Unsteady fluid flow calculations using a machine independent parallel programming environment [1]

A. Gursoy[a], L.V. Kale[a], and S.P. Vanka[b]

[a]Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana IL 61801, USA

[b]Department of Mechanical and Industrial Engineering, University of Illinois at Urbana-Champaign, Urbana IL 61801, USA

Abstract

A portable parallel implementation of a fractional step method to solve 3D unsteady incompressible Navier-Stokes equations on MIMD machines is described. The context currently considered involves uniform grid spacing in one of the directions. A parallel decomposition strategy and its implementation on a portable parallel programming system (Charm) is described. Some optimizations including one that reduces the cost of parallel convergence tests are described. Performance results for calculation of flow in a driven cavity on different parallel computers are presented.

1. INTRODUCTION

The performance of MIMD multiprocessor systems has increased significantly in recent years. Many hope that massively parallel systems will be used routinely to solve difficult, computationally intensive scientific applications. However, programming and portability issues still pose a problem which could limit the expected usage of these machines.

Computational Fluid Dynamics (CFD) is one of the most computationally demanding application areas. Until recently, only the supercomputers such as a CRAY have been providing the required computing power. Powerful vectorizing compilers facilitated the task of programming on these machines. More recently, these supercomputers evolved into multiple vector processors systems delivering four to eight times more performance than their uniprocessor versions. With the availability of low-cost, high-performance nonshared memory parallel machines including intel iPSC and NCUBE hypercubes, there has been growing interest in the scientific community to use these machines [1]. Programming on these machines is obviously more difficult than sequential programming. It is necessary to simplify and support the task of writing parallel applications, and also to ensure that the investment in parallel software is protected through architectural advances and new generation of parallel machines.

[1]This work was supported in part by the NSF grant CCR-90-07195 and NASA-NAG 3-1208

In this paper, we will discuss a machine independent parallel implementation of 3D unsteady incompressible flows on MIMD machines. Machine independence is achieved by using Charm parallel programming environment [2]. Charm is a runtime support system which allows machine independent parallel programming across MIMD machines. It provides an explicit parallel language which uses C as its base language. It hides the details of underlying machine architecture, communication and process management from the user. It has already been implemented on various shared and nonshared memory machines including Sequent Symmetry, Alliant FX/8, intel's iPSC/2 and iPSC/860, NCUBE/2, and is currently being implemented on a network of Sun workstations.

In section 2 we will discuss briefly the relevant Charm language features. The CFD problem and its parallel implementation will be described in sections 3 and 4, followed by the performance results.

2. CHARM LANGUAGE

A Charm program consists of chare definitions, message definitions, and declarations of specifically shared objects in addition to regular C language constructs (except global variables). A chare is a medium grained process which can dynamically create other chares, send messages to other chares, and share information through specifically shared objects. The Charm system takes the responsibility of scheduling chares, dynamic load balancing, resource management, and efficient machine specific implementation of specifically shared objects on different architectures.

A chare definition consists of local variable declarations, entry-point definitions and private function definitions as illustrated in Figure 1. Local variables of a chare are shared among the chare's entry-points and private functions. Private functions are not visible to other chares, and can be called only inside the owner chare. However, C functions that are declared outside of chares are visible to any chare. Entry-point definitions start with an entry name, a message name, followed by a block of C statements and Charm system calls. Some of the important Charm system calls are:

CreateChare(chareName, entryPoint, message)

This call is used to create an instance of a chare named as *chareName*. As all other Charm system calls, CreateChare is a non-blocking call, that is, it immediately returns. Eventually as the system creates an instance of chare *chareName*, it starts to execute the *entryPoint* with the message *message*.

SendMsg(chareID, entryPoint, message)

This call deposits *message* to be sent to the *entryPoint* of chare instance *chareID*. *chareID* represents an instance of a chare. It is obtained by a system call *MyChareID()*, and it may be passed to other chares in messages.

The runtime system is message driven. It repeatedly selects one of the available messages from a pool of messages in accordance with a user selected queueing strategy, restores the context of the chare to which it is directed, and initiates the execution of the code at the entry point.

A BranchOffice chare (BOC) is a form of chare that is replicated on all processors. An instance of BOC has a branch on every processor. All the branches have the same ID. A BOC definition is similar to a chare definition except it contains public functions which can be called by other chares. BOC's are useful for some computations such as

```
chare chare-name {
        local variable declarations
        entry EP1 : (message MSGTYPE *msgptr) {C code block}
        ..
        entry EPn : (message MSGTYPE *msgptr) {C code-block}
        private function-1() {C code block}
        ..
        private function-m() {C code block }
}
```

Figure 1: Chare Definition

reduction operations (i.e., collecting some information locally on each processor, and then combining it across processors), as well as for expressing static load balancing.

