A Parallel Three-Dimensional Incompressible Navier-Stokes Solver with a Parallel Multigrid Kernel

John Z. Lou* and Robert Ferraro **

Abstract

The development and applications of a parallel, time-dependent, three-dimensional 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 implemented in the parallel elliptic kernel, which is used 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 kernel on all fine and coarse grids. Numerical experiments and parallel performance measurements show the parallel solver package is numerically stable, physically robust and computationally efficient. Both the multigrid kernel and the flow solver scale well to a large number of processors on Intel Paragon and Cray T3D/T3E for two and three dimensional problems with moderate granularity. The solver package has been carefully designed and implemented so that it can be easily adapted to solving a variety of interesting scientific and engineering flow problems. The code is portable to parallel computers that support MPI, PVM and NX for interprocessor communications.

1. Introduction

The 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, non-smooth flow problems must be numerically stable, physically robust and computationally efficient. Results from numerical experiments in [2], [3] and [4] indicate that a second-order projection method proposed in [2] is a promising candidate for simulations of complex incompressible flows.

Our task 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 portabilit y 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 rect-

* lou@acadia.jpl. nasa.gov, Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91109

** ferraro@zion.jpl .nasa.gov, Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91109

angular grid or on a composite grids with each of its component 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 while the use of a finite-element type scheme may not be desirable for some applications 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 non-rectangular domains, such mappings can indeed be constructed (for more detail, see [10]).

A projection method for solving incompressible Navier-Stokes equations was first described in a paper by Chorin [7]. 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 stability. The 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 (1 or 2 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 IV-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 was designed to be a general-purpose elliptic solver. It can solve N-dimensional ($N \leq 3$) 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-partitioning: 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 they are therefore “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 can not 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 [6] [9], 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. [6] [9] [1 l]). 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 grids, In fact, the percentage of time spent on those coarse grids is insignificant compared to the total computation time. A similar observation was also made in [9]. As shown by a simple asymptotic analysis in [8], 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 [5] and [9], 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 **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.

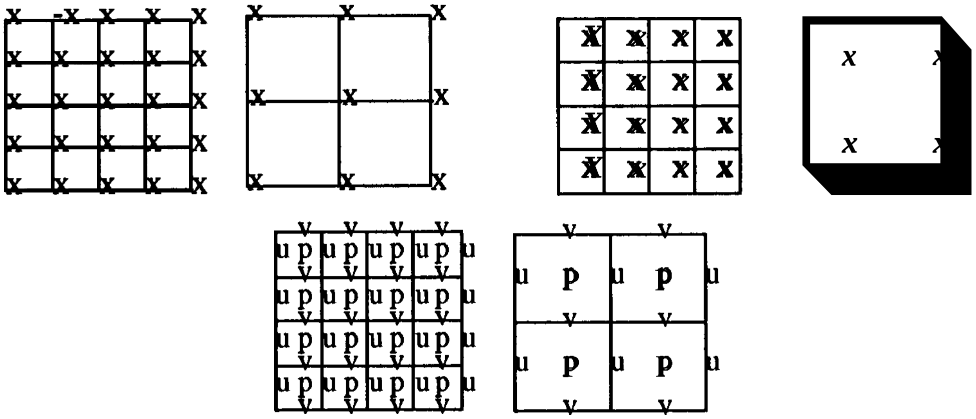
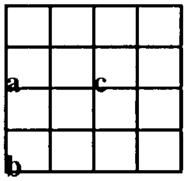


Figure 1: Coarsening of three types of grids: vertex-centered (top-left), cell-centered (top-right) and staggered (bottom).

Atypical 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 1 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 fill-weighting stencil (9-point averaging on a 2D grid) to make the V-cycle scheme converge well when a pointwise red-black Gauss-Seidel (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 newest-neighbor restriction stencil with the point-wise red-black 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. We think the failure of point-wise red-black GS with the nearest-neighbor restriction stencil on vertex-centered grid is due to the decoupling of the two sets of grid points. 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 constructions of restriction and interpolation operators for different types of grids can be found in [14].

Our multigrid solver can solve Dirichlet and Neumann problems for the grids depicted in Figure 1. A periodic boundary condition is also implemented for a special case used in the NS flow solver. 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 Figure 2, where a 9-point full weighting stencil is used for restriction. This can be done naturally for the interior point c. For boundary 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:



$$\underset{c}{\frac{1}{16}\begin{bmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix}} \qquad \underset{a}{\frac{1}{16}\begin{bmatrix}0 & 2 & 2\\ 0 & 4 & 4\\ 0 & 2 & 2\end{bmatrix}} \qquad \underset{b}{\frac{1}{16}\begin{vmatrix}0 & 4 & 4\\ 0 & 4 & 4\\ 0 & 0 & 0\end{vmatrix}}$$

Figure 2: Restriction stencils for interior point c and boundary points *a* and *b*.

$$\sum_{I,J} U_{IJ} \text{ x } A_{IJ} = \sum_{i,J} U_{ij} \text{ x } a_{ij} \tag{1}$$

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 equation (1) are given in Figure 2.

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

$$\begin{gathered}\frac{ou}{\partial t} + (\boldsymbol{u} \cdot \nabla)\boldsymbol{u} = -\, \boldsymbol{Vp} + Re^{-1}\Delta \boldsymbol{u} \\ \nabla \cdot \boldsymbol{u} = 0\end{gathered} \tag{2}$$

where $\boldsymbol{u} \in \boldsymbol{R}^n (n = 2$ or $3)$ is the velocity field, $\boldsymbol{p} \in \boldsymbol{R}$ ***is the*** pressure field and ***Re*** is the Reynolds number. Atypical problem is to find ***u*** and ***p*** satisfying (2) in a domain Ω for a given initial velocity field $\boldsymbol{u}_0$ in Ω and a velocity boundary condition $\boldsymbol{u}_b$ on $\partial\Omega$. The projection method for solving equations (2) is based on the Hedge decomposition which states that any vector field ***u can*** be uniquely decomposed into a sum of $\boldsymbol{U}_1 + U_2$ with $\nabla \cdot \boldsymbol{U}_1 = \boldsymbol{O}$ and $U_2 = \nabla\phi$ 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

$$\frac{\partial \boldsymbol{u}}{at} = \mathbf{P}(Re^{-1}\Delta \boldsymbol{u} - (\boldsymbol{U} \cdot \nabla)\boldsymbol{u}) \tag{3}$$

where **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 **P** to the momentum equation in

