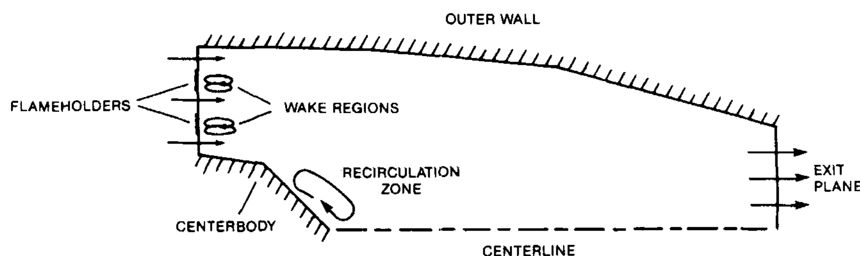
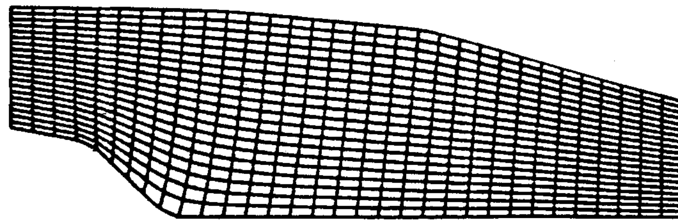
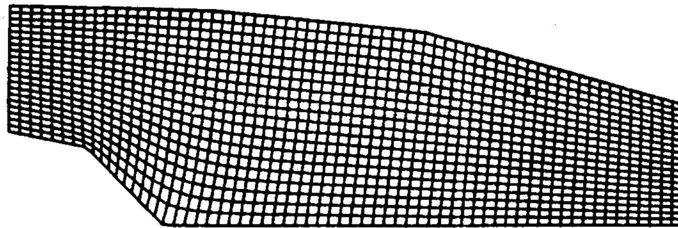


Figure 1. Axisymmetric afterburner configuration

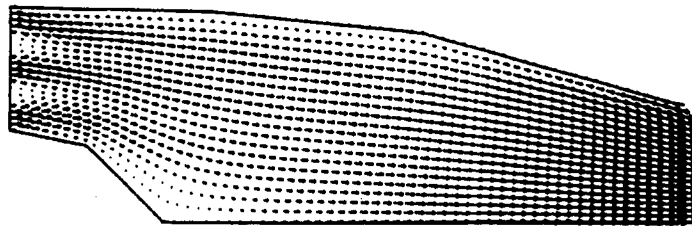


(a)

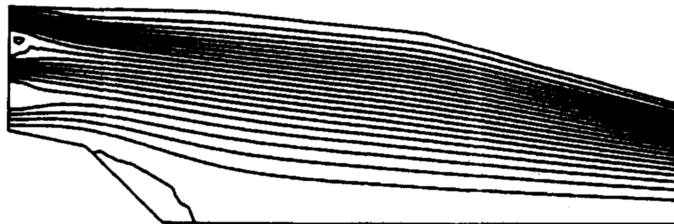


(b)

Figure 2. Computational grids used for test problem: (a) coarse 32×20 grid; (b) fine 64×20 grid



(a)



(b)

Figure 3. Converged solution for the fine grid: (a) velocity vectors; (b) streamlines

Although the performance of the original serial algorithm is affected by the choice of such parameters as underrelaxation factors for the velocities and pressure, and the number of line sweeps for each variable, previous studies[16] have shown that the sensitivity is not that great provided reasonable values are used. In this work, no attempt has been made to optimize these factors for each run; rather, reasonable values of these parameters were held fixed for all runs. The underrelaxation factors for the x- and y-momentum equations were taken equal to 0·3, and that for pressure was taken equal to 0·5. Two sweeps of the line-by-line procedure were used for both x- and y-momentum, with three sweeps taken for pressure correction.

The number of iterations required for convergence is dependent on the choice of convergence criterion. Here the solutions were taken to be converged when the normalized mass residual was less than 10^{-3}. It was confirmed that the converged solutions obtained with the concurrent algorithm were independent of the number of processors used and identical to those obtained with the original serial algorithm.