In addition to messages and BOC's, Charm provides some other ways of information sharing:

readonly A readonly variable is initialized at the beginning of a Charm program, and its value can be accessed by *ReadValue* call from any part of the program.

write-once A write-once variable is created and initialized at any point of the execution (only once). The system provides a global ID for the write-once variable, and this ID is used to access its value on any processor.

dynamic table A dynamic table is a set of entries with key and data fields. A number of asynchronous access and update calls are allowed on table entries.

The Charm system provides other information sharing mechanisms such as monotonic and accumulator variables. Details about these features can be found in [6]. It also provides a sophisticated module system that facilitates reuse, and large-scale programming for parallel software.

3. PROBLEM

3.1. Governing Equations and Numerical Method

We consider three dimensional, unsteady incompressible flows governed by the following equations:

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x}(uu) + \frac{\partial}{\partial y}(uv) + \frac{\partial}{\partial z}(uw) = -\frac{\partial p}{\partial x} + \frac{1}{Re}\nabla^2 u \tag{1}$$

$$\frac{\partial v}{\partial t} + \frac{\partial}{\partial x}(uv) + \frac{\partial}{\partial y}(vv) + \frac{\partial}{\partial z}(vw) = -\frac{\partial p}{\partial y} + \frac{1}{Re}\nabla^2 v \tag{2}$$

$$\frac{\partial w}{\partial t} + \frac{\partial}{\partial x}(uw) + \frac{\partial}{\partial y}(vw) + \frac{\partial}{\partial z}(ww) = -\frac{\partial p}{\partial z} + \frac{1}{Re}\nabla^2 w \tag{3}$$

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0 \tag{4}$$

where u, v, and w are non-dimensional velocities, p is non-dimensional pressure, and Re is the Reynolds number.

These equations are discretized on a staggered grid using the central difference scheme of Harlow-Welch [3]. A fractional step method [4] is used for the time integration. The resulting discretized equations are:

$$\frac{\hat{u} - u^n}{\Delta t} = \frac{3}{2}H_u^n - \frac{1}{2}H_u^{n-1} \tag{5}$$

$$\frac{\hat{v} - v^n}{\Delta t} = \frac{3}{2}H_v^n - \frac{1}{2}H_v^{n-1} \tag{6}$$

$$\frac{\hat{w} - w^n}{\Delta t} = \frac{3}{2}H_w^n - \frac{1}{2}H_w^{n-1} \tag{7}$$

$$\nabla^2 p = -\frac{1}{\Delta t}\left(\frac{\partial \hat{u}}{\partial x} + \frac{\partial \hat{v}}{\partial y} + \frac{\partial \hat{w}}{\partial z}\right) \tag{8}$$

$$u^{n+1} = \hat{u} - \Delta t \frac{\partial p}{\partial x} \tag{9}$$

$$v^{n+1} = \hat{v} - \Delta t \frac{\partial p}{\partial y} \tag{10}$$

$$w^{n+1} = \hat{w} - \Delta t \frac{\partial p}{\partial z} \tag{11}$$

where

$$H_u^n = -\left(\frac{\partial}{\partial x}(uu) + \frac{\partial}{\partial y}(uv) + \frac{\partial}{\partial z}(uw)\right) + \frac{1}{Re}\nabla^2 u \tag{12}$$

$$H_v^n = -\left(\frac{\partial}{\partial x}(uv) + \frac{\partial}{\partial y}(vv) + \frac{\partial}{\partial z}(vw)\right) + \frac{1}{Re}\nabla^2 v \tag{13}$$

$$H_w^n = -\left(\frac{\partial}{\partial x}(uw) + \frac{\partial}{\partial y}(vw) + \frac{\partial}{\partial z}(ww)\right) + \frac{1}{Re}\nabla^2 w \tag{14}$$

There are two distinct patterns of computation in these equations: the first one deals with the evaluation of the intermediate velocity fields and the second one is the calculation of the pressure field. At time-step n, intermediate velocities $\hat{u}$, $\hat{v}$, and $\hat{w}$ are calculated by using velocity values from previous time-steps. Next, a Poisson equation is solved for the pressure field, p. For the current problem, we assumed that the grid is uniform along the z direction. Therefore, the Poisson equation is solved by applying FFT in that direction. The resulting penta-diagonal equations are solved with Stone's method [5] (a strongly implicit iterative method). Finally, the intermediate velocity fields are corrected using the pressure values. Algorithm-1 shows the basic flow of this computation.

