FINAL REPORT

submitted to

NASA LEWIS RESEARCH CENTER

21000 Brookpark Road
Cleveland, Ohio 44135

for

**Parallel Computation of Unsteady Flows
on a Network of Workstations**
(NAG3-1577)

for the period of 1993-1997

INTRODUCTION

Parallel computation of unsteady flows requires significant computational resources. The utilization of a network of workstations seems an efficient solution to the problem where large problems can be treated at a reasonable cost. This approach requires the solution of several problems:

.the partitioning and distribution of the problem over a network of workstation,

.efficient communication tools,

.managing the system efficiently for a given problem.

Of course, there is the question of the efficiency of any given numerical algorithm to such a computing system.

NPARC code was chosen as a sample for the application. For the explicit version of the NPARC code both two- and three-dimensional problems were studied. Again both steady and unsteady problems were investigated. The issues studied as a part of the research program were:

*how to distribute the data between the workstations,

*how to compute and how to communicate at each node efficiently,

*how to balance the load distribution.

In the following, a summary of these activities is presented. Details of the work have been presented and published as referenced.

SUMMARY OF THE WORK PERFORMED:

A. Parallelization of the NPARC code:

PARC2D code was initially supplied by NASA Lewis Research Center for this study. This code was parallelized, by using GPAR, on the LACE cluster at Lewis. This results of this study was presented
in reference 1. A variable time-stepping algorithm was proposed This algorithm was first tested for steady flows.
Later, a version of NPARC code was parallelized for both two and three-dimensions. Variable time stepping was further implemented. These studies were reported in references 2 and 3.

B. Load Balancing

A dynamic load balancing procedure was developed for supporting an heterogenous cluster of work stations. NPARC code was used to test this capability for both steady and unsteady computations. Variable time-stepping was incorporated to the load balancing algorithm, such that each block and interface can choose their own time step as shown in figure 1. Figure 2 shows the overall computational procedure.
Also, research was conducted for determining the communication cost for a workstation cluster connected with Ethernet. Results of this research are summarized in references 4,5 and 6.

C. Filtering Techniques

To improve the efficiency of parallel algorithms, filtering techniques were developed. By using these techniques the communication and computation cost of the given parallel algorithm can be reduced significantly. The description of the methodology and obtained results are summarized in references 7, 8, 9, and 10.

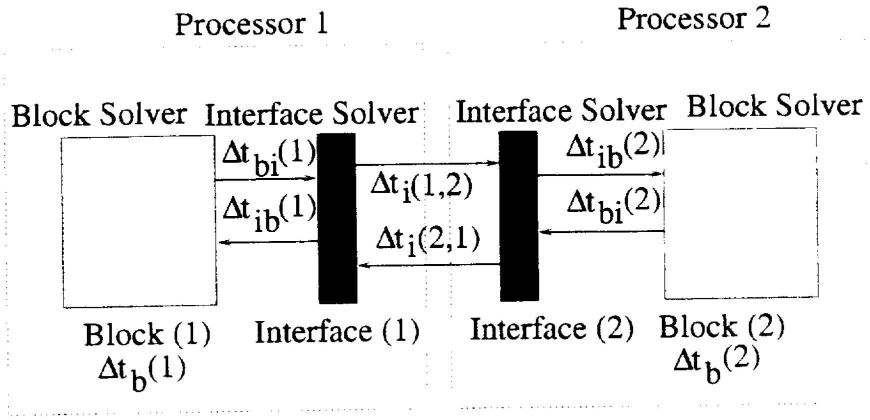


Figure 1. Blocks and Interfaces

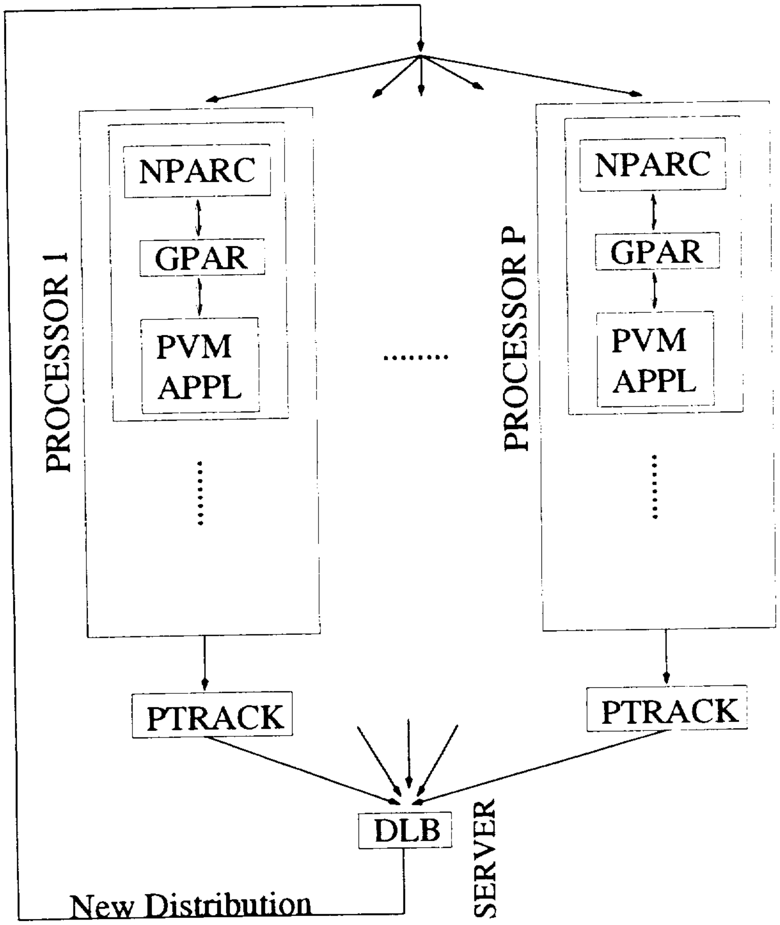


Figure 2. Parallel Execution

List of References

1. Akay, H.U., and Ecer, A., "Efficiency Considerations for Explicit CFD Solvers on Parallel Computers," *Proceedings of the International Workshop on Solution Techniques for Large-Scale CFD Problems*, Montreal, pp. 289-314, 1994.

2. Gopalaswamy, N., Chien, Y.P., Ecer, A., Akay, H.U., Blech, R.A, and Cole, G.L., "An Investigation of Load Balancing Strategies for CFD Applications on Parallel Computers," *Parallel CFD '95*, pp. 703-710, Pasadena, 1995.

3. Gopalaswamy, N., Akay, H.U., Ecer, A., and Chien, Y.P., "Parallelization and Dynamic Load Balancing of NPARC Codes," AIAA Paper No. 96-3302, Lake Buena Vista, FL, July 1-3, 1996.

4. Ecer, A., Akay, H.U., Chien, Y.P., and Gopalaswamy, N., "Load Balancing Issues in Parallel Computing," *Sixth International Symposium on Computational Fluid Dynamics*, Lake Tahoe, Nevada, September 4-8, 1995.

5. Secer, S., "Genetic Algorithms and Communication Cost Function for Parallel CFD Problems," M.S. Thesis, Department of Electrical Engineering, May 1997.

6. Chien, Y.P., Ecer, A., Akay, H.U., Carpenter, F., and Blech, R.A., "Dynamic Load Balancing on a Network of Workstations for Solving Computational Fluid Dynamics Problems," *Computer Methods in Applied Mechanics and Engineering*, vol. 199, 1994, pp. 17-33.

7. Gopalaswamy, N., Ecer, A., Akay, H. U., and Chien, Y.P., "Efficient Parallel Communication Schemes for Explicit Solvers of NPARC Codes," AIAA Paper No. 97-0027, Reno, January 1997.

8. Ecer, A., Akay H.U., and Gopalaswamy, N., "Filtering Techniques in Parallel Computing," *Computational Sciences for the 21st Century*, Edited by J. Periaux et al., Tours, France, May 5-7, 1997.

9. Ecer, A., Gopalaswamy, N., Akay, H.U., and Chien, Y.P., "Digital Filtering Techniques for Parallel Computation of Explicit Schemes," AIAA Paper No. 98-0616, 36th Aerospace Sciences Meeting, Reno, Nevada, January 12-15, 1998.

10. Gopalaswamy, N., "Parallel Computation of the Euler Equations," Ph.D. Thesis, Department of Mechanical Engineering, May 1998.

APPENDIX: Copies of selected references.

Solution Techniques for Large-scale CFD Problems

Edited by

Wagdi G. Habashi
Professor, Concordia University
Director-Industry, CERCA
Aerodynamics Consultant, Pratt & Whitney, Montreal, Canada

JOHN WILEY & SONS
Chichester · New York · Brisbane · Toronto · Singapore

EFFICIENCY CONSIDERATIONS FOR EXPLICIT CFD SOLVERS ON PARALLEL COMPUTERS

H.U. AKAY and A. ECER

A parallel algorithm, based on subdividing the flow field into several subdomains and distributing each subdomain onto available computers, is presented for the solution of Euler equations on workstation clusters. Each block is treated as a different process in available computers on the network and the load distribution is dynamically balanced. Machine independence is achieved by combining the flow code with a general CFD data base and a machine portable library. Strategies are explored for integrating the unsteady flow equations explicitly in time by taking advantage of the local flow conditions and the grid point distribution in each block.

1. INTRODUCTION

Solution of large-scale CFD problems requires, and will continue to require, computer resources beyond those readily available. Memory and CPU requirements are still the key factors affecting the progress in this area. Whether it is an implicit or explicit scheme, efficiency still remains a major problem. Recently, considerable effort has been directed towards modifying algorithms for efficiency and significant progress has been made in vectorizing and parallelizing these algorithms.

Our earlier work on parallel computations of CFD has led to the development of a CFD data base program, GPAR[1], which manages computational grids. GPAR utilizes a machine portable library, APPL[2], for implementations on different distributed memory systems. Using the GPAR program, together with APPL, we were able to parallelize a number of flow codes[3,4]. In addition to using machines with specific parallel architectures, we have explored the use of clusters of workstations for parallel computations. For cases where the number of solution blocks are greater than the number of workstations, multi-processing is exercised in machines containing multiple blocks. For such cases, we have also incorporated load balancing algorithms[5,6].

The following factors affect efficiency in parallel computing:

- Ease in programming
- Ease in portability and scalability
- Ability to use heterogeneous systems
- Ease in load balancing
- Speed-up and scalability

In this chapter, we present an explicit solution strategy for the solution of CFD problems, which can readily be used on a network of heterogeneous workstations. We discuss the issues related to the implementation of this scheme.

2. EULER EQUATIONS

2.1 *Formulation*

A finite element discretization of the compressible Euler equations can be formulated by adding two diffusion operators as follows:

$$\frac{\partial U}{\partial t}+\frac{\partial F_i}{\partial x_i}-\frac{\partial}{\partial x_i}\left(\alpha A_i\frac{\partial F_j}{\partial x_j}\right)-\frac{\partial}{\partial x_i}\left(\varepsilon\frac{\partial U}{\partial x_i}\right)=0 \tag{1}$$

where U is the vector of conservation variables and F_i are flux vectors defined as

$$U=\begin{bmatrix}\rho\\ \rho u_1\\ \rho u_2\\ \rho u_3\\ \rho E\end{bmatrix} \qquad F_i=\begin{bmatrix}\rho u_i\\ \rho u_1u_i+p\delta_{1i}\\ \rho u_2u_i+p\delta_{2i}\\ \rho u_3u_i+p\delta_{3i}\\ (\rho E+p)u_i\end{bmatrix} \tag{2}$$

Also,

$$A_i=\frac{\partial F_i}{\partial U} \tag{3}$$

are the Jacobian matrices corresponding to flux vectors F_i, ρ is the density, u_i the velocity components, and E is the total specific energy. The static pressure p is calculated from the equation of state:

$$p=(\gamma-1)\rho\left(E-\frac{1}{2}u_iu_i\right) \tag{4}$$

where γ is the ratio of specific heats.

The third and fourth terms in Eq. 1 were introduced to stabilize the equations by adding artificial diffusion[7-10]. Here, α is the streamwise diffusion coefficient for upwinding of the flux vectors F_i and ε is an additional diffusion coefficient added in all directions, which can be computed from local flow conditions and mesh characteristics as follows:

$$\alpha=c_1\frac{\tau}{\bar{q}^2} \qquad \varepsilon=c_2\tau+c_3\frac{\tau^2}{\bar{q}^2}\left|\frac{u_i}{q}\frac{\partial q}{\partial x_i}\right| \tag{5}$$

where

$$\tau=\frac{1}{2}\left\{\sum_{\xi=1}^{3}\left(h_\xi\bar{u}_\xi\right)^2\right\}^{1/2} \qquad q=\left(u_iu_i\right)^{1/2} \qquad \bar{q}=q+a \tag{6}$$

In the above equations, h_ξ and $\bar{u}_\xi$ represent the element characteristic lengths and velocity components, respectively, in the direction of the natural coordinates ξ_i of an isoparametric finite element. $a=\sqrt{\gamma p/\rho}$ is the speed of sound, and c_i are user-specified constants used to control non-physical oscillations of the numerical scheme. For most subsonic and transonic flows, we use $c_1=1.0$, $c_2=0.15-0.5$ and $c_3=1.0-2.0$.

2.2 *Finite Element Discretization in Space*

Using a weighting function vector $W(x_i)$, the weighted residual form of Eq. 1 is expressed as follows

$$\int_{\Omega^e} W \cdot \left[\frac{\partial U}{\partial t} + \frac{\partial F_i}{\partial x_i} - \frac{\partial}{\partial x_i}\left(\alpha A_i \frac{\partial F_j}{\partial x_j} \right) - \frac{\partial}{\partial x_i}\left(\varepsilon \frac{\partial U}{\partial x_i} \right) \right] d\Omega = 0 \tag{7}$$

and by applying the Green-Gauss theorem on the last two terms of Eq. 7, a weak variational form is obtained

$$\int_{\Omega^e} \left[W\left(\frac{\partial U}{\partial t} + \frac{\partial F_j}{\partial x_j} \right) + \alpha \left(\frac{\partial W}{\partial x_i} \right) G_i + \varepsilon \left(\frac{\partial W}{\partial x_i} \right)\left(\frac{\partial U}{\partial x_i} \right) \right] \; d\Omega - \oint_{\Gamma^e} W \cdot H_i n_i \, d\Gamma = 0 \tag{8}$$

where

$$G_i = A_i \frac{\partial F_j}{\partial x_j} \qquad \text{and} \qquad H_i = \left(\alpha A_i \frac{\partial F_j}{\partial x_j} + \varepsilon \frac{\partial U}{\partial x_i} \right) \tag{9,10}$$

Ω^e is the element area, n_i are the directional cosines of the outward normal vector on the element boundaries Γ^e, and H_i are the boundary flux vectors resulting from diffusion operators. We introduce the following interpolations to each conservation variable ϕ, as follows

$$\phi(x_i, t) = N_j(x_i)\, \phi_j^e(t) \tag{11}$$

where N_j are the spatial element interpolation functions, ϕ_j^e are the nodal point values of the conservation variable ϕ in an element e. Equal-order linear interpolations are used for all variables. After using the same interpolation functions as weighting functions, we obtain the following decoupled system of ordinary differential equations for each conservation variable ϕ

$$M_{ij}^e \, \dot{\phi}_j^e = f_i^e \tag{12}$$

where

$$\dot{\phi}_j^e = \frac{d\phi_j^e}{dt} \tag{13}$$

$$M_{ij}^e = \int_{\Omega^e} N_i N_j \, d\Omega \tag{14}$$

$$f_i^e = - \int_{\Omega^e} \left\{ N_i \frac{\partial F_j}{\partial x_j} + \alpha \, G_k \frac{\partial N_i}{\partial x_k} + \varepsilon \frac{\partial N_i}{\partial x_k} \frac{\partial \phi}{\partial x_k} \right\} d\Omega + \oint_{\Gamma^e} N_i H_k \, n_k \, d\Gamma \tag{15}$$

F_j, G_k and H_k are calculated for each conservation variable ϕ using Eqs. 2, 9 and 10, respectively. The term G_k in Eq. 15 may also be replaced with $u_k \, \partial F_j / \partial x_j$, where u_k is the local flow velocity vector, eliminating the need to calculate the inner product involving the Jacobian matrix in Eq. 10. Although this provides more efficient results for transonic problems with subsonic inflows and moderately high Mach numbers, it does not appear to provide enough stability at high supersonic speeds[4].

2.3 Boundary Conditions

Inflow and outflow boundaries are treated differently using the *characteristic boundary method* based on whether the local flow conditions are subsonic or supersonic[11]. For subsonic inflow boundaries with known Mach number conditions, Riemann variables are used together with the values extrapolated from the interior elements closest to the boundary. Similarly, for subsonic outflows, the exit static pressure is specified together with the values extrapolated from the closest interior elements. For supersonic inflows, all values of the conservation variables are fixed. For supersonic outflows, values of conservation variables are extrapolated from the nearest interior elements. A zero normal mass flux boundary condition $\rho u_i n_i = 0$ is imposed on solid boundaries.

2.4 Time-Integration of the Equations

Assembly of the element equations leads to the following system of equations for each of the conservation variables

$$M_{ij} \dot{\phi}_j = f_i \tag{16}$$

Using forward-differencing in time, the time derivative of ϕ is expressed as

$$\dot{\phi}_j^n = \left(\phi_j^{n+1} - \phi_j^n\right) \Big/ \Delta t^n \tag{17}$$

where n denotes a time step and Δt^n is the time increment at time step n. Substituting the above into Eq. 16, we devise the following explicit scheme to calculate the solution at $n+1$:

$$\phi_i^{n+1} = \phi_i^n + \Delta t^n \left(\overline{M}_{ij}\right)^{-1} f_j^n \tag{18}$$

where $\overline{M}_{ij}$ is the global matrix assembled from a lumped matrix approximation of M_{ij}^e in Eq. 14. Due to the explicit nature of the scheme in Eq. 18, the element Courant number limitation

$$\Delta t < C \, min \left\{ 1 \Big/ \left[\left(\overline{u}_\xi + a \right) \Big/ h_\xi \right] \right\} \tag{19}$$

must hold in each element for stability of the numerical integration[12], where $\overline{u}_\xi$ is the local flow speed, a is the speed of sound, h_ξ is the element length in the local ξ direction, and C is a constant less than unity.

For steady-state problems, the residual norm of each conservation variable, $\phi = \left\{\rho, \rho u_i, \rho E\right\}$, is used for monitoring the convergence to steady-state and is calculated at each time step n as

$$\left(R_\phi^n\right)_{avg} = \sum_{i=1}^{nodes}\left[\left(\frac{\phi_i^{n+1}-\phi_i^n}{\Delta t^n}\right)^2\right]^{1/2} \tag{20}$$

Steady-state is considered reached when $\left(R_\phi^n\right)_{avg} < 10^{-6}$.

3. PARALLEL COMPUTING ENVIRONMENT AND EXPLICIT SOLVERS

Solution of the Euler equations by both implicit and explicit methods has been greatly studied for steady and unsteady flows. In this chapter, the emphasis is on the parallel implementation of explicit solvers. In comparing parallelization of explicit and implicit schemes, one can make the following observations:

- Explicit solvers are easier to parallelize since the data to be organized on a parallel computer is simpler.
- The ratio of communication versus computational cost is higher for explicit schemes, which reduces the efficiency of parallelization.

Also, for explicit solvers, scalability becomes more crucial due to the relative importance of the communication costs. In the following, specific parallelization issues are discussed as applied to the scheme described above.

For the numerical integration of a system in the form $d\phi/dt = f(\phi, t)$, parallelization requires distribution of the data over a number of processors. For an explicit solver, the calculation of $f(\phi, t)$ can be localized on a processor quite easily. In the present application, we assign elements to blocks, and blocks to processors. Parallel implementation of the algorithm involves the following steps:

- The computational grid is divided into a number of blocks equal to or greater than the number of available computers, with one layer of elements overlapped at inter-block boundaries as shown in Fig. 1.
- The block and interface information is incorporated into the data base, which is distributed to different machines.
- Eq. 18 is solved inside a *block solver* on each processor locally.
- The inter-block information is transferred to an *interface solver* which also manages the communication between neighboring blocks.
- The problem is load balanced dynamically for a given system of processors.

The flow chart of the parallel CFD algorithm is shown in Fig. 2. The first step is to define the grid in terms of blocks and interfaces. The assembly of grid points and their connectivities are defined for each block and interface. The data base program, GPAR, manages such block-based data and distributes it to the appropriate processors. Blocks are divided among the available processors and interfaces attached to each block are also stored on that processor. Since each interface is attached to two blocks, they appear in two processors and are prepared to communicate with each other. The same basic solution algorithm is used in all blocks which is defined as a *block solver*. The *interface solver* transfers data from each block to its interfaces, performs necessary computations on interface nodes and communicates the results back to the corresponding blocks. In return, the interfaces update the neighboring blocks as shown in Fig. 3. The interface solver performs all necessary communications with relatively small amount of computations. The data base management program GPAR supports block and interface solvers in terms of propagating the data as soon as they are computed by the block solvers. The block data are communicated to the interfaces which inform their duplicates on other processors. The interfaces then update the blocks. The user does not have to employ any machine-specific *send* or *receive* commands. Insteac the basic message passing commands of GPAR are issued in the interface solver for managing th interface data. Although APPL was used as the parallel message passing library for thi implementation, the use of other message passing libraries such as PVM[13] is also possible.