5.1. Initial results

In this sub-section, the results obtained with the basic concurrent algorithm (without the global block correction for the pressure correction equation) are described. As discussed earlier in Section 4, as the grid size is increased, the speed-up per iteration of the algorithm should approach a linear speed-up with the number of processors, since the penalties associated with overlapping and communication become less significant. Figure 4 demonstrates that the speed-ups obtained on the test problem for the two grid sizes exhibit this trend. With 16 processors, the maximum speed-up per iteration increased from 11·12 for the coarse grid to 12·84 for the fine grid, representing parallel efficiencies of about 70% and 80% respectively. The major contribution to

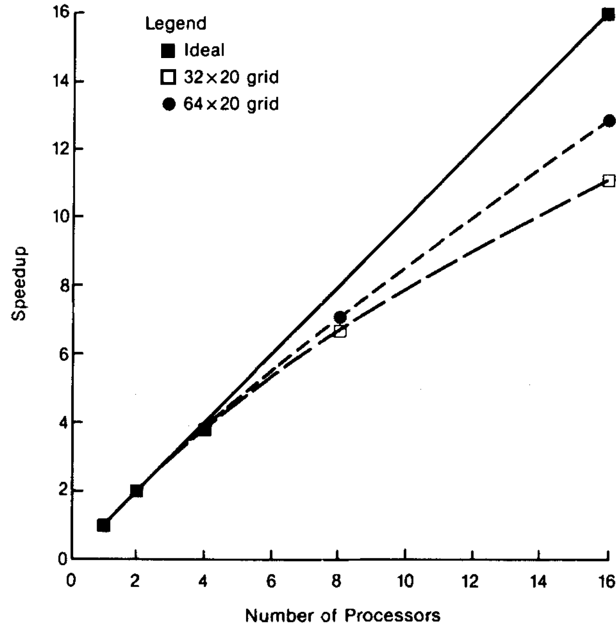


Figure 4. Speed-up per iteration for initial concurrent algorithm on Intel iPSC

the efficiency being less than 100% comes from the overlapping of the subdomains rather than from the communications costs, which appear minimal.

However, the speed-up in terms of total computational time, shown in Figure 5, does not show the same improvement as the number of grid points is increased, owing to a degradation in the convergence rate of the algorithm as the number of grid points and the number of processors are increased. The convergence paths, as a function of the number of processors, are shown for the fine grid in Figure 6. Note that the original serial algorithm (one node) shows a smooth monotonic reduction in the mass residual, reaching an asymptotic rate of convergence that is substantially less than the initial convergence rate. The concurrent algorithm shows a noisier convergence path with a slower initial rate of convergence, but with a similar asymptotic rate of convergence. With 16 processors, the convergence rate was 24% less for the coarse grid and 40% less for the fine grid than the corresponding rates for the serial algorithm.

Earlier studies of the serial algorithm have shown that the elliptic nature of the pressure correction equation makes it more difficult to converge than the momentum equations, which tend to be convection-dominated.[17] Schwarz's alternating method is also known to be slowly convergent for highly elliptic problems when many subdomains are used.[18] A series of runs was made, varying the number of line sweeps for the pressure correction equation, for the fine grid test case using the concurrent algorithm with 16 processors. Figure 7 shows that the convergence rate of the concurrent algorithm approaches that of the serial algorithm as the number of pressure correction sweeps is increased, as expected. This indicates that the reason for the slower convergence of the concurrent algorithm in the original case was a poorer solution of the pressure correction equation with the same number of sweeps used in the serial procedure. It is interesting to note from Figure 8 that the total computational time required by the initial concurrent algorithm is relatively insensitive to the number of line sweeps used for pressure correction, provided that at least three sweeps are performed. Any improvement in convergence rate achieved