Algorithm-1
Basic Flow of Sequential Algorithm

---

1. Initialization
2. Time-stepping loop
   (a) compute intermediate velocities
   (b) calculate the right-hand side of the pressure equation
   (c) FFT along the z direction
   (d) solve for pressure in all xy-planes with Stone's method
   (e) inverse FFT along z
   (f) correct velocities

Figure 2: Decomposition of the computational domain

4. PARALLEL IMPLEMENTATION

In this section, we will discuss parallelization of the above algorithm. A natural way to implement parallel solution of partial differential equations is to divide the computational domain into several subdomains, and distribute them to the processors. In most finite difference applications, a static and rectangular domain partitioning is implied by the inter-grid dependencies (e.g., the dependencies in a five-point stencil etc). A decomposition scheme must address some computational issues such as load balancing and minimization of communication cost. The optimum decomposition scheme depends on machine and problem characteristics such as the number of processors, inter-grid dependencies, numerical solution techniques, etc.

We chose to partition the computational domain into $n \times m$ rectangular boxes which extend along the z direction, where $n \times m$ is equal to the number of processors. In Figure 2-a, this 2D partitioning scheme is illustrated for the four processors case. The grid points along the z direction belong to one processor which makes this partitioning scheme more suitable than other partitioning schemes as explained in following sections.

Parallel implementation of Algorithm-1 requires communication at some points. The parallel version, Algorithm-2, shows the points where communication is necessary. The sequential one contains two major phases as described previously: momentum equations and the pressure equation. Boundary values for intermediate and modified velocity variables are exchanged after their calculation (Algorithm-2, steps b and h). Since the grid points along the z direction are local to a processor, no communication takes place in that direction.

The solution of the pressure equation, steps d through f in Algorithm-2, involves FFT and solution of penta-diagonal linear systems. One of the advantages of 2D partitioning over 3D appears in the FFT phase. Each FFT along the z direction is a local operation.

Algorithm 2
Basic Flow of Parallel Algorithm

---

1. Initialization
2. Time-stepping loop
   (a) compute intermediate velocities
   (b) exchange intermediate velocity boundary values
   (c) calculate the right-hand side of pressure equation
   (d) FFT along z direction
   (e) Pressure loop
      i. solve local subdomain (all xy-planes)
      ii. exchange boundary values for pressure
      iii. test global convergence
      iv. if all processors converged then exit Pressure loop
   (f) inverse FFT along z
   (g) correct velocities
   (h) exchange boundary values of corrected velocities

Figure 3: Dynamic messages

Each processor applies FFT to its own data which results in full parallelism over the computation domain. FFT removes the inter-grid dependencies along the z direction which results in n_z independent xy-planes to be solved (each of which is a penta-diagonal linear system), where n_z is the number of grid points in the z direction, as illustrated in Figure 2-b. Due to the 2D partitioning, an xy-plane is spread across processors. These xy-planes are solved iteratively with Jacobi-like relaxation across processors. The local part of an xy-plane is solved with Stone's method. Each processor first solves for local points and exchanges information at boundaries. The global convergence is then tested. This process continues until the desired global convergence is reached. The xy-plane solving phase involves two kinds of communication patterns: nearest neighbor communication for pressure values, and a spanning tree communication for global convergence check. For a particular plane, if at least one of the processors reports non-convergence, then all the processors iterate one more level for that plane. Each processor calculates its local convergence. When all of its children processors report their convergence information, the combined information is propagated to the parent processor. Finally the root processor broadcasts the result to other processors.

There are n_z planes to be solved. Iterations last until the last plane converges. In order to reduce both computation and communication cost, the converged planes are dynamically eliminated from the computation. In the first iteration, all planes are in the computation list. Therefore boundary values for all xy-planes are exchanged. After every iteration, xy-planes that are globally converged are not solved locally, and boundary messages contain only boundary values from non-converged planes as shown in Figure 3. The load is also automatically balanced among processors with a 2D partitioning scheme, because each processor owns an equal sized piece of every xy-plane. After all planes have converged, inverse FFT is applied locally to get the pressure values. Then the velocities are corrected, and new values of velocities are exchanged to be used in next time step.