(2) which can be rewritten as

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

The projection operation removes the pressure gradient in (4) because Vp is orthogonal to the projection, Thus if we let the right-hand side of (4) be a vector field ***V,*** then Vp = **(I - P)V. The second-order** projection method in [2] is a modification to the original projection method proposed in [7] to achieve a second-order temporal accuracy and an improved numerical stability for the nonlinear convection. It use-s the following temporal **discretization** on the momentum equation at each half time step $n+$ 1/2

$$\frac{\bar{u}^{n+1,\ k}\ ‘u^{n}}{At} + [(u \cdot \nabla)u]^{n+1/2} = -\nabla p^{n+1/2,k} + \frac{1}{Re}\Delta\left(\frac{\bar{u}^{n+1,k} + u^{n}}{2}\right) \qquad (5)$$

where we assume the velocity u^n ***is*** known, and $\bar{u}^{n+1,k}$ 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 halftime step $n+$ 1/2, The superscripts ***k* in** (5) indicate that an iterative process is used for computing the **velocit** y at next time step u^{n+1}, and the pressure at next half time step $p^{n+1/2}$: given a divergence-free field 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 $\bar{u}^{n+1,k}$. **Since** the **correct pressure** $p^{n+1/2}$ is not known, the computed $\tilde{u}^{n+1,k}$ is usually not **divergence-free;** but $\bar{u}^{n+1,k}$ can be used as a **guess** for $u^{n}+^{1}$ and it is used to compute $p^{n+1/2,k}$, a new guess for $p^{‘+1/2}$, by solving a pressure equation, Once ***we*** have a new guess for $p^{n+1/2}$, it is used in (5) to compute $\bar{u}^{n,k+1}$. This **iterative** procedure is performed at each time step until $Vp^{n+1,2,k} \rightarrow \nabla p^{n+1,2}$ and $\bar{u}^{n+1,k} \rightarrow \tilde{u}^{n+1}$. **This iterative process** converges because it can be shown that the mapping of errors from state ***k*** to state ***k+*** 1 is contractile [2]. In practice, we found 1 to 2 iterations would be enough to get a satisfactory convergence.

The convection term $(u \cdot \nabla)u$ is evaluated at the half time step ***n+*** 1/2, using only the velocity u“ and the pressure $p^{‘-1/2}$.On the staggered grid shown in Figure 1, 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 $(i$ - l/2) Ax, $(j - 1/2)\Delta y$ for $i = 1 \ldots I$ and j = 1 . . . J. $(u” \nabla)u$ is then **evaluated** at i, j - 1/2 for u component and i - 1/2, j for v component. The **discretization** for u component, for example, has the following form

$$[(u “ \nabla)u]_u = \frac{‘i-1/2,\ j-1/2 + u_{i+1/2,}\ j-1/2}{2}\left(\frac{‘i-1/2,\ j-1/2\ ‘i+1/2,\ j-1/2}{Ax}\right)$$

$$+ \frac{vi,_{j}\ \cdot\_,\ ,j}{2}\left(\frac{u_{i,j} - u_{i,j-1}}{\Delta y}\right)$$

where $U_{i\pm1,2,\ j\pm1/2}$ *are* velocities at cell centers, $u_{i,}$ $_{‘j}$ **and** vi, $_j$ are velocities at **cell corners** and **all** velocities are assumed to be at time ***n+*** 1/2. Since 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 comers, we use Taylor expansions of second-order accuracy in both time and space, as was done in [2] and

[3], to find velocities at appropriate locations and at the halftime step $n+1/2$ 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

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

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

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

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 constructions of these derivatives are given in [2] and [3].

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

$$-\Delta\bar{u}^{n+1,k} + \frac{2Re}{\Delta t}\bar{u}^{n+1,k} = 2Re\left(-[(u" \nabla)u]^{n+1/2} + \frac{1}{\Delta t}u^n + \text{Au}^n - \nabla p^{n-1/2}\right) . \tag{8}$$

We notice that the matrix resulted from equation (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 Euler (inViscid) flow problems where $Re = \infty$, $\bar{u}^{n+1,k}$ ***can be*** computed explicitly from equation (5). Once $\bar{u}^{n+1,k}$ is computed, a projection step is performed to find the pressure field $p^{n+1/2,k+1}$ by Solving a Poisson equation:

$$\text{Ap} = R(u^n, u^{n+} ‘) \tag{9}$$

where $\bar{u}^{n+1,k}$ is used in place of u^{n+1} in the right-hand side of equation (5). Mathematically, equation (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 equation (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 pressure boundary condition for 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 equations (8) and (9). After the pressure field $p^{n+1,k+1}$ is found, $u^{n+1,k+1}$ ***can be*** computed by using equation (5) with $p^{n+1,k+1}$ in place of

