JOURNAL OF COMPUTATIONAL PHYSICS **125,** 225–243 (1996)
ARTICLE NO. 0090

A Parallel Incompressible Flow Solver Package with a Parallel Multigrid Elliptic Kernel

JOHN Z. LOU[1] AND ROBERT FERRARO[2]

Jet Propulsion Laboratory, California Institute of Technology, Pasadena, California 91109

Received June 30, 1995; revised January 11, 1996

The development and applications of a parallel, time-dependent incompressible Navier–Stokes flow solver and a parallel multigrid elliptic kernel are described.The flow solver is based on a second-order projection method applied to a staggered finite-difference grid. The multigrid algorithms inplemented in the parallel elliptic kernel, which is needed by the flow solver, are V-cycle and full V-cycle schemes. A grid-partition strategy is used in the parallel implementations of both the flow solver and the multigrid elliptic kernel on all fine and coarse grids. Numerical experiments and parallel performance tests show the parallel solver package is numerically stable, physically robust, and computationally efficient. Both the multigrid elliptic kernel and the flow solver scale very well to a large number of processors on Intel Paragon and Cray T3D for computations with moderate granularity. The solver package has been carefully designed and coded so that it can be easily adapted to solving a variety of interesting two- and three-dimensional flow problems. The solver package is portable to parallel systems that support MPI, PVM, and NX for interprocessor communications.

1. INTRODUCTION

The final objective of this work is to develop a parallel and scalable incompressible flow solver package which can be used for solving a variety of practical and challenging incompressible flow problems arising from physics and engineering applications. A few examples are convective turbulence modeling in astrophysics, thermally driven flows in cooling systems, and combustion process modeling. A Navier–Stokes algorithm for successfully solving these complicated, nonsmooth flow problems must be numerically stable, physically robust, and computationally efficient. Results from numerical experiments in [2–4] indicate that a second-order projection method proposed in [2] is a promising candidate for simulations of complex incompressible flows.

Our task here is to develop an efficient, flexible, and portable parallel flow solver package for multiple applications. In terms of efficiency, we want the solver to have high numerical efficiency, as well as parallel computing efficiency, which is the reason to use a parallel multigrid elliptic kernel as a convergence accelerator for the parallel flow solver. Flexibility and portability have been emphasized throughout our design and implementation of the solver package. We want to develop the solver package so that it can be used either as a stand-alone flow solver for several types of flow problems or as a flow solver template which can be modified or expanded by the user for a specific application. A reusable or template partial differential equation (PDE) solver, in our view, is a PDE solver package that can be adapted or expanded to solving a variety of problems using different (component) numerical schemes as needed without a major rewriting of the solver code.

A basic assumption in our solver package is the use of finite-difference methods on a rectangular grid or on a composite grid with each of its components a rectangular grid. The use of rectangular grids has several advantages: (1) finite-difference is easy to implement, and for many applications, stable and robust finite-difference methods already exist and the use of a finite-element type scheme may not be desirable due to physical and numerical considerations; (2) multigrid is easy to apply; (3) parallel implementations are easier than on unstructured grids. Many practical problems are, however, defined in irregular domains. One way to extend our solver package to problems in an irregular domain is to construct a mapping between the irregular domain and a rectangular region. For a variety of nonrectangular domains, such mappings can indeed be constructed (for more detail; see [9]).

A projection method for solving incompressible Navier–Stokes equations was first described in a paper by Chorin [6], which is a finite-difference method for solving the incompressible Navier–Stokes equations in primitive variables. Bell *et al.* [2, 3] extended the method to second-order accuracy in both time and space and used a Godunov procedure combined with an upwind scheme in the discretization of the convection term for improved numerical sta-

[1] E-mail address: lou@acadia.jpl.nasa.gov.
[2] E-mail address: ferraro@zion.jpl.nasa.gov.

0021-9991/96 $18.00

bility. A projection method is a type of operator-splitting method which separates the solutions of velocity and pressure fields with an iterative procedure. In particular, at each time step, the momentum equations are solved first for an intermediate velocity field without the knowledge of a correct pressure field and therefore no incompressibility condition is enforced. The intermediate velocity field is then ''corrected'' by a projection step in which we solve a pressure equation and then use the computed pressure to produce a divergence-free velocity field. Our projection step, which is based on a pressure equation derived in [1] and makes use of the highly efficient elliptic multigrid kernel we developed, is mathematically equivalent to, but algorithmically different from, the projection step described in [2]. In actual flow simulations, this prediction–correction type procedure is usually repeated a few times (one or two iterations seem to be enough from our experiments) until reasonably good velocity and pressure fields have been reached for that time step. In each time step for computing an N-dimensional ($N = 2$ or 3) viscous flow problem, we need to solve $m \times N$ Helmholtz equations for the velocity field and m Poisson equations for the pressure field, where m is the number of iterations performed at each time step. A fast multigrid elliptic solver is thus very useful to improve the computational performance of the flow solver. The multigrid kernel we developed was designed to be a general-purpose elliptic solver. It can solve N-dimensional ($N \leq 3$) elliptic problems on vertex-centered, cell-centered and staggered grids, and it can deal with a variety of different boundary conditions as well.

Since the solver package is implemented on rectangular grids, a natural parallel implementation strategy is grid-partition: the global computational grid is partitioned and distributed to a logical network of processors; message exchanges are performed for grid points lying on ''partition boundary-layers'' (whose thickness is usually dictated by the numerical schemes used) to ensure a correct implementation of the sequential numerical algorithms on the global computational grid. In our implementation of the parallel multigrid V-cycle and full V-cycle schemes, we apply this grid-partition to all coarse grids as well. This means on some very coarse grid, only a subset of allocated processors will contain at least one grid point on that grid and therefore they are ''active'' on that grid, whereas processors which do not contain any grid point will be idle when processing that grid. The appearance of idle processors certainly introduces some complexity for a parallel implementation. For example, the logical processor mesh on which the original computational grid is partitioned cannot be used for communications on those coarse grids for which idle processors appear. Depending on the type of finite-difference grid and coarsening scheme, one may also need to consider, on those coarse grids, how to correctly apply boundary conditions in ''boundary processors'' which contain at least one grid point next to the boundary of the global grid, since boundary processors may change from one grid to another. Grid-partition on all coarse grids is certainly not the only possibility for parallel multigrid. Another approach, e.g., is to duplicate some of the global coarse grids in every processor allocated, so that processing on those coarse grids can be done without further interprocessor communication, but this coarse-grid-duplication approach involves quite some global communication for grid duplication and it needs some extra storage for global coarse grids. These requirements may severely affect the scalability of the solver when running on a large number of processors. One may also stop further grid coarsening at the coarsest grid for which no idle processor appears yet and solve the coarse grid problem by some direct or iterative methods. But the cost in solving the coarse grid problem with those methods is not competitive, compared to further grid coarsening. Although it seems no approach is perfect for implementing a parallel classical multigrid cycle [5, 8], we do believe the use of grid-partition at all grid levels is an appropriate approach for implementing a general-purpose parallel multigrid solver. The degradation of parallel efficiency due to the idle processors on some coarse grids has been discussed in many papers (e.g., [5, 8, 10]). The performance measurements from our parallel implementations indicate our multigrid solver scales quite well on a 512-node Intel Paragon and on a 256-node Cray T3D for both 2D and 3D problems with moderate sizes of local finest grid. In fact, the percentage of time spent on those coarse grids is insignificant compared to the total computation time. Similar observation was also made in [8]. As shown by a simple asymptotic analysis in [7], the parallel efficiency of multigrid schemes with the grid-partition approach is not qualitatively different from that of a single grid scheme.

The rest of the paper is organized as follows: Section 2 presents numerical algorithms for the multigrid kernel and the second-order projection method for the incompressible flow solver; in Section 3, discussions are made on issues related to the parallel implementations of the solver package; numerical results and parallel performances from the implemented parallel solvers are shown in Section 4; Section 5 gives some of our observations and conclusions.

2. THE NUMERICAL METHODS

A. *The Multigrid Algorithms*

The multigrid schemes implemented are the so-called V-cycle and full V-cycle schemes for solving elliptic PDEs, discussed in some detail in [4, 8]. The full V-cycle scheme is a generalization of the V-cycle scheme which first restricts the residual vector to the coarsest grid and then performs a few smaller V-cycle schemes on all coarse grids, followed by a complete V-cycle scheme on all grids. The

V-Cycle Scheme:

$V^h \leftarrow MV^h(v^h, f^h)$

1. Relax n_1 times on $A^h u^h = f^h$ with initial V^h
2. If Ω^h = coarsest grid, then go to 4
   else
   $$f^{2h} \leftarrow I_h^{2h}(f^h - A^h V^h)$$
   $$V^{2h} \leftarrow 0$$
   $$V^{2h} \leftarrow MV^{2h}(V^{2h}, f^{2h})$$
3. Correct $V^h \leftarrow V^h + I_{2h}^h V^{2h}$
4. Relax n_2 times on $A^h u^h = f^h$ with V^h

Full V-Cycle Scheme:

$V^h \leftarrow FMV^h(v^h, f^h)$

1. If Ω^h = coarsest grid, then go to 3
   else
   $$f^{2h} \leftarrow I_h^{2h}(f^h - A^h V^h)$$
   $$V^{2h} \leftarrow 0$$
   $$V^{2h} \leftarrow FMV^{2h}(V^{2h}, f^{2h})$$
2. Correct $V^h \leftarrow V^h + I_{2h}^h V^{2h}$
3. $V^h \leftarrow MV^h(v^h, f^h)$

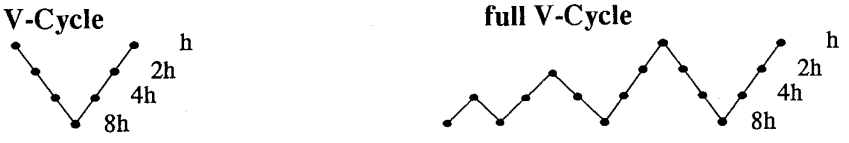

FIG. 1. Multigrid V-cycle and full V-cycle schemes.

full V-cycle scheme often offers a better numerical efficiency than the V-cycle scheme by using a much better initial guess of the solution in the final V-cycle. The parallel efficiency for the full V-cycle scheme, however, is poorer than the V-cycle scheme because it does more processing on coarse grids. Figure 1 shows pseudo-codes for the two schemes in a recursive fashion and their graphic representations.

A typical multigrid cycle consists of three main components: relax on a given grid, restrict the resulting residual to a coarse grid, and interpolate a correction back to a fine grid. Our multigrid solver can handle several different types of finite-difference grids commonly used in numerical computations. Figure 2 shows how coarse grids are derived from fine grids for vertex-centered, cell-centered, and staggered grids. Although the main steps in a V-cycle are the same for all these grids, restriction and interpolation operators can have different forms on different grids. On a vertex-centered grid we use a full-weighting stencil (9-point averaging on a 2D grid) to make the V-cycle scheme converge well when a pointwise red–black Gauss–Seidal (GS) smoother is used; whereas on a cell-centered grid, a nearest-neighbor stencil (4-point on a 2D grid) can be used with the pointwise red–black GS smoother to achieve a good convergence rate. We also point out that, on a vertex-centered grid, the use of the nearest-neighbor restriction stencil with the point-wise GS smoother does not even result in convergence on our test problems, but the use of a Jacobi smoother with the nearest-neighbor restriction stencil results in convergence but with a slower rate. The operator for transferring from a coarse grid to a fine grid is basically bilinear interpolation for all grids. Since fine and coarse grid points do not overlap on cell-centered and staggered grids, one needs to set the values for grid points at the boundary of coarse grids before a bilinear interpolation operator can be applied. More details on the construction of restriction and interpolation operators for different types of grids can be found in [13].