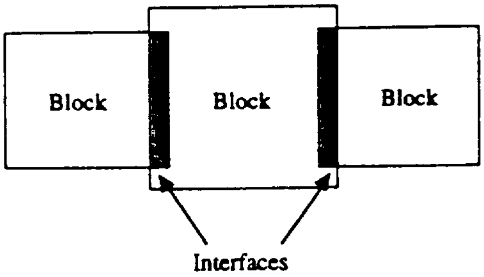


FIGURE 1: Blocks and interfaces.

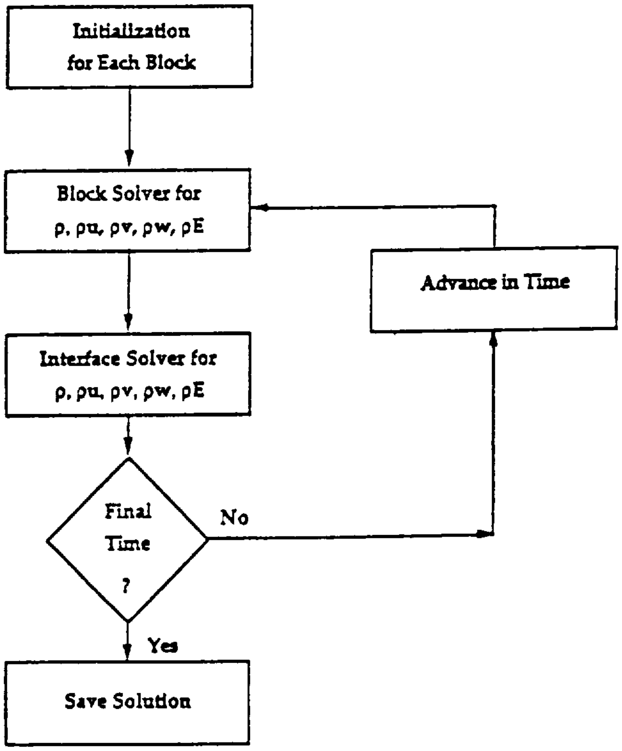

Initialization for Each Block
Block Solver for ρ, ρu, ρv, ρw, ρE
Advance in Time
Interface Solver for ρ, ρu, ρv, ρw, ρE
Final Time ?
No
Yes
Save Solution

FIGURE 2: Flow chart of the parallel CFD code.

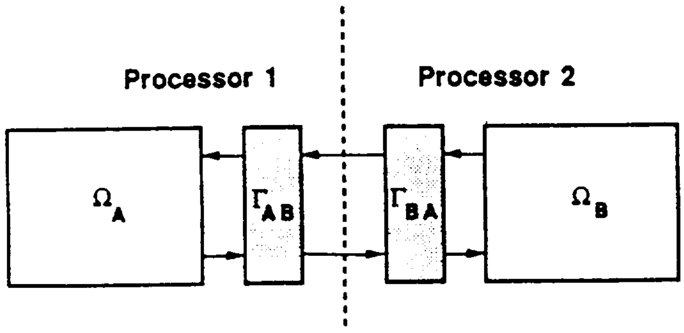


FIGURE 3: Communication between blocks and interfaces
(Ω_A *and* Ω_B are blocks, Γ_{AB} *and* Γ_{BA} are interfaces).

In the present application, a *block* is defined as an entity in the data base consisting an assembly of finite elements. After the unsteadiness $d\phi/dt$ is calculated locally in each block, one has to communicate between the processors to propagate these changes. At the end of the calculation of the $d\phi/dt$ vector at each node in each block, the interfaces are automatically updated by the *interface solver.*

Steps of an explicit parallel scheme can then be outlined as follows:

- **Initialize the data base.**
- ***Do for each time step:***
    - ***Do for each block:***
        - Integrate the equations for the nodes in that block.
    - ***Do for each interface:***
        - Gather information from the neighboring blocks to update the flow variables for the nodes on that interface and send this information to the neighboring blocks.

Although the above particular scheme seems straightforward, there are several decisions to be made. These are related to the problem to be solved and the computer system to be utilized. In terms of solving a specific flow problem let us consider the flow around an airfoil. When one generates a computational grid, certain regions are refined to account for the details of the flow field. A uniformly refined grid increases the computational cost exponentially, especially for three-dimensional problems. Different levels of grid refinement also suggest different choice of time steps for stability requirements of the explicit scheme. If we compute with an explicit scheme by using a constant time step for the entire grid, based on the stability requirements around the leading edge of the airfoil, the scheme becomes quite expensive. Also, when considering the flow field in different regions, one can observe that time step requirements can be different in terms of the accuracy of the solution. The above discussion suggests that running an explicit algorithm on an equally spaced grid with a constant time step is not a preferred solution. In a parallel computation of such flow problems, one should consider the refinement of block grids differently to represent the local flow conditions properly and perform the computations in a selective manner.

Another issue is the availability of hardware. In parallel computing, one can start with the assumption that blocks will utilize similar computer resources which are readily available. The correct problem should be stated as solving a given problem on a given computer system most efficiently. Furthermore, a portion of a specific computer system may be available to a specific user on a given day and the computer system may be upgraded periodically. One would like to use all available computer resources in a most efficient manner in such an hardware environment when running a parallel code, in contrast to running on a single processor.

The use of an efficient data base management system is also critical in utilizing the available computer resources. By using the above described system one can utilize any given computer system supported by the message passing library, including heterogeneous systems. However, one of the important features of parallel computers is the dynamic nature of available resources. One would like to run the algorithm on a given machine efficiently without requiring an excessive amount of computer time. For this purpose, a dynamic load balancing capability was developed[6]. Based on an initial distribution of the blocks on available processors, the cost data is gathered in terms of communication and computation for each block and interface. Better distribution of blocks on available processors is then determined as more data is collected during these computations. This algorithm can be periodically used to improve load balancing to account for changing loading conditions of the individual processors in a given computer system. The interaction of the CFD code with GPAR, APPL and the load balancing program is shown in Fig. 4.

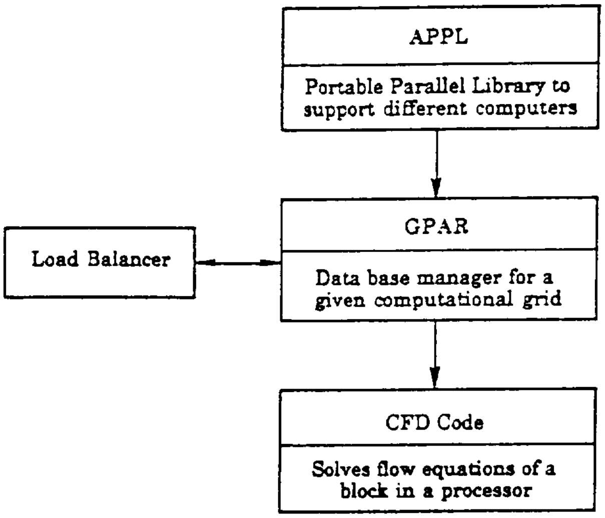

APPL
Portable Parallel Library to support different computers
GPAR
Data base manager for a given computational grid
Load Balancer
CFD Code
Solves flow equations of a block in a processor

FIGURE 4: Relationship between different parallel tools and a CFD application code.

4. EFFICIENCY CONSIDERATIONS FOR PARALLEL COMPUTATIONS

4.1 Load Balancing

As discussed above, dynamic load balancing capability is a critical factor in improving the efficiency of parallel computations. While running parallel jobs by subdividing the solution domain into several blocks on loosely-coupled systems such as networked workstations, one is faced with the following situations:

- The computational grid may be divided into many solution blocks with varying sizes,

- A process is assigned to each block and there may be more than one process on each processor,
- The available multi-user and multi-tasking networked processors may have different computational speeds and memory,
- The load of each processor may vary due to initialization or termination of processes of other users.

One way of achieving load balancing between the processors is to distribute an equal load, or number of solution blocks, to available processors. This may require some effort in subdividing the computational grid into a number of blocks several times greater than the number of processors, which can be done at the initiation of execution[5]. In the development of this strategy, it was originally assumed that the processes would be executed synchronously. During the numerical integration of the equations, each block (or process) was synchronized at each time step[3]. This results in certain restrictions to the load balancing. Here, we consider a solution strategy which exploits networked workstation environments in which each processor can handle multiple tasks and their loads may vary considerably. In addition, for multi-user environments considered here, computer loads can change dynamically since other users may start new processes anytime. Consequently, the effective computational speed of a computer may change dynamically during the duration of parallel computations which may increase the elapsed time of computations. This situation is improved by the dynamic load balancing algorithm[6]. By checking the status of processors during the time-integration of a problem, the loads are redistributed to available machines as unbalanced loading conditions are detected.

4.2 Variable Time-Stepping - Steady and Unsteady Flows

Parallelization of explicit schemes is a rather straightforward task when the domain is subdivided into several solution blocks. It has been demonstrated that for well balanced cases it is possible to reach efficiencies of 75% or more, with 20 or more machines[3]. However, due to the Courant number restriction on the time increment, the solution of large size problems is prohibitive, even after parallelization. One has to consider more adapted schemes rather than performing similar computations on all machines. Due to the Courant number restriction in Eq. 19, the time increment Δt is directly proportional to element size and inversely proportional to local speed. Hence, Δt becomes most restrictive in regions with high flow speeds and denser grid points. Shown in Fig. 5 is the scheme used in exchanging interface data between blocks with spatially constant time steps for all blocks. In this case, the interface data is sent to neighbors at each time step with all blocks having the same time increment. One remedy for steady flow problems is to use spatially variable time-stepping. On the other hand, drastic spatial variations in Δt delay the convergence. Hence, it is more appropriate to subdivide the domain into regions with different Δt in each. In such cases, we have found that one does not have to march in time with the same speed in all the blocks. Of course, for unsteady flows, a time accurate solution of interfaces is necessary. The block-based parallel approach adopted here makes the variation of Δt in time and space both for steady and unsteady problems quite convenient.

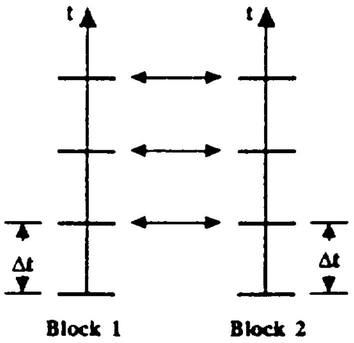


FIGURE 5: Exchange of interface data in spatially constant time-stepping scheme.

For a flow domain subdivided into solution blocks for parallelization as implemented here, we propose to select a local time step for each block as multiple of a minimum preset value Δt_{min}. For each nth time step and mth block we determine

$$\Delta t_m^n < C \; min\left\{ 1 \Big/ \left[\left(\bar{u}_\xi + a \right) \big/ h_\xi \right] \right\} \tag{21}$$

and choose the actual time-step for each block as

$$\Delta \bar{t}_m^n = k \; \Delta t_{min} \leq \Delta t_m^n \tag{22}$$

where k is an integer. An upper limit on k is needed (e.g., 5) to minimize extrapolation errors. This way, each block may advance with different time increments. For impulsively started flows. it is safer to initially advance all blocks with a constant time step until the solution starts developing. A similar technique was introduced by Löhner et al.[14] in conjunction with a domain splitting technique on serial machines.

In the case of unsteady flows, a block, advancing with a smaller time step than one of its neighbors, can calculate its own interface conditions by extrapolation from the previously received interface data. While blocks with smaller time steps are solved more frequently, the blocks with larger time steps will be solved less frequently, thus using the resources (available machines) less. In addition, there will be less interchange of interface data as shown in Fig. 6. Since the Courant number limitation is usually small enough for time accuracy too, there is no appreciable danger in loss of accuracy because of extrapolations.

For steady flows, there may not be any need for using the extrapolated interface data, since it is not necessary to maintain time accuracy between blocks. Hence, the time increments of blocks can be determined directly from Eq. 21 without having to use the constraint in Eq. 22. In this case, the latest available interface results from the neighboring blocks are employed as shown in Fig. 7.

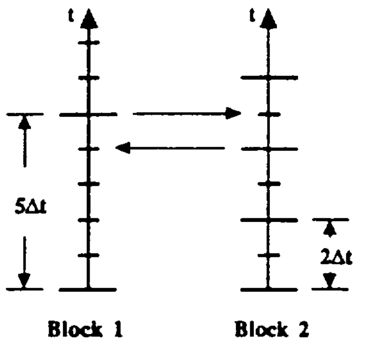


FIGURE 6: Exchange of data in spatially variable time-stepping scheme for unsteady flows.

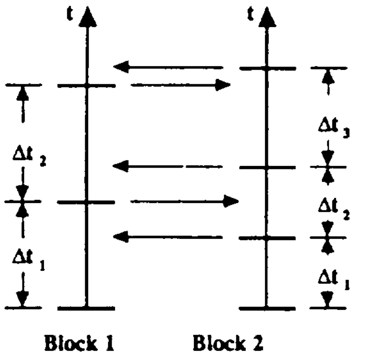


FIGURE 7: Exchange of data in spatially variable time-stepping scheme for steady flows.

4.3 Local Residual Criterion for Steady Flows

For steady flow problems, the residuals in each block often reach sufficiently small values at different times, indicating local convergence to a steady state. Hence, depending on the local flow conditions, the solution of a block may be stopped as soon as convergence is detected in that block. This way, additional efficiency can be obtained by minimizing the utilization of the available computer resources without sacrificing accuracy. It will be shown that for fully supersonic blocks, convergence of the blocks can be obtained sequentially. For subsonic and transonic blocks, after the convergence of the blocks is obtained, it is necessary to restart the solution in all the blocks to check the global convergence to a steady-state solution. Such savings may be substantial in large-scale problems with varying local flow and grid characteristics.

5. NUMERICAL RESULTS

In this section, we present numerical examples demonstrating applications of the algorithms discussed above. All computations were done on a network of IBM RS-6000/540 workstations.

5.1 Parallel Performance Example

To illustrate the parallel performance of the algorithm, transonic flow around a NACA0012 airfoil was considered[4]. A C-grid with 304 K grid points and 20 blocks was employed. The topology of solution blocks is shown in Fig. 8. The CPU and elapsed times of 5, 10 and 20 block subdivisions of this problem are summarized in Fig. 9. Elapsed to CPU time ratios of 1.4, 1.9 and 2.3 were measured for cases with 5, 10 and 20 machines. Differences between the elapsed and CPU times are attributed to communication loads and delays, and the presence of other processors or users on the machines. As may be observed, the difference between the CPU and elapsed time gets larger as the number of processors increases, indicating that larger block sizes yield more efficiency. These results were obtained by static distribution of the loads over computers with a single user and a constant time step was used in all blocks.

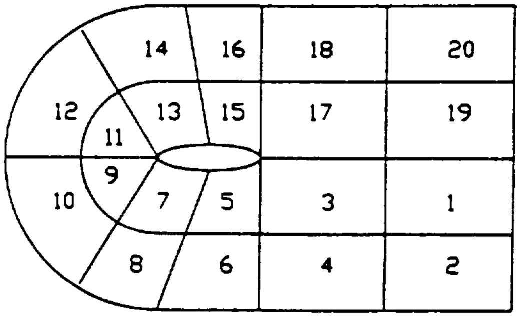


FIGURE 8: Block topology for parallel performance example.

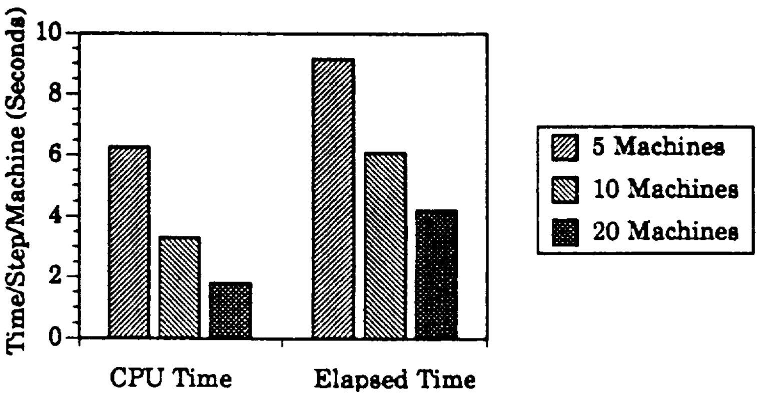


FIGURE 9: Performance of the algorithm for a grid of 304,000 points and 20 solution blocks.

5.2 *Load Balancing Example*

In this case, a NACA0012 airfoil with 65 K grid points was analyzed by distributing 30 blocks on 5 computers[6]. Initially six blocks were assigned to each computer. There were also other users on these processors. As it is shown in Fig. 10, the load balancer checks the status of computers every n number of steps. When it detects unbalances due to appearance of extraneous processes on the system, it redistributes the loads accordingly. As may be observed, the algorithm provides a better performance by periodically checking the loads on the computers and redistributing the loads. The dashed line in Fig. 10 indicates the elapsed time estimated by the balancer, while the solid line denotes the actual measured time. The estimate of the elapsed time is made from the size of blocks and interfaces of a given computational grid.

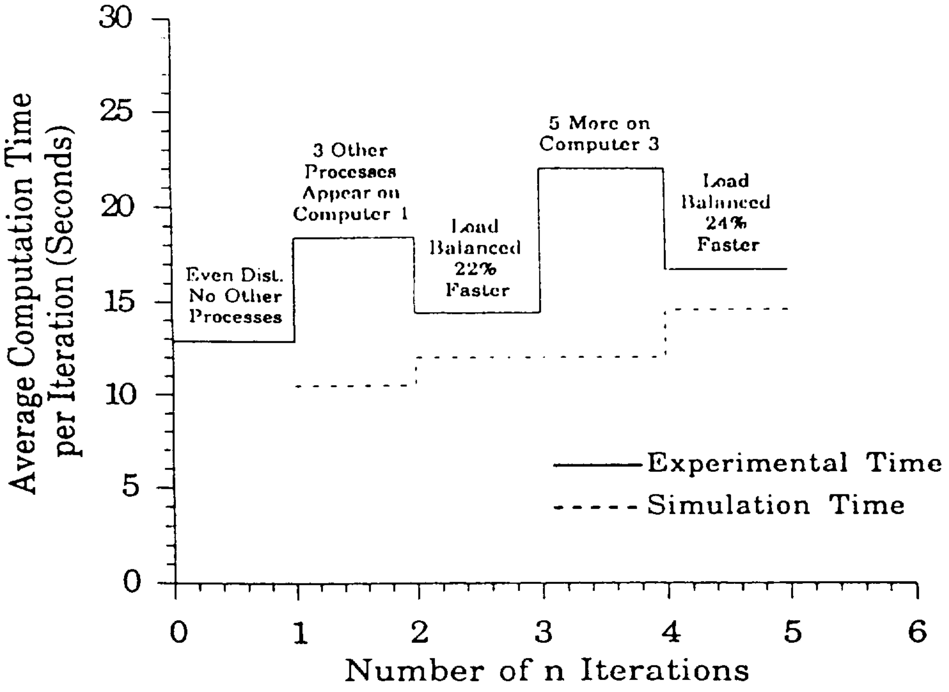
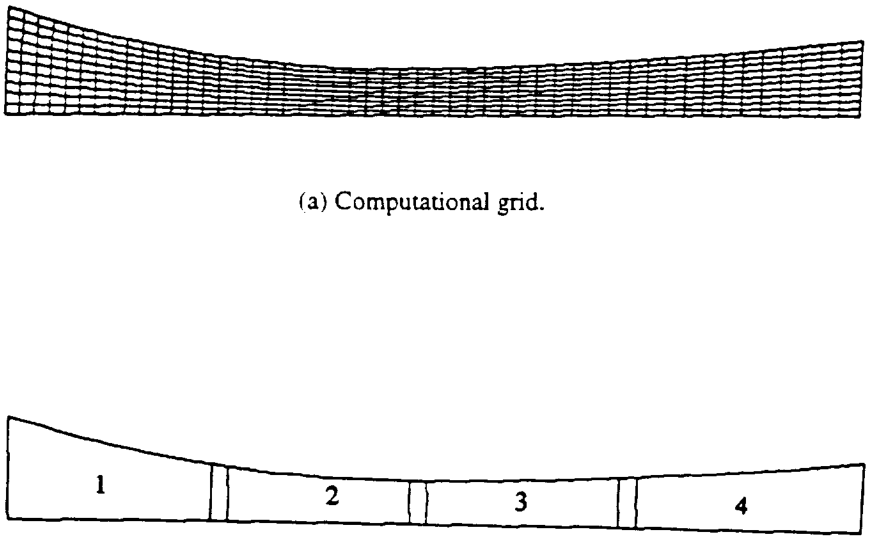


FIGURE 10: Timing results of the dynamic load balancing case.