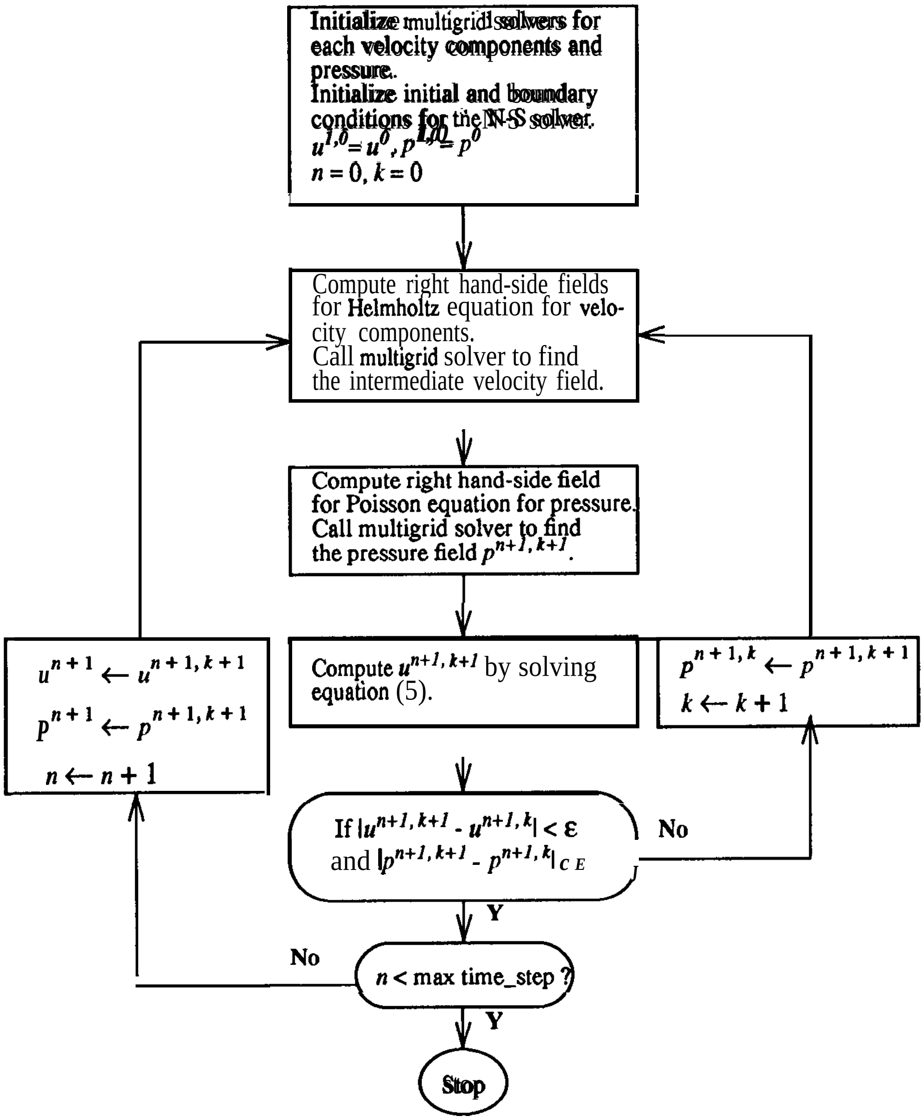

Initialize multigrid solvers for each velocity components and pressure.
Initialize initial and boundary conditions for the N-S solver.
$u^{1,0} = u^0, p^{1,0} = p^0$
$n = 0, k = 0$
Compute right hand-side fields for Helmholtz equation for velocity components.
Call multigrid solver to find the intermediate velocity field.
Compute right hand-side field for Poisson equation for pressure.
Call multigrid solver to find the pressure field $p^{n+1,k+1}$.
Compute $u^{n+1,k+1}$ by solving equation (5).
$u^{n+1} \leftarrow u^{n+1,k+1}$
$p^{n+1} \leftarrow p^{n+1,k+1}$
$n \leftarrow n+1$
$p^{n+1,k} \leftarrow p^{n+1,k+1}$
$k \leftarrow k+1$
If $|u^{n+1,k+1} - u^{n+1,k}| < \varepsilon$ and $|p^{n+1,k+1} - p^{n+1,k}| c \varepsilon$
No
Y
No
$n <$ max time_step ?
Y
Stop

Figure 3: Flow diagram for the Navier-Stokes solver

$p^{n+1,k}$, and this completes one iteration in the computations for the time step n +1. 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 Figure 3.

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 $(M \leq N)$ logical processor meshes. For example, Figure 4 shows the partition of a three-dimensional grid and the

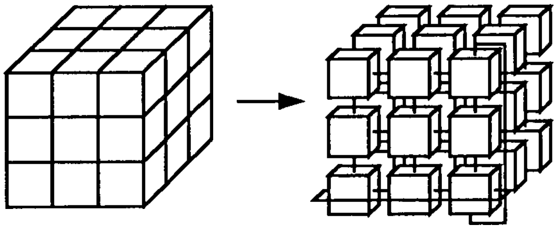


Figure 4: A 3D Grid partition and mapping to a processor mesh. Only two wrap-around connections were shown in the logical processor mesh.

assignment of the partitioned subgrids to a three-dimensional torus processor mesh. As shown in Figure 5, 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 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 processors in the fine processor mesh. This process is repeated recursively until all coarse grids and coarse processor meshes have been constructed. As an illustration, Figure 5 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 also knows its neighboring processors on that grid level.

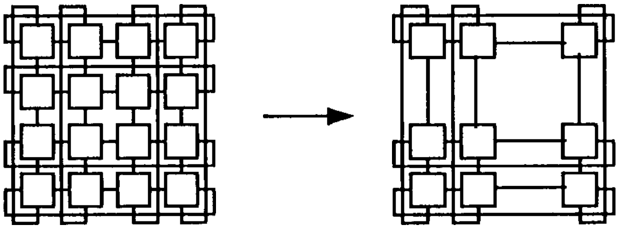


Figure 5: If the left processor mesh contains a 5x5 grid for a Neumann problem on a vertex-centered grid, then the derived coarse processor mesh is the one on the right.

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"

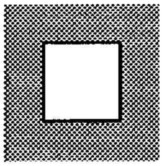


Figure 6: A local subgrid (white area) with surrounding ghost points (shaded area).

which are duplicates of grid points contained in other processors, as shown in Figure 6. The number of ghost points on each side of the subgrid depends on 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 sand-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 Figure 7. Since processors Pi and P_j in Figure 7 are not nearest neighbors, direct data exchange between them will introduce a more complicated message-passing pattern. Fortunately, direct data exchange between Pi and P_j is not necessary to get the diagonal ghost grid points. It can be verified that **all** data exchanges that we need are of nearest neighbor types, as indicated in Figure 8 for 2D problems. As can be seen in Figure 8 that when data lying on partition boundaries are exchanged, the sending blocks always include ghost grid points. After data exchanges in Figure 8 are performed, **all** ghost grid points shown in Figure 6 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 Figure 8 can be used.

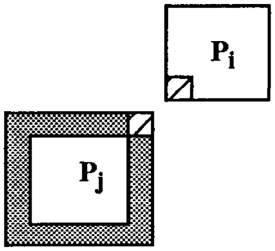


Figure 7: The data in the lower left corner of the **subgrid** in **processor** Pi are needed by processor P_j, **and stored** in P_j 's ghost grid points at upper right corner,

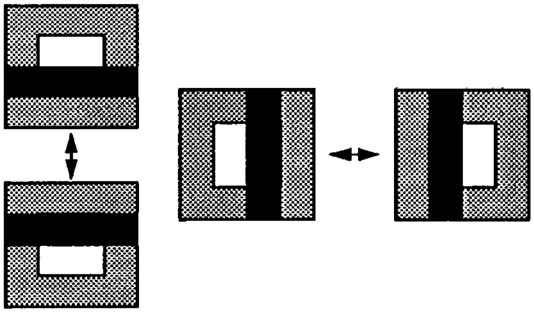


Figure 8: Data exchanges between neighboring processors for 2D problems. The data in black blocks in each processor are sent out., which is stored in the blocks for ghost grid points in the neighboring processors.

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 the Intel Paragon, we implemented the second set of operations above in the multigrid solver and the flower solver using asynchronous message-passing calls. For one full V-cycle in the elliptic solver, for example, the performance improvement on a 256x256 grid partitioned among 256 processors is about 15%, and the improvement on a 256^3 grid partitioned among512 processors is about 22%. Faster and asynchronous interprocessor communication can also be achieved on the 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 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 exchange of partition boundary data in the flow solver and for 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 routine 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 to 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 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 share&memory communication library on the 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 data 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 performance in terms of speed-up and parallel scaling of the solvers on the Intel Paragon and the Cray T3D systems for problems with different sizes and granularity.

A. The Elliptic Multigrid **Solver**

The multigrid elliptic solver was first tested on a Helmholtz equation with known exact solutions, Tables 1 and 2 show the convergence rates of the multigrid solver for solving 2D and 3D Helmholtz equations of the form

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

with a Dirichlet boundary condition. The runs were performed on the Intel Paragon, Errors displayed are the normalized maximum norm of the difference between the computed solution and the exact solution. The tables show the number of 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. For comparison, the results of using the red-black Gauss-Seidel scheme were also displayed in the last row of each table. Although the full V-cycle scheme has been shown numerically more efficient than the V-cycle scheme in sequential processing [5], the same statement may not be true in parallel processing. For the 2D test case in Table 1 with a 2048^2 grid, we found the execution time needed for the computed solution to reach a fixed accuracy is about the same for V-cycle and full V-cycle schemes. As shown for a 3D test case in Table 2 with a 256^3 grid, the full V-cycle scheme is still a little more efficient. With a grid-partition 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. 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.