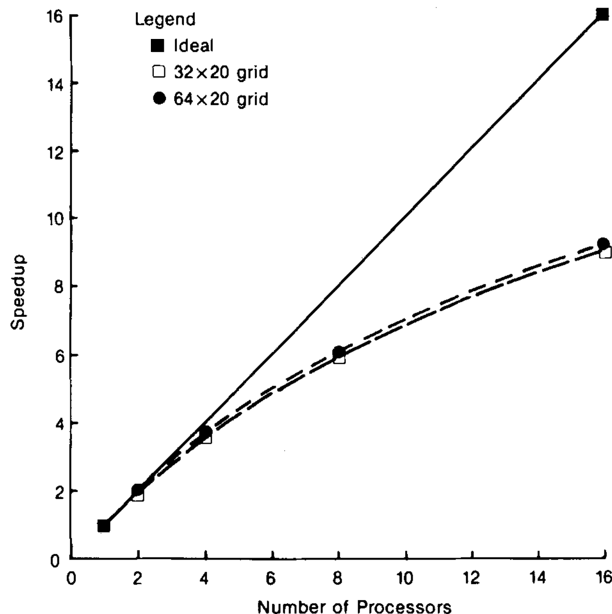


Figure 5. Speed-up of total time for initial concurrent algorithm on Intel iPSC

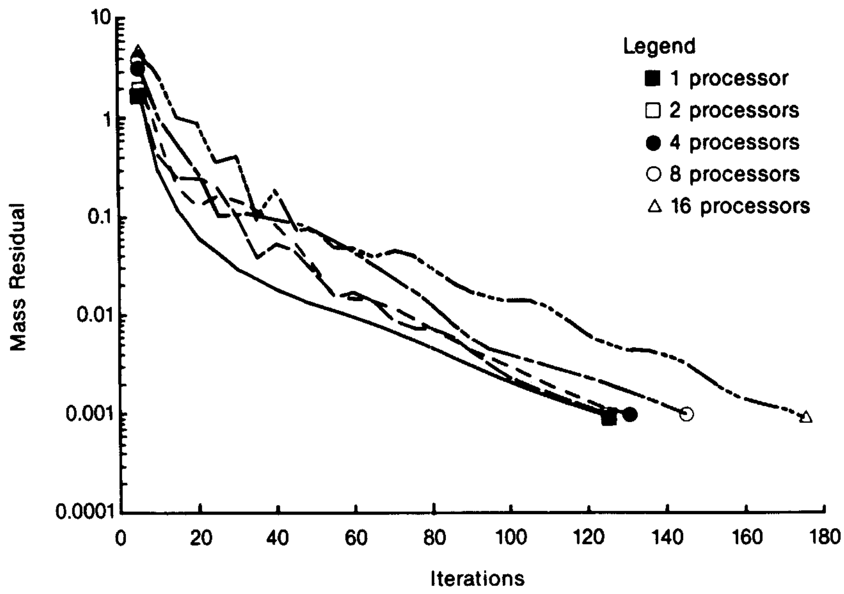


Figure 6. Convergence behaviour of initial concurrent algorithm on fine 64 x 20 grid

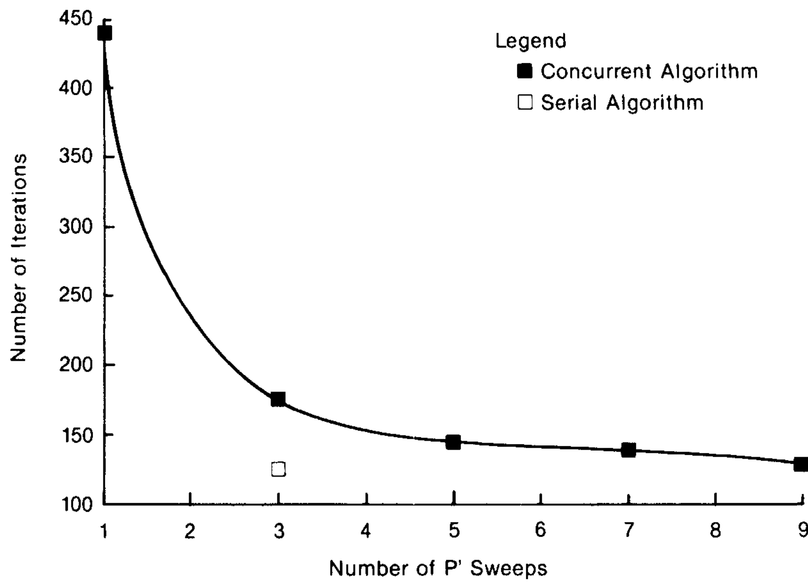


Figure 7. Effect of number of line sweeps for pressure correction equation on number of iterations required for convergence (64 x 20 grid, 16 processors)