5.3 *Example of Variable Time-Stepping for Steady Flows*

To illustrate applications of the block-based variable time-stepping algorithm discussed above, we have selected a converging-diverging channel geometry with a computational grid of 50X10 elements shown in Fig. 11a. The channel has an inlet to throat area ratio of 2.5, exit to throat area ratio of 1.5 and total length to throat height ratio of 20. For the purpose of demonstrating the ideas presented, a four-block subdivision of the channel was considered as shown in Fig. 11b. Since the steady-state was of interest, the equations in each block were integrated by using locally determined time steps from Eq. 21 in each block until the average residuals in each block reduced below 10^{-6}. Five distinct flow regimes were considered.



(a) Computational grid.

(b) Block and interface topology.

FIGURE 11: Computational grid and a four-block subdivision of the converging-diverging channel test case.

5.3.1 Subsonic flow For a uniform inlet Mach number of 0.24 at the inlet and an exit to inlet pressure ratio of 0.9136, the flow remains completely subsonic throughout the channel as may be seen in Fig. 12. The number of time steps required for each block to reach a steady-state solution is summarized in Fig. 13. As it can be observed, due to the subsonic nature of the flow conditions, all blocks reach the steady state at about the same number of steps. As it will be seen from the results of the other cases, this case required the highest number of steps for convergence.

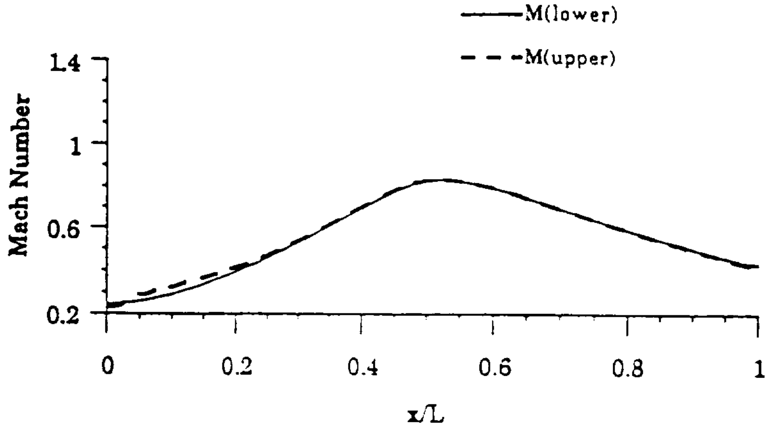


FIGURE 12: Computed Mach number distribution along lower and upper surfaces of the channel for the subsonic flow case.

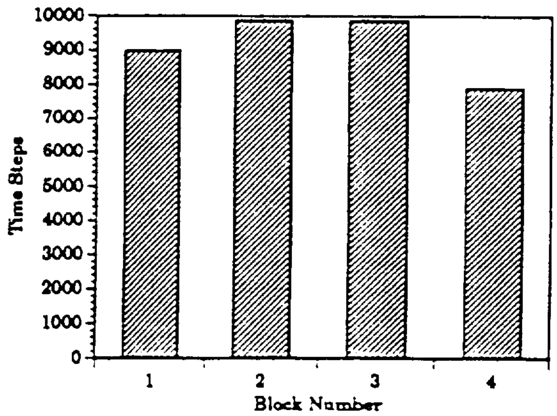


FIGURE 13: Convergence of each block to steady state for the subsonic flow case.

5.3.2 Transonic flow with subsonic inlet and exit To create a shock in the downstream of the channel, a uniform inlet Mach number of 0.24 and an exit to inlet pressure ratio of 0.8435 were applied as inflow and outflow boundary conditions, respectively. Shown in Fig. 14 is the variation of Mach number along the upper and lower surfaces of the channel. The number of time steps needed for each block to reach a steady-state solution is summarized in Fig 15. As may be seen, the first two blocks converged to the steady state earlier than the last two blocks. Compared to the single block solution of the problem, it was seen that about 15% less computation was needed. Fig. 16 illustrates the effect of using variable Δt at each grid point versus using a different Δt in each block. While the grid point based variations of Δt yield oscillatory residuals and slow the convergence rate, the block-based Δt variations perform better.

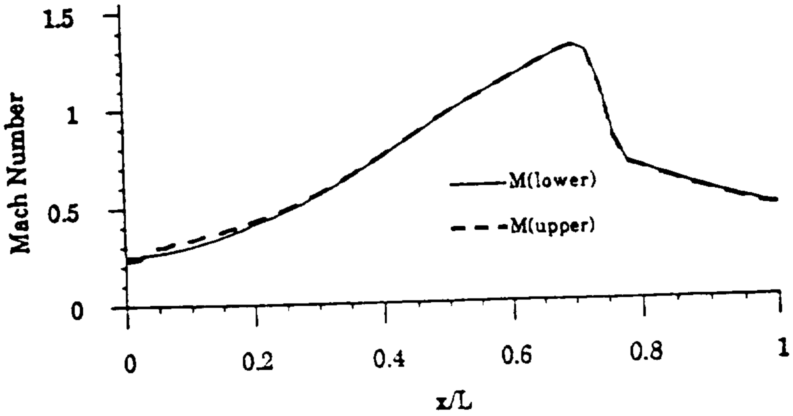


FIGURE 14: Computed Mach number distribution along lower and upper surfaces of the channel for the transonic flow case with subsonic inlet and exit.

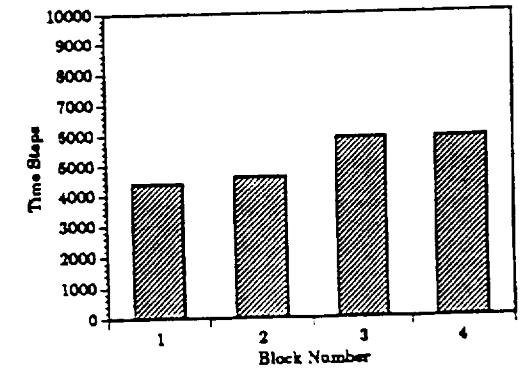


FIGURE 15: Convergence of each block to steady-state for the transonic flow case with subsonic inlet and exit.

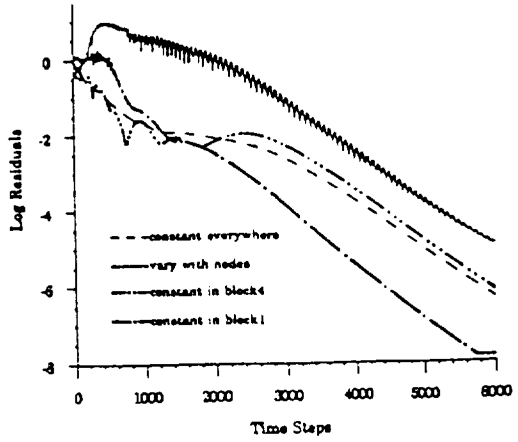


FIGURE 16: Effect of different spatial variations of Δt.

5.3.3 Transonic flow with subsonic inlet and supersonic exit For an inlet Mach number of 0.24, the exit conditions were left free yielding a supersonic exit. Shown in Fig. 17 is the Mach number variation along the upper and lower wall surfaces. The corresponding convergence requirements of this problem are shown in Fig. 18. As may be observed, all blocks reach the steady-state after almost the same number of time steps. The maximum number of steps required for convergence was considerably less than in the two previous cases.

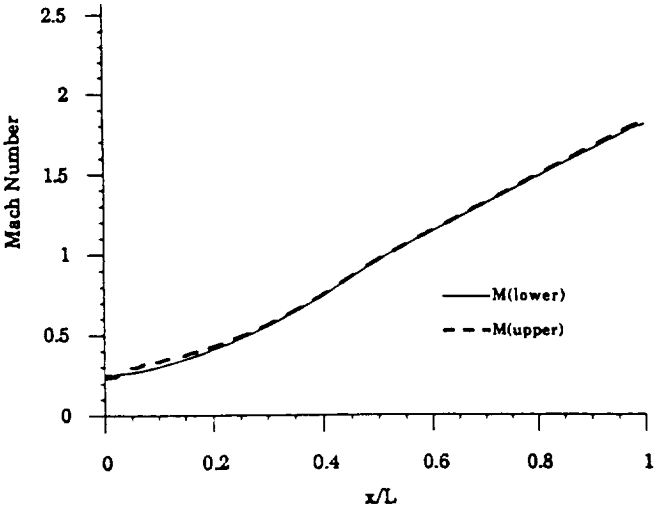


FIGURE 17: Computed Mach number distribution along lower and upper surfaces of the channel for the transonic flow case with subsonic inlet and supersonic exit.

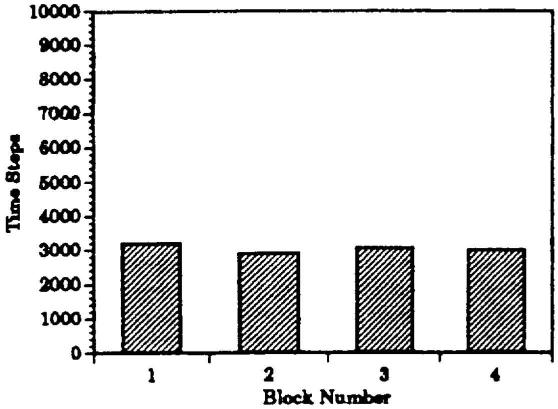


FIGURE 18: Convergence of each block to steady-state for the transonic flow case with subsonic inlet and supersonic exit.

5.3.4 Supersonic flow with supersonic inlet and exit In this case, the inlet Mach number was s to 2.65 and the exit was left free. The Mach number variation along the upper and lower wa surfaces is given in Fig. 19. The corresponding convergence requirements of this problem ar shown in Fig. 20. It is observed that block 1 reaches the steady state much earlier than the other By stopping the solution of blocks reaching the residual criterion of 10^{-6}, about 30% savings : computations were reached. Among the five cases considered here, this case required the lea. number of steps to reach the steady-state.

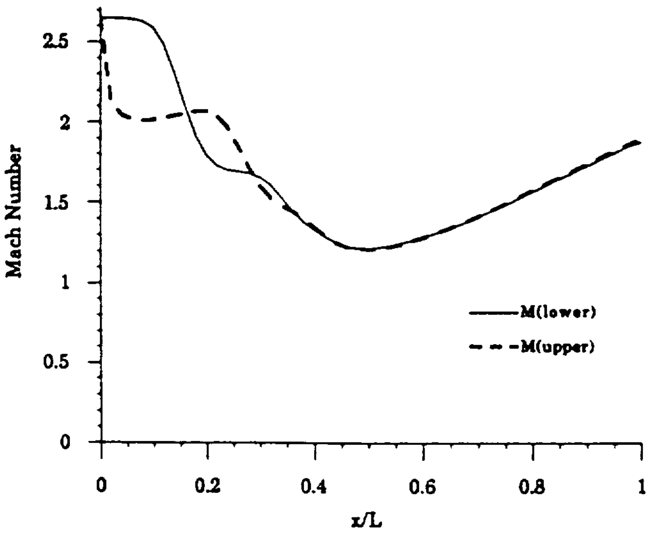


FIGURE 19: Computed Mach number distribution along lower and upper surfaces of the channel for the supersonic flow case with supersonic inlet and exit.

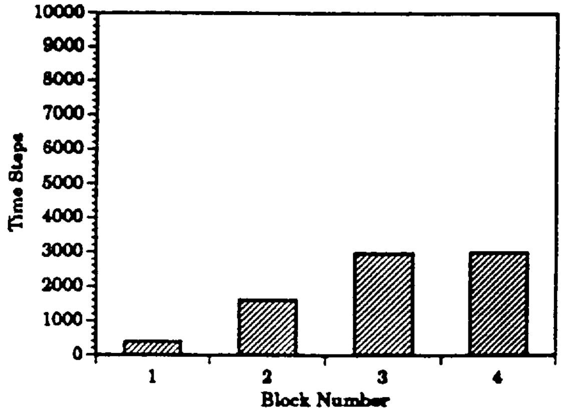


FIGURE 20: Convergence of each block to steady-state for the supersonic flow case with supersonic inlet and exit.

5.3.5 Transonic flow with supersonic inlet and subsonic exit In this case, the inlet Mach number was set to 2.65 and the back-pressure was specified to yield an exit to inlet pressure ratio of 13.72. The Mach number variation along the upper and lower wall surfaces is given in Fig. 21. The corresponding convergence requirements of this problem are shown in Fig. 22. As may be observed, blocks 1 and 2 reach steady state much earlier than the other blocks. By stopping the solutions in blocks 1 and 2 at earlier stages, a 40% efficiency was achieved. As shown in Fig. 23, the savings are even more substantial when a mesh of 120, 000 grid points and 20 blocks is used for the solution of a similar problem.

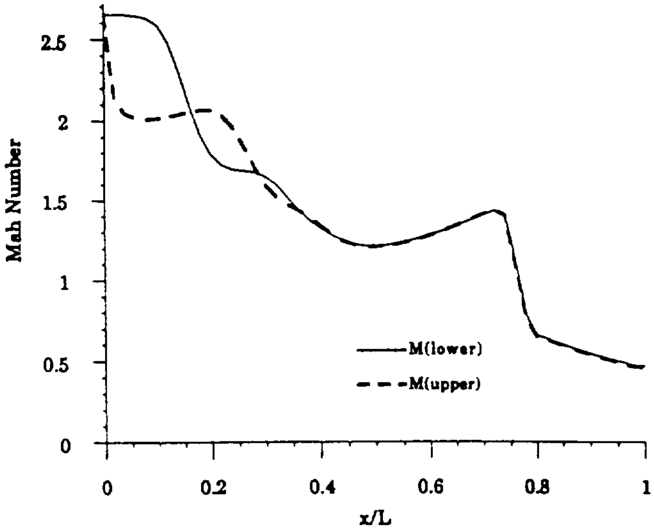


FIGURE 21: Computed Mach number distribution along lower and upper surfaces of the channel for the transonic flow case with supersonic inlet and subsonic exit.

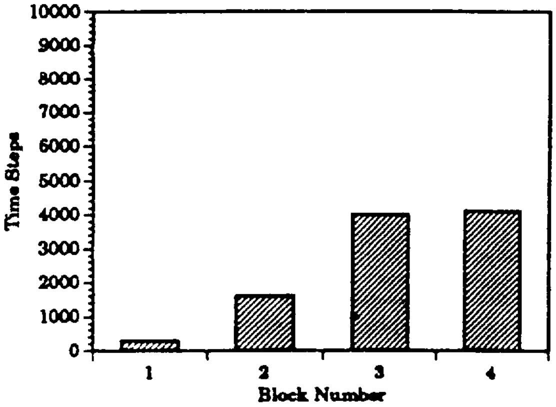


FIGURE 22: Convergence of each block to steady-state for the transonic flow case with supersonic inlet and subsonic exit.

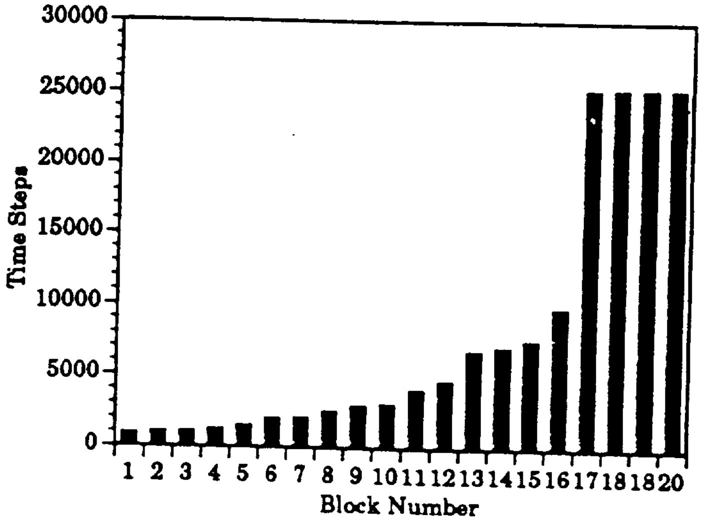


FIGURE 23: Convergence of each block to steady-state for a mesh of 12, 000 grid points and 20 blocks.

5.4 *Example of Variable Time-Stepping for Unsteady Flows*

For illustration of the variable time-stepping algorithm in unsteady flows, we have selected the same channel geometry of Example 3 and considered a sinusoidal variation of the exit pressure of Case 5 in the form:

$$p(t) = p_o + 0.04\, p_o \sin \omega t \qquad (23)$$

where ω is the frequency of oscillations. This corresponds to ±4 % variations in back-pressure p_o of the case in section 5.3.5. The results of the case with a frequency of 0.02 rad/s are summarized in Fig. 24. As may be observed, ±4 % variations in downstream pressure changes the shock location significantly, while the supersonic region from inlet to throat remains undisturbed. Since in such unsteady problems it is necessary to maintain the time-accuracy of the solutions, the constraint in Eq. 22 was used together with Eq. 21 for the selection of time increments in each block.

For the purpose of studying the efficiency of the variable time-stepping algorithm, we have also considered two additional grids:

Grid 1: 10,800 grid points, 5 blocks, 5 machines.

Grid 2: 200,000 grid points, 20 blocks, 5 machines.

Blocks of nearly equal sizes were distributed to 5 machines. By using the algorithm described in Section 4.2, for a block advancing with a smaller time step than its neighbors, the boundary conditions were determined from the interface data by linear extrapolations in time (Fig. 6). The

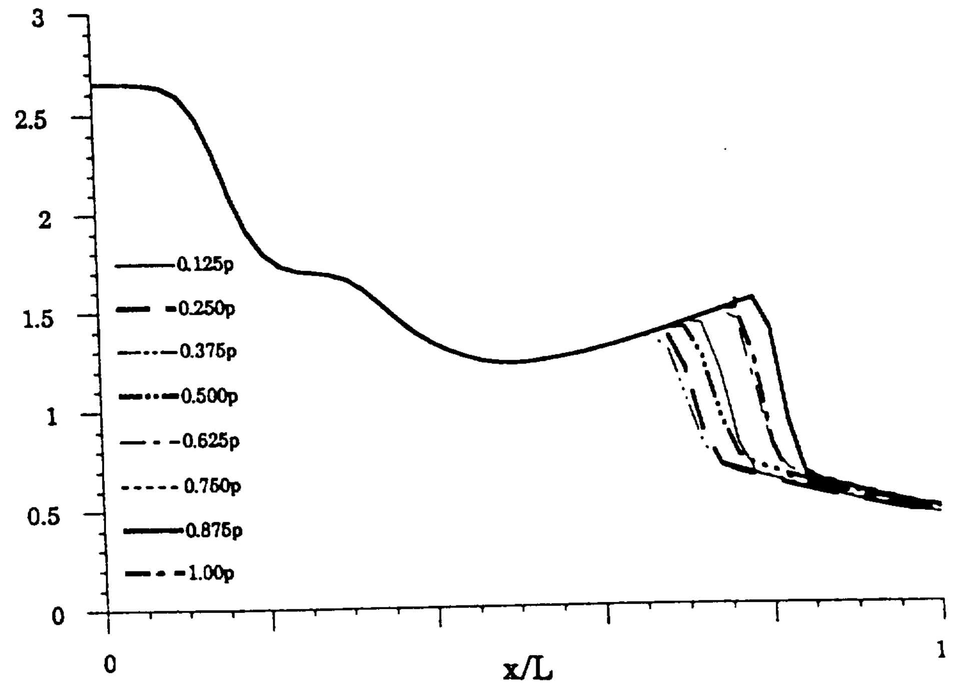
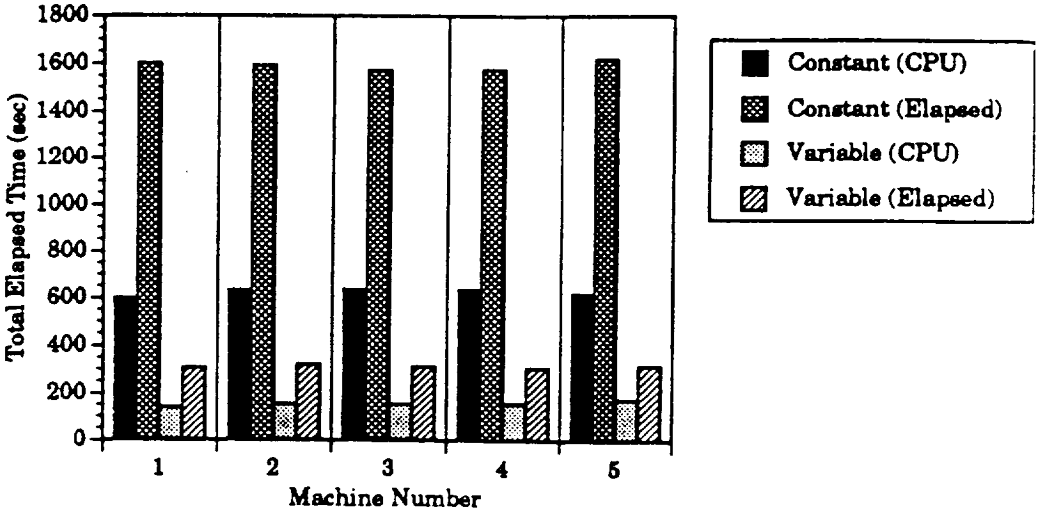


FIGURE 24: Computed Mach number distribution along the lower surface of the channel for unsteady variations of back-pressure (at $t = 1/8$th period positions).