Table 1: Numerical Convergence: 2D **Helmholtz** Solver

Scheme	Grid size	Error	#Processors	# Cycles	CPU sec.
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

Table 2: Numerical Convergence: 3D **Helmholtz** Solver

Scheme	Grid	Error	#Processors	# Cycles	CPU sec.
Full V-cycle	256^3	1.2×10^{-5}	64	4	71.3
V-cycle	256^3	2.6×10^{-5}	64	8	86.7
Single grid	256^3	8.3×10^{-1}	64	400 R-B sweeps	652.1

The parallel performance of an application code is usually judged by two measurements: speed-up and scaling. Speed-up 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 speed-up can be obtained for many applications with a small number of processors, the reduction in CPU time often diminishes rapidly as the number of processors used exceeds some threshold. This phenomenon is largely inevitable, as stated in Amdahl's law. Because as the number of processors used increases for a fixed problem size, the local communication cost and the cost for global operations will eventually become dominant over the local computation cost after a certain stage. This high ratio of communication to computation makes the influence of a further reduction in the local computation very small on the overall cost of running the application, On the other hand, the scaling performance seems to be a more realistic measure of an application's 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, and this is indeed one of the main reasons to use a parallel machine. Figure 9 displays two speed-up 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 the Intel Paragon and the Cray T3D systems. For a comparison, an ideal speed-up curve for one test case is also shown. The code was compiled with the-02 switch on both machines. The grid size for the 2D problem is 5122, and the grid size for the 3D problem is 64^3. 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 x 8 x 8 was used) and512 processors were used on the Paragon.

In terms of single processor performance, we found, for our multigrid solver, that the Cray T3D is about 4 times faster than the 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 speed-up 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, speed-up 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 speed-up, we can still see some reduction in CPU time when the largest number of processors was used in each case.

Figure 10 shows the scaling of the parallel multigrid solver on the Intel Paragon for problems with three different granularity, using up to 512 processors. Figure 11 shows the scalings of the same problems on the Cray T3D, using up to 256 processors, Shown in the plots are the ratio of CPU times of using n processors versus using one processor. On each of the scaling curves, we fix the local 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 full V-cycle, 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 speed-up 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 performance on each specific machine.

We can see from all the plots in Figure 10 and Figure 11 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 [8]. 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) though the size of each message is larger. We can also see V-cycle scheme scales somewhat better than full V-cycle scheme, which is expected since the latter does more operations on coarse grids. The scaling plots also show 2D test cases scales bet-

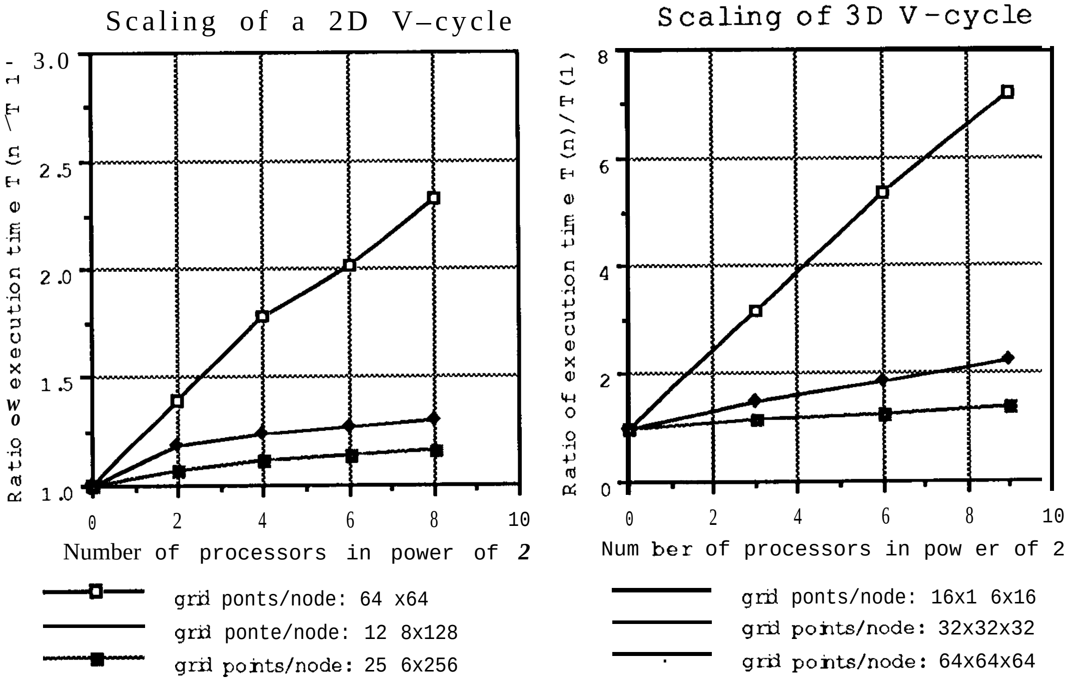


Figure 10: Scaling of the **multigrid** solver on Intel Paragon.

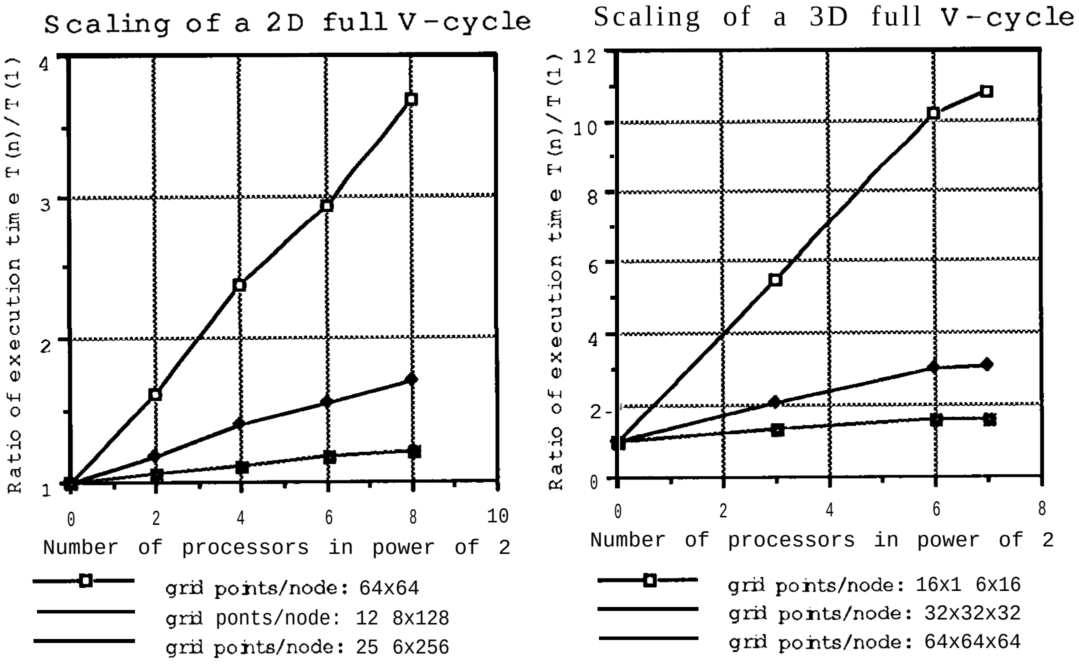


Figure 11: Scaling of the **multigrid** solver on Cray T3D.