Out multigrid solver can solve Dirichlet and Neumann problems for the grids depicted in Fig. 2. A periodic boundary condition is also implemented for a special case used in the NS flow solver (to be discussed later). The Dirichlet and Neumann boundary conditions are applied only to the original (finest) grid; a homogeneous (zero) boundary condition is used on all coarse grids since residual equations are solved there. In the case of a Neumann boundary condition, where the unknowns are solved on all grid points, including those on the grid boundary, restriction stencils are not well-defined for boundary grid points. Take, for example, the vertex-centered grid in Fig. 3, where a 9-point full weighting stencil is used for restriction. This can be done naturally for the interior point c. For boundary

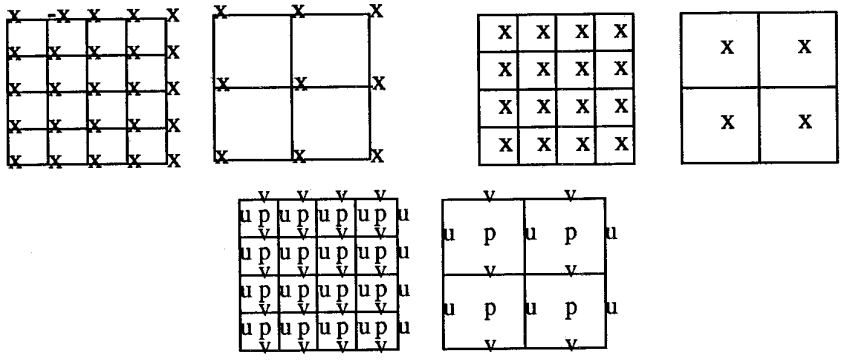

FIG. 2. Coarsening of three types of grids: vertex-centered (top-left); cell-centered (top-right); and staggered (bottom).

points a and b, however, only a subset of the neighboring points are within the grid and, therefore, weighting stencils on those points still need to be defined in some way. On the other hand, it is reasonable to have the following discrete integral condition satisfied between a pair of coarse and fine grids:

$$\sum_{I,J} U_{IJ} \times A_{IJ} = \sum_{i,j} u_{ij} \times a_{ij}, \tag{1}$$

where U_{IJ} and u_{ij} are solutions on coarse and fine grids and A_{IJ} and a_{ij} are areas of grid cells on coarse and fine grids, respectively. Restriction stencils for interior and boundary points that satisfy Eq. (1) are given in Fig. 3.

When solving a Poisson equation with a Neumann boundary condition, the solution is determined up to a constant. We use the following strategy to make sure the application of multigrid cycles converges to a fixed solution; after every relaxation on each grid, we perform a normalization step by adding a constant to the computed solution so that its value at a fixed point (we pick the point located at the center of the grid) is zero. Our numerical tests show this simple step results in a good convergence rate for Neumann problems.

B. *The Second-Order Projection Method*

We now give a brief description of the second-order projection method for solving the incompressible Navier–Stokes equations in a dimensionless form

$$\begin{aligned} \frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} &= -\nabla p + \mathrm{Re}^{-1} \Delta \mathbf{u} \\ \nabla \cdot \mathbf{u} &= 0, \end{aligned} \tag{2}$$

where $\mathbf{u} \in R^n$ ($n = 2$ or 3) is the velocity field, $p \in R$ is the pressure field, and Re is the Reynolds number. A typical problem is to find $\mathbf{u}$ and p satisfying (2) in a domain Ω for a given initial velocity field $\mathbf{u}_0$ in Ω and a velocity boundary condition $\mathbf{u}_b$ on $\partial\Omega$. The projection method for solving Eqs. (2) is based on the Hodge decomposition which states that any vector field $\mathbf{u}$ can be uniquely decomposed into a sum of $\mathbf{u}_1 + \mathbf{u}_2$ with $\nabla \cdot \mathbf{u}_1 = 0$ and $\mathbf{u}_2 = \nabla\phi$ for some scalar function ϕ. The projection method proceeds as a type of fractional step method by first writing the momentum equation in (2) in an equivalent form,

$$\frac{\partial \mathbf{u}}{\partial t} = \mathbf{P}(\mathrm{Re}^{-1} \Delta \mathbf{u} - (\mathbf{u} \cdot \nabla)\mathbf{u}), \tag{3}$$

where $\mathbf{P}$ is an orthogonal projection operator which projects a smooth function onto a divergence-free subspace. Equation (3) can be viewed as the result of applying $\mathbf{P}$ to the momentum equation in (2) which can be rewritten as

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla p = \mathrm{Re}^{-1} \Delta \mathbf{u} - (\mathbf{u} \cdot \nabla)\mathbf{u}. \tag{4}$$

The projection operation removes the pressure gradient in (4) because ∇p is orthogonal to the projection. Thus if we let the right-hand side of (4) be a vector field $\mathbf{V}$, then $\nabla p = (\mathbf{I} - \mathbf{P})\mathbf{V}$. The second-order projection method in [2] is a modification to the original projection method proposed in [6] to achieve a second-order temporal accuracy and an improved numerical stability for the nonlinear con-

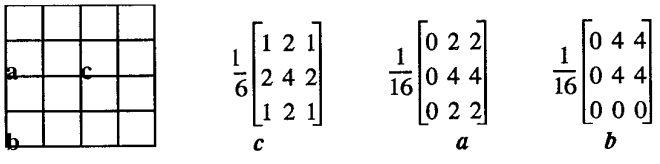

FIG. 3. Restriction stencils for interior point c and boundary points a and b.

vection. It uses the following temporal discretization on the momentum equation at each half time step $n + \frac{1}{2}$

$$\frac{\overline{\mathbf{u}}^{n+1,k} - \mathbf{u}^n}{\Delta t} + [(\mathbf{u} \cdot \nabla)\mathbf{u}]^{n+1/2} = -\nabla p^{n+1/2,k} + \frac{1}{\mathrm{Re}} \Delta \left(\frac{\overline{\mathbf{u}}^{n+1,k} + \mathbf{u}^n}{2} \right), \tag{5}$$

where we assume the velocity $\mathbf{u}^n$ is known, and $\overline{\mathbf{u}}^{n+1,k}$ is an intermediate velocity field that satisfies the same boundary condition as the physical velocity at time step $n + 1$. The temporal discretization in (5) is second-order accurate, provided that the nonlinear convection term in (5) can be evaluated to the second-order accuracy at the half time step $n + \frac{1}{2}$. The superscripts k in (5) indicate that an iterative process is used for computing the velocity at next time step $\mathbf{u}^{n+1}$ and the pressure at the next half time step $p^{n+1/2}$: given a divergence-free field $\mathbf{u}^n$ and the corresponding pressure field $p^{n-1/2}$, we first set $p^{n+1/2,0} = p^{n-1/2}$. For $k \geq 1$, we solve (5) for $\overline{\mathbf{u}}^{n+1,k}$. Since the correct pressure $p^{n+1/2}$ is not known, the computed $\overline{\mathbf{u}}^{n+1,k}$ is usually not divergence-free; but $\overline{\mathbf{u}}^{n+1,k}$ can be used as a guess for $\mathbf{u}^{n+1}$ and it is used to compute $p^{n+1/2,k}$, a new guess for $p^{n+1/2}$, by solving a pressure equation. Once we have a new guess for $p^{n+1/2}$, it is used in (5) to compute $\overline{\mathbf{u}}^{n,k+1}$. This iterative procedure is performed at each time step until $\nabla p^{n+1/2,k} \to \nabla p^{n+1/2}$ and $\overline{\mathbf{u}}^{n+1,k} \to \mathbf{u}^{n+1}$. This iterative process converges because it can be shown that the mapping of errors from state k to state $k + 1$ is contractive [2]. In practice, we found 1 to 2 iterations would be enough to get a satisfactory convergence.

The convection term $(\mathbf{u} \cdot \nabla)\mathbf{u}$ is evaluated at the half time step $n + \frac{1}{2}$, using only the velocity $\mathbf{u}^n$ and the pressure $p^{n-1/2}$. On the staggered grid shown in Fig. 2, the pressure p is defined at cell centers; horizontal velocity u and vertical velocity v are defined at cell edges. Let us denote cell (i, j) as the cell whose center is located at $(i - \frac{1}{2})\Delta x$, $(j - \frac{1}{2})\Delta y$ for $i = 1 \cdots I$ and $j = 1 \cdots J$. $(\mathbf{u} \cdot \nabla)\mathbf{u}$ is then evaluated at i, $j - \frac{1}{2}$ for the u component and $i - \frac{1}{2}$, j for the v component. The discretization for the u component, for example, has the form

$$[(\mathbf{u} \cdot \nabla)\mathbf{u}]_u = \frac{u_{i-1/2,j-1/2} + u_{i+1/2,j-1/2}}{2} \times \left(\frac{u_{i-1/2,j-1/2} - u_{i+1/2,j-1/2}}{\Delta x} \right) + \frac{v_{i,j-1} + v_{i,j}}{2} \left(\frac{u_{i,j} - u_{i,j-1}}{\Delta y} \right),$$

where $u_{i\pm1/2,j\pm1/2}$ are velocities at cell centers, $u_{i,j}$ and $v_{i,j}$ are velocities at cell corners and all velocities are assumed to be at time $n + \frac{1}{2}$. Since $\mathbf{u}^n$ is the only velocity available at the start of computations for time step $n + 1$ and velocity values are not defined at cell centers and cell corners, we use Taylor expansions of second-order accuracy in both time and space, as was done in [2, 3], to find velocities at appropriate locations and at the half time step $n + \frac{1}{2}$ for computing the discrete convection term. To improve numerical stability, a Godunov-type procedure combined with an upwind scheme is used in determining velocity values at cell centers and cell corners. To compute the u velocity component at the cell center of cell (i, j), for example, we first compute

$$\begin{aligned} u^R &= u^n_{i-1,j-1/2} + \frac{\Delta x}{2} u^n_{x,i-1,j-1/2} + \frac{\Delta t}{2} u^n_{t,i-1,j-1/2} \\ u^L &= u^n_{i,j-1/2} - \frac{\Delta x}{2} u^n_{x,i,j-1/2} + \frac{\Delta t}{2} u^n_{t,i,j-1/2}, \end{aligned} \tag{6}$$

where the expansions for u^R and u^L are evaluated on the right side of edge $(i - 1, j - \frac{1}{2})$ and on the left side of edge $(i, j - \frac{1}{2})$, respectively. The choice of $u^{n+1/2}_{i-1/2,j-1/2}$ is then made by the following upwind scheme:

$$u^{n+1/2}_{i-1/2,j-1/2} = \begin{cases} u^R, & \text{if } u^L > 0, u^L + u^R > 0, \\ 0, & \text{if } u^L < 0, u^R > 0, \\ u^L, & \text{otherwise.} \end{cases} \tag{7}$$

The spatial derivatives in (6) are computed by first using a centered differencing and then applying a slope-limiting step to avoid forming new maxima and minima in the velocity field. Temporal derivatives in (6) are computed by using the momentum equation (4). Derivatives at cell corners are computed in a similar way. More details for the construction of these derivatives are given in [2, 3].