CPU and elapsed times were obtained for 5000 steps of the constant time-stepping option. Shown in Fig. 25 are the comparisons of CPU and elapsed times of constant and variable time-stepping algorithms for Grid 1. Although, the elapsed times of the variable time-stepping scheme are about 20% of the constant time-stepping scheme, the elapsed times are approximately twice the CPU times. This is attributed to the significance of communication times compared to block solver times in small size grids in each machine. The corresponding results for the larger case (Grid 2) are given in Fig. 26. As may be seen, since the block sizes are larger, the differences between CPU and elapsed times are insignificant, indicating that the communication times are negligible compared to the computations performed in the block solvers. The elapsed times of the variable time-stepping scheme are about 20% of the constant time-stepping scheme in this case too.



EXAMPLE 25: Unsteady problem: time comparisons between constant and variable time-stepping algorithms (10, 800 grid points).

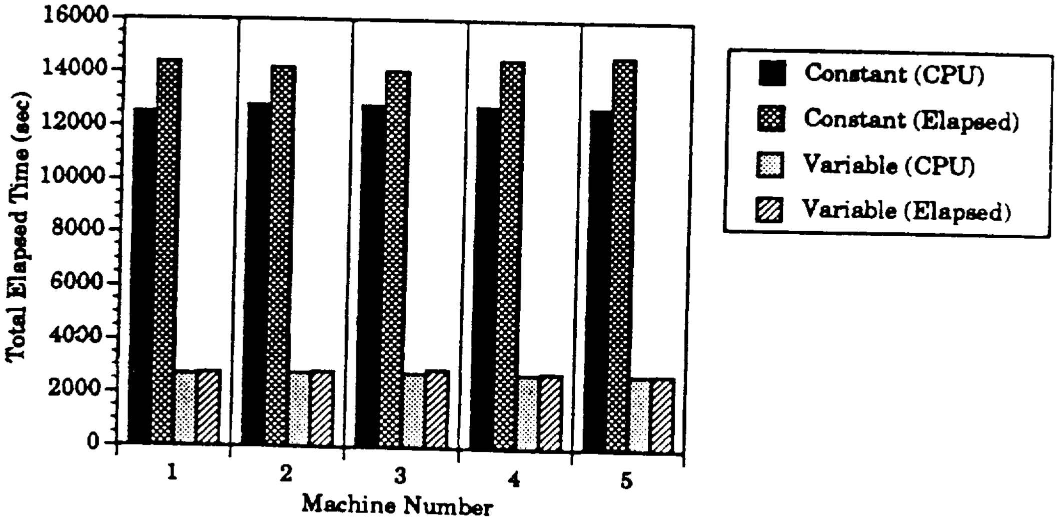


FIGURE 26: Unsteady problem: time comparisons between constant and variable time-stepping algorithms (200, 000 grid points).

6. CONCLUSIONS

In this chapter, we have summarized some of the considerations involved in solving large CFD problems on network of workstations using explicit solution algorithms. Specifically, the issues concerning load balancing, efficiency and time increment restrictions are addressed. The domain partitioning approach used here allows:

- Ease in programming and data base management.
- Flexibility in load balancing.
- Control in the implementation of block-based variable time-stepping.

One can observe that the combination of dynamic load balancing and variable time-stepping schemes can be a powerful tool in computing unsteady flows. Work is in progress in extending the variable time-stepping algorithm to unsteady external flows, where the benefits may be more pronounced due to the nature of the variations in local flow conditions and grid spacing.

REFERENCES

1. Akay, H.U., Blech, R.A., Ecer, A., Ercoskun, D., Kemle, W.B., Quealy, A. and Williams, A. (1993), A Database Management System for Parallel Processing of CFD Algorithms, *Proceedings of Parallel Computational Fluid Dynamics '92*, R.B. Pelz et al. (eds.), North-Holland, 9-23.
2. Quealy, A., Cole, G.L. and Blech, R.A. (1993), Portable Programming on Parallel Networked Computers Using the Application Portable Parallel Library (APPL), *NASA TM 106238*, Lewis Research Center, Cleveland, Ohio, USA.
3. Ecer, A., Akay, H.U., Kemle, W.B., Wang, H., Ercoskun, D. and Hall, E.J. (1994), Parallel Computation of Fluid Dynamics Problems, *Computer Methods in Applied Mechanics and Engineering*, **112**, 91-108.
4. Akay, H.U. and Ecer, A. (1994), Parallel Computation of Unsteady Flows on Network of Workstations, *Proceedings of the 2nd Japan-US Symposium on Finite Element Methods in Large-Scale Computational Fluid Dynamics*, Tokyo, Japan, 63-66.
5. Chien, Y.P., Carpenter, F., Ecer, A. and Akay, H.U. (1993), Computer Load Balancing for Parallel Computation of Fluid Dynamics Problems, *Proceedings of Parallel Computational Fluid Dynamics '93*, North Holland.
6. Chien, Y.P., Ecer, A., Akay, H.U. and Carpenter, F. (1994), Dynamic Load Balancing on a Network of Workstations for Solving Computational Fluid Dynamics Problems, *Computer Methods in Applied Mechanics and Engineering* (in print).
7. Hughes, T.J.R. and Tezduyar, T.E. (1984), Finite Element Computation of Compressible Flows with the SUPG Formulation, *Computer Methods in Applied Mechanics and Engineering*, **45**, 217-284.
8. LeBeau, G.J., Ray, S.E., Aliabadi, S.K. and Tezduyar, T.E. (1992), SUPG Finite Element Computation of Compressible Flows with the Entropy and Conservation Variables Formulations, *University of Minnesota Supercomputer Institute Research Report 92/96*.
9. Donea, J. (1984), A Taylor-Galerkin Method for Convective Transport Problems, *International Journal for Numerical Methods in Engineering*, **20**, 101-120.
10. Löhner, R., Morgan, K. and Zienkiewicz, O.C. (1984), The Solution of Non-Linear Hyperbolic Equation Systems by the Finite Element Method, *International Journal for Numerical Methods in Fluids*, 4, 1043-1063.
11. Hirsch, Ch. (1990), *Numerical Computation of Internal and External Flows*, Volume 2, John Wiley & Sons.
12. Roache, P.J. (1976), *Computational Fluid Dynamics*, Hermosa Publishers, Albuquerque, New Mexico.
13. Geist, G.A. and Sunderam, V.S. (1992), Network-Based Concurrent Computing on the PVM System, *Concurrency: Practice and Experience*, 4(4), 293-311.
14. Löhner, R., Morgan, K. and Zienkiewicz, O.C. (1984), The Use of Domain Splitting with an Explicit Hyperbolic Solver, *Computer Methods in Applied Mechanics and Engineering*, **45**, 313-329.

Parallel CFD '95, June 26-29, 1995, Pasadena, California

An investigation of load balancing strategies for CFD applications on parallel computers

N. Gopalaswamy[a], Y.P. Chien[a], A. Ecer[a], H.U. Akay[a], R.A. Blech[b] and G.L. Cole[b]

[a]Purdue School of Engineering and Technology, IUPUI, Indianapolis, Indiana

[b]Computational Technologies Branch, NASA Lewis Research Center, Cleveland, Ohio

1. INTRODUCTION

As the use of parallel computers is becoming more popular, more attention is given to manage such systems more efficiently. In this paper, several issues related to the problem of load balancing for the solution of parallel CFD problems are discussed. The load balancing problem is stated in a general fashion for a network of heterogeneous, multi-user computers without defining a specific system. The CFD problem is defined in a multi-block fashion where each of the data blocks can be of a different size and the blocks are connected to each other in any arbitrary form. A process is attached to each block where different algorithms may be employed for different blocks. These blocks may be marching in time at different speeds and communicating with each other at different instances. When the problem is defined in such general terms, the need for dynamic load balancing becomes apparent. Especially, if the CFD problem is a large one, to be solved on many processors over a period of many hours, the load balancing can aid to solve some of the following problems:

- load of each processor of a system can change dynamically on a multi-user system: one would like to use all the processors on the system whenever available.
- an unbalanced load distribution may cause the calculations for certain blocks to take much longer than others, since the slowest block decides the elapsed time for the entire problem. This may occur during different instances of the execution if the algorithm is dynamic, i.e., solution parameters change with the solution.

Based on the above considerations, the load balancing problem was treated by dynamically adjusting the distribution of the blocks among available processors during the program execution, based on the loading of the system. The details of the load balancing algorithm was presented previously [1,2]. Here, only the basic steps of the dynamic load balancing process are listed as follows:

- Obtain reliable computational cost information during the execution of the code.
- Obtain reliable communication cost information during the execution of the code.
- Determine the total cost in terms of computation and communication costs of the existing block distribution on the given system.
- Periodically, re-distribute the blocks to processors to achieve load balancing by optimizing the cost function.

In the present paper, an Euler and Navier-Stokes code, PARC2D, is employed to demonstrate the basic concepts of dynamic load balancing. This code solves unsteady flow equations using conservation variables and provides different order Runge-Kutta time-stepping schemes [3]. Parallel implementation of the explicit time-integration algorithm involves the following steps:

- Division of the computational grid into a greater number of blocks than the number of available processors with one layer of elements overlapped at inter-block boundaries.
- Introduction of the block and interface information into the data base management program, GPAR [4].
- Distribution of the blocks and associating interfaces among the available processors by using GPAR.
- Definition of PARC2D as a block-solver for the solution of the equations inside each block either for Euler or Navier-Stokes equations.
- Preparation of an interface solver to communicate with its parent block and its twin interface. As can be seen from Figure 1, each block automatically sends information to its interfaces after each iteration step. The interfaces will then send information to their twins whenever necessary for the twin to update its parent block. The task assigned to each block may not be identical, due to factors such as: size of the block, choice of either Euler vs. Navier-Stokes equations for a particular block, size of the time-step for solving each block and the time-step for communicating between the interfaces. Thus, controlling communications and computations in such a truly heterogeneous environment becomes even more critical.

These issues are discussed below in detail, for a sample problem.

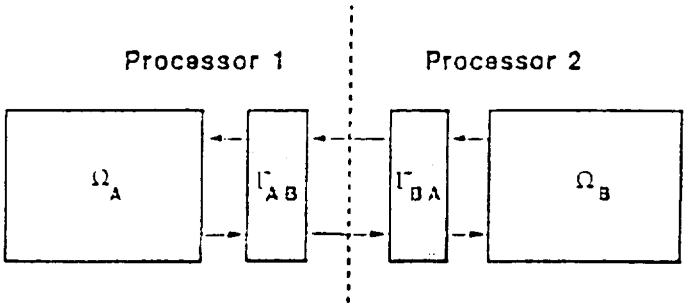


Figure 1. Communication between two neighboring blocks and related interfaces (Ω_A and Ω_B are blocks. Γ_{AB} and Γ_{BA} are interfaces).

2. INVESTIGATION OF DYNAMIC LOAD BALANCING STRATEGIES

Numerical experiments were chosen to demonstrate some strategies in dynamic load balancing for managing computer systems and algorithms. The chosen test problem is a two-dimensional grid for an inlet with 161,600 grid points as shown in Figure 2. The flow region is divided into 17 blocks as shown in Figure 3, each with approximately 10,000 grid points.

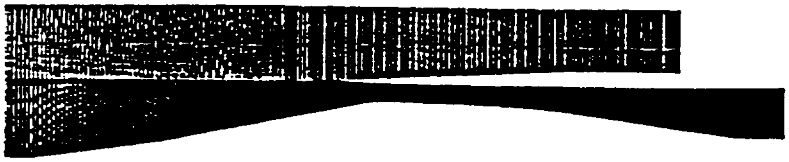


Figure 2. Computational Grid for the Inlet (161,600 nodes).

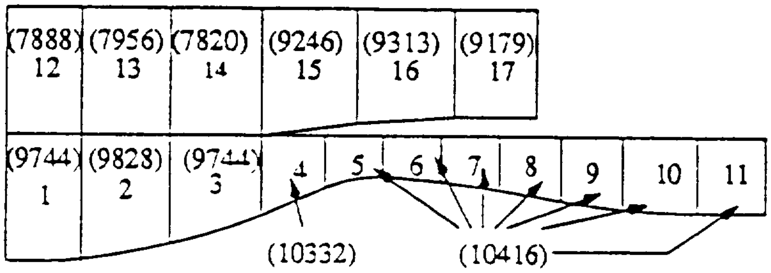


Figure 3. Division of the flow-field into Blocks for the Inlet Grid (number of grid points in each block are shown in parentheses).

2.1. Load Balancing for Variations in the System Loading

One type of load balancing strategy involves controlling the computer system. It may be a heterogeneous and multi-user system. It is also dynamic in a sense that it changes over a long run. The test problem was run on four processors over a period of approximately twelve hours. Communication and computation costs for each process were recorded and a load balancing was performed after approximately every thirty minutes. Figure 4 summarizes the results of this computation for a controlled environment. As shown in Figure 4a, the loading of certain machines was increased by adding extraneous processes, while on other machines no other jobs are running. The response of the load balancer is summarized in Figure 4b. Over 24 load balance cycles, the elapsed time for each iteration varies between 1.5 to 4 seconds. The load balancer recorded the communication and computation cost data over a cycle and predicted the elapsed time for a suggested load balanced distribution. As can be seen from this figure, the prediction is quite accurate and reduced the elapsed time by removing the bottlenecks. Figure 5 illustrates the same problem run on an uncontrolled environment. Four heavily used processors were chosen during the daytime operation. The load balancer responded in a similar fashion to a rather irregular loading pattern of the system. It is interesting to note that in this case, the total elapsed time was not excessive in comparison with the elapsed time for the dedicated machine.

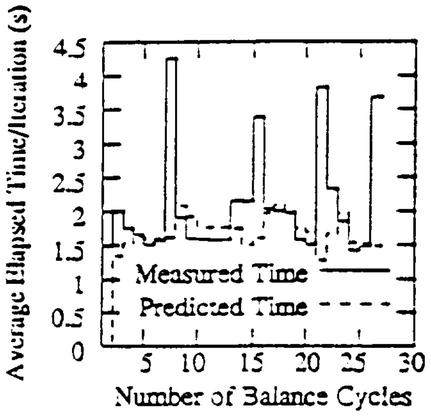


Figure 4a. Load balancing in a controlled environment.

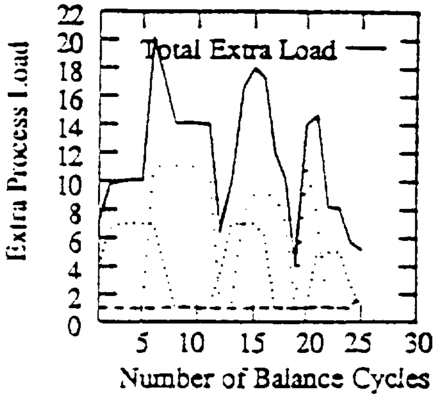


Figure 4b. Extra load variation on the system for the controlled environment.

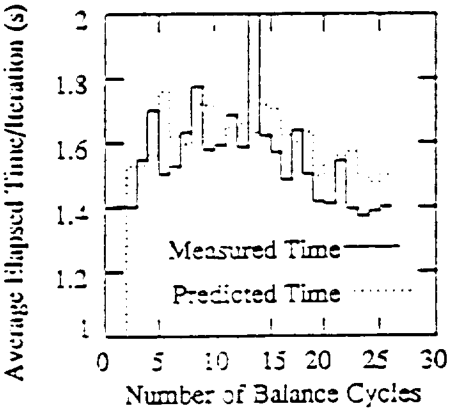


Figure 5a. Load balancing in a multi-user environment.

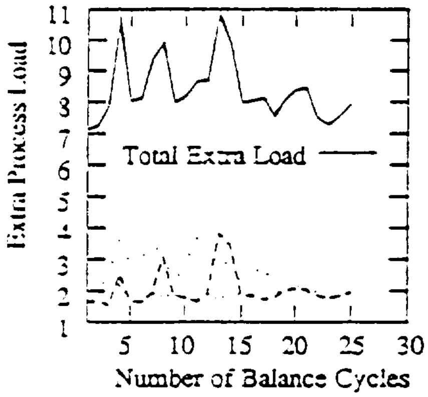


Figure 5b. Extra load variation on the system for the multi-user environment.

2.2. Load Balancing for Heterogeneous Algorithms in Parallel Computing

The second type of load balancing strategy involves optimizing the algorithm on a parallel system and dynamically load balancing the problem as the algorithm adapts to the solution. When running the PARC2D code. one can specify a time step for each block from the CFL condition as defined below:

$$\Delta t = \text{CFL} \ / \ \underset{i}{\text{Max}} \left[\left(|U_j| + a|K_i^j| \right) + \frac{2}{\text{Re}} \frac{\mu}{\rho} |K_i^j|^2 \right] \qquad (1)$$

where U_j are the contravariant velocities, a is the speed of sound, Re is the Reynolds number, μ is the viscosity, ρ is the density, and K_i^j is the Jacobian matrix. This time step is calculated for all the grid points inside a block, depending on the local flow conditions and grid size; the minimum value of all such time steps is chosen as the time step for that particular block. Since, the flow conditions are changing, the time step for each block changes over the history of a complete run [5].

Variable time-stepping with variable communications is illustrated in Figure 6a for two neighboring blocks on two different processors. In this case, Δt_{min} is a global reference time step for all the blocks. The first block at this instant is operating with a time step of $3\Delta t_{min}$, while the second block is running with $2\Delta t_{min}$. The arrows indicate the instances at which an interface of a block sends a message to its neighbor. Figure 6b shows a non-optimum solution. Here, while the computations are performed for each block solver with its own time step, each block is sending information to its neighbor at every global time step.

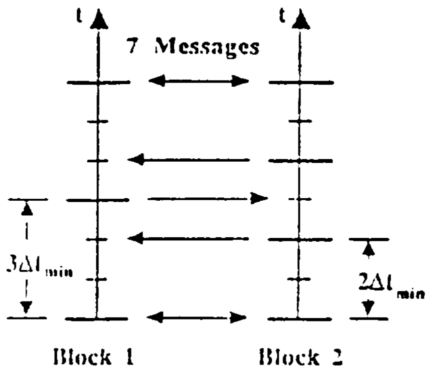


Figure 6a. Communication occurs when necessary

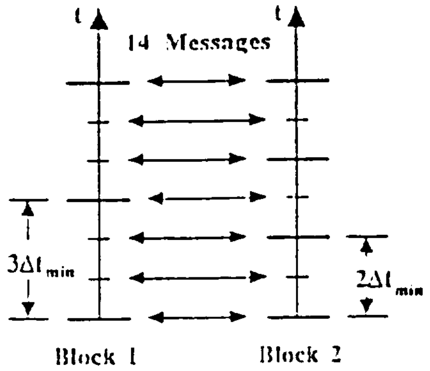


Figure 6b. Communication occurs at the global time step.

Figure 7 provides a summary of the computations with fixed and variable time-stepping. The reference case is case 1, where the time step is the same for all the blocks.

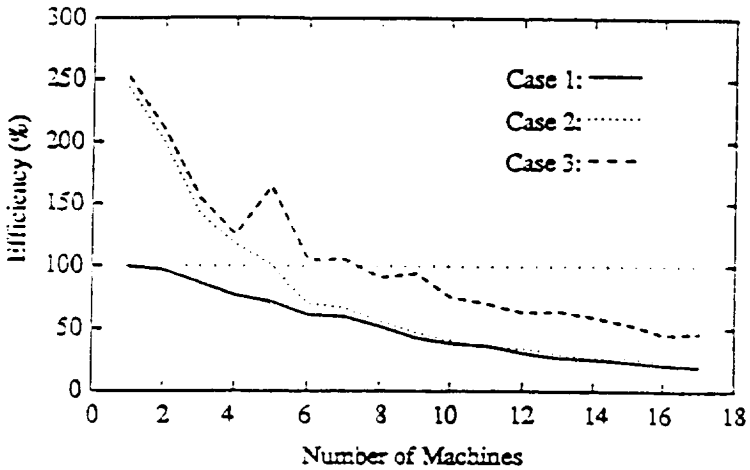


Figure 7. Parallel efficiency vs. number of machines.