4.1. Initial Results

In this part, initial performance results for the driven cavity problem will be discussed. The problem domain involves a unit cube, and the flow is driven by moving the top wall. The performance results are gathered from runs with Reynolds number set to 100 for 50 time steps. Table 1 shows the performance results on a shared memory machine, Sequent Symmetry, for fixed domain size. Shared memory results are satisfactory. The speedup is below the linear speedup because as the number of processors increase, number of iterations to solve the pressure equation increases due to the relative weakness of Jacobi-like relaxation. The Mflop rate is almost doubled as the number of processors doubled.

Table 2 and Table 3 show the performance on two nonshared memory machines: iPSC/860, and NCUBE/2. The subdomain size is kept fixed in nonshared memory runs because the number of processors is varying over a wide range. As the physical size of the cavity is still fixed as a unit cube, this leads to a finer grid. The execution time increases as the number of processors increases despite the fact that the subdomain size is fixed. This is due to the slower rate of convergence of the Poisson solver. However, the Mflop rate is perhaps a better indicator for the effect of parallelization strategy. The Mflop rate is tripled as the number of processors is quadrupled as shown in the tables.

Table 1
Performance of non-adaptive scheme on shared memory machines

Sequence Symmetry				
#PE	Domain	Subdomain	time(sec)	Mflops
1	32x32x33	32x32x33	732	0.11
4	32x32x33	16x16x33	205	0.42
8	32x32x33	8x16x33	108	0.80
16	32x32x33	8x8x33	57	1.53

Table 2
Performance of non-adaptive scheme on nonshared memory machines

#PE	Domain	Subdomain	dt	time(msec)	Mflops
		NCUBE/2			
4	16x16x33	8x8x33	0.006	36058	3.05
16	32x32x33	8x8x33	0.005	47487	9.46
64	64x64x33	8x8x33	0.002	58106	31.21
256	128x128x33	8x8x33	0.0012	73342	96.98
		iPSC/860			
4	16x16x33	8x8x33	0.006	10611	10.35
16	32x32x33	8x8x33	0.005	16709	26.89
64	64x64x33	8x8x33	0.003	21481	84.44

Table 3
Performance of sequential algorithm on nonshared memory machines

Machine	Domain	dt	time(msec)	Mflops
NCUBE	16x16x33	0.005	105380	0.96
i860	16x16x33	0.005	25498	3.96

4.2. Improved algorithm

Some further improvements in the performance can be obtained by restructuring some communication strategies. Remember that there are two patterns of communication in the algorithm: nearest neighbor, and the tree communication for checking convergence. In hypercubes the cost of nearest neighbor communication is O(1), whereas the cost spanning tree reduction is $O(\log n)$ where n is the number of processors. Therefore treewise reduction might be causing the performance degradation significantly. We inspected execution times of different phases of the algorithm, and we observed that there is considerable idle time in the reduction phase. Note that, the convergence test is performed after each iteration. To reduce cost of this phase, the number of convergence tests should be reduced. A close inspection of the iteration counts versus timesteps, Figure 4-a, shows that the number of iterations for a xy-plane changes only gradually from one step to the next one. Therefore an adaptive convergence test scheme is applied for consecutive time steps, as described below.

Let n_i denote the number of iterations for xy-plane i recorded from the previous time

Figure 4: Iteration count, and Mflops

Table 4
Performance of adaptive scheme on nonshared memory machines

#PE	Domain	Subdomain	dt	time(msec)	Mflops
		NCUBE/2			
4	16x16x33	8x8x33	0.006	35993	3.04
16	32x32x33	8x8x33	0.005	44732	9.82
64	64x64x33	8x8x33	0.002	48564	36.18
256	128x128x33	8x8x33	0.0012	48594	144.02
		iPSC/860			
4	16x16x33	8x8x33	0.006	10436	10.48
16	32x32x33	8x8x33	0.005	14890	29.49
64	64x64x33	8x8x33	0.003	16676	105.37

step. Based on the correlation in Figure 4-a, we can predict the number of iterations in this time step to be close to n_i. To make sure that the algorithm adapts downwards (as well as upwards), that is even when the number of iterations required decreases in succeeding time steps, we choose $\max(n_i - 1, 1)$ as the predicted number of iterations for xy-plane i. The new algorithm performs $n_{max} - 1$ iterations without any convergence check, where n_{max} is the largest among n_i. At the end of j^{th} iteration of this phase, computation and communication is stopped for any xy-plane for which $\max(n_i - 1, 1)$ is equal to j. After $n_{max} - 1$ iterations, convergence tests for all xy-planes are performed in a single reduction operation. Iterations for xy-planes that are not converged yet are continued with the convergence test as in the base algorithm above.