After evaluation of the convection term, the intermediate velocity $\overline{\mathbf{u}}^{n+1,k}$ can be found by solving the following Helmholtz equation for each velocity component:

$$-\Delta \overline{\mathbf{u}}^{n+1,k} + \frac{2\mathrm{Re}}{\Delta t} \overline{\mathbf{u}}^{n+1,k} = 2\mathrm{Re} \left(-[(\mathbf{u} \cdot \nabla)\mathbf{u}]^{n+1/2} + \frac{1}{\Delta t} \mathbf{u}^n + \Delta \mathbf{u}^n - \nabla p^{n-1/2} \right). \tag{8}$$

We notice that the matrix derived from Eq. (8) becomes more diagonally dominant as the Reynolds number increases for a fixed grid size and a fixed time step, which is fortunate for computing flows with large Reynolds numbers. For inviscid flow problems where $\mathrm{Re} = \infty$, $\overline{\mathbf{u}}^{n+1,k}$ can be computed explicitly from Eq. (5). Once $\overline{\mathbf{u}}^{n+1,k}$ is computed, a projection step is performed to find the pressure

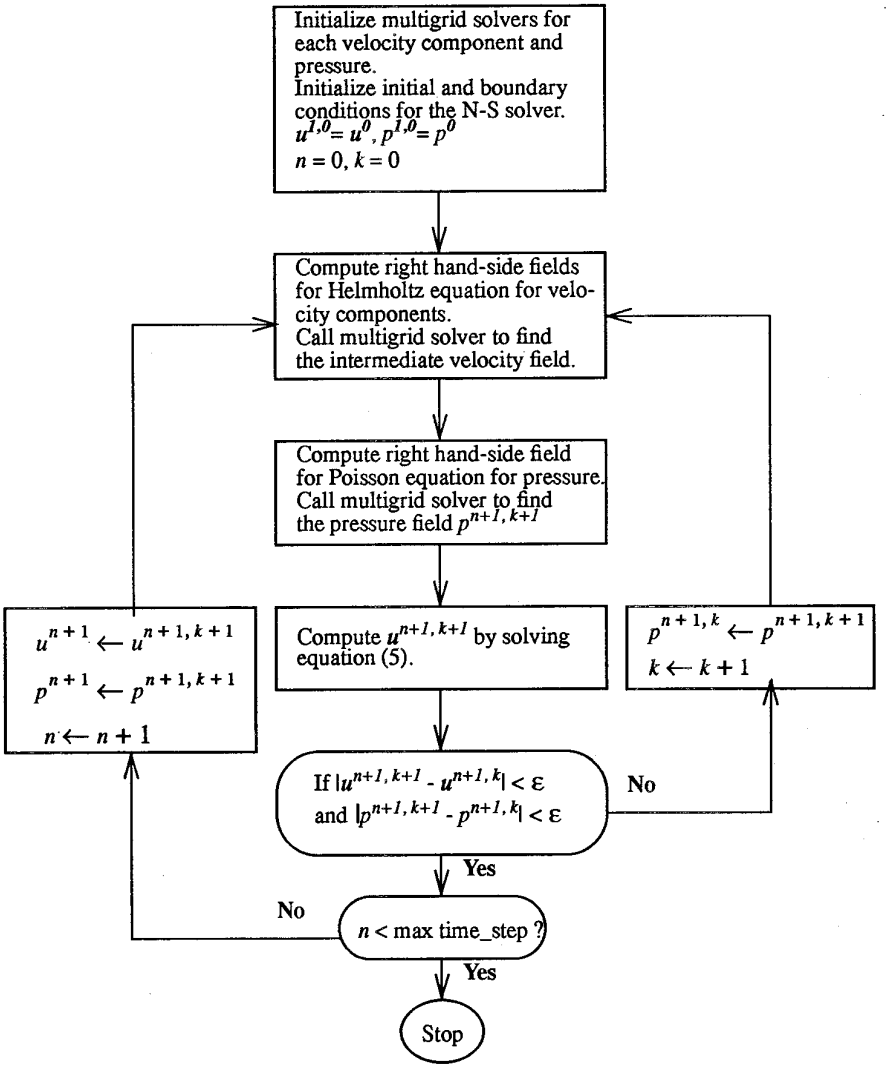

FIG. 4. Flow diagram for the Navier–Stokes solver.

field $p^{n+1/2,k+1}$ by solving a Poisson equation,

$$\Delta p = R(\mathbf{u}^n, \mathbf{u}^{n+1}), \tag{9}$$

where $\bar{\mathbf{u}}^{n+1,k}$ is used in place of $\mathbf{u}^{n+1}$ in the right-hand side of Eq. (5). Mathematically, Eq. (9) is the result of applying a divergence operator to the momentum equations in (2). Since no boundary condition is defined for the pressure field, some special treatments are needed at the boundary grid points in solving Eq. (9). The details of deriving the pressure equation (9) on a staggered grid with appropriate treatments at boundary grid points for Dirichlet velocity boundary condition is given in [1]. The treatment of the pressure boundary condition for a periodic velocity boundary condition using a multigrid scheme is discussed in Section 4. In computing a viscous flow, the multigrid elliptic solver is used to solve both Eqs. (8) and (9). After the pressure field $p^{n+1,k+1}$ is found, $\mathbf{u}^{n+1,k+1}$ can be computed by using Eq. (5) with $p^{n+1,k+1}$, in place of $p^{n+1,k}$, and this completes one iteration in the computations for the time step $n+1$; $\mathbf{u}^{n+1}$ and p^{n+1} are then obtained at the end of the last iteration. The flow of control for our incompressible Navier–Stokes solver is shown in Fig. 4.

3. PARALLEL IMPLEMENTATIONS

A. *Grid Partition and Logical Processor Mesh*

The approach we adopted in parallel implementations of the multigrid elliptic solver and the incompressible flow solver is grid-partition. Our goal is to develop parallel solvers that can partition any N-dimensional ($N \leq 3$) rectangular grids and run on any M-dimensional ($M \leq N$)

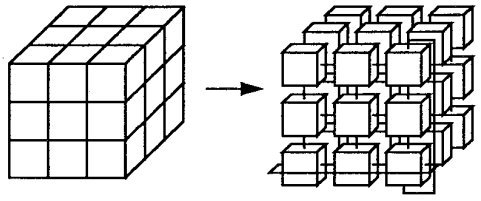

FIG. 5. A 3D Grid partition and mapping to a processor mesh. Only two wrap-around connections were shown in the logical processor mesh.

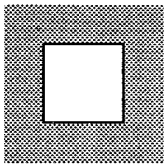

FIG. 7. A local subgrid (white area) with surrounding ghost points (shaded area).

logical processor meshes. For example, Fig. 5 shows the partition of a three-dimensional grid and the assignment of the partitioned subgrids to a three-dimensional torus processor mesh. As shown in Fig. 6, logical processor meshes in our code are always constructed as toroidal meshes. Toroidal meshes are useful in the construction of nested coarser processor meshes for the multigrid solver and for dealing with problems with a periodic boundary condition.

In the multigrid solver, coarse grids and coarse logical processor meshes are constructed automatically and recursively, based on information on a given fine grid. All grid storages are allocated dynamically during the grid coarsening process. In particular, for each multigrid level, a local coarse grid is derived from the local fine grid and storages are allocated for the coarse grid. Processors which will get at least one grid point on that coarse grid will be in an active state on that grid; otherwise they will be in an idle state on that grid. A flag is then set in each processor for that level, depending on the value of the state. A coarse processor mesh for that coarse grid can then be established by communicating the states among the processors in the fine processor mesh. This process is repeated recursively until all the coarse grids and the coarse processor meshes have been constructed. As an illustration, Fig. 6 shows a processor mesh and its derived coarse mesh for a problem with a Neumann boundary condition. In our multigrid solver, we put this construction process in an initialization routine which must be called before the first time the multigrid solver itself is called. The cost of running the initialization routine is relatively small when one needs to call the multigrid solver a large number of times, as is the case for the Navier–Stokes flow solver. After executing this initialization routine, every processor knows its "role" at each level of the multigrid cycle, and it also knows its neighboring processors on that grid level.

B. *Interprocessor Communications*

To implement the multigrid scheme and the projection method on a partitioned grid, we need to exchange data which are close to the partition boundaries of each subgrid local to a certain processor. Each processor contains a rectangular subgrid surrounded by some "ghost grid points" which are duplicates of grid points contained in other processors, as shown in Fig. 7. The number of ghost points on each side of the subgrid depends on the numerical algorithms. For the multigrid elliptic solver using a standard Laplacian stencil, one ghost grid point on each side is needed for the local subgrid at each level, whereas for the second-order projection method, three ghost grid points on each side are needed in computing the nonlinear convection term using Taylor series and upwind schemes. Therefore in the Navier–Stokes flow solver, we allocate storages for three ghost grid points for the fine local grid and one ghost grid point for each coarse grid. For certain operations in the multigrid scheme (e.g., restriction and interpolation) and for computing the convection term in the projection method, ghost grid points in the diagonal neighbor are also needed, as shown in Fig. 8. Since processors P_i and P_j in Fig. 8 are not nearest neighbors, direct data exchange between them will introduce a more complicated message-passing pattern. Fortunately, direct data exchange between P_i and P_j is not necessary to get the diagonal ghost grid

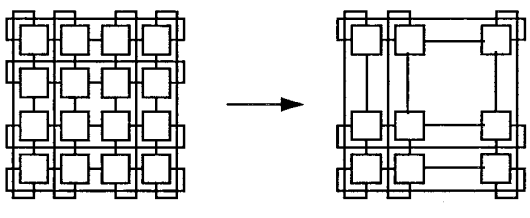

FIG. 6. If the left processor mesh contains a 5×5 grid for a Neumann problem on a vertex-centered grid, then the derived coarse processor mesh is the one on the right.

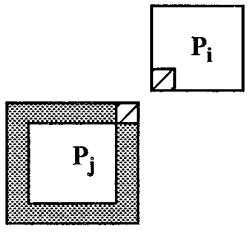

FIG. 8. The data in the lower left corner of the subgrid in processor P_i are needed by processor P_j, and stored in P_j's ghost grid points at upper right corner.

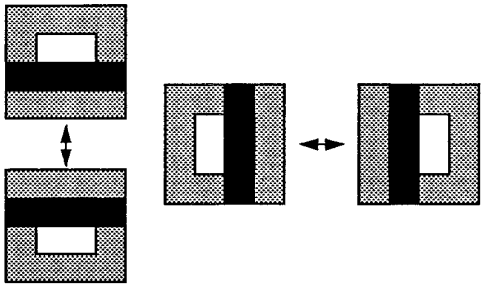

FIG. 9. Data exchanges between neighboring processors for 2D problems. The data in black blocks in each processor are sent out, which is stored in the blocks for ghost grid points in the neighboring processors.

points. It can be verified that all the data exchanges we need are of nearest neighbor types, as indicated in Fig. 9 for 2D problems. As can be seen in Fig. 9, when data lying on the partition boundaries are exchanged, the sending blocks always include ghost grid points. After the data exchanges in Fig. 9 are performed, all the ghost grid points shown in Fig. 7 will be obtained by appropriate neighboring processors. Each processor, therefore, only needs to know its nearest neighboring processors on each logical processor mesh. In solving problems with periodic boundary conditions, data exchanges are also required among processors lying on the boundary of a processor mesh, and the same message-passing operations as shown in Fig. 9 can be used.