The equations are solved for each block at each time step and the blocks communicate with each other after each time step. As can be seen from this figure, the parallel efficiency falls below 50% after 8 processors for the sample problem. Case 2 indicates the importance of variable time-stepping that is local to each block. In this case, each block chooses its own time step for solving the equations for that block. however communicates with its neighbors based on the global time step, as described in Figure 6b. As can be seen from Figure 7, after 6 processors, communication cost becomes the dominant factor for this case. Case 3 illustrates the need for intelligent communication as suggested by Figure 6a. In this case, a block sends a message to its neighbor only when necessary since each block is solved only when necessary. In this case, the parallel efficiency can be maintained at a higher level even when the communication cost becomes dominant. For example, around the leading edge of an airfoil with very fine grids, one can choose time steps of different order than other blocks and save computation time. Also, these blocks may not need to talk to their neighbors after each solution time step. The computational savings, discussed above, are purely due to the refinements in the use of the algorithm. When performing parallel computing, one can localize the algorithm according to the flow conditions and grids, especially for the solution of large problems with complex grids. It should be remembered that all of the above cases were load balanced to determine the most efficient distribution under given conditions. These experiments were possible only after a reliable load balancing procedure was developed.

The second example involves the solution of Euler and Navier-Stokes solutions at different blocks. The time step restriction for viscous computations is more restrictive than Euler computations as can be observed from Equation 1. Figure 8 illustrates a case when the computations were started by an Euler computation for blocks 12-17 and Navier-Stokes solution for blocks 1-11.

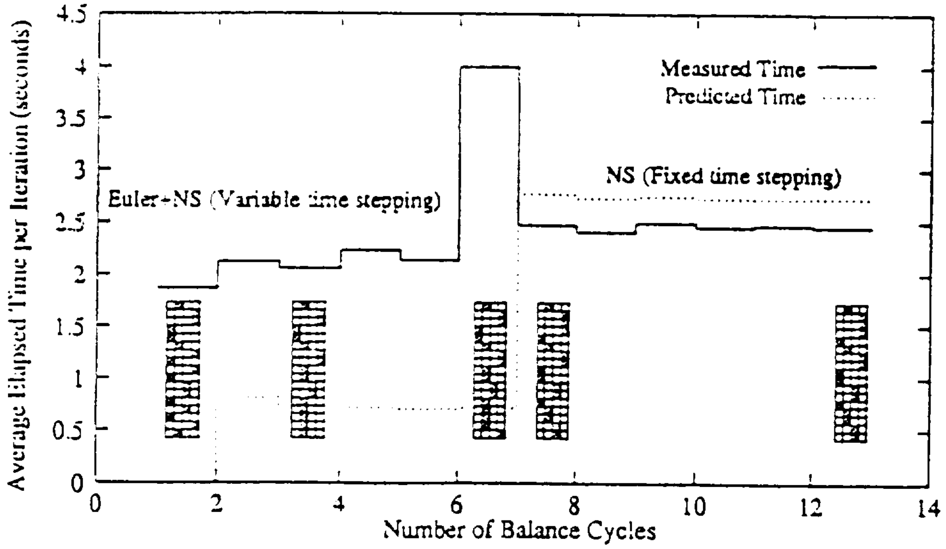


Figure 8. Load balancing due to change in solution algorithm.

The numerical integration took approximately 2.1 seconds per time-step. Local time-stepping was employed for all blocks. The distribution of the 17 blocks among 4 machines is also shown in the figure. Afterwards, blocks 12-17 were switched to a Navier-Stokes solver and global fixed time-stepping was employed for all blocks. As can be seen again from this figure. the load balancer provided a new distribution which eliminated the bottleneck by removing several processes from machine 2 and loading machines 1 and 3. Again in this case, it is shown that an algorithm can be defined and executed locally on a flow region for improving efficiency. By defining the parallel computing in a heterogeneous environment, one can employ an algorithm in a most efficient manner whenever necessary.

The third example relates to the development of algorithms which communicate in a selective manner. The cost of communication is still the dominant factor in parallel computing. It only makes sense to develop intelligent interfaces to communicate between the blocks-processes. Figure 9 shows two blocks in a one-dimensional flow field which are sending messages to each other at different speeds.

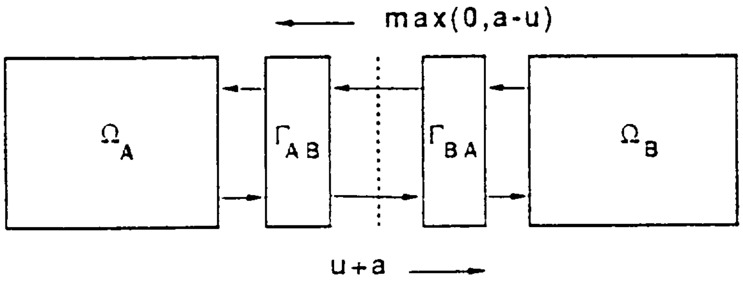


Figure 9. Communication in subsonic and supersonic flows.

Also by remembering the grid requirements for the grid points on an interface. send and receives between the two neighboring blocks can be executed at different time intervals. The test case is a specific one where most of the flow is supersonic except for blocks 8-11 which are located inside the inlet. In this case for all supersonic interfaces, one can send messages only in one direction. Figure 10 demonstrates such a case. The time-integration started where each block was communicating with its neighbors as discussed above. The distribution of the blocks among the processors is also shown in the figure. The solution scheme was then modified where the supersonic flow regions the messages were sent only in one direction. The load distribution was also modified as shown in this figure which reduced the elapsed time per iteration from 2.5 to 1.8 seconds. This figure also shows a change in the loading of the system after the 13th balance cycle which was corrected by the load balance: a block was moved from machine 3 to machine 4.

The above examples illustrate the advantages of parallel computing defined in a general fashion. Concepts such as heterogeneity and asynchronous computations in terms of both algorithms and computer systems can help to improve efficiency of parallel computing.

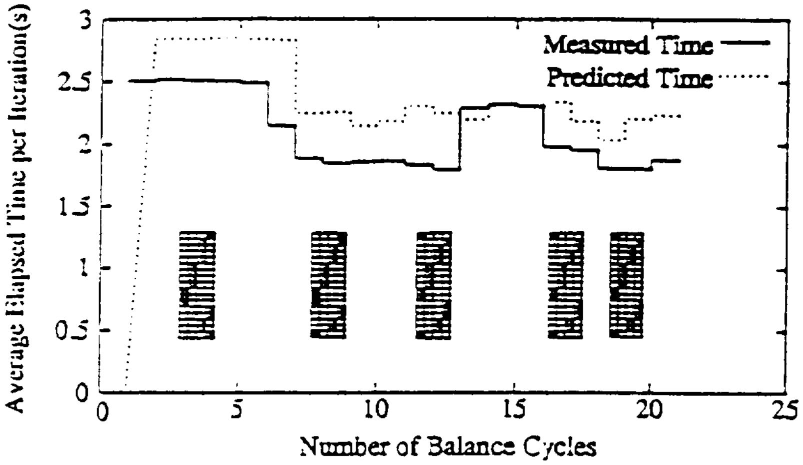


Figure 10. Load balancing in subsonic and supersonic flows.

ACKNOWLEDGMENTS

This research was supported by the NASA Lewis Research Center under NAG3-1577. Computer access provided by NASA and IBM is gratefully acknowledged. The authors thank S. Secer of IUPUI's CFD Laboratory for his assistance on the implementation of the load balancing algorithm.

REFERENCES

1. Y.P. Chien, A. Ecer, H.U. Akay, F. Carpenter and R.A. Blech, "Dynamic Load Balancing on a Network of Workstations for Solving Computational Fluid Dynamics Problems," *Computer Methods in Applied Mechanics and Engineering*, vol. 119, pp. 17-33, 1994.
2. Y.P. Chien, A. Ecer, H.U. Akay and R.A. Blech, "Environment Requirements for Using Dynamic Load Balancing in Parallel Computations," *Proceedings of Parallel CFD '94*, Edited by A. Ecer et al., Elsevier, Amsterdam, 1995.
3. G.K. Cooper and J.R. Sirbaugh, "The PARC Code: Theory and Usage," Arnold Engineering Development Center, TR-89-15, 1989.
4. H.U. Akay, R.A. Blech, A. Ecer, D. Ercoskun, B. Kemle, A. Quealy and A. Williams, "A Database Management System for Parallel Processing of CFD Algorithms," *Parallel CFD '92*, Edited by R.B. Pelz, et al., Elsevier, Amsterdam, pp. 9-23, 1993.
5. H.U. Akay and A. Ecer, "Efficiency Considerations for Explicit CFD Solvers on Parallel Computers," *Proceedings of the International Workshop on Solution Techniques for Large-Scale CFD Problems*, Montreal, Quebec, Canada, pp. 289 - 314, September 26-28,1994.

AIAA 96-3302

Parallelization and Dynamic Load Balancing of NPARC Codes

N. Gopalaswamy, H.U. Akay, A. Ecer and Y.P. Chien
Computational Fluid Dynamics Laboratory
Purdue School of Engineering and Technology, IUPUI
Indianapolis, IN

32nd AIAA/ASME/SAE/ASEE Joint Propulsion Conference

July 1-3, 1996 / Lake Buena Vista, FL

For permission to copy or republish, contact the American Institute of Aeronautics and Astronautics
1801 Alexander Bell Drive, Suite 500, Reston, VA 22091

PARALLELIZATION AND DYNAMIC LOAD BALANCING OF NPARC CODES

N. Gopalaswamy, H.U. Akay, A. Ecer and Y.P. Chien
Computational Fluid Dynamics Laboratory
Purdue School of Engineering and Technology, IUPUI
Indianapolis, Indiana

Abstract

Parallelization and dynamic load balancing of the 2D and 3D NPARC flow codes of NASA are presented. A previously developed parallel database package GPAR, and a dynamic load balancer program DLB are used for both the 2D and 3D versions of NPARC. Performance characteristics of the implemented algorithms in 2D and 3D internal flow configurations are explored. Dynamic load balancing studies are carried out with the two parallel codes for an engine inlet configuration. The benchmark cases consist of a 2D case with 4,592 grid points and two 3D cases: one with 50,950 grid points, and the other with 240,000 grid points. The grids are decomposed into solution blocks and parallel computations are carried out with varying number of processors. The pressure response to unsteady perturbations of the inlet temperature is calculated using a variable time-step approach specifically developed for parallel computations which takes into account the time-step variations in blocks with optimized communications between the blocks. It is found that time accuracy is maintained with the benefits of increased speedup with the above approach. Load balancing is found to be effective only in large cases where block computation costs are more dominant than the communication costs.

Introduction

Our current research efforts are aimed at achieving an efficient computing paradigm on parallel computers. Parallel computers can be of many types, including MIMD and SIMD computers though our attention will be primarily focused on MIMD computers. The codes chosen for parallelization are the NPARC codes (2D and 3D) obtained from NASA LeRC[1]. Each code lends itself easily to parallelization by the method of domain decomposition. A software package called GPAR[2] developed earlier in CFD Lab in conjunction with a dynamic load balancer called DLB[3] was used for parallelization. The non-dimensional form of the governing equations for viscous flows are cast in conservation form in the following fashion:

$$\frac{\partial Q}{\partial t} + \frac{\partial F_j}{\partial X_j} = \frac{1}{\mathrm{Re}} \frac{\partial G_j}{\partial X_j} \qquad (1)$$

where $Q = (\rho, \rho u, \rho v, \rho w, \rho E)^T$, F_j, are the inviscid flux vectors, G_j are the viscous flux vectors and Re is the reference Reynolds number. The conservation laws are solved in strong conservation law form after transformation to computational coordinates. Although both implicit and explicit flow solution options exist, the explicit Runge-Kutta time-integration scheme is used for solution of unsteady flows. In this paper we show parallelization of the explicit solvers of the NPARC codes for unsteady flows.

The problem to be solved over a given domain is parallelized by dividing the domain into many sub-domains, called blocks, and solving the governing equations over these blocks. The blocks are connected to each other through the inter-block boundaries, called interfaces. These blocks are allocated to processors in the parallel computing environment, and the solution of the problem over the entire domain is achieved by solving the governing equations over each block, with the information exchange between the blocks handled by the interfaces.

The NPARC codes (2D and 3D) already use a block-structured solution approach. It is only necessary to write the interface communication part, called the "Interface Solver" to implement parallelization. The interior point algorithm which operates on the points inside the block, is termed the "Block Solver" and is essentially unmodified. This block-structured approach is highly suited for parallel computing since each block can possess its own set of parameters which describe the flow-field within the block more accurately, instead of using a global set of parameters applicable over the whole domain. For instance blocks far away from no-slip surfaces, in the absence of free-stream turbulence, can be modeled adequately with the Euler equations, without the additional expense of computing the viscous terms which are negligible. Such an approach has been used in the parallelization of the NPARC codes in conjunction with an improved communication strategy.

During parallel computations, often bottlenecks occur due to communication between the blocks over the network of processors. We expect that with the availability of larger number of processors over the coming years the capability to solve larger problems with more CPUs will also increase. The communication cost may become more critical as such developments occur. Another important objective has also been the identification and optimization of communication cost in a heterogeneous environment. The following sections describe the communication strategies and load balancing algorithm used to implement efficient execution of the NPARC codes.

Variable Time-Stepping Approach

When the flow-field has been decomposed into solution blocks, we can select a time-step for each block for computing unsteady flows in two ways. The default NPARC algorithm picks the most restrictive time-step among all blocks, and all blocks are advanced in time with this time-step for transient flows. An approach, called the variable time-stepping method[4], is considered in this paper. In this approach, the block time-step is picked as a multiple of the most restrictive time-step and at the interfaces, and linear interpolation is carried out between the two time levels to obtain the updated boundary conditions. The time-step for a particular block is determined from the CFL condition for stability of the explicit Runge-Kutta time-stepping algorithm from the following expression:

$$\Delta t = \frac{C}{Max_i\left[\left|U_j\right| + a\left|K_j^i\right| + \frac{2}{\mathrm{Re}}\frac{\mu}{\rho}\left|K_j^i\right|^2\right]} \qquad (2)$$

where, C is the Courant number, U_j are the contravariant velocities, a is the speed of sound, and K_j^i are the metrics of transformation. The viscous correction includes the reference Reynolds number Re, the non-dimensional viscosity μ and density ρ.

The time-step is chosen as the minimum of the above expression over all the grid points i. Thus for each block the most restrictive time-step can be different depending upon the grid and flow conditions. It has been usually found that the stability condition also satisfies the accuracy requirement. Hence, for each block we can pick a certain multiple of a global minimum time-step which satisfies the stability conditions for all blocks.

$$\Delta t_j = n_j \Delta t_{\min} \quad ; j = 1, 2, \ldots N; \;\; n_j \le n_{jmax} \qquad (3)$$

where n_j is an integer determined from the CFL condition in Equation (2), n_{jmax} is a maximum limit for n_j (typically $n_{jmax} = 5$ for time-accurate solutions, for steady flows n_{jmax} is the maximum permissible time-step ratio in each block). N is the total number of blocks. Even for blocks which are of equal size, depending on the flow conditions, the computational effort required to advance a certain amount in time can be different if the time-steps chosen are different. Some variable time-stepping studies have been carried out previously,[5] and their efficiency investigated.

Communication Strategies

While the Block Solver takes care of the solution of the grid points inside the block, the Interface Solver handles communications between the blocks. The Interface solver evaluates the information it receives from the block and decides if it should be sent to the neighboring interface solver which is located in another processor. The interface solver may also modify the information before it sends. For instance, for unsteady computations for each block we choose a time step for the computations. This is currently based on calculating the Courant numbers for all grid points and utilizing the critical Courant number inside a block as a basis for choosing the time step for the particular block. Each block marches with its own time step and stores the boundary information into its interface. The interface can store and interpolate the data and communicate with its neighbor based on the critical Courant number of the grid points local to that interface. Another consideration is the magnitude of the wave speeds across the interface at each direction: $u + a$ versus $u - a$ which give some preferential direction to the communication process. Thus, the time step necessary for communication between the interfaces does not have to be the same for the interface in the upstream block and the interface in the downstream block. In summary, the following types of communication algorithms were employed.

Scheme 1: Same time-steps for blocks and interfaces

The block time-step and the communication frequency of the interfaces belonging to the parent block are chosen to be identical. However, the blocks can possess different time-steps in a variable time-stepping scenario. Figure 1 shows the relation between the block time-stepping and the interface communication frequency. Each block advances in time with a certain time-step, chosen to be a multiple of a certain fixed global minimum time-step.

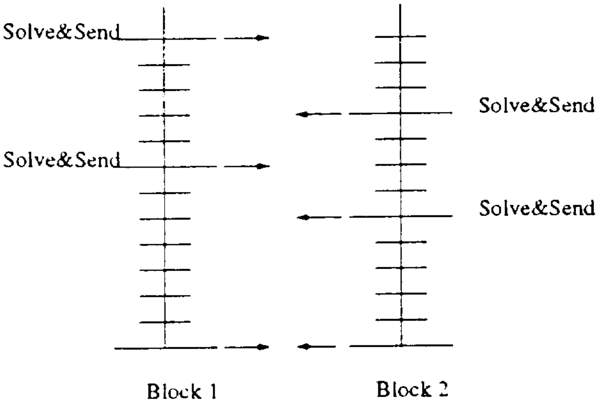


Figure 1. Interface communication frequency based on the block time-step.

Since the interface nodes belong to the parent block, these nodes are also solved for during the block solution step. Then the interface nodes are updated with the values at the advanced time-step from the block and sent to the neighboring interfaces. Also, in order to solve for the next time-step, the block needs boundary conditions at its interfaces from the neighboring blocks. Hence, it waits till the information is available for all the interfaces before it proceeds to the solve for the next time-step. If a block proceeds with a smaller time-step than its neighbor, it receives information from the neighboring block which indicates that it is at a higher time-step, and hence, the slower block linearly interpolates (in time) the boundary values from the neighboring block at its current time level.

Scheme 2: Different time-steps for blocks and interfaces

Since the partial differential equations of fluid mechanics are usually very stiff, the time-steps needed to integrate the differential equation are quite small in order to satisfy stability. Since the rest of the solution develops along the rest of the eigenvalues of the system which are smaller than the maximum, which controls stability, satisfying stability also satisfies the requirements of accuracy. Accuracy of a scheme is achieved when the solution is integrated with a time-step which contains all the eigenvalues. This observation, coupled with the frequently encountered scenario that the block time-step is decided by a relatively small region of the block, allows us to propose a scheme in which the interfaces need be updated only infrequently, for example for a time interval corresponding to the maximum stable time-step for the interface nodes. For instance, for highly stretched grids, the maximum stable time-step at the interfaces present in the region where the element lengths are large could be about 100 times a certain global minimum time-step, while the block time-step might be restricted by regions where the element lengths are very small, for example close to the wall. Thus, the interfaces need be updated only infrequently relative to the block time-step. The elements or nodes in the interface are solved by the block but the interface is itself updated only at an interval corresponding to the stability restriction for the interface nodes alone. After the update, the neighboring interfaces exchange the information required for the next time step for the block solution. It usually happens that the block time-step may be such that the interface update interval may not coincide with a block solution time-step. In such a case, the solution for the interface nodes is interpolated from two block solution time-levels, and then sent to the interface. In case of matching and overlapping interfaces, the neighboring interface may not have exactly the same stable time-step as its neighbor, since the metrics calculated numerically for one interface may not be the same as that for the neighbor, and hence the non-dimensional stable time-step for one interface may not be the same as its neighbor. Local grid stretching effects also play a part in yielding differing stable time-steps for neighboring interfaces. In such a situation, the interface which is at an advanced time-step, sends the information first, and then waits till the other interface catches up or passes. The slower interface interpolates the values at the boundary, from the solution received previously and the information currently available at the higher time-level for the neighboring interface. Figure 2 illustrates the strategy followed. This strategy results in less communication compared to the previous case which may yield savings in the elapsed time for computation of the overall problem.

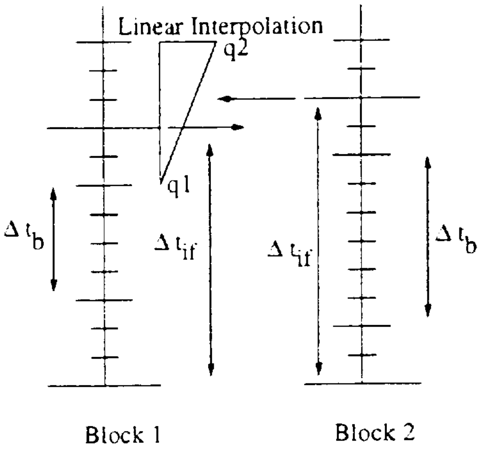


Figure 2. Interface communication frequency based on the local stability condition for the interface nodes (Δt_b = block time-step, Δt_{if} = interface time-step).