In Table 4, performance results of the improved version on nonshared memory machines are listed. It is seen that this algorithm performs better than the non-adaptive case. For example, when the number of processors increases from 64 processors to 256 on NCUBE/2, the Mflop rate is tripled in the non-adaptive version whereas it is quadrupled in the adaptive version, which is optimal. The time increase from 4 to 16 to 64 processors is probably accounted for by the increase in the amount of neighbor communication. With 4 processors, each one communicates with 2 neighbors. With 16, four processors communicate with four neighbors, while others communicate with fewer. With 64, most of them have four neighbors. Comparing Table 2 and Table 4, we notice that the impact of the new scheme increases with the number of processors. This is as expected, because the time complexity of reduction increases with the number of processors. In Figure 4-b, performance difference between these two schemes is depicted.

4.3. Pipelining

A further improvement for this algorithm is the pipelined solution of the xy-planes. The algorithm described above exchanges the boundary values of all non-converged xy-planes. After the exchange is complete, it starts to solve those xy-planes. However, the time spent in computation and communication can be overlapped by solving xy-planes in a pipelined fashion (note that xy-planes can be solved independent of each other). The

Figure 5: Pipelined Execution

pipelined execution can be performed as follows: First, xy-planes are divided into groups. As soon as a group of planes is solved, the boundary values from this group is sent while simultaneously initiating computations for the next group of xy-planes (Figure 5-a). By the time of next iteration, boundary values of the first group would have arrived, and the processor can perform computations without waiting. The performance of this approach depends on the work distribution (i.e., the amount of computation necessary to solve xy-planes), and the choice of xy-planes in the concurrent groups. In our case, however, the empirical result shows (Figure 5-b) that the work load of the first few planes is significant, while the other planes converge rapidly. Therefore, for this particular problem, pipelined execution does not provide significant performance gain. The distribution of workload for this problem also implies that a 3D partitioning will suffer from load balancing.

5. CONCLUSION

In this paper, we have described the parallelization of a fractional step method for solving unsteady Navier-Stokes equations in the context of a 3D grid that is uniformly spaced in one dimension. A 2D decomposition scheme was used. In addition to rendering the FFT a completely local operation, this decomposition also induces a uniform load distribution. Although a 3D decomposition may reduce the communication cost somewhat, we believe that the cost of load imbalance and the parallelization of the FFT far outweigh the reductions in the communication cost.

It was found that parallel convergence tests during the Poisson solver constitute a significant portion of the elapsed time. An adaptive technique for reducing the number of these expensive tests was developed and was shown to improve the performance significantly for systems with a large number of processors.

The Charm parallel programming system facilitated development of this code by providing portability, so that we could focus on the algorithm expression without getting involved in machine specific details. The program once developed ran on different parallel machines without change. The full flexibility of this system was not used in the current implementation. Because the system is message driven, it provides many opportunities for overlapping computation and communication. This can be exploited by overlapping convergence tests and iterative computations,for example. Other algorithmic improvements in future may include replacing the block Jacobi with a red-black Gauss-seidel iterative scheme.

6. REFERENCES

[1] M.E.Braaten, "Application of parallel computing in computational fluid dynamics: A review", GE Corporate R&D report 89CRD121, July 1989.

[2] W.Fenton, B.Ramkumar, V.A.Salatore, A.B.Sinha, L.V.Kale, "Supporting machine independent programming on diverse parallel architectures", *Proceedings of the International Conference on Parallel Processing*, Vol. II, Aug 1991, pp.193-201.

[3] F.H.Harlow, J.E.Welch, "Numerical calculation of time dependent viscous incompressible flow of fluid with free surface", *Phys. of Fluids*, Vol. 8, No. 112, 1965, pp.2182-2189.

[4] J.Kim, P.Moin, "Application of a fractional step method to incompressible Navier-Stokes equations", *J.Comp.Phys*, Vol. 59, 1985, pp.308-323.

[5] H.L.Stone, "Iterative solution of implicit approximations of multidimensional partial differential equations", *SIAM J.Numer.Anal*, Vol. 5, No. 3, September 1968, pp.530-558.

[6] The CHARM(3.0) programming language manual, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, 1992.