The parallel efficiency of a parallel code is largely determined by the ratio of local computations over interprocessor communications. In our solvers, the best parallel efficiency is achieved on the finest grid, where the communication cost could be easily dominated by a large amount of computations, and the parallel efficiency degrades as the grid gets coarser. One way to hide communication overhead and thus improve parallel efficiency on all grids is to overlap communications with computations. In several places within our solvers, we have the following sequence of operations for each processor:

(1) Exchange data lying on partition boundaries;

(2) Perform processing on all local grid points.

To overlap communications with computations, we can perform the following sequence of operations for the same result:

(1) Initiate the data exchange for partition boundaries;

(2) Perform processing on interior grid points that do not need ghost grid points;

(3) Wait until data exchange in (1) is complete;

(4) Perform processing on the remaining grid points.

On intel Paragon, we implemented the second set of operations above in the multigrid solver and the flow solver using asynchronous message-passing calls. For one full V-cycle in the elliptic solver, for example, the performance improvement on a 256×256 grid partitioned among 256 processors is about 15%, and the improvement on a 256^3 grid partitioned among 512 processors is about 22%. Faster and asynchronous interprocessor communication can also be achieved on a Cray T3D by using its shared-memory communication model, in which direct memory copy is used at either sending or receiving processors for data exchanges between different processors. Some synchronization between sending and receiving processors, however, is needed before or after a direct memory copy is performed to ensure the correctness of a message-passing. On a T3D processor synchronization is provided only for a group of processors with a fixed stride in their processor indexes, this shared-memory communication model can be easily used for the exchange of partition boundary data in the flow solver and for the multigrid elliptic solver on some fine grids in which data exchanges only occur between nearest-neighbor processors on the original processor mesh.

C. *Software Structures*

Our solvers were implemented in C because we think it is the language that provides adequate support for implementing advanced numerical software without incurring unreasonably large overhead. Since our goal is to develop reusable and high-performance PDE solvers which can be used either as library routines or as extensible, template-type code for different applications, we emphasize in our code design both generality and flexibility. First, we want the solvers to be able to run on any M-dimensional rectangular processor meshes for any N-dimensional rectangular grids with ($M \leq N$) (for multigrid processing, N is usually a power of 2). This requirement introduces some complexities in coding the multigrid solver in terms of determining the right global indices for local grids at each grid level. Storages for all grid variables are allocated at run time. For the multigrid solver, storages for local coarse grids are allocated as they are derived recursively from local fine grids. The user is given the option either to supply the storage for variables defined on the original grid or to let the solver allocate those storages. An array of pointers to an N-dimensional grid (i.e., an N-dimensional data array) is allocated, and each of the pointers points to a grid in the grid hierarchy. N-dimensional data array is constructed recursively from one-dimensional data arrays. This strategy of dynamic memory allocation offers a greater flexibility in data structure manipulations and more efficient use of memory than a static memory allocation, and the user is also alleviated from the burden of calculating storage requirements for multigrid processing.

There are two major communication routines in the

TABLE I

Numerical Convergence: 2D Helmholtz Solver

Scheme	Grid size	Error	No. of processors	No. of cycles	CPU seconds
Full V-cycle	2048^2	3.0×10^{-7}	64	3	14.7
V-cycle	2048^2	3.5×10^{-7}	64	6	14.5
Single grid	2048^2	9.7×10^{-1}	64	400 R-B sweeps	124.7

solvers: the communication routine for the flow solver exchanges partition boundary data only on the original grid; the communication routine for the multigrid solver can exchange partition boundary data for all fine and coarse grids, using a hierarchy of processor meshes. To make the code portable across different message-passing systems, we defined our own generic message-passing library as an interface with our solvers. To use a new message-passing system, we only need to extend the generic message-passing library to that system without changing any code in our solvers. Currently, our generic message-passing library can accommodate NX, MPI, and PVM. A separate data exchange routine has also been implemented for the flow solver, which uses the shared-memory communication library on Cray T3D.

Simple user interfaces to the parallel solvers have also been constructed. The elliptic multigrid solver can be used as a stand-alone library routine with both C and Fortran interfaces. After initialization of the problem to be solved and some algorithm parameters, a preprocessing routine must be called before the first time the multigrid solver routine is called. The preprocessing routine constructs the set of nested grids and the corresponding set of logical processor meshes. The flow solver can be used as a general-purpose incompressible fluid flow solver on a rectangular, staggered finite-difference grid for problems with Dirichlet or periodic velocity boundary conditions. To use the multigrid solver as a kernel for evaluating velocity and pressure fields, the preprocessing routine must be called for each velocity component and the pressure, since they are defined on different grid points on a staggered grid. Therefore separate structures will be constructed in the preprocessing routine for each velocity component and the pressure, which will be used in subsequent calls to the multigrid solver.

4. NUMERICAL EXPERIMENTS AND PARALLEL PERFORMANCES

We now report numerical experiments made to examine the numerical properties of the parallel solvers on a few test problems, and parallel performances in terms of speedup and parallel scaling of the solvers on Intel Paragon and Cray T3D systems for problems with different sizes and granularity.

A. *The Elliptic Multigrid Solver*

The multigrid elliptic solver was first tested on Helmholtz and Poisson equations with known exact solutions. Table I and Table II show the convergence rates of the multigrid solver from solving 2D and 3D Helmholtz equations

$$-\Delta u + u = f$$

with Dirichlet boundary conditions. The runs were performed on Intel Paragon. Errors displayed are the normalized maximum norm of the difference between a computed solution and the exact solution. The tables show the number cycles needed in each case to reach the order of discretization error (or truncation error). At each grid level, two red–black relaxations were performed. Although the full V-cycle scheme is usually considered a lot more efficient than the V-cycle scheme in sequential processing [4], it seems not necessarily the case in terms of total execution

TABLE II

Numerical Convergence: 3D Helmholtz Solver

Scheme	Grid	Error	No. of processors	No. of cycles	CPU seconds
Full V-cycle	256^3	1.2×10^{-5}	64	4	71.3
V-cycle	256^3	2.6×10^{-5}	64	86	86.7
Single grid	256^3	8.3×10^{-1}	64	400 R-B sweeps	652.1

time cost in parallel processing. As shown in the tables, for the 2D problem, even though the V-cycle scheme takes three more cycles to reach the same order of error than the full V-cycle scheme, the CPU time for both schemes are about the same for that test problem; but for the 3D test case in Table II, the full V-cycle scheme is still a little more efficient. With a domain decomposition of both fine and coarse grids, parallel efficiency degrades as the processing moves to coarser grids. Since the full V-cycle scheme does more processing on coarse grids, its parallel efficiency is worse than the V-cycle scheme. For a large computational grid with many levels of coarse grids, the higher numerical efficiency of the full V-cycle scheme may not improve overall computational performance, due to its worse parallel efficiency, as is the case for the 2D case in Table I. For an appreciation of the effectiveness of the multigrid scheme which has a rate of convergence independent of grid sizes, the errors after a large number of red–black relaxations on the finest grid are also shown in the last rows of the tables. Although not shown in the paper, the convergence rates of the multigrid solver were also measured for cell-centered grid and staggered grid. We found for the same model problem the convergence speeds are slightly slower on those grids, which could be due to the use of different restriction operators.

The parallel performance of an application code is usually judged by two measurements: speedup and scaling. Speedup is measured by fixing the problem size (or grid size in our case) and increasing the number of processors used. Scaling is measured by fixing the local problem size in each processor and increasing the number of processors used. Although a nice speedup can be obtained for many applications with a small number of processors, the reductions in CPU time often become very small when a large number of processors is used. This phenomenon is largely inevitable, as stated in Amdahl's law, because as the number of processors used increases for a certain fixed problem size, the communication cost (e.g., the latency for message-passing) and the cost for global operations will eventually become dominant over local computation cost after a certain stage, which makes the influence of a further reduction in local computations very small on the overall cost of running the application. On the other hand, the scaling performance of an application seems to be a more realistic measure of its parallel performance, since a code with a good parallel scaling implies, given enough processors, it can solve a very large problem in about the same time as it requires for solving a small problem, which is, indeed, one of the main reasons to use a parallel machine. Figure 10 displays two speedup plots for a multigrid V-cycle and a full V-cycle for solving 2D and 3D Helmholtz equations with a Dirichlet boundary condition on a vertex-centered grid, measured on Intel Paragon and Cray T3D systems. For a comparison, an ideal speedup curve for one test case is also shown. The code was compiled with the −O2 switch on both machines. The grid size for the 2D problem is 512×512, and the grid size for the 3D problem is $64 \times 64 \times 64$. The maximum number of processors used for the 2D problem is 256 on both machines. For the 3D problem, all 256 processors were used on the T3D (in which case a rectangular processor mesh of dimensions $4 \times 8 \times 8$ was used) and 512 processors were used on the Paragon.

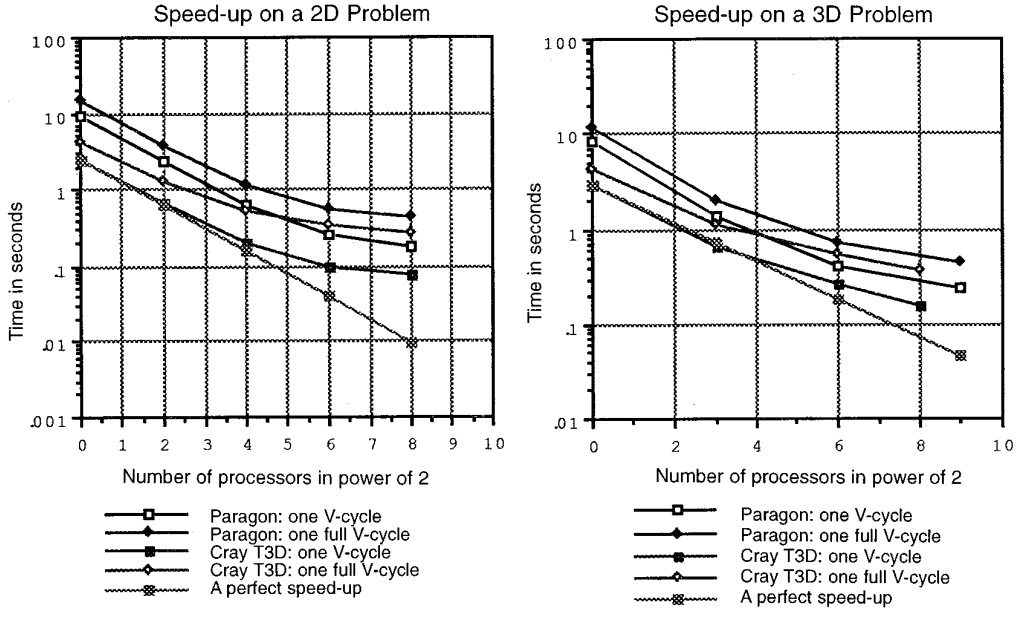

FIG. 10. Speedup performances of the elliptic multigrid solver.