Scheme 3: Interface time-steps based on interface characteristic speeds

A communication frequency based on the local characteristic speeds in the interface region has also been proposed. Figure 3 shows a simple one dimensional example of the above discussion:

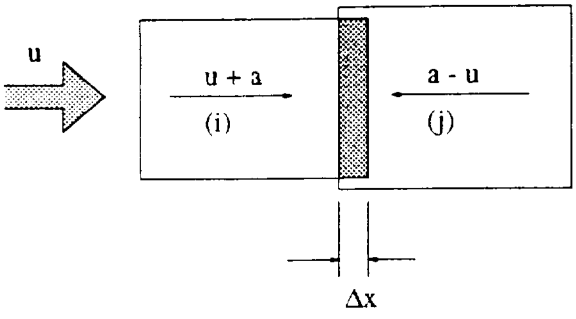


Figure 3. Communication between blocks based on local characteristics.

As can be seen from the figure, if a block were completely supersonic, then from the direction of the characteristics or eigenvalues, it is not necessary for the block downstream to communicate with the upstream block. However, the upstream block must necessarily send information downstream. If the blocks were subsonic, then the communication frequency between the blocks would depend on the following ratios:

$$\Delta t_i = \frac{\Delta x}{u + a}$$

$$\Delta t_j = \frac{\Delta x}{a - u} \qquad (4)$$

$$\frac{\Delta t_i}{\Delta t_j} = \frac{a - u}{u + a} \quad ; \ (a - u) > 0$$

where Δx is the local element length in the interface region, and u is the velocity of the fluid and a is the local speed of sound. If the flow were completely supersonic, then $a - u < 0$, and hence no messages would be sent upstream to the neighboring interface. Thus $\Delta t_j \rightarrow \infty$ and there would be a significant reduction in the communication. The block solves for the solution variables on the interface nodes. The interface nodes are updated with the block solution variables at a time interval corresponding to the communication frequency calculated from Equations (4). Since the block time-step may be such that the time interval at which to update the interface may not coincide exactly with a block solution time-step, two block solution levels are stored, and the interface is updated with a linearly interpolated value from the two solution levels. Similarly, when an interface receives information, it may not coincide with the block solution time-level, and hence the interface solution variables are stored over two time levels. This way the solution variables for the block boundaries can be obtained by an extrapolation of the interface solution variables stored over the two previous time-levels (interface time-steps or time-levels). Also, the solution may be developing, which means that a shock initially located in a particular block may start moving upstream as the solution progresses and eventually cross over into a block which was completely supersonic. Thus, if communication were completely cut-off from the downstream block to the upstream block, the shock would be stalled in the downstream block for the whole duration of the solution yielding a final solution which would be incorrect. Hence, even if the interface nodes currently appear to have supersonic flow, an upstream communication is still enabled as a certain multiple of the block time-step, in order to allow a developing flow to let a shock move upstream across blocks if needed. A brief illustration of the communication strategy is shown in Figure 4.

For multi-dimensional flows, we can extend the above reasoning by considering each direction separately. The physical coordinates map to the computational coordinates (ξ, η, ζ). Since the interfaces are usually aligned along a constant index or computational coordinate, the contravariant velocities (U, V, W) along each computational coordinate direction should be considered. Along each interface, only one contravariant velocity will exist in a direction crossing the interface, the other two are parallel since they are mutually orthogonal to each other. For example, if there is an interface along a constant ξ direction, only U exists in a direction crossing the interface, the other two contravariant velocities V and W are parallel to this interface.

$$U = \xi_x u + \xi_y v + \xi_z w \qquad (5)$$

$$\frac{\Delta t_i}{\Delta t_j} = \frac{U + a\sqrt{\xi_x^2 + \xi_y^2 + \xi_z^2}}{-U + a\sqrt{\xi_x^2 + \xi_y^2 + \xi_z^2}} \qquad (6)$$

where the pair of interfaces are denoted by the subscripts i and j. The same approach is used for interfaces aligned along constant η and constant ζ directions.

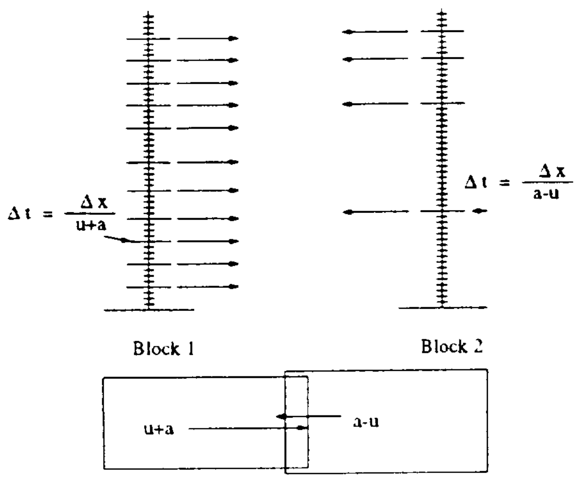

$\Delta t = \frac{\Delta x}{u+a}$
$\Delta t = \frac{\Delta x}{a-u}$
Block 1
Block 2
u+a
a-u

Figure 4. Communication frequency between the interfaces based on local characteristic speeds.

Variable time-stepping for each block and interface has been implemented in a parallel environment. For cases with variations in grid size and flow conditions, computational efficiency can be improved significantly.

Load Balancing

Following the above discussion, the objective is to reduce both the computation and the communication cost by making parallel computing optimally suited to local conditions. Apart from the algorithmic considerations, one also needs to consider the performance of the overall computation itself in terms of the processor speeds and communication speeds. Bottlenecks can also arise due to the computational load of the processors and communication times between the processors. Computing on a network of workstations or on dedicated multi-processor systems has its own set of issues to be addressed in order to obtain maximum efficiency, or in other words, a solution in the shortest possible time for a given set of resources. Obtaining maximum efficiency leads to the necessity of load balancing, or balancing the computational load on each processor during the execution. For large problems, it is typical to have a greater number of blocks compared to the number of processors or machines. As an example, for the computation of a three-dimensional wing section which has been decomposed into 150 blocks, there may be only 10 machines available. In many cases, it is advantageous to decompose the problem into more blocks than the number of machines available, since load balancing can be used to alleviate bottlenecks due to a portion of the domain, or the processor itself.

The load balancing program or the "balancer" needs statistics about the execution of the application code in terms of the computational and communication cost for each block on every processor, and also the number of extraneous processes on those processors. This is then factored into calculating the cost function. For example, the cost of computation can vary due to a change in the solution algorithm, or due to an increase in extraneous processes started or stopped by other users. The response to the two causes is different. A program called "Ptrack" (process tracker), executes concurrently on each processor on which the blocks are executing, and monitors the extraneous process load on the respective processors. The execution scenario is illustrated in Figure 5.

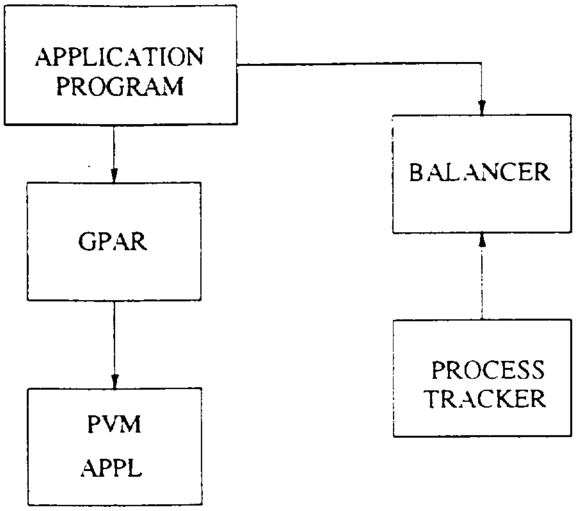


Figure 5. Balancing of the application program.

Load balancing can be of two types, static or dynamic load balancing. In static load balancing, the blocks and interfaces are allocated to the machines (or processors) in a fashion that the resulting overall computational speed of the problem achieves a maximum. Factors which come into consideration here are the block and interface sizes, proximity of blocks and interfaces, speed of the individual machines, and the communication speed and bandwidth of the network, all or some of which can vary. In dynamic load balancing, this initial distribution can change according to external factors such as extraneous processes added to or removed from the machines during computation, and also due to changes in computational speed of the blocks and interfaces themselves on the machines due to changes in the solution algorithm or the solution behavior. In a heterogeneous computing environment, load balancing becomes extremely important if efficient utilization of the given resources is desired.

The load balancing scheme developed at CFD Lab is based on the greedy algorithm[3], which tries to minimize the total cost of executing all the blocks. The formulation of the cost function can be described in the following way:

i. Let the computation cost of processing block i on a processor j be c_i^j. Here, i can take values from 1 to n where n is the number of blocks executing, and j can take values from 1 to P where P is the total number of processors the blocks are executing on.

ii. Let the communication cost of sending interface data from a processor j to its neighboring interface be b^{jk}, which may be on a different processor k.

iii. The computation cost of executing blocks on a computer j is also influenced by the waiting time W_i for each block i, since it has to wait to receive the interface information. The total cost of computation on a processor j is:

$$C^j = \Sigma(c_i^j + b_i^{jk} + W_i) \qquad (7)$$

where b_i^{jk} is the communication cost required per block i on processor j.

The load distribution problem then reduces to minimizing the maximum of the above computation costs among all the blocks, since it is the slowest block which is the bottleneck. Hence if $C = \max(C^j)$ then C should be minimized to achieve load balancing. The greedy algorithm is used to minimize C, the computational work for that being equal to $O(nP^2)$ where n is the number of data blocks, and P is the number of processors being used.

The computation and communication cost must be computed in order to serve as the input to the load balancing algorithm. This involves placing some time-stamps inside the application program to obtain the time spent by the application program in computing the data block and the time spent by the application program waiting to receive information and the time to send the required information. Based on this information, the cost function C^j for each processor is calculated, and the data blocks i are redistributed among the processors if necessary to balance the computational load. This process is done periodically during the execution of the code for every specified interval, called the balance cycle, to monitor the progress of the computation. Typical balance cycles are in the order of 100-500 time-steps.

The NPARC codes have been enabled for load balancing by means of certain calls to the DLB library functions which monitor the time spent in communication and computation in terms of both CPU and elapsed time. Information about the number of bytes exchanged between processors is also recorded, which is factored into the calculation of the communication cost.

Test Cases Considered

Three test cases were considered. All the grids for the following test cases were supplied by NASA LeRC[6]. The Euler/inviscid version of the NPARC code was used to compute the test cases with the 3-stage version of the Pseudo Runge-Kutta time-stepping scheme. The plane of the axisymmetric inlet is shown in Figure 6 together with the steady-state solution from which perturbation is started. The 17-block division used for this 2D/axisymmetric case is also shown on the same figure.

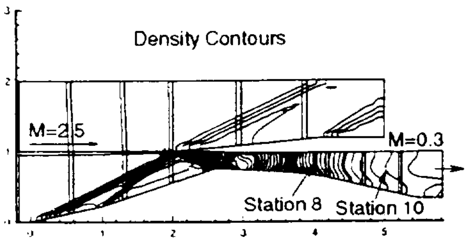


Figure 6. Steady-state density contours for the axisymmetric inlet.

Test Case 1:

A two-dimensional case with 4,592 grid points was used to study the pressure response to a sinusoidal temperature perturbation with a frequency of 225 Hz. The grid was divided into 17 blocks and the number of machines was varied from 1 to 8. The inlet Mach number was 2.5 and the exit was subsonic with a Mach number of 0.3. The exit boundary condition is based on a scheme developed previously for NPARC[7]. The reference inlet pressure was 117.8 lb/ft^2, and the reference inlet temperature was 395 Rankine. The cowl-tip radius of the inlet, $Rc=18.61$ inches, was used to non-dimensionalize the lengths. The amplitude of the sinusoidal temperature perturbation was 5% of the mean value (395 Rankine). The pressure response was measured at two locations, $X/Rc=4.08$ and $X/Rc=5.01$, downstream of the normal shock in the diverging section of the inlet.

Test Case 2:

A three-dimensional case with 50,950 grid points corresponding to a 60 degree sector of the axisymmetric inlet was divided into 16 blocks and subjected to the same inlet temperature perturbation with a sinusoidal frequency of 225 Hz and the pressure response studied. The inlet Mach number was 2.5 and the subsonic exit had a Mach number of 0.3.

Test Case 3:

A three-dimensional case with 240,000 grid points corresponding to a 360 degree O-grid of the axisymmetric inlet was divided into 20 blocks and a steady state solution was sought for an inlet Mach number of 2.5 and subsonic exit Mach number of 0.3 as shown in Figure 7.

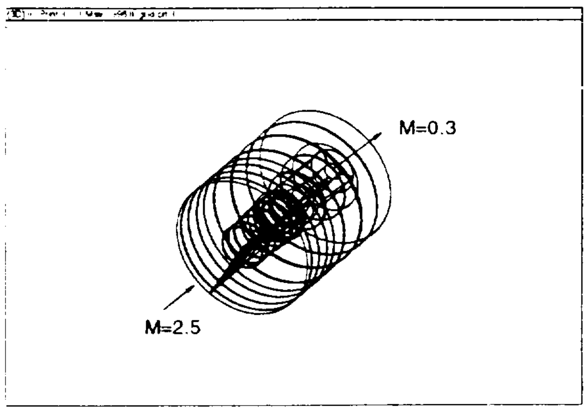


Figure 7. Three-dimensional grid for Test Case 3 with 240,000 grid points.

For each of the above test cases, the following strategies were considered to investigate the performance of the new algorithms.

Default Scheme

The base case is chosen to be global time-stepping with the same time-step for all the blocks. A time-step of 6 μs is chosen as the global time-step and computations are performed for approximately 5000 steps till a periodic response is achieved. This corresponds to an interval of approximately 0.035 seconds. Then, the same case is run for all three grids and this time dynamic load balancing is enabled and the block distribution after load balancing, and the resultant elapsed time and CPU time is recorded.

Scheme 1

This time, the variable time-stepping option is enabled, in which each block picks a certain multiple of the global time-step depending upon the critical Courant number inside the block. The initial distribution of the blocks is the same as obtained from the previous step with constant global time-stepping. The interfaces communicate with their neighbors at each block solution step as outlines in Figure 1. Again the elapsed time and CPU time are recorded for this case with and without load balancing enabled. The pressure response is plotted for the 2 stations with time, and compared to the base case.

Scheme 2

Variable time-stepping in addition to interface communication which takes place at an interval corresponding to the critical Courant number for the interface nodes is studied. The communication scheme used is shown in Figure 2. The elapsed time and CPU time are recorded and the pressure response plotted with time. Load balancing is enabled and the same case is rerun with all parameters recorded.

Scheme 3

Finally, variable time-stepping in addition to interface communication which takes place according to the characteristic speeds of the solution variables in the interface nodes is investigated. This corresponds to the communication scheme shown in Figure 4. As before, all parameters are recorded for cases with and without load balancing.

Results

The timing information for the cases considered is presented in the form of speedup and efficiency which are defined in the next two equations.

$$S_n = \frac{\text{Elapsed Time with 1 Machine (Default)}}{\text{Elapsed Time with } n \text{ Machines}} \quad (8)$$

$$\text{Efficiency} = \frac{S_n}{n} \quad (9)$$

where S_n is the speedup with n machines. The total elapsed time for solving the test case using the default communication scheme in the NPARC codes is used as a basis for comparison when speedup is calculated.

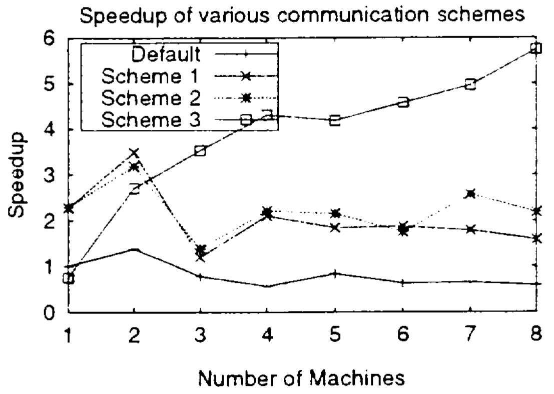


Figure 8. Speedup for the 2-D grid with 4,592 nodes divided into 17 blocks.

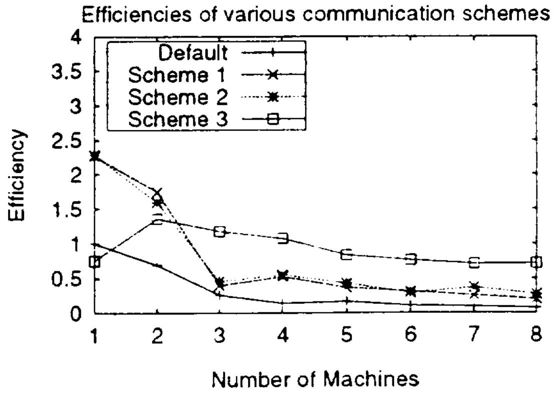


Figure 9. Efficiency for the 2-D grid with 4,592 nodes divided into 17 blocks.

From the results of the 2D case, it is found that it is highly communication bound, and load balancing yields little improvement in efficiency. Hence, the load balanced computational speedup is not displayed here. Also, the choice of communication algorithm makes a significant difference among the variable time-stepping cases. Only the case with Scheme 3 shows linear speedup as the number of machines increases.

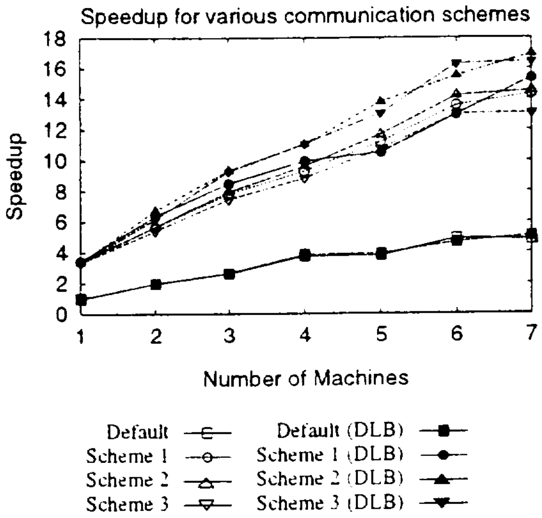


Figure 10. Speedup for the 3-D grid with 50,950 nodes divided into 16 blocks.

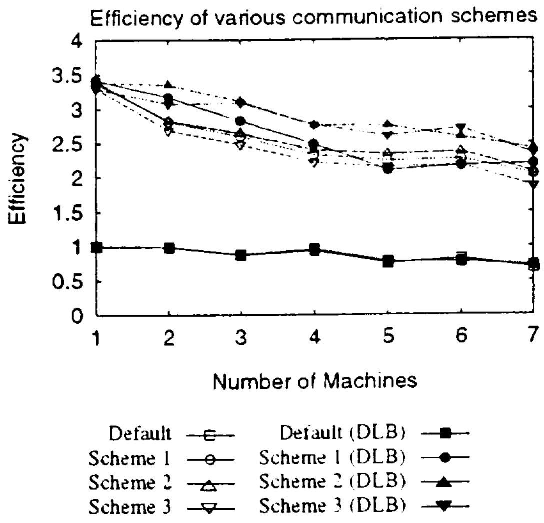


Figure 11. Efficiency for the 3-D grid with 50,950 nodes divided into 16 blocks.

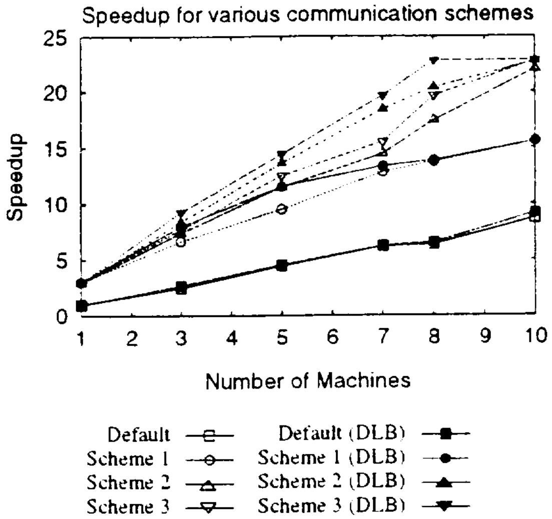


Figure 12. Speedup for the 3-D grid with 240,000 nodes divided into 20 blocks.

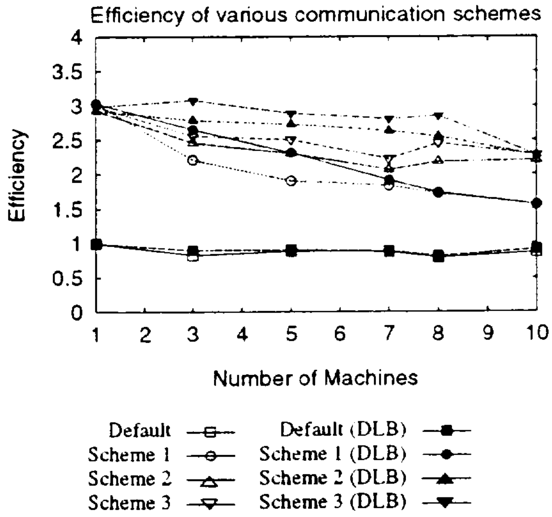


Figure 13. Efficiency for the 3-D grid with 240,000 nodes divided into 20 blocks.