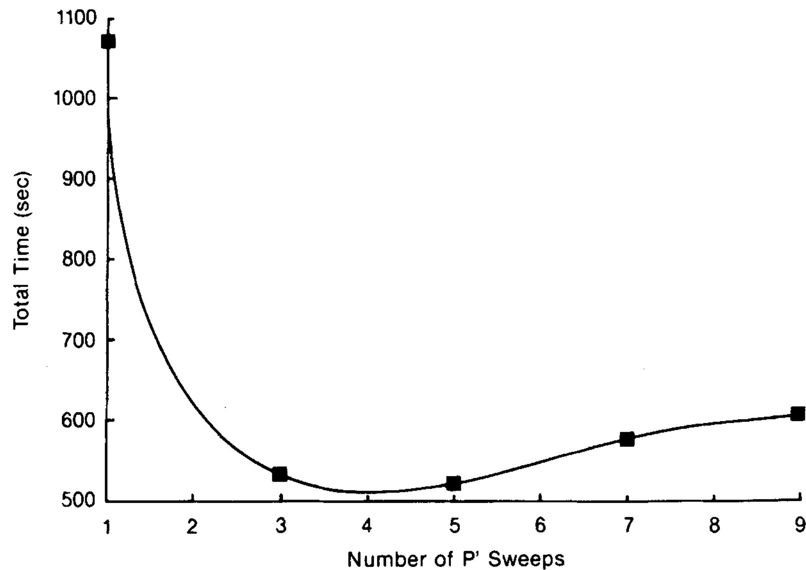


Figure 8. Effect of number of line sweeps for pressure correction equation on total time required by initial concurrent algorithm (64 × 20 grid, 16 processors)

by performing more than five sweeps is more than offset by the increased cost per iteration that results.

6. GLOBAL BLOCK CORRECTION PROCEDURE

At this point, the use of block correction or multigrid techniques to improve the parallel solution of the pressure correction equation were considered, in an attempt to minimize the degradation of the convergence rate noted for large numbers of processors without significantly increasing the cost per iteration. Multigrid methods had been previously implemented on hypercubes[19] and had been shown to improve the convergence of the pressure correction equation for the serial algorithm.[20] The importance of the block correction technique used in the original serial algorithm was demonstrated by turning off the procedure and noting a similar degradation in overall convergence rate as was noted in the concurrent procedure.

A brief review of the block correction procedure used in the serial procedure[21,22] is in order. In this procedure, a sequence of one-dimensional corrections is made to the tentative solution, first along vertical columns of grid cells and then along horizontal rows of grid cells. The correction equations are obtained by summing the original difference equations, equation (3), over the columns (or rows) of grid cells. For the corrections along the columns, the corrected values take the form

$$\phi_{ij} = \phi^*_{ij} + \psi_i, \tag{16}$$

where ϕ^*_{ij} is the tentative solution for the variable ϕ at the grid point (i,j) and ψ_i is the correction for column i.

The correction equations take the form

$$B_P\psi_i = B_E\psi_{i+1} + B_W\psi_{i-1} + R_i, \tag{17a}$$

where

$$B_P = \sum_j (A_P - A_N - A_S), \tag{17b}$$

$$B_E = \sum_j A_E, \tag{17c}$$

$$B_W = \sum_j A_W, \tag{17d}$$

$$R_i = \sum_j [(S_\phi)_P - A_P\phi_P^* + A_E\phi_E^* + A_W\phi_W^* + A_N\phi_N^* + A_S\phi_S^*]. \tag{17e}$$

The equations for the corrections along the rows appear similarly, with the summations taken over i rather than j. These equations form a tridiagonal system which can be solved directly via the tridiagonal matrix algorithm (TDMA) (see e.g. Reference 10). The resulting corrections are then added to the tentative solution. The corrected values resulting from the application of this procedure satisfy overall conservation over each row (or column) of control volumes.

In the concurrent implementation of this procedure, the computation of the corrections requires global communication along the processors. For the column corrections, each processor computes the coefficients of the correction equations corresponding to the columns of control volumes within its subdomain. These coefficients are then passed via a spanning tree to a single processor (designated as the root node), which computes the corrections using TDMA. The root node then broadcasts the corrections back to the other processors, which apply them to the solution within their respective subdomains. For the row corrections, each processor computes a partial sum for each correction coefficient, representing the contribution of the columns within its subdomain to the total. These contributions are then passed to the root node via a spanning tree, and as before, the root node computes the corrections and broadcasts them back to the other nodes.

Since the time required for the global communication of the coefficients dominates the time required for the block correction, the key to effective implementation is the use of efficient global communications routines based on spanning trees which allow much of the communication to be done in parallel. In Reference 17 it was demonstrated that the momentum equations converge very quickly without the need for block correction or multigrid, owing to their convection-dominated nature. In Reference 22 it was found that multiple sweeps of the iterative solver between applications of the block correction procedure proved more efficient than doing only one sweep of the solver per application of the block correction. Based on these observations, the block correction procedure was performed only for the pressure correction equation. For each iteration of the pressure correction equation, the block correction is done only once, followed by a fixed number of sweeps of the line-by-line procedure. This procedure was found to significantly reduce the time required to perform the block correction while retaining its full benefits.