ter than 3D test cases for the elliptic **multigrid** solver, which we think could be explained by the fact that coarse grid inefficiency (the existence of more idle processors on coarse grids) gets worse with a 3D grid than with a 2D grid. As for a comparison between the Intel Paragon and the Cray T3D, our results show that the scaling on Paragon is slightly better than on T3D. This could be explained by the fact that single processor speed on T3D is much faster than on Paragon, whereas the speed of interprocessor communication on T3D is not proportionally faster when the PVM library is used for communication,

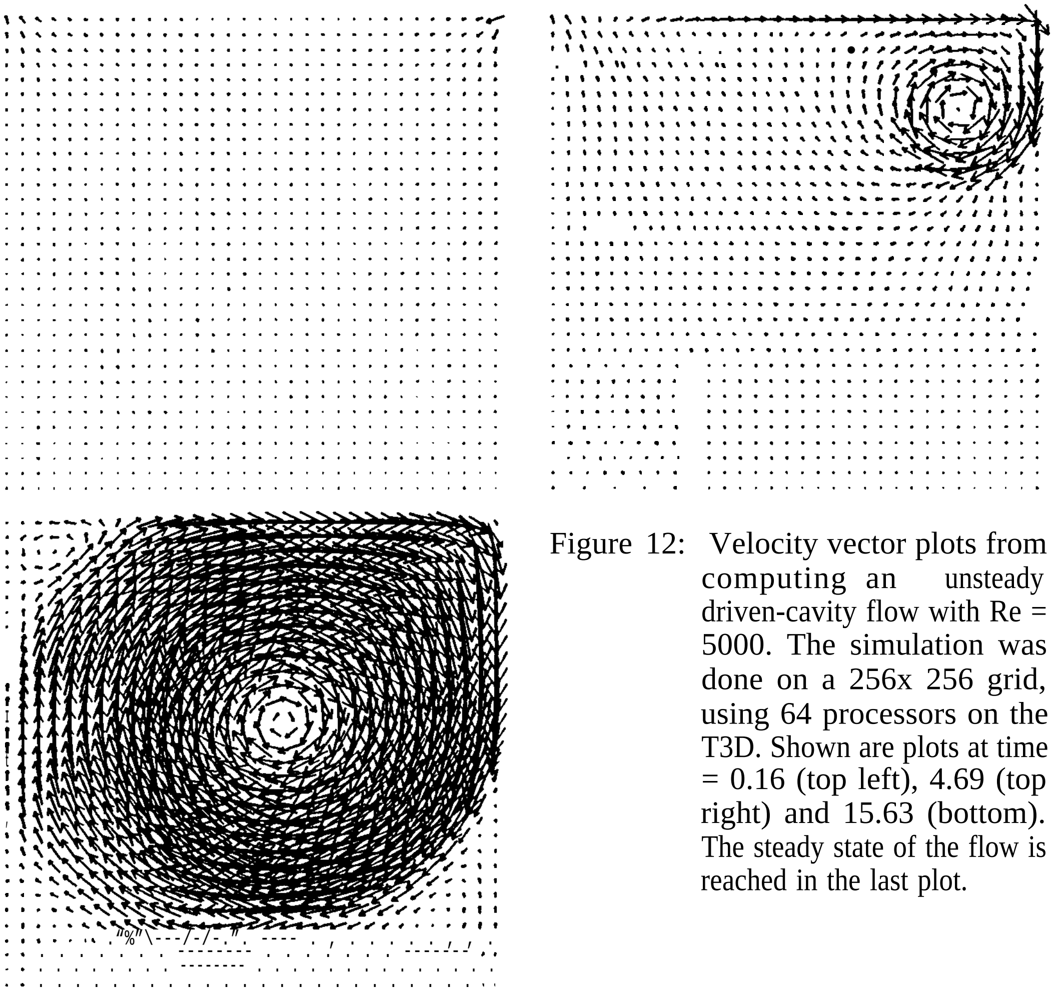


Figure 12: Velocity vector plots from computing an unsteady driven-cavity flow with Re = 5000. The simulation was done on a 256x 256 grid, using 64 processors on the T3D. Shown are plots at time = 0.16 (top left), 4.69 (top right) and 15.63 (bottom). The steady state of the flow is reached in the last plot.

B. The Incompressible **Navier-Stokes** Solver

The parallel Navier-Stokes flow solver was first tried out on a test problem in a unit square with a known exact solution. The purpose of the test is to examine the convergence rate of the flow solver on smooth problems. A second-order convergence rate was obtained on the test problem for Reynolds numbers up to 5000, as expected. For numerical stability of the Godunov scheme used in discretizing the convection term, the time step, At, is restricted by the following CFL condition

$$\frac{\Delta t}{\Delta x}U_{max} < \alpha, \qquad \text{(lo)}$$

in all our tests, where Δx is the size of grid cells, a is a constant around 0.5, and U_{max} is the maximum value in the current velocity field, A detailed description of such a testis given in [2].

Our first numerical flow experiment with the solver was to simulate an evolving 2D