The 3D cases show a much higher speedup as number of machines increases compared to the 2D case. Also, the load balancing improves the speedup and efficiency by an additional 15-25 percent for most cases.

Next, the pressure response to the sinusoidal temperature perturbation is plotted for the two monitoring stations. As can be seen from the figures, all three schemes preserve good time-accuracy with respect to the globally uniform time-stepping case.

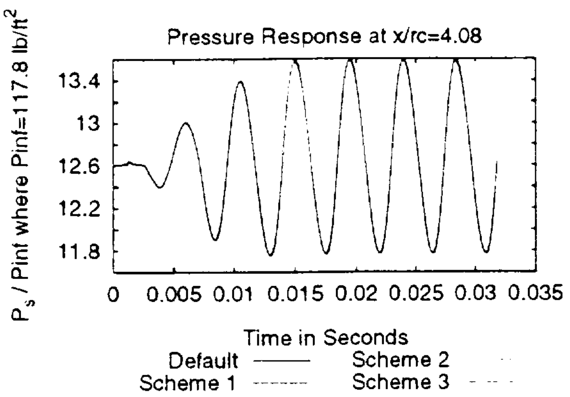


Figure 14. Pressure response at station 8 (X/Rc=4.08) to a 5% sinusoidal inlet temperature perturbation.

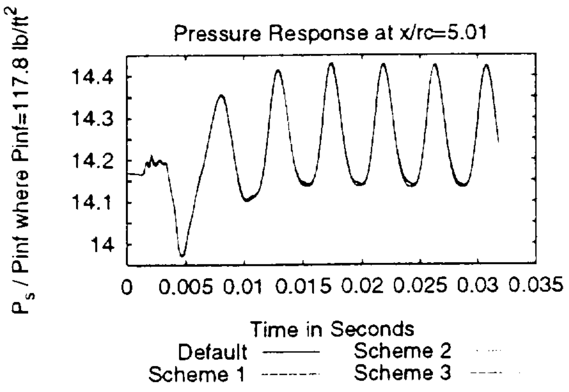


Figure 15. Pressure response at station 10 (X/Rc=5.01) to a 5% sinusoidal inlet temperature perturbation.

Conclusions

The 2D/axisymmetric and 3D versions of NPARC have been parallelized and enabled for dynamic load balancing. A variable time-stepping block solution algorithm is implemented in addition to various communication schemes and their efficiency is explored with the help of three test cases. The combination of the variable time-stepping approach and the communication schemes are shown to be time accurate for unsteady computations. Significant savings in total elapsed time can be achieved with the developed variable time-stepping schemes when the interface time-steps and characteristic speeds are considered. The dynamic load balancing provides additional efficiency when the problem size increases. The variable time-stepping tools introduced here can significantly reduce the cost of solving unsteady perturbation problems with NPARC codes. The reduction in total elapsed times is 4-5 times than in constant time-stepping algorithm when large size problems are solved.

Acknowledgments

This research was funded by the NASA Lewis Research Center under NAG3-1577. The authors appreciate the support provided by Rich Blech, Gary Cole and Joongkee Chung of Computational Technologies Branch of NASA LeRC.

References

[1] G.K. Cooper and J.R. Sirbaugh, "The PARC Code: Theory and Usage," *Arnold Engineering Development Center TR-89-15*, 1989.

[2] H.U. Akay, R.A. Blech, A. Ecer, D. Ercoskun, B. Kemle, A. Quealy and A. Williams, "A Database Management System for Parallel Processing of CFD Algorithms," Parallel CFD '92," Edited by R.B. Pelz, et al., Elsevier, Amsterdam, pp. 9-23, 1993.

[3] Y.P. Chien, A. Ecer, H.U. Akay, F. Carpenter and R.A. Blech, "Dynamic Load Balancing on a Network of Workstations for Solving Computational Fluid Dynamics Problems," Computer Methods in Applied Mechanics and Engineering, vol. 199, pp. 17-33, 1994.

[4] N. Gopalaswamy, Y.P. Chien, A. Ecer, H.U. Akay, R.A. Blech and G.L. Cole, "An Investigation of Load Balancing Strategies for CFD Applications on Parallel Computers," Parallel CFD '95, June 26 - 29, 1995, Pasadena, California, U.S.A.

[5] H.U. Akay and A. Ecer, "Efficiency Considerations for Explicit CFD Solvers on Parallel Computers," *Proceedings of the International Workshop on Solution Techniques for Large-Scale CFD Problems*, Montreal, Canada, pp. 289-314, September 26-28, 1994.

[6] J. Chung and G.L. Cole, "Comparison of Compressor Face Boundary Conditions for Unsteady CFD Simulations of Supersonic Inlets," *NASA Technical Memorandum 107194*, ICOMP 96-5, March 1996.

[7] J.K. Chung, "Numerical Solution of a Mixed Compression Supersonic Inlet Flow," AIAA Paper No. 940583, 32nd Aerospace Sciences Meeting, Reno, Nevada, 1994. .

Sixth International Symposium on Computational Fluid Dynamics, September 4-8, 1995, Lake Tahoe, Nevada, USA

Load Balancing Issues in Parallel Computing

A. Ecer*, H.U. Akay*, Y.P. Chien** and N. Gopalaswamy*

*Department of Mechanical Engineering
**Department of Electrical Engineering
Purdue School of Engineering and Technology, IUPUI
Indianapolis, Indiana - USA

INTRODUCTION

Parallel computation of CFD problems involves utilization of many processors to solve a single problem. The efficiency of a parallel scheme generally depends on allocating the data on individual processors and managing the communication in an efficient manner since one has to be aware of both computation and communication costs. The problem is usually simplified into an homogenous form by assuming that the operations on each processor are identical and the load is distributed evenly among identical processors. In computational fluid dynamics, this is similar to solving a problem on a square grid where the difference operator, the solution scheme, the grid size and the machines are the same for all processes. In this case, the load balancing problem is equivalent to dividing a given problem into a given number of equal tasks. In the solution of complex three-dimensional problems, however, the issues are quite different. The grid spacing around an aircraft may vary several orders of magnitude with appropriate stretching. To pick up a boundary layer or the leading edge separation, much finer grids may be required in comparison with the inviscid freestream. The stability requirements for computing with such grids may vary considerably over the entire flow field. Time-accurate solutions of such problems also require a wide range of time-integration steps since the unsteadiness may vary both in time and space. When the problem is described in the above fashion, the definition of parallel computing has to be generalized since the allocation of the data to individual processors depends on the resources available on each processor, as well as the level of computations required for the particular subset of data. If the subset is defined as a collection of grid points, the local refinement of the grid and the local characteristics of the flow dictate the allocation of such data to an individual processor.

While for serial algorithms, the elapsed time is simply a summation of all computational costs, for parallel algorithms the elapsed time is controlled by the *bottlenecks* due to information exchange between the processes. Thus, the efficiency of an algorithm strongly relates to detecting and eliminating bottlenecks. For this reason, load balancing becomes critical for solving large CFD problems. In order to propose solutions to these problems, the present authors have devised a dynamic load balancing technique which dynamically takes into account: 1) computational effort in each processor, 2) inter-communication loads, 3) presence of other users on each processor, and then periodically redistributes the loads for better efficiency as needed [1,2]. In this paper, we summarize our recent experiences with the coupling of explicit CFD algorithms and a dynamic load balancing strategy on network of computers.

PARALLEL CFD ALGORITHMS

For the parallel CFD algorithms we have studied so far, the computational domain is divided into a number of subdomains called *solution blocks* [3,4]. Each block consists of a set of grid points and their connectivity. Also, each block is associated with the neighboring blocks through a group of grid points called *interfaces*. An interface includes all the grid points required to define the connection of two neighboring blocks. An interface is duplicated and stored on both processors where the two neighboring blocks are stored (Figure 1). For pseudo time-integrations of the nonlinear set of equations in steady flows or real-time integrations in unsteady flows, the solution blocks are solved using an explicit scheme. Any computations on a block are communicated to its interfaces. An interface decides when to communicate with its identical twin on the other block. When an interface receives information from its twin, it updates the block it is attached to. Thus, during the computation process two basic decisions are made: 1) when to compute in each block and 2) when to transfer data from one interface to its twin. In general, each block and interface will have different requirements depending on the local flow conditions and grid refinement.

The algorithms thus described are very suitable for parallel computations on distributed multi-user systems such as workstation networks. For parallelization we have developed a grid-based parallel database program, GPAR [3,4], which utilizes portable parallel library routines such as PVM [5] and APPL [6]. Using GPAR, a CFD user-programmer needs to code only a *block solver* and an *interface solver* without being concerned with parallel computing primitives such as *send, receive, wait*, etc. This database program and its applications were presented elsewhere, see e.g., [3,4,7]. Depending on the size of the problem and the availability of computers, the solution blocks are typically distributed to several processors on the network. Each solution block is treated as a separate process while each processor may handle one or more of such processes.

Our experience with such systems has shown that the total elapsed time for these calculations is a function of:

1. Size of each solution block.
2. Size and number of interfaces.
3. Balance in size of solution blocks and interfaces.
4. Number of times the exchange of interface information is needed.
5. Speed and memory of each machine and non-heterogeneity of the system.
6. Change of loading on each machine at a given time.

When studied in detail, it becomes apparent that the above problem is not static. The computer resources may vary over a computer run of many hours. Also, the computational requirements for a block may change due to changes in local flow conditions.

DYNAMIC LOAD BALANCING

Although balancing the size of solution blocks and interfaces is usually under the control of a user, for complicated geometries this may not be readily achieved and may require extra amount of effort. What is not at user's control in multi-user/multi-task environments is the change of loading on each machine during executions. To alleviate such problems, we developed a high-level load balancer which is intrinsically connected to the database program GPAR and the corresponding CFD application code. The load balancer computes the computational cost of block and interface solvers, including the communication costs, and distributes the load into available computers. It also periodically checks the loading of each processor and redistributes the loads if significant load unbalances are detected during the parallel computations due to change in loading status of processors [1,2]. The following steps are to be performed when a parallel CFD code is used with the present dynamic load balancing algorithm:

1. A computational grid is generated in the form of blocks and interfaces and stored in the database.
2. Each block is assigned to a *block solver* which solves the equations for each block and also updates its interfaces.
3. Each interface is assigned to an *interface solver* which sends the information to its twin interface which belongs to the neighboring block (Figure 1).
4. Blocks are distributed among the existing processors along with their respective interfaces.
5. Program is executed and computation time of each block and execution time of each interface are recorded.
6. Based on the recorded data, a load balancing is performed to distribute the given problem to available processors in a most efficient manner.
7. Steps 4 through 7 are repeated periodically to include the changes in the problem, the solution algorithm and computer conditions.

NUMERICAL INTEGRATIONS IN TIME

In this paper, an explicit time integration technique is chosen to demonstrate the concept of load balancing. The stability requirements of such schemes are usually defined in terms of a CFL number. For example, for the scheme to be stable, the limiting time step is directly proportional to the element size and inversely proportional to the local velocity. Hence, the flow regions with denser grid distributions and higher velocities are severely penalized. This severe restriction makes the solution of large problems prohibitively time-consuming even after parallelization. However, it is possible to further improve the computational efficiency by exploiting the parallel data structure of the proposed algorithms as described in the following sections.

Block-Based Variable Time-Stepping Strategies

If a group of grid points is identified as a block, the CFL condition (i.e., Courant number) suggests that the time integration step for that block is dictated by the grid point with the highest CFL number in that block. As we divide the entire grid into a larger number of blocks, we have the opportunity to utilize the most efficient time step for each region. For instance, we do not wish the leading edge of an airfoil to dictate the integration time step for the entire problem. The flow regions with denser grid distributions and high velocities are severely penalized. Although increasing the number of blocks decreases the *block solver* times, it increases the relative importance of communication times. To overcome this difficulty, we proposed using time-steps which vary with time based on a rule in each block independently to meet the condition set by the CFL number. While the blocks advance in time with different time steps decided by the Courant number, interface information exchange is made whenever needed and the missing information is calculated by linear interpolations within a time step. The rule used in selecting the time steps is based on using a minimum preset value Δt_{min} and an integer k such that the time step used in each block m at a time step n is calculated from:

$$\Delta \bar{t}_m^n = k \Delta t_{min} \leq \Delta t_m^n$$

where, in each block the variable time step Δt_m^n is calculated from the CFL condition. An upper limit on the integer multiplier k is needed (e.g., 5) to minimize the interpolation errors at the interfaces. Exchange of interface information selectively only when needed, instead of at every Δt_{min}, significantly improves the efficiency of overall calculations.

Zonal Solution Strategies

One may also use a zonal approach for which a complete Navier-Stokes solver is used only at selected flow regions for efficiency purposes. Some blocks may be treated as inviscid while others as viscous. Thus, solution time for each block may not only be a function of number of grid points but also the solution algorithm utilized for the specific block. Based on the above considerations, one can define a time step locally for each block and solver for improving efficiency. Such a procedure may also be extended to zones with potential, Euler and Navier-Stokes solvers combined with the load balancer.

Interface-Based Variable Communication Strategies

Since the communication cost is still the critical factor in parallel computing, one can obtain considerable efficiency by selectively sending the interface information based on the direction of the flow at the interfaces. For instance, if the flow is supersonic the upstream block sends messages to downstream but does not need information from the downstream block. When the flow is subsonic, the speed and hence the frequency of information exchange are different in upstream and downstream directions. This way, one can optimize communication costs by studying the flow conditions and grids at the interfaces. Again this process is dynamic and depends on the local flow conditions.

BENCHMARK STUDIES

The problem considered in this paper is the flow through an axisymmetric engine inlet as shown in Figure 2 (see e.g., Chung [8]), where we divided the flow region into seventeen blocks. Each block contains between 8,000 and 10,000 grid points. The flow is supersonic in most regions except in blocks 9-11. The PARC2D unsteady flow code [9], which was parallelized via GPAR, was used for the test cases.

Example 1 Load Balancing

This case illustrates the basic functions of the load balancing program. Shown in Figure 3 is a typical load balancing sequence which may occur in a multi-user heterogeneous environment. Initially seventeen blocks were distributed among four machines. The loads were monitored by the load balancing program periodically at each cycle of computations, where one cycle in this case is equal to 800 time steps of unsteady integrations. As may be observed, a sudden change in the loading of one of the machines was detected at seventh balancing cycle, after which the load balancer redistributed the loads for a better performance. The new distribution was dynamically determined from the measured computational and communication costs and the cost calculation formulae of the balancing program. Similar situations happen at later cycles too. Each time the load balancer program corrects the problem.

Example 2 Zonal Approach

This case illustrates the effects of zonal approach where certain blocks in the flow zone are switched from an Euler to Navier-Stokes solver (blocks 12-17). As may be observed from Figure 4, following the switch of the solvers at balancing cycle 6, the load balancer improves the efficiency by redistributing the solution blocks.

Example 3 Interface-Based Variable Communication

This case illustrates the obtained savings in elapsed time when communications are made selectively at supersonic interfaces. Figure 6 shows the savings when the information in supersonic blocks is passed only from upstream to downstream direction after the balancing cycle 5. It is to be noted that an unexpected load increase on the system which happened at cycle thirteen was later corrected by the load balancer.

Example 4 Block-Based Variable Time-Stepping

This case illustrates the effects of the block-based variable time-stepping algorithm and communication costs. Shown in Figure 6 are the efficiency curves of variable time-stepping algorithm compared with the constant time-stepping. While there is a severe drop in efficiency after seven machines are used in the case of constant time-stepping, the same drop in efficiency takes place only after fourteen machines when the variable time-stepping plus variable communication algorithms are used.

ACKNOWLEDGMENTS

This research was supported by NASA Lewis Research Center under grant NAG3-1577. Computer accesses provided by NASA and IBM are gratefully acknowledged. The authors thank S. Secer of IUPUI's CFD laboratory for his assistance on the implementation of the load balancing algorithm.

REFERENCES

1. Y.P. Chien, A. Ecer, H.U. Akay, F. Carpenter and R.A. Blech, "Dynamic Load Balancing on a Network of Workstations for Solving Computational Fluid Dynamics Problems", *Computer Methods in Applied Mechanics and Engineering*, vol. 119, pp. 17-33, 1994.
2. Y.P. Chien, A. Ecer, H.U. Akay and R.A. Blech, "Environment Requirements for Using Dynamic Load Balancing in Parallel Computations," *Proceedings of Parallel CFD '94*, Edited by A. Ecer, et al., Elsevier, Amsterdam, 1995.
3. H.U. Akay, R. Blech, A. Ecer, D. Ercoskun, B. Kemle, A. Quealy and A. Williams, "A Database Management System for Parallel Processing of CFD Algorithms," *Parallel CFD '92*, Edited by R.B. Pelz, et al., Elsevier, Amsterdam, pp. 9-23, 1993.
4. A. Ecer, H.U. Akay, W.B. Kemle, H. Wang, D. Ercoskun and E.J. Hall, "Parallel Computation of Fluid Dynamics Problems," *Computer Methods in Applied Mechanics and Engineering*, vol. 112, pp. 91-108, 1993.
5. G.A. Geist, A.L. Beguelin, J.J. Dongarra, W. Jiang, R. Manchek and V. Sunderam, "PVM 3 User's Guide and Reference Manual," *Oak Ridge National Laboratory, ORNL/TM-12187*, 1993.
6. A. Quealy, G.L. Cole and R.A. Blech, "Portable Programming on Parallel/Networked Computers Using Application Portable Library (APPL)," *NASA Technical Memorandum, 106238*, 1993.
7. H.U. Akay and A. Ecer, "Efficiency Considerations for Explicit CFD Solvers on Parallel Computers", *Proceedings of the International Workshop on Solution Techniques for Large-Scale CFD Problems*, Montreal, pp. 289-314, September 26-28, 1994.
8. J.K. Chung, "Numerical Solution of a Mixed-Compression Supersonic Inlet Flow," *AIAA Paper No: 940583, 32nd Aerospace Sciences Meeting*, Reno, Nevada, 1994.
9. G.K. Cooper and J.R. Sirbaugh, "The PARC Code: Theory and Usage," *Arnold Engineering Development Center, TR-89-15*, 1989.

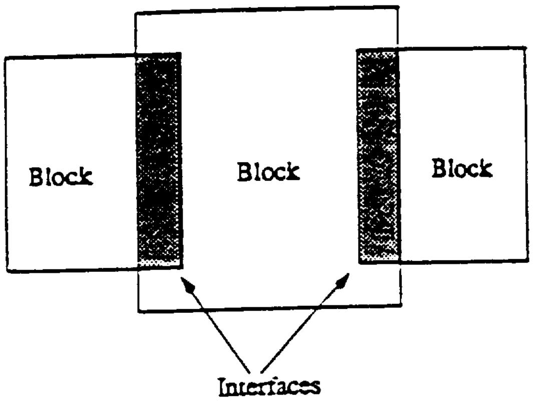


Figure 1. Blocks and overlapped interfaces.

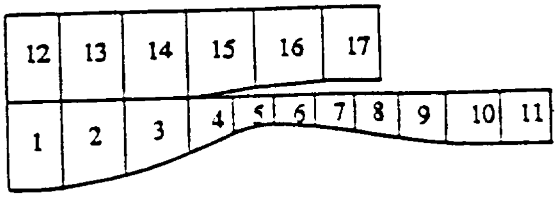


Figure 2. Block structure used for an engine inlet problem.

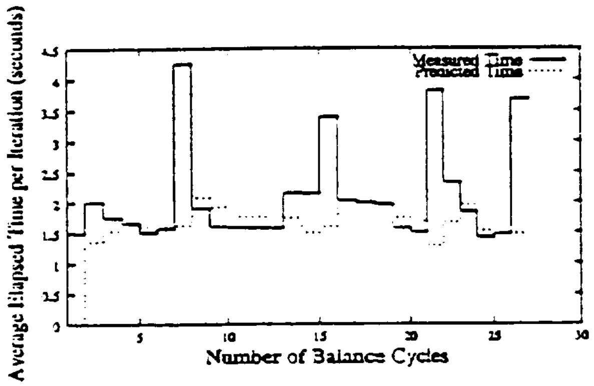


Figure 3. Load balancing in a multi-user and heterogeneous environment.

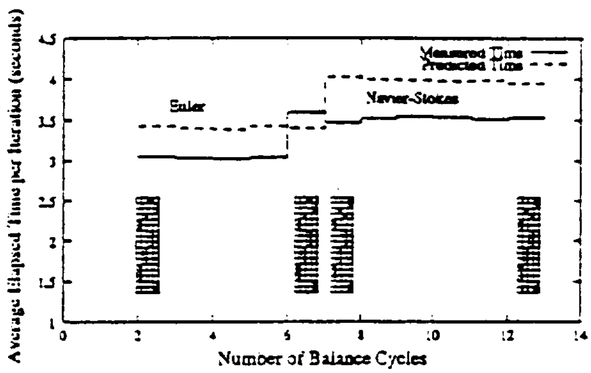


Figure 4. Zonal solvers coupled with load balancing.