The earlier series of test cases was repeated using the revised concurrent algorithm with the global block correction scheme applied to the solution of the pressure correction equation. As shown in Figure 9, the convergence behaviour of the block-corrected algorithm is virtually equivalent to that of the serial algorithm, and almost independent of the number of processors. This demonstrates the success of the block correction procedure in restoring the convergence rate of the parallel solution of the pressure correction equation to that obtained with the original serial procedure. In fact, the convergence rate of the block-corrected procedure sometimes even improves slightly as the number of processors is increased. Figure 10 compares the speed-up in total time for the block-corrected concurrent algorithm to the initial concurrent algorithm and the original serial algorithm. The addition of the global block correction is clearly effective in

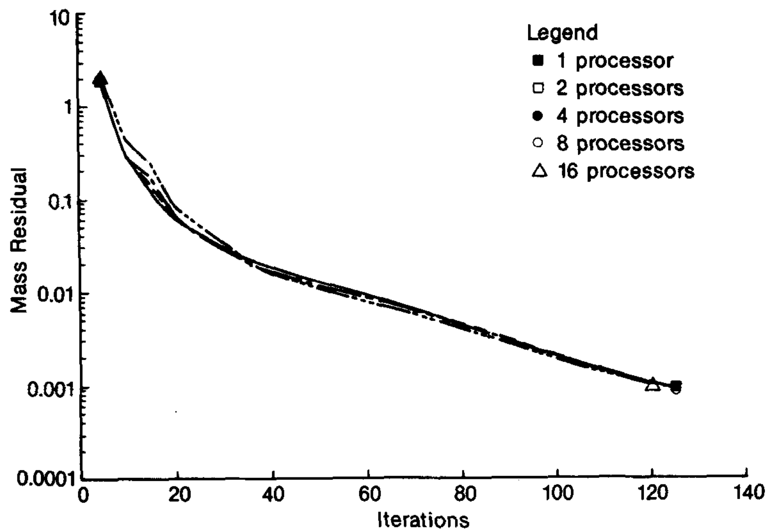


Figure 9. Convergence behaviour of revised (block-corrected) concurrent algorithm on fine 64 × 20 grid

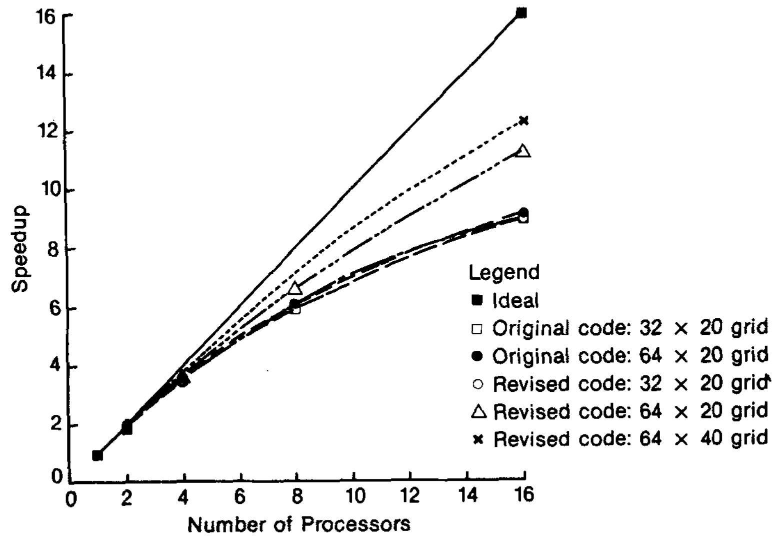


Figure 10. Speed-up of total time for revised concurrent algorithm on Intel iPSC

increasing the parallel efficiency of the concurrent algorithm as the grid size is increased. For a finer 64 × 40 grid, similar to those shown in Figure 2, a maximum speed -up of 12·3 was obtained with 16 processors, which represents a parallel efficiency of 77%.

7. CONCLUDING REMARKS

A concurrent algorithm for the solution of the Navier–Stokes equations expressed in curvilinear co-ordinates has been developed and demonstrated on a distributed memory parallel processor.