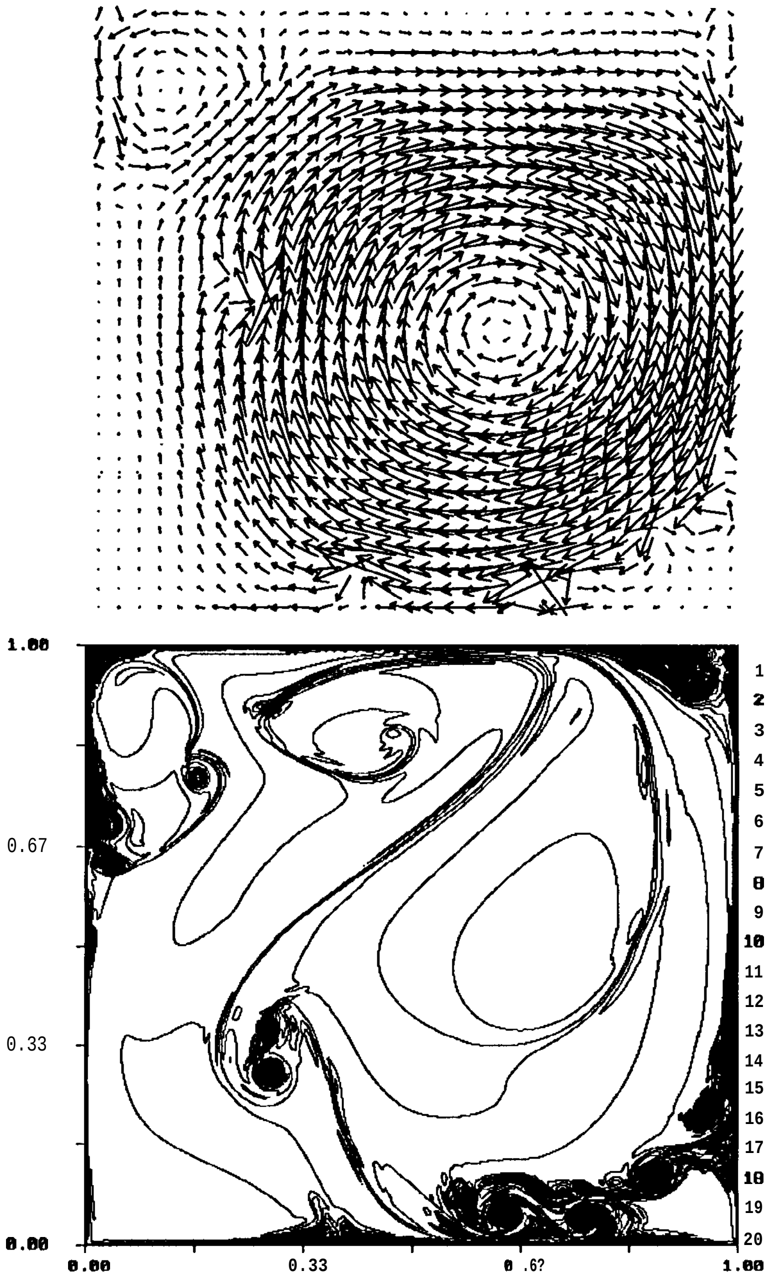


Figure 13: Velocity vector plot (top) and vorticitycontourplot (bottom) fromadriven-cavity flow with Re= 10^6, at time= 5.47. Grid size= 512x 512.

driven-cavity flow. The computational domain is still in a unit box 0< x, y ≤ 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 tested the solver on the problem in which the velocity initial condition is specified b y u = O inside the domain, and the velocity at the top boundary is always one. Figure 12 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 256x 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 were computed to reach the last state at time = 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. [12]). In running the parallel solver on the Cray T3D, the global staggered grid was partitioned and distributed to an 8 x 8 logically rectangular processor mesh. In computing the velocity vector field, velocity components defined on cell edges were averaged to the center of cells. For better visibility, the vector fields shown in Figure 12 are actually 32x 32 data arrays which were obtained by averaging the 256 x 256 velocity vectors from the simulation, Vorticity fields were computed at cell corners by central differencing. The velocity vector plots in Figure 12 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 comer (e.g. [12]), We also noticed, when reaching the final stage in Figure 12, 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 comers 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 13 displays the result from a calculation with Re = 10^6 for a total of 7000 time steps on a 512x512 grid. The computation was performed on the Cray T3D using 64 processors. We noticed the computation using our solver at such a high Reynolds numbers is 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 structure at this Reynolds numbers, however, is quite different from that obtained from the computed flow with Re = 5000. First, at this high Reynolds numbers, the computed flow does not show any sign of approaching a steady state after computing the large number 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 much smaller number of time steps. Secondly, we can see some interesting flow patterns which do not exist in the flow with Re = 5000. As shown by the vorticity contour in Figure 13, the vorticity structures in this high Reynolds number flow is much more complicated. We can see that a large amount of vortices are generated from the top boundary and then being flushed down along the right wall. Once these vortices reach the neighborhood of the lower right comer, they are pushed toward the interior of the box. We found the vorticity plot in Figure 13 is similar to what reported in [13] where a different algorithm was used on the same problem.

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

$$u = \begin{cases} \tanh(y - 0.25)/\rho & \text{for} \quad y \geq 0.5 \\ \tanh(0.75 - y)/\rho & \text{for} \quad y > 0.5 \end{cases} \tag{11}$$

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

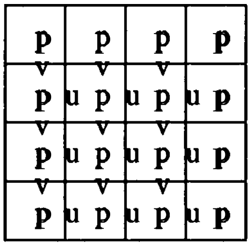


Figure 14: An example of a staggered grid used for computing the doubly periodic shear flow with $N = 4$. Unknowns for velocity and pressure in the grid are shown.

where p = 0.03 and δ = 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 u^{n+1} because the pressure can be computed without the knowledge of 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 velocity field is only related to the pressure gradient in the momentum equations, it makes sense to have the velocity defined on an $(N - 1)$x $(N - 1)$ grid, as shown in Figure 14. Therefore there are (N- 1)^2 unknowns for each velocity component. Since the velocity is periodic, a periodic domain should have a dimension of $(N - 1)$ x $(N - 1)$. Since the pressure gradient is a function of velocity, it must have the same dimension of periodicity. Thus the physical boundary condition for the pressure equation (9) in the horizontal direction, for example, can be specified as

$$‘O, j \;{}^{=}\; ' 1, \quad j + P_{N-2,\, j} - P_{N-1,} \quad j \quad P_{N+1,\, j} {}^{=} P_{N,\, j} + P_{2,\, j} - P_{1,}. \tag{12}$$

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 blowup the computation quickly). Since the unknown vector on a coarse grid is the difference between an exact solution and an 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

$$P_0 = P_N \; P_{N+1} = P_1. \tag{13}$$

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 Cray T3D,

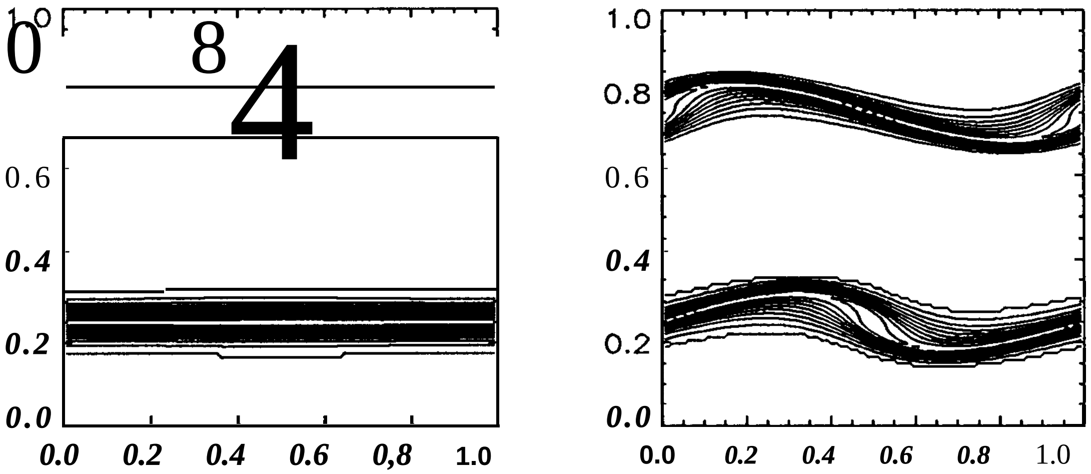


Figure 15: Vorticity contour plots from the periodic shear flow at time = 0.0 (left) and 0.62 (right). Grid size = 128x 128.