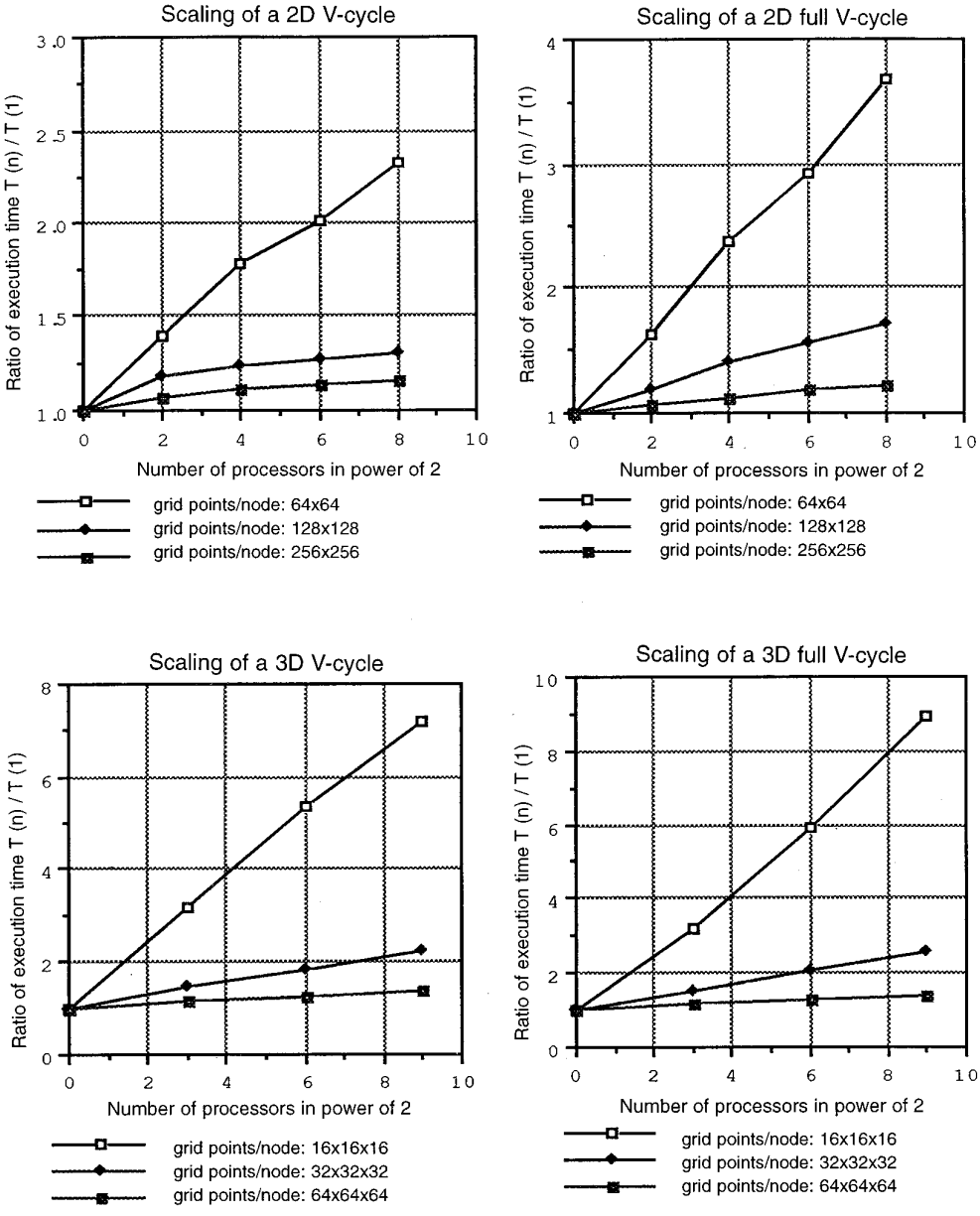

FIG. 11. Scaling of the multigrid solver on the Intel Paragon.

In terms of single processor performance, we found for our multigrid solver that Cray T3D is about 4 times faster than Intel Paragon. But since the implementation of PVM on T3D, which we used in our code for message-passing, is relatively slow for interprocessor communication, the performance difference for a parallel application on both machines tends to become smaller as granularity of the problem gets finer. We can see for both 2D and 3D problems that the V-cycle scheme has a slightly better speedup performance than the full V-cycle scheme, which is expected since the full V-cycle scheme does more processing on coarse grids. For the 2D problem, speedup started to degrade when more than 16 processors were used, and for the 3D problem, the degradation started when more than 8 processors were used. Despite the degradation in speedup, we can still see some reduction in CPU time when the largest number of processors was used in each case.

Figure 11 shows the scalings of the parallel multigrid solver on the Intel Paragon for problems with three different granularities, using up to 512 processors. Figure 12 shows the scalings of the same problems on the Cray T3D, using up to 256 processors. Shown in the plots are the ratios of CPU times using n processors versus using one processor. On each of the scaling curves, we fix the local

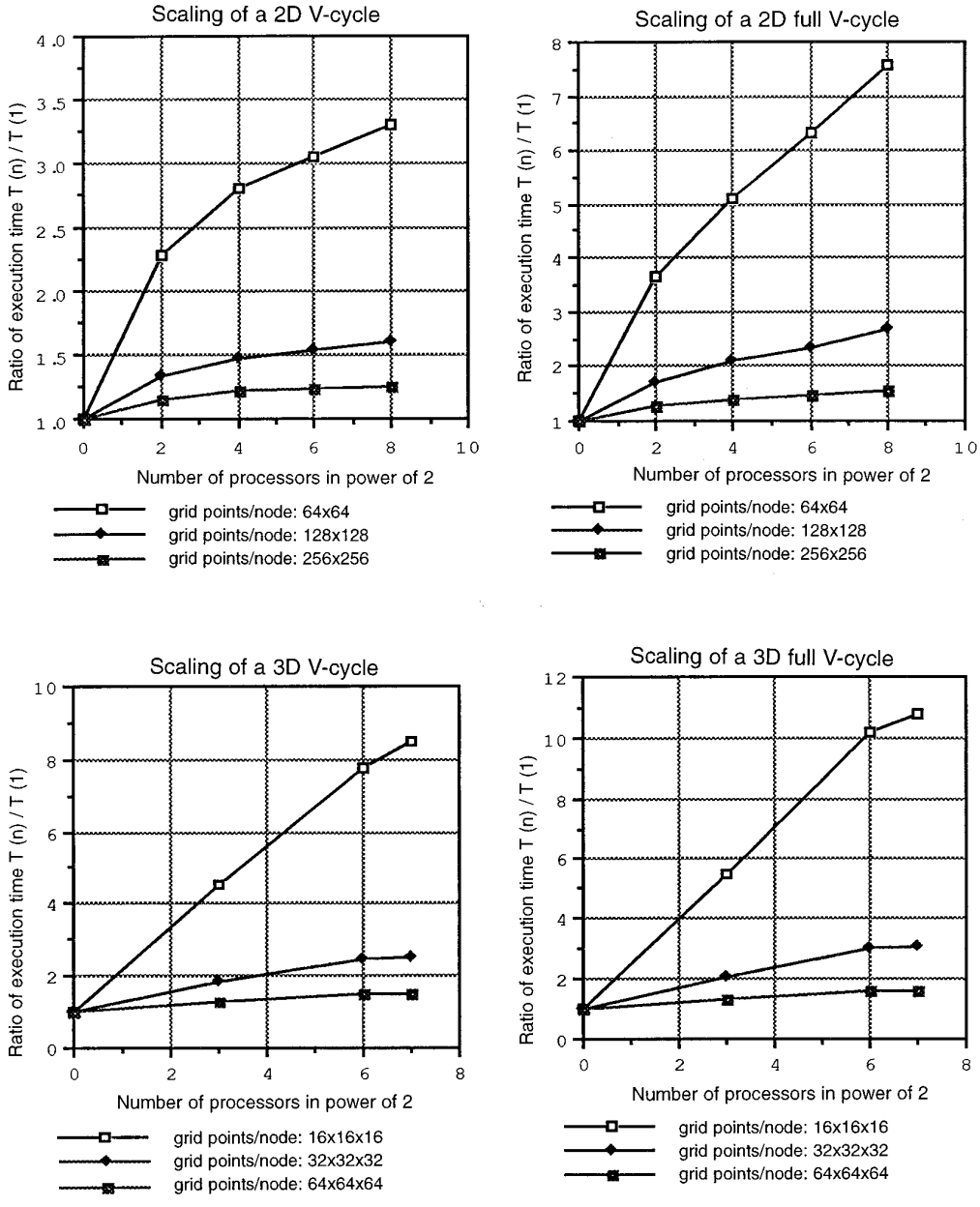

FIG. 12. Scaling of the multigrid solver on the Cray T3D.

grid size and increase the number of processors, so a flat curve means a perfect scaling. Since a larger global grid has more coarse grid levels for a complete V-cycle or a full V-cycle, the cost for processing on coarse grids also rises as the number of processors increases, and therefore it has a negative effect on the scaling performance. Like speedup performance, scaling performance is also largely determined by the ratio of local computation cost versus communication cost. This ratio can be dependent on both numerical/parallel algorithms and hardware/software performances on each specific machine. We can see from all the plots in Fig. 11 and Fig. 12 that scaling performance improves as the size of local grid increases. This improvement is expected for an iterative scheme on a single grid, since the computation cost scales as $O(n)$, where n is the number of grid points in the local grid, whereas the communication cost scales as $O(n^{1/2})$. For a multigrid scheme, it can still be shown that both computation cost and communication cost scale with the same orders as on a single grid [7]. In addition, message-passing latency does not increase as proportionally since the number of messages communicated is still roughly the same for a larger local grid (not

exactly the same because more coarse levels are involved) although the size of each message is larger. We can also see the V-cycle scheme scales somewhat better than the full V-cycle scheme, which is expected since the latter does more operations on coarse grids. The scaling plots also show 2D test cases scale better than 3D test cases, which we think is due to the fact that 3D grids have a higher surface to volume ratio than the 2D grids and, thus, the ratio of computations to communications is smaller for 3D cases. As for a comparison between the Intel Paragon and the Cray T3D, our results show that scalings on the Paragon are slightly better than on the T3D. This could be explained by the fact that single processor speed on the T3D is much faster than on the Paragon, whereas the speed of interprocessor communication on the T3D is not proportionally faster when PVM is used for communication.