Initially, the concurrent algorithm was based on a straightforward domain decomposition. However, a reduction in the convergence rate as the number of processors was increased, caused by a poorer solution of the pressure correction equation, limited the speed-up in terms of the total computational time relative to the original serial algorithm to significantly less than linear. The addition of a global block correction procedure remedies this problem, making the convergence rate of the concurrent algorithm comparable to that of the serial algorithm, and significantly increasing the parallel efficiency. The revised algorithm looks very promising, with the potential for giving near-supercomputer performance on a distributed memory machine with faster processors.

Further work is needed in the following areas.

1. More test problems should be studied to verify the performance of the concurrent algorithm under a wider range of conditions. Problems involving turbulence and chemical reactions should be included in the study.
2. The revised concurrent algorithm should be run on a distributed memory machine with vector processors, such as Intel's iPSC/VX, to measure the speed-up relative to a single processor and the total computational time. A successful demonstration will prove the practical utility of such machines for running CFD codes.

ACKNOWLEDGEMENTS

The author would like to thank his colleague S. S. Tong of the GE Research and Development Center for his assistance in getting the hypercube simulator operational on his SUN workstation, Professors G. Pecelli and S. Smith of the Computer Science department of the University of Lowell for allowing me access to the university's Intel hypercube, and P. Kautz and G. MacDonald of Intel for their assistance. Thanks are also due to my colleagues R. Mani and W. Shyy for helpful discussions, and to C. Cowsert for typing the equations.

REFERENCES

1. W. Shyy and M. E. Braaten, 'Combustor flow computations in general coordinates with a multigrid method', *AIAA-87-1156-CP, AIAA 8th CFD Conf.*, Honolulu, Hawaii, 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. Chessire, C. Moler, P. Ross and D. Scott, 'Seismic modeling on the Intel iPSC', *Application Brief AB-05-687*, Intel Scientific Computers, 1987.
4. W. D. Gropp and E. B. Smith, 'Computational fluid dynamics on parallel processors', *YALEU/DCS/RR-570*, Yale University, 1987.
5. J. Townsend, T. A. Zang and D. L. Dwoyer, 'Fluid dynamics parallel computer development at NASA Langley Research Center', in A. K. Noor (ed.), *Parallel Computations and Their Impact on Mechanics*, ASME, New York, 333–343 (1987).
6. D. E. Keyes and M. D. Smooke, 'A parallelized elliptic solver for reacting flows', in A. K. Noor (ed.), *Parallel Computations and Their Impact on Mechanics*, ASME, New York, 375–402 1987).
7. 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).
8. 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).
9. G. D. Raithby and G. E. Schneider, 'Numerical solution of problems in incompressible fluid flow: treatment of the velocity–pressure coupling', *Numer. Heat Transfer*, **2**, 417–449 (1979).
10. S. V. Patankar, *Numerical Heat Transfer and Fluid Flow*, Hemisphere, New York, 1980.
11. O. A. McBryan and E. F. Van de Velde, 'Hypercube algorithms and implementations', *SIAM J. Sci. Stat. Comput.*, **8**, 5227–5287 (1987).

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. C. L. Seitz, 'The cosmic cube', *Comm. ACM*, **28**, 22–33 (1985).
14. 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.
15. J. M. Ortega and R. G. Voigt, 'Solution of partial differential equations on vector and parallel computers', *SIAM Rev.*, **27**, 149–240 (1985).
16. W. Shyy, 'Numerical outflow boundary condition for Navier–Stokes flow calculations by a line iterative method, *AIAA J.*, **23**, 1847–1848 (1985).
17. W. Shyy and M. E. Braaten, 'Three-dimensional analysis of the flow in a curved hydraulic turbine draft tube. *Int. j. numer. methods fluids*, **6**, 861–882 (1986).
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, Minneapolis, MN, 1985.
19. T. F. Chan and R. S. Tuminaro, 'Implementation of multigrid algorithms on hypercubes', *RIACS Technical Report 86.30*, NASA Ames Research Center, Moffett Field, CA, 1986.
20. M. E. Braaten and W. Shyy, 'Study of pressure correction methods with multigrid for viscous flow calculations in nonorthogonal curvilinear coordinates', *Numer. Heat Transfer*, **11**, 417–442 (1987).
21. S. V. Patankar, 'A calculation procedure for two-dimensional elliptic situations', *Numer. Heat Transfer*, **4**, 409–425 (1981).
22. A. Settari and K. Aziz, 'A generalization of the additive correction methods for the iterative solution of matrix equations', *SIAM J. Numer. Anal.*, **10**, 506–521 (1973).