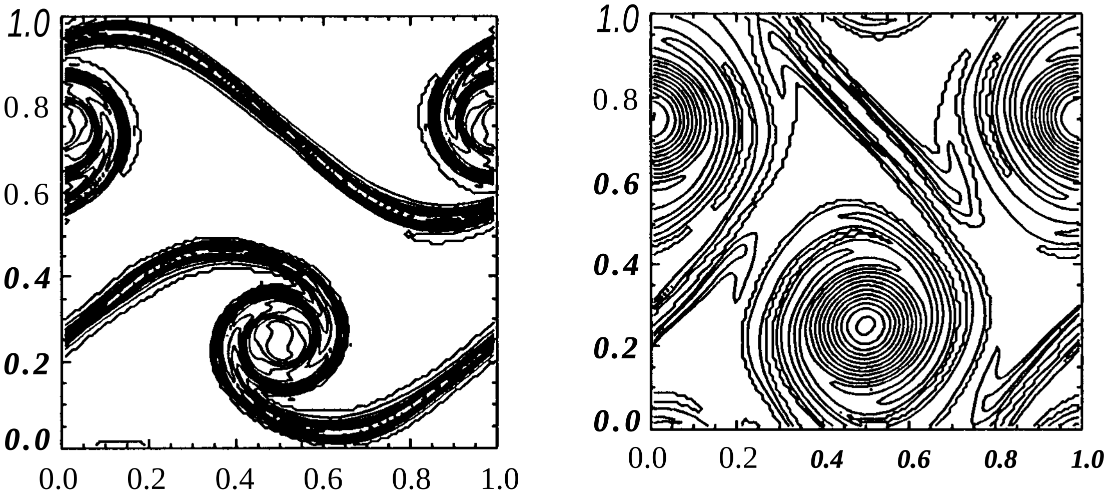


Figure 16: Vorticity contour plots from the periodic shear flow for time= 1.25 (left) and time = **2.50** (right). Grid size= 128x 128.

using 64 processors in all cases. Figure 15 shows vorticity contours of two early states of the inviscid periodic shear flow. Figure 16 and 17 show vorticity contours of the flow at time = 1.25 and 2.50, computed on a 128 x 128 grid, and a 512x512 grid, respectively. The CFL number used in the computations is still 0.4. On the 512x512 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 aperiodic array of vortices, with the shear layer between the rolls stretched and thinned by the large straining field there. A comparison between Figures 17 and 18 clearly shows the resolution of the vorticit y structure improves as the computational grid gets finer,

The three dimensional flow solver was tested on a driven-cavity flow in a unit cube. The

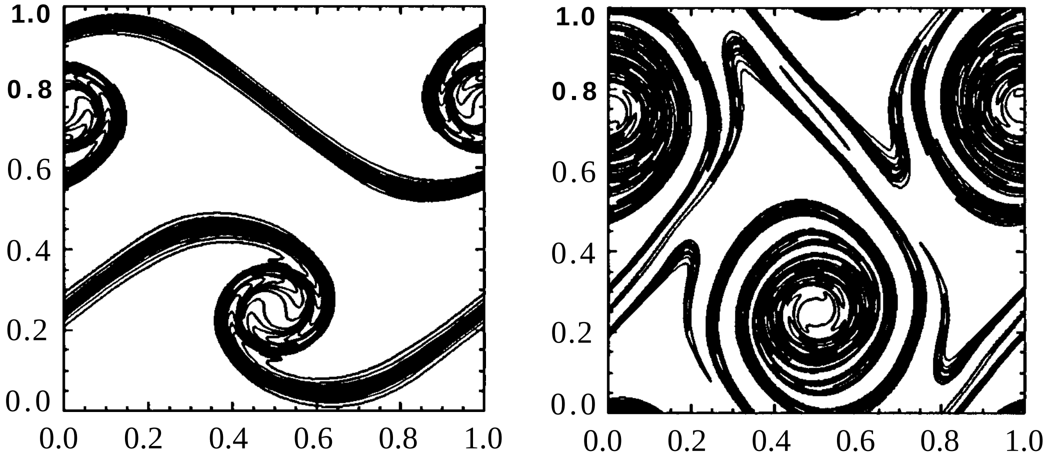


Figure 17: Vorticity contour plots from the periodic shear flow for time= 1.25 (left) and time =2.50 (right). Grid size = 512x 512.

initial and boundary conditions of the flow is set similarly to the 2D case: the fluid in the cubic box is at rest in the beginning, and no-slip boundary condition is enforced at the boundary except for the top of the cube where velocity is set to 1. We ran the 3D flow solver with a few Reynolds numbers in the range of 500 to 2500, and found the flows approach steady states in these simulations. Figure 18 shows plots of vorticity projection in three streamwise-vertical planes from a run with Re = 2500, on a 128^3 grid. The two planes close to the cube boundary are one grid point away from its corresponding grid boundary.

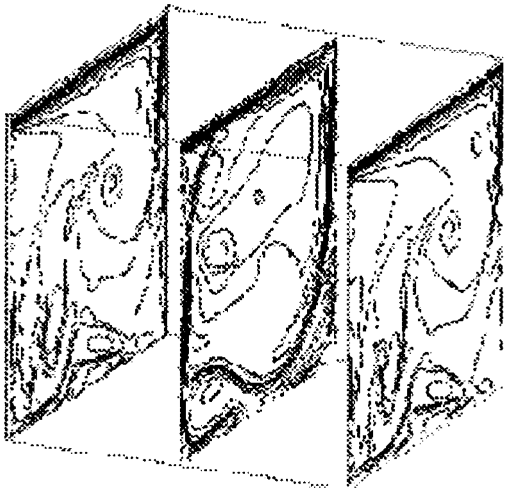


Figure 18: Projections of vorticity contour in streamwise-vertical planes in a cubic driven-cavity flow, with Re = 2500, at time= 11,72, Grid size= 128^3.

The parallel performance of the incompressible flow solver was also evaluated in terms of speed-up 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 19 shows the speed-up and scaling curves of the 2D flow solver on the Cray T3D for three different problem sizes (an ideal speed-up curve is shown again for comparison). The speed-up performance improves as the problem size increases, as expected. For the 512 x512 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 **T3D** is about five times faster than the Paragon for the code compiled with the -02 switch, But on 256 processors, T3D runs only about 1.5-2.0 times faster than Paragon depending on problem sizes, because the speed-up performance of the flow solver on Paragon is better than on **T3D.** Figure 19 also shows the scaling performance of the parallel flow solver on T3D for three local problem sizes. Again, we see the scaling improves as the size of local grid increases. In measuring the scaling of the flow solver, we used smaller local problem sizes than we did for the **multigrid** elliptic solver (see Figure 10 and 11). We expect the flow solver to have better scaling than the **multigrid** elliptic solver because the flow solver, even though calling the elliptic solver several times at each time step, does substantially additional processing on the finest grid. Indeed, this scaling difference between the two solvers can be verified by looking at the scaling curves for the 64x 64 local grid for the flow solver in Figure 19 and for the **multigrid full** V-cycle (which is used in the flow solver) in Figure 10 and 11. Similar performance trends were observed on Intel Paragon. In view of the scaling performance in Figure 19, we would claim that our parallel 2D flow solver scales well on large numbers of processors as long as the local grid size is not smaller than 64 x 64.