B. *The Incompressible Navier–Stokes Solver*

The parallel Navier–Stokes solver was first tried out on a test problem with the following exact solution:

$$u = -\cos(x)\sin(y)e^{-2t/\mathrm{Re}} \quad v = \sin(x)\cos(y)e^{-2t/\mathrm{Re}}$$
$$\frac{\partial p}{\partial x} = \frac{1}{2}\sin(2x)e^{-4t/\mathrm{Re}} \qquad \frac{\partial p}{\partial y} = \frac{1}{2}\sin(2y)e^{-4t/\mathrm{Re}}. \tag{10}$$

The purpose of the test is to examine the convergence rate of the flow solver.The computation was performed in the unit square $0 \le x, y \le 1$ with initial and (time dependent) boundary conditions specified by the given solution. For numerical stability of the Godunov scheme used in discretizing the convection term, the time step, Δt, is restricted by the CFL condition

$$\frac{\Delta t}{\Delta x} u_{\max} < 0.5, \tag{11}$$

where Δx is the size of grid cells and $U_{\max}$ is the maximum value in the current velocity field. In using multigrid solver for solving velocity and pressure equations, we perform four full V-cycles in each call to the elliptic solver. On a 64×64 grid, we found four full V-cycles can reduce the residual error to the order of 10^{-10} for the test problem whose solution norm is one, which we think is good enough for our purpose. Table III shows the convergence rate of the computed velocity field to the exact velocity field using three Reynolds numbers. The velocity was computed to time = 3.12, which is about 400 time steps on a 64×64 grid. The error in Table III was measured by

$$e(u) = \|u_{\mathrm{exact}} - u_{\mathrm{comp}}\|_{\max} \quad e(v) = \|v_{\mathrm{exact}} - v_{\mathrm{comp}}\|_{\max}$$
$$E(\mathbf{u}) = \max(e(u), e(v)) \qquad \mathrm{Rate} = \log_2\left(\frac{E(\mathbf{u})^k}{E(\mathbf{u})^{k+1}}\right),$$

TABLE III

Second-Order Convergence Rate for Several Reynolds Numbers

Re	64×64	Rate	128^2	Rate	256^2	Rate	512^2
1000	4.628 E-5	1.961	1.195 E-5	1.982	3.010 E-6	2.035	7.568 E-7
3000	2.253 E-5	1.879	6.120 E-6	1.952	1.583 E-6	1.996	3.964 E-7
5000	1.478 E-5	1.824	4.181 E-6	1.845	1.087 E-6	1.870	2.970 E-7

where $E(\mathbf{u})^k$ is the error measured on a grid of size $2^k \times 2^k$. A second-order rate of convergence can be seen from the data in Table III. The rate improves slightly as the grid become finer, possibly due to better accuracy on finer grids. The rate also drops slightly as the Reynolds number increases, which could be a result of more numerical noise introduced at higher Reynolds number calculations. Table III shows smaller errors at higher Reynolds number, which in fact does not contract the intuition about higher Reynolds numbers being harder to compute because their exact solution has less variation in time at higher Reynolds number.

Our next numerical experiment on the flow solver is to simulate an evolving 2D driven-cavity flow. The computational domain is still in a unit box $0 \le x, y \le 1$. The no-slip velocity boundary condition is applied to all boundaries except at the top boundary, where the velocity value is given. We first test the solver on the problem in which the velocity initial condition is specified by $\mathbf{u} = 0$ inside the domain, and the velocity at the top boundary is always one. Figure 13 displays the velocity vector fields which show three stages for time = 0.16, 3.91, and 15.63 in the evolution of the flow, computed on a 256×256 grid with the Reynolds number = 5000. The CFL number (i.e., the right-hand side of (10)) used in the calculation is 0.4, and a total of 10,000 time steps was computed to reach the last state at time = 15.63. Figure 14 shows the vorticity structures at time = 3.91 and 15.63. We found the vorticity structure at time = 15.63 is similar to those obtained by solving the steady incompressible Navier–Stokes equations (e.g., [11]). In running the parallel solver on a Cray T3D, the global staggered grid was partitioned and distributed to an 8×8 logically rectangular processor mesh. In computing the velocity vector field, velocity components defined on the cell edges were averaged to the center of the cells. For better visibility, the vector fields shown in Fig. 13 are actually 32×32 data arrays which were obtained by averaging the 256×256 velocity vectors from the simulation. Vorticity fields were computed at cell corners by central differencing. The velocity vector plots in Fig. 13

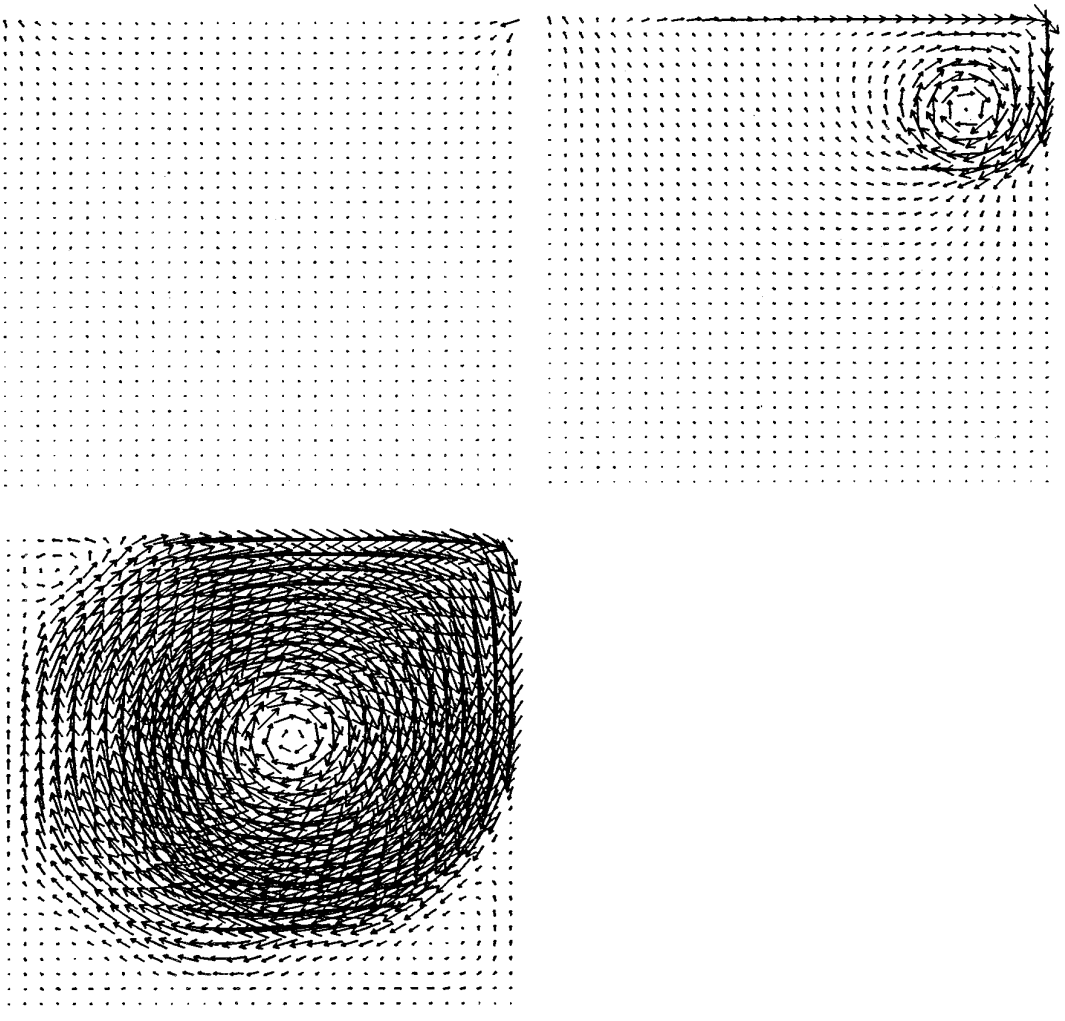

FIG. 13. Velocity vector plots from computing an unsteady driven-cavity flow with Re = 5000. The simulation was done on a 256 × 256 grid, using 64 processors on the T3D. Shown are plots at time = 0.16 (top left), 4.69 (top right), and 15.63 (bottom). A steady state of the flow can be very slowly reached.

show clearly how the cavity flow develops from its initial state to the final steady state which is characterized by a primary vortex in the center of the unit box and two secondary vortices at the two bottom corners and a small vortex at the upper left corner (e.g., [11]). We also noticed, when reaching the final stage in Fig. 13, that the change of numerical divergence of the computed velocity field before and after projection is very small. This is because, when the steady state is reached, the intermediate velocity field would be computed using the correct pressure field to result in a correct velocity field. Even though the initial condition is not continuous along the top boundary and the boundary condition is not continuous at the two upper corners of the unit box, the numerical computation of the flow solver turns out to be quite stable.

The flow solver was next tried on a driven-cavity flow problem with some smooth initial and boundary conditions. The top boundary now moves with a slip velocity $u_t(x) = 16x^2(1 - x^2)$ and the initial velocity field is specified through a stream function $\psi_0(x, y) = (y^2 - y^3)u_t(x)$. The velocity is then computed by $u = -\psi_y$ and $v = \Psi_x$. In this case, we wanted to test the numerical stability of the flow solver on problems with large Reynolds numbers which will result in a very thin boundary layer at the top boundary. Figure 15 displays a velocity vector field and a vorticity contour from a calculation with Re = 10^5, at time = 4.69 for a total of 3000 time steps using a 256 × 256 grid. Figure 16 displays the result from a calculation with Re = 10^6 for a total of 7000 time steps on a 512 × 512 grid. These computations were performed on Cray T3D using 64 processors. We noticed the computations of our solver at these Reynolds numbers seem to be still numerically stable, which we can judge by checking the convergence rate of the pressure equation and the numerical divergence of the

computed velocity. The computed flow structures at these Reynolds numbers, however, are quite different from that obtained from the computed flow with Re = 5000. First, at these high Reynolds numbers, the computed flows did not show any sign of approaching a steady state after computing the large numbers of time steps; while with Re = 5000, for the same initial and boundary conditions, we found a steady state can be reached after computing a smaller number of time steps. Second, we can see some interesting flow patterns which do not exist in the flow with Re = 5000. As shown by the vorticity contours in Figs. 15 and 16, the vorticity structures in these high Reynolds number flows are much more complicated. We can see a large amount of vortices are generated from the top boundary and are then being flushed down along the right wall. Once these vortices reach the neighborhood of the lower

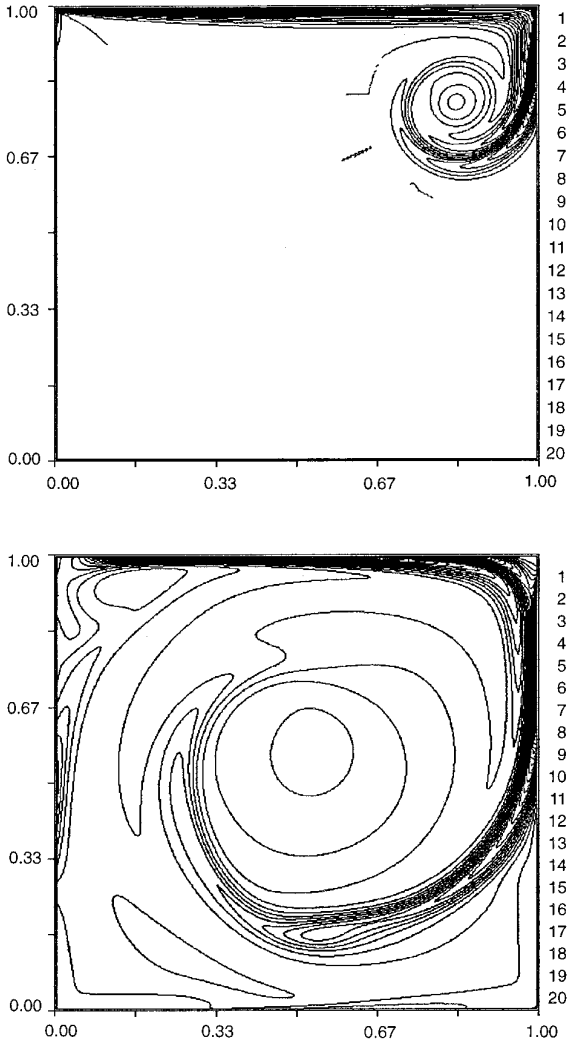

FIG. 14. Vorticity contour plots from a driven-cavity flow with Re = 5000, at time = 3.91 (top) and time = 15.63 (bottom). Grid size = 256 × 256.

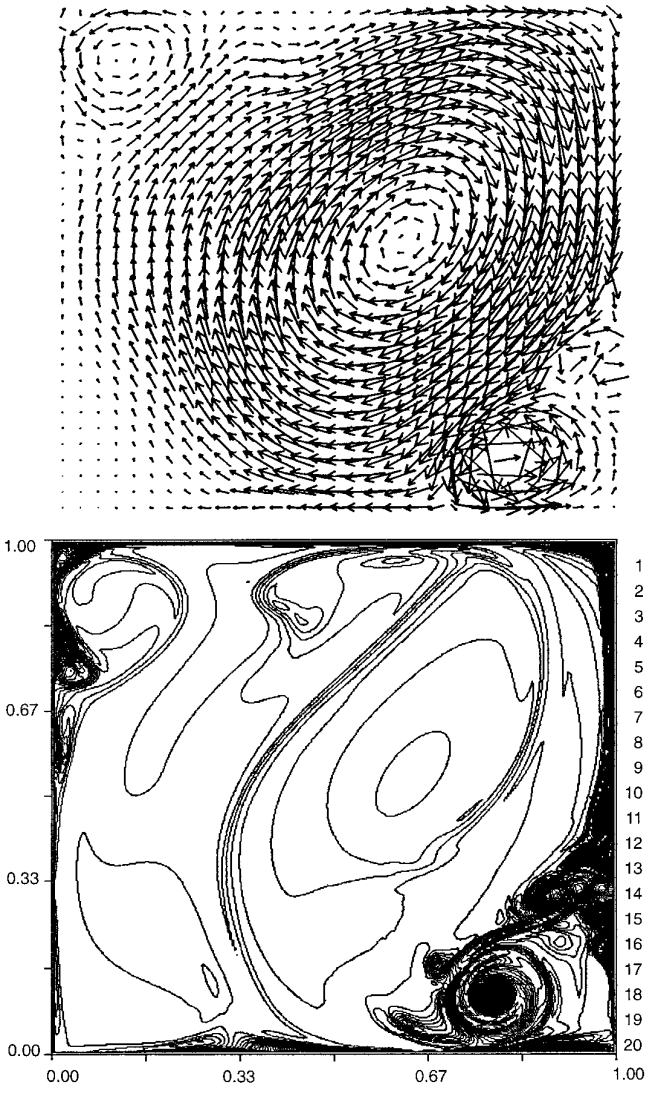

FIG. 15. Velocity vector field (top) and vorticity contour plot (bottom) from the driven-cavity flow with Re = 10^5, at time = 4.69. Grid size = 256 × 256.

right corner, they are pushed toward the interior of the box. We found the vorticity plot in Fig. 16 is similar to that reported in [12], where a different algorithm was used on the same problem.

The second problem we tested on our flow solver is an inviscid fluid flow for which the Euler equations are solved. The computational domain is again restricted to a unit box, and a periodicity of one is assumed in both horizontal and vertical directions. The initial velocity field is given by

$$u = \begin{cases} \tanh(y - 0.25)/\rho & \text{for } y \geq 0.5 \\ \tanh(0.75 - y)/\rho & \text{for } y > 0.5 \end{cases} \qquad (12)$$

$$v = \delta \sin(2\pi x),$$

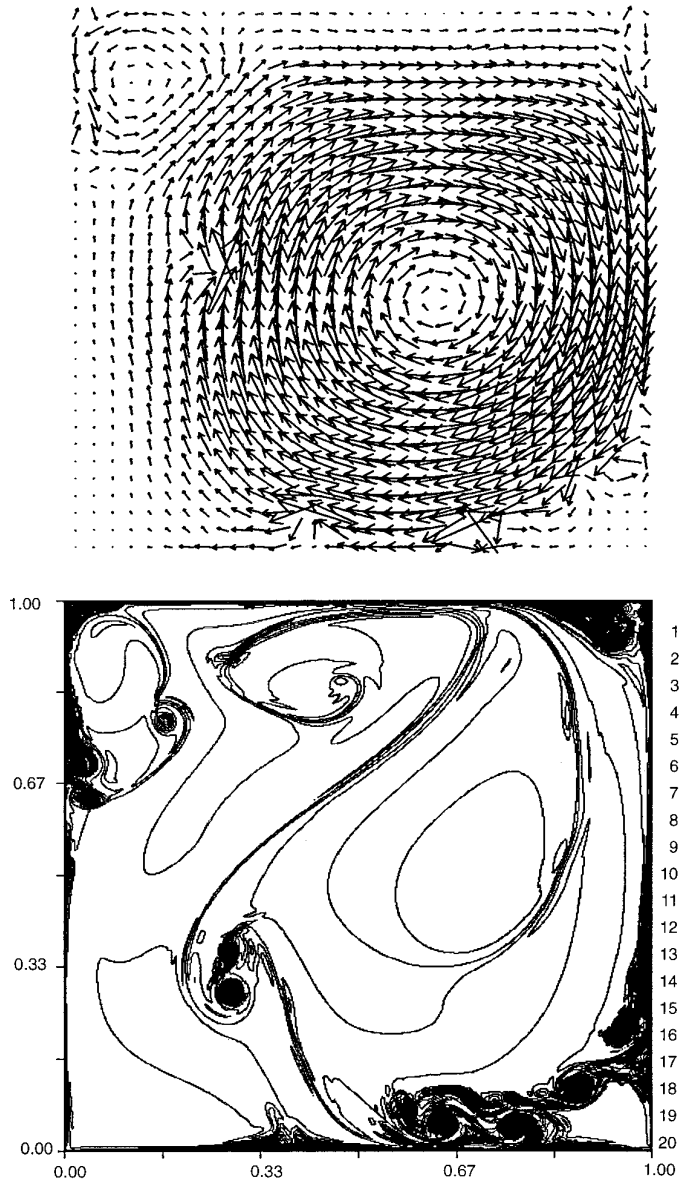

FIG. 16. Velocity vector plot (top) and vorticity contour plot (bottom) from a driven-cavity flow with Re = 10^6, at time = 5.47. Grid size = 512×512.

where $\rho = 0.03$ and $\delta = 0.05$. Thus the initial flow field consists of a jet which is a horizontal shear layer of finite thickness, perturbed by a small amplitude of vertical velocity. Since the viscous term is dropped, the pressure can be updated without computing the intermediate velocity field and the multigrid elliptic solver is only used for solving the pressure equation (9). In addition, only one iteration at each time step is needed for computing $p^{n+1/2}$ and $\mathbf{u}^{n+1}$ because the pressure can be computed without the knowledge of $\mathbf{u}^{n+1}$. On the staggered grid we used, the pressure field is defined on a cell-centered grid whose linear dimension, say N, is preferably taken as a power of 2 for the convenience of applying grid coarsening. Thus there are N^2 unknowns for the pressure. Since the velocity field is only related to the pressure gradient in the momentum equations, it makes sense to have the velocity defined on an $(N-1) \times (N-1)$ grid, as shown in Fig. 17. Therefore there are $(N-1)^2$ unknowns for each velocity component. Since the velocity is periodic, a periodic domain should have a dimension of $(N-1) \times (N-1)$. Since the pressure gradient is a function of velocity, it must have the same dimension of periodicity. Thus the numerical boundary condition for the pressure gradient in the finite-difference equations derived from pressure equation (9) in the horizontal direction, for example, can be specified as

$$\begin{aligned} P_{0,j} &= P_{1,j} + P_{N-2,j} - P_{N-1,j}, \\ P_{N+1,j} &= P_{N,j} + P_{2,j} - P_{1,j}. \end{aligned} \tag{13}$$

In a multigrid solution of the pressure equation, the boundary condition (13) is clearly for use on the original, finest grid. The use of condition (13) on any coarse grid, however, is incorrect (our numerical experiments indicate the use of (13) on coarse grids will blow up the computation quickly). Since the unknown vector on a coarse grid is the difference between an exact solution and a approximate solution on the fine grid restricted to that coarse grid, the solution on a coarse grid can be regarded as an approximation to the derivative of the solution on the fine grid. Since a derivative of the pressure field of any order is still periodic with the same period as the velocity field, a reasonable boundary condition for pressure on coarse grids is

$$P_0 = P_N, \quad P_{N+1} = P_1. \tag{14}$$

Although condition (14) imposes a period which is one grid cell (of the finest grid) larger than the velocity period, we found it is easy to apply it to all the coarse grids, and our numerical results show it works well.

Numerical experiments for the inviscid periodic shear flow were performed on a Cray T3D, using 64 processors

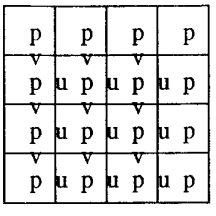

FIG. 17. An example of a staggered grid used for computing the doubly periodic shear flow with $N = 4$. Unknowns for velocity and pressure in the grid are shown.

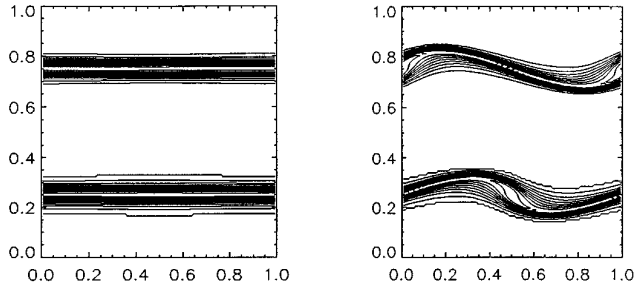

FIG. 18. Vorticity contour plots from the periodic shear flow at time = 0.0 (left) and 0.62 (right). Grid size = 128 × 128.

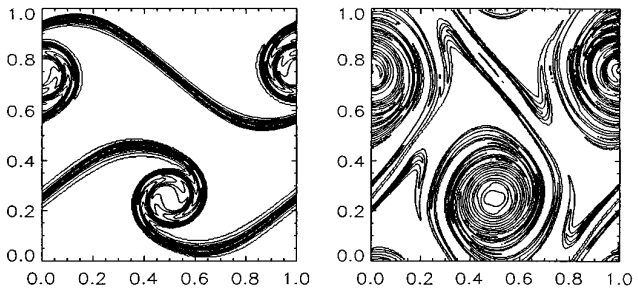

FIG. 20. Vorticity contour plots from the periodic shear flow for time = 1.25 (left) and time = 2.50 (right). Grid size = 256 × 256.

in all cases. Figure 18 shows the vorticity contours of two early states of the inviscid periodic shear flow. Figures 19, 20, and 21 show the vorticity contours of the flow at time = 1.25 and 2.50, computed on a 128 × 128 grid, a 256 × 256 grid, and a 512 × 512 grid, respectively. The CFL number used in the computations is still 0.4. On the 512 × 512 grid, a total of 3200 time steps were computed to reach time = 2.50. These vorticity plots show how the shear layers, which form the boundaries of the jet, evolve into a periodic array of vortices, with the shear layer between the rolls stretched and thinned by the large straining field there. A comparison between Figs. 19–21 clearly shows the resolution of the vorticity structure improves as the computational grid gets finer.