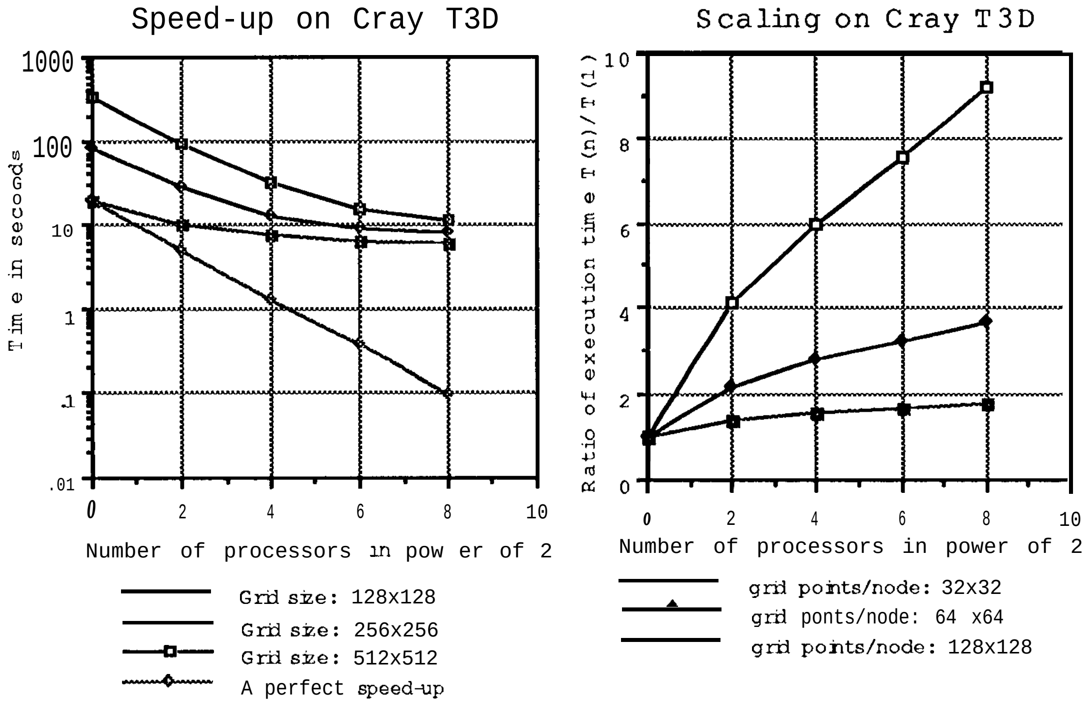


Figure 19: Speed-up and scaling of the 2D parallel Navier-Stokes solver on Cray T3D.

Scaling performance of the 3D flow solver was measured on Cray **T3D** and **T3E** systems, using one time step in the cubic driven-cavity flow. The results are shown in Tables 3-5. The last row in the tables shows the ratio of CPU time on *n* processors over one processor. Compared to the 2D case, the 3D flow solver scales much better on T3D to a maximum of 512 processors forming an 8 x 8 x 8 array with a 256 x 256x 256 global grid, The superior 3D sealing over the 2D scaling for the flow solver is clearly a result of a much higher volume to surface ratio for the 3D grid than for the 2D grid. Since the costs of computation and communication of the flow solver are proportional to the volume and surface of the finest grid, respectively, the relative cost of doing communication becomes much smaller in the 3D case, and thus the 3D scaling is greatly improved. Table 5 shows the flow solver scales slightly better on Cray T3E. The CPU times in Tables 4 and 5 show that the flow solver runs about three times faster on Cray T3E than on T3D, which is in the range of the performance difference of the two systems.

Table 3: Scaling of 3D flow solver on Cray T3D

Processor array	1X1X1	2 x 2 x 2	4 x 4 x 4	8 x 8 x 8
Grid size	16^3	32^3	64^3	128^3
CPU time (sec.)	6.82	19.4	22.9	24.0
T(n)/(T(1)	1.0	2.84	3.36	3.52

Table 4: Scaling of 3D flow **solver on Cray T3D**

Processor array	1X1X1	2 x 2 x 2	4 x 4 x 4	8 x 8 x 8
Grid size	32^3	64^3	128^3	256^3
CPU time (sec.)	57.1	70.8	75.8	80.0
T(n)/(T(1)	1.0	1.24	1.33	1.40

Table 5: Scaling of 3D flow solver on Cray T3E

Processor array	1X1X1	2 x 2 x 2	4 x 4 x 4
Grid size	32^3	64^3	128^3
CPU time (sec.)	19.4	22.9	24.0
T(n)/(T(1)	1.0	1.18	1.24

5. Conclusions

In this paper we presented elliptic multigrid schemes and a second-order **projection** method for solving Navier-Stokes equations for two and three dimensional incompressible flow problems. 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 is also discussed. We designed and implemented these solvers in a highly modular fashion so that they can be used either as stand-alone solvers or as extensible code modules for different applications. Several message-passing protocols (MPI, PVM and Intel NX) have been coded into the solvers so that the codes 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 expected orders of the 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 the Intel Paragon and the Cray T3D/T3E 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, message-passing systems. For future work, we plan to generalize this parallel flow solver package to thermally-driven flows and variable density flow problems.

Acknowledgments:

The authors wish to thank Dr. Stefan Vandewalle (California Institute of Technology) and Dr. Steve McCormick (University of Colorado) for some helpful discussions on multigrid methods.The authors also would like to thank Dr. Allan Stagg of Cray Research for the timing work on a Cray T3E. 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 the Intel Paragon parallel computers operated by JPL and by the Concurrent Supercomputing Consortium at Caltech, and on the Cray T3D parallel computers operated by JPL and by NASA Goddard Space Flight Center.

References:

1. C. Anderson, “Derivation and Solution of the Discrete Pressure Equations for the Incompressible Navier-Stokes Equations,” Lawrence Berkeley Laboratory Report, LBL-26353, 1988, Berkeley, CA (unpublished)

2. J. B. Bell, P. Colella and H. Glaz, “A Second-Order Projection Method for the Incompressible Navier-Stokes Equations,” J. Comp. Phys., 85:257-283, 1989

3. J. B. Bell, P. Colella and L. H. Howell, “An Efficient Second-Order Projection Method for Viscous Incompressible Flow.” Proceedings, 10th AIAA Computational Fluid Dynamics Conference, Honolulu, HI, pp.360-367, 1991

4. J. B. Bell and D. L. Marcus, “A Second-Order Projection Method for Variable-Density Flows.” J. Comp. Phys., Vol 101, No 2, pp. 334-348, 1992

5. W. Briggs, “A Multigrid Tutorial” SIAM, Philadelphia, 1987