The parallel performances of the incompressible flow solver were also evaluated in terms of speedup and scalability. In each of the parallel performance measurements, we ran the flow solver on the driven-cavity problem for one time step, excluding any initialization and assignment of initial and boundary conditions. Figure 22 shows the speedup curves of the flow solver on the Intel Paragon and the Cray T3D systems for three different problem sizes (ideal speedup curves are shown again for comparison). The speedup performance improves as the problem size increases, as expected. For the 512 × 512 grid, no significant reduction in execution time could be obtained after more than 64 processors were used. By running the flow solver on a single processor, we found the T3D is about 5 times faster than the Paragon for the code compiled with the -O2 switch. But on 256 processors, the T3D runs only about 1.5–2.0 times faster than the Paragon, depending on problem sizes, because, as shown in Fig. 22, the speedup performance of the flow solver on the Paragon is better than on the T3D. Figure 23 shows the scaling performances of the parallel flow solver on the T3D and the Paragon for three local problem sizes. Again, we see the scaling improves as the size of local grid increases on both machines. In measuring the scalings of the flow solver, we used smaller local problem sizes than we did for the multigrid elliptic solver (see Figs. 11 and 12). We expect the flow solver to have better scalings than the multigrid elliptic solver because the flow solver, even though calling the elliptic solver several times at each time step, does substantially more additional processings on the finest grid. Indeed, this scaling difference between the two solvers can be verified by looking at the scaling curves for the 64 × 64 local grid for the flow solver in Fig. 23 for the multigrid full V-cycle (which is used in the flow solver) in Figs. 11 and 12. In view of the scaling performances in Fig. 23, we would claim that our parallel flow solver scales quite well on large numbers of processors, as long as the local grid size is not smaller than 64 × 64.

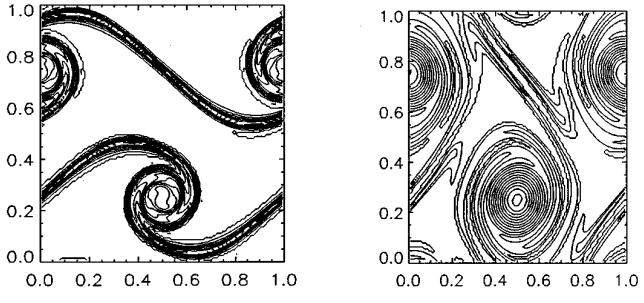

FIG. 19. Vorticity contour plots from the periodic shear flow for time = 1.25 (left) and time = 2.50 (right). Grid size = 128 × 128.

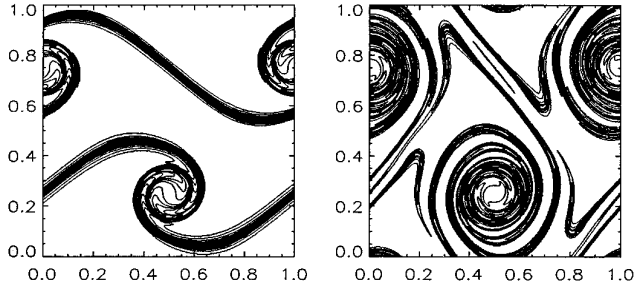

FIG. 21. Vorticity contour plots from the periodic shear flow for time = 1.25 (left) and time = 2.50 (right). Grid size = 512 × 512.

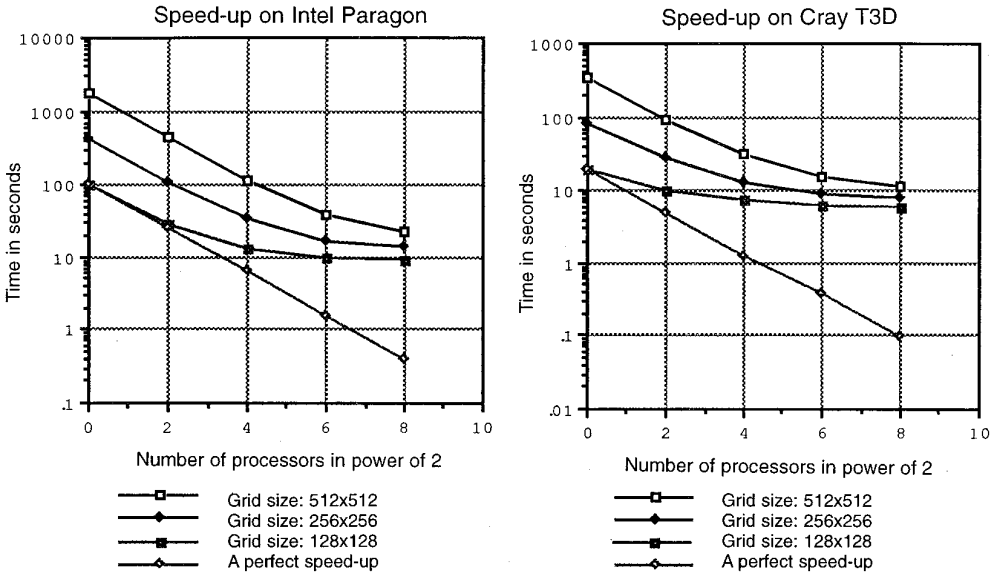

FIG. 22. Speedup performances of the parallel Navier–Stokes.

5. CONCLUSIONS

In this paper we presented multigrid schemes for solving elliptic PDEs and a second-order projection method for solving the Navier–Stokes equations for incompressible fluid flows. Our parallel implementation strategies based on grid-partition are discussed for implementing these algorithms on distributed-memory, massively parallel computer systems. Our treatment of various boundary conditions in implementing these parallel solvers are also discussed. We designed and implemented these solvers in a highly modular approach so that they can be used either as stand-alone solvers or as expandable template codes which can be used in different applications. Several message-passing protocols (MPI, PVM, and Intel NX) have been coded into the solvers so that they are portable to systems that support one of these interfaces for interprocessor communications.

Numerical experiments and parallel performance measurements were made on the implemented solvers to check their numerical properties and parallel efficiency. Our numerical results show the parallel solvers converge with the

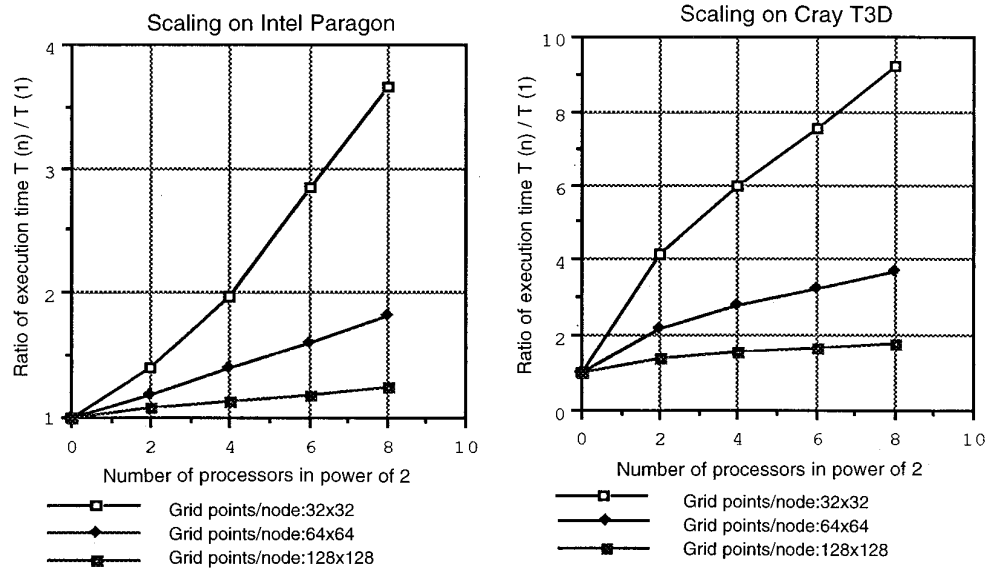

FIG. 23. Scaling performances of the parallel Navier–Stokes solver.

order of numerical schemes on a few test problems. Our numerical experiments also show the flow solver is stable and robust on viscous flows with large Reynolds numbers as well as on an inviscid flow. Our parallel efficiency tests on Intel Paragon and Cray T3D systems show that good scalability on a large number of processors can be achieved for both the multigrid elliptic solver and the flow solver as long as the granularity of the parallel application is not too small, which we think is typical for applications running on distributed-memory, MIMD machines. For future work, we plan to generalize the parallel solver package to thermally driven flows and variable density flow problems (actually, these works are underway at the time of writing this paper) and to extend the flow solver to 3D problems.

ACKNOWLEDGMENTS

The authors thank Dr. Sefan Vandewalle (California Institute of Technology) and Dr. Steve McCormick (Univeristy of Colorado) for some helpful discussions on multigrid methods. This work was carried out at the Jet Propulsion Laboratory (JPL), California Institute of Technology (Caltech), under a contract with the National Aeronautics and Space Administration (NASA) and as a part of the NASA High-Performance Computing and Communications for Earth and Space Sciences Project. The computations were performed on Intel Paragon parallel computers operated by JPL and by the Concurrent Supercomputing Consortium at Caltech, and on the Cray T3D parallel computer operated by JPL.

REFERENCES

1. C. Anderson, Lawrence Berkeley Laboratory Report, LBL-26353, 1988, Berkeley, CA (unpublished).
2. J. B. Bell, P. Colella, and H. Glaz, *J. Comput. Phys.* **85,** 257 (1989).
3. J. B. Bell, P. Colella, and L. H. Howell, ''An Efficient Second-Order Projection Method for Viscous Incompressible Flow,'' in *Proceedings, 10th AIAA Computational Fluid Dynamics Conference, Honolulu, HI, 1991,* p. 360.
4. J. B. Bell and D. L. Marcus, *J. Comput. Phys.* **101**(2), 334 (1992).
4. W. Briggs, *A Multigrid Tutorial* (SIAM, Philadelphia, 1987).
5. T. F. Chan and R. S. Tuminaro, ''A Survey of Parallel Multigrid Algorithms,'' in *Parallel Computations and Their Impact on Mechanics,* edited by A. Noor, Vol. 86, 1986.
6. A. J. Chorin, *Math. Comput.* **22,** 745 (1968).
7. G. Fox *et al., Solving Problems on Concurrent Processors,* Vol. I (Prentice–Hall, Englewood Cliffs, NJ, 1988).
8. S. F. McCormick, *Multilevel Adaptive Methods for Partial Differential Equations,* Frontiers in Appl. Math. (SIAM, Philadelphia, 1989).
9. W. D. Henshaw, Ph.D. thesis, Dept. Appl. Math., California Institute of Technology, Pasadena, CA, 1985 (unpublished).
10. F. Roux and D. Tromeur-Dervout, manuscript, 1994.
11. R. Schreiber and H. B. Keller, *J. Comput. Phys.* **49,** 310 (1983).
12. W. E. and J.-G. Liu, manuscript, 1994.
13. P. Wesseling, *An Introduction to Multigrid Methods,* Pure & Appl. Math. (Wiley, New York, 1991).