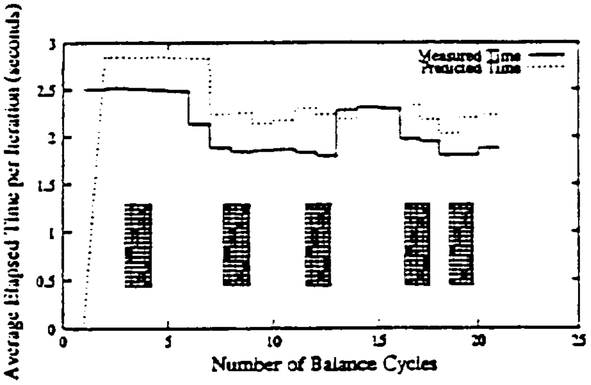


Figure 5. Interface-based variable communication coupled with load balancing.

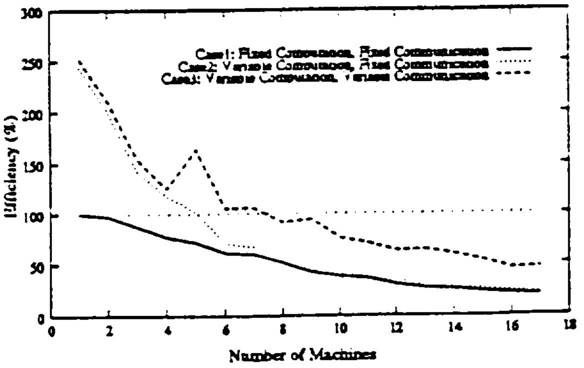


Figure 6. Effect of variable time-stepping and variable interface communication algorithms.

Reprinted from

Computer methods in applied mechanics and engineering

Comput. Methods Appl. Mech. Engrg. 119 (1994) 17–33

Dynamic load balancing on a network of workstations for solving computational fluid dynamics problems

Y.P. Chien[a,*], A. Ecer[a], H.U. Akay[a], F. Carpenter[a], R.A. Blech[b]

[a] *Purdue University, School of Engineering and Technology at Indianapolis, 723 W. Michigan Street, Indianapolis, IN 46202, USA*

[b] *NASA Lewis Research Center, Cleveland, OH 44135, USA*

Received 24 November 1993
Revised manuscript received 3 March 1994

ELSEVIER

Comput. Methods Appl. Mech. Engrg. 119 (1994) 17–33

Computer methods in applied mechanics and engineering

Dynamic load balancing on a network of workstations for solving computational fluid dynamics problems

Y.P. Chien[a,*], A. Ecer[a], H.U. Akay[a], F. Carpenter[a], R.A. Blech[b]

[a] *Purdue University, School of Engineering and Technology at Indianapolis, 723 W. Michigan Street, Indianapolis, IN 46202, USA*

[b] *NASA Lewis Research Center, Cleveland, OH 44135, USA*

Received 24 November 1993
Revised manuscript received 3 March 1994

Abstract

Distributed computing on a network of computer workstations is being considered as a practical tool for parallel CFD applications. Presently, workstations are commonly arranged in the dedicated, single-user mode for executing such computations. Since workstations are generally employed in a multi-user environment, running the workstations in the dedicated mode causes scheduling problems for system administrators and inconvenience to other users. A methodology is presented in this paper for dynamic balancing of the computation load on a network of multi-user computers for parallel computing applications. In order to distribute the computation load in a multi-user environment, it becomes necessary to determine the effective speed of a multi-user workstation to a parallel application. In the present approach, it was assumed that (i) multi-user and multi-tasking networked computers may have different computation speeds, (ii) application data can be divided into many small data blocks with possibly different sizes, (iii) a process is assigned to each block, and (iv) the number of computers is much less than the number of processes. The developed dynamic load balancing procedure uses the greedy method for optimizing computation load distribution. Due to dynamic changes of the computer loads in a multi-user and multi-tasking environment, the loads on computers are periodically examined and parallel application processes may be re-distributed to reduce the computation time. The developed method has been tested on two computer clusters and its applicability has been demonstrated for two case studies.

1. Introduction

Solution of large CFD problems requires access to large computer systems. In the past, supercomputers were utilized to solve such problems where vectorizing was the main tool for speed improvements. Presently, parallel computers are being considered to treat such problems in terms of obtaining higher computational speeds and solving larger problems. The development of parallel computers during the last decade has progressed mostly towards developing tightly coupled systems. Whether a parallel computer is configured as a SIMD or MIMD, an expandable, yet fixed configuration, was proposed. This resulted in the development of computers with many processors which communicate with each other in a prescribed fashion [1, 2]. These developments have been mostly of an experimental nature and parallel supercomputers are only been realized during that last couple of years [3]. Access to 512 or 1024 processors are being made possible to researchers to solve large CFD problems. The term 'massively parallel' is being realized as such systems are being assembled.

After experimenting with the present parallel supercomputers, one can make several observations:

- These computers have been developed up to a level exceeding the performance of older

[*] Corresponding author.

SSDI 0045-7825(94)00074-3

supercomputers. It seems possible that much larger systems can be put together at a reduced cost in the near future.

- Although attempts are being made to provide a virtual environment where the communication between different processors is not visible to the user, the communication cost is still an important factor for most of the applications.
- Many existing software packages are not suitable to work in a parallel environment. Usually it is costly to convert or there is not sufficient interest to justify the cost of conversion.
- Since these machines are still at the development stage, changes are being made rather rapidly. Therefore, only a few specific production codes are running on these machines at this time.

Also, during the last decade the development of UNIX workstations has attracted considerable attention. A large amount of scientific computing previously performed on main frames has been shifted to workstations. The wide popularity of such hardware has driven the costs down. Many organizations have already purchased a number of workstations which have brought forward the possibility of a network of workstations as a cost effective means to parallel computing. The use of distributed workstations for parallel computing has drawn significant interest from the research community, mainly due to the potential for high performance. It has also drawn interest from the management community, which looks to this new technology as a means to significantly reduce computing costs. These different objectives are causing some confusion.

The 'performance' seekers are driving the dedicated cluster approach. This has promoted investigations into more efficient communication software and high performance networks. Performance seekers will try anything to acquire more computing power. They will even write their own load-balancing schemes (static or dynamic) into their codes. The 'efficiency' seekers are driving the scheduling/load balancing software development. This software is meant to keep as many machines as possible as busy as possible [4]. Traditionally, this has been done through scheduling multiple single-processor jobs. The situation is complicated with the addition of parallel jobs. Scheduling/load balancing software primarily meant to 'capture idle cycles' could conflict with applications developed to achieve high performance. However, some of the load balancing techniques built into this software (task migration, checkpointing) could be useful to applications seeking performance. The key is to have schedulers/load balancers which are flexible enough to recognize and support both situations. The distributed network could be viewed as multiple 'clusters', where a cluster could consist of only a single workstation or multiple workstations.

A network of loosely-coupled, multi-user workstations for solving large problems requires answers to further questions. If one compares a network of loosely coupled workstations to existing parallel machines, one can make the following observations:

- A user can access to much larger memory on the existing workstations (256 to 512 Mbytes per processor).
- The communication between the workstations is still being improved at this time.
- A system of loosely-coupled networked workstations has more possibilities in terms of expandability, yet it is much more difficult to schedule and load balance parallel applications than a tightly-coupled parallel machine.
- A system of loosely coupled networked workstations is dynamic. The number of available workstations and their load may change day-to-day.
- The network of workstations is suitable for a multi-user environment. The variety of resources on such a system enables efficient utilization by several users simultaneously. This is quite different to users sharing a single computer which was the supercomputing environment of the last two decades.
- Software development on networked workstations prevents the software package from becoming machine dependent. The present parallel supercomputers require specific software tools for improving the efficiency of their particular systems.

Based on the considerations listed above, our work on parallel computing has been directed towards the utilization of a network of loosely coupled workstations. We consider a network of multi-user UNIX workstations as our basic system. For solving large CFD problems on such a system, we try to answer the following questions:

- How can we distribute a large CFD problem over a network when sharing resources with other users?
- How can we utilize a network of heterogenous UNIX workstations with different brands and models?
- How do we develop parallel algorithms and computer codes without knowing the details of such a complicated computer network?
- How can we maintain 'high performance computing' in such an environment?

In this paper, we describe a dynamic computer load balancing methodology suitable for a network of loosely-coupled workstations. In order to maximize the utilization of computing power of the network, we assume that the network supports the multi-user environment. In order to support the load balancing tasks for a variety of parallel, portable CFD application codes, we do not require the detailed knowledge of the parallel code for load balancing. The dynamic load balancing is based on the on-line performance measurements of a given CFD code on existing network of heterogenous workstations. By utilizing the developed methodology, one can ensure the scalability, portability and the efficiency of a parallel algorithm on a given network.

2. Background

One can develop parallel CFD algorithms by parallelizing the access to data at different levels. Our experience with MIMD machines has been based on parallelizing the CFD algorithm and duplicating the same algorithm on different processors [5]. In parallel CFD, one simple strategy is to divide the computational grid into a series of blocks and perform parallel computations on each of these blocks. Again, a simple approach is to divide the data into the same number of blocks as the number of computers or processors used for processing such data. Examples of blocking the data to fit a given number of processors can be found in literature [6–8]. Computer load balancing using these methods is achieved by varying the sizes of the data blocks. These methods simplify the load balancing problem by assuming that there are no restrictions on how the data can be divided into blocks and the computing environment is static. However, they may become complicated when there are restrictions imposed on data blocking and the computers are in the multi-user mode.

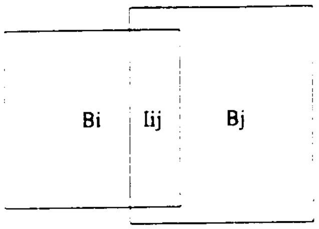
To develop a general yet efficient computational environment for parallel CFD on a network of multi-user workstations, we proposed to arrange the data into a large number of data blocks where each block corresponds to an assembly of grid points. We first developed the methodology for managing such data efficiently on a network [9]. We then defined load balancing in terms of optimum allocation of these blocks to different processors where the number of blocks exceeds the number of processors [10]. In this paper, we extend this discussion to dynamic load balancing. To introduce further details of the procedure, we formulate the problem based on the following assumptions:

(1) A set of m multi-tasking, multi-user networked computers are used.

(2) Computation speeds of computers may be different.

(3) There is a program (grid divider) to divide the original data into a set of n small data blocks $D = \{d_i \mid i = 1, \ldots, n\}$, where d_i is data block i and $n > m$. The data can be cut into blocks with preferred sizes and geometry. Each data block is associated with the description of the shape of the block, the number of nodes and elements in the block, the number of interfaces of the block (see Fig. 1), the block numbers of its neighboring blocks, and the data to be exchanged with its neighboring blocks. (Usually, this grid divider is executed only once in the beginning. One can later combine two small blocks into one. This is much simpler than further dividing blocks into smaller pieces.)

(4) The parallel CFD algorithm is characterized by two components: computations for each block and communications between neighboring block interfaces. Block computation component includes the computation instructions for all the grid points in a single block while the block interface communication component consists of the instructions for interface data communication and processing. The computation time used for a CFD problem depends on the complexity of the computational component of the CFD code and the number of grid points in the data. The



Bi = Solution domain of i-th block

Iij = Interface between blocks i and j

Fig. 1. Definition of blocks and interfaces in the solution domain.

communication time for an interface depends on the complexity of the communication component and the amount of information to be exchanged between the neighboring blocks.

(5) Since the effective computational speed of a computer to a user in a multi-user environment changes dynamically, processing time for the same data on different computers and communication time between different pairs of computers vary with time.

(6) In optimal computer load balancing, the cost is represented in terms of the total time elapsed during the program execution.

We define the following parameters to describe the cost of computing:

(1) The computation cost for processing of data block a on computer j be c^j_a (subscript denotes the data block number, superscript denotes the computer number).

(2) The communication cost for sending all required information of adjacent data blocks from computer j to computer k be u^{jk}.

(3) The computation of a data block cannot be completed until the interface data from adjacent blocks are obtained. The cost of using computer j to process all data blocks on computer j is $C^j = \sum (c^j_a + u^{jk}_a + W_a)$ for all data blocks d_a on computer j, where u^{jk}_a is the cost of collecting required data from all computer k to computer j in order to process d_a, $1 \leq k \leq m$, and W_a is the elapsed time during which block a is waiting for interface data from adjacent blocks to become available.

Hence, the optimal load distribution task for parallel computing is to minimize the maximum of the execution costs for all computers. This is equivalent to the following statement

$$\text{minimize } C = \max(C^j) \quad \text{for all } 1 \leq j \leq m .$$

When a network of computers are in the dedicated mode (single user mode), the cost functions reflect the hardware specifications of the computer and is static. When computers operate in a multi-user mode, the cost functions to a specific problem change dynamically depending on the extraneous load on the computers.

We have previously reported the development of a static computer load balancing method [10] based on the greedy algorithm [11] for solving parallel CFD problems on a dedicated network of workstations. Before describing the extension of this static load balancing method to the dynamic load balancing in a multi-user environment, we first summarize the method. In static load balancing, we first find the computation and communication cost functions (measured CPU time used for computations with respect to the number of nodes in a block processed per time step) based on several trial executions of the code. These time measurements can be easily implemented once a CFD code is expressed as a combination of block and interface solvers (shown by time stamps in Fig. 2). Computations for the grid points occupying a block is performed inside the block solver. All communications between the neighboring blocks are in the interface solver. The static load balancing method is used to direct the simulated movement of data blocks among the computers until the cost cannot be reduced further. The block diagram of the static load balancing procedure is depicted in Fig. 3. This procedure generates a near minimum cost load distrubtion on all computers. The computational cost of the static load

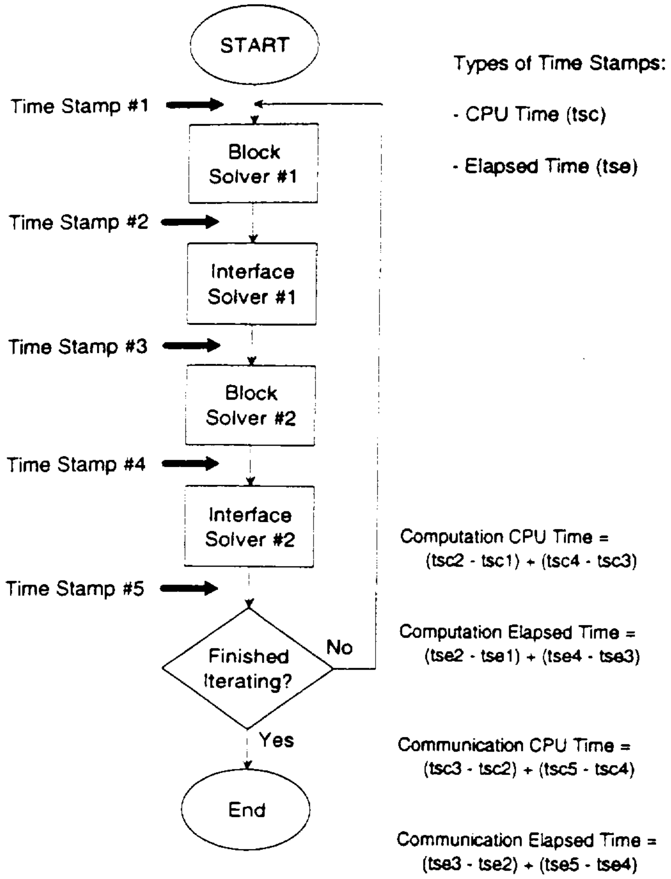


Fig. 2. Flow chart of FLOW3P with time stamps.

balancing method for both the best case and worst case situation is proportional to mn^2, where n is the number of blocks and m is the number of computers. After the method generates a balanced block distribution, the block and interface data are distributed accordingly and the CFD code is executed.

3. Dynamic computer load balancing

In a multi-user environment, computer load can change dynamically since other users can start new processes anytime. Consequently, the effective computational speed of a computer to a user changes dynamically. In this case, it becomes unsatisfactory to rely on a static load balancing algorithm. Fig. 4 shows the variation of the CFD code execution time on a initially statically load balanced network of workstations due to the load change on only one of the workstations. An unbalanced load distribution on computers causes the processing time of certain blocks to be much longer than that of the other blocks on other computers. Since the solution time for the entire problem depends on the slowest process, the computation time can increase drastically whenever the loads are not balanced. It is obvious that we need to periodically examine the progress of the code execution and re-distribute the data blocks if necessary. In order to do so, we have implemented a dynamic load balancing loop which contains the following four steps:

(1) Obtain reliable computational cost information periodically during the code execution.
(2) Obtain reliable communication cost information periodically during the code execution.
(3) Determine the cost functions based on the collected cost information.
(4) Re-distribute data blocks to computers to achieve load balancing.

In the following, the implementation of these steps are described.

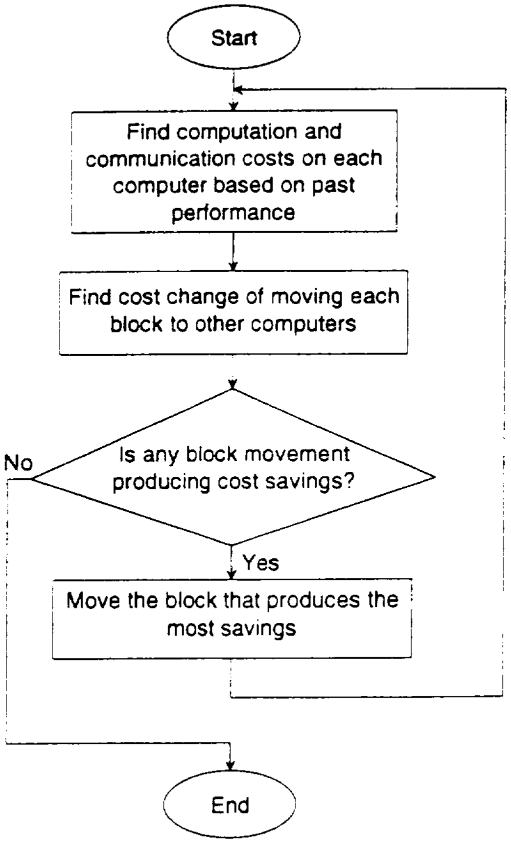

Start
Find computation and communication costs on each computer based on past performance
Find cost change of moving each block to other computers
Is any block movement producing cost savings?
No
Yes
Move the block that produces the most savings
End

Fig. 3. Block diagram of the dynamic load balancing algorithm.

3.1. Determination of the dynamic computation cost function

In a dynamic environment, the computational cost of solving a given number of blocks on computer j, C^j, is a function of four parameters: (i) the computational complexity of the algorithm, (ii) the speed of the computer, (iii) the total number of grid points processed by the computer, and (iv) the total number of active processes on that computer. Since the time complexity analysis of a CFD program only provides a loose relationship between the number of grids points and the computation time, it does not provide an accurate timing information. Besides, it becomes difficult to gather the speed information for a variety of computers used for executing different size blocks. To avoid calculating the computational cost based on the complexity of the algorithm and on unreliable computer speed

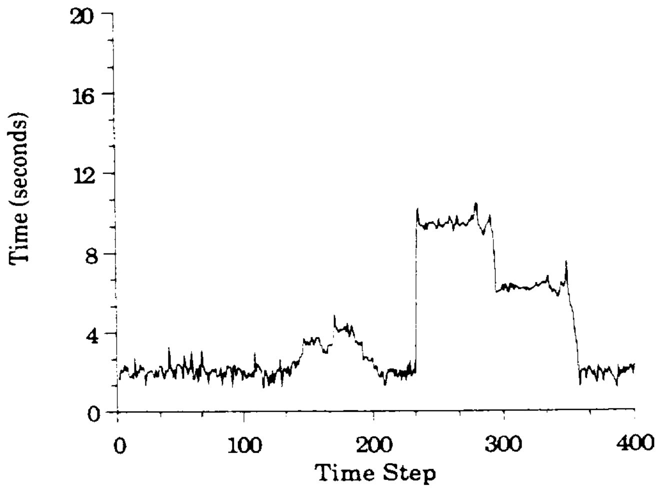


Fig. 4. Variation of total elapsed time for a statically load balanced case for FLOW3P.

information. we calculate the computational cost functions for all computers periodically. based on the timing data measured during the execution of the CFD code on the existing system.

In the static load balancing method. we obtained the computation cost function by directly interpolating the measured CPU time per time step for each data block (Fig. 5). However. this approach is not appropriate for multi-user environments. when there are extraneous processes on the computer. In this case. one has to consider the total number of processes. as well as the CPU time of the process for a given block. We have tried several ways to find a reliable computational cost function for dynamic load balancing for the multi-user environment. Here. we include the failed attempts to our discussion since we believe they also provide useful insight to dynamic load balancing.

The first approach. for obtaining the dynamic computation cost function. was to interpolate the measured elapsed computation time per time step for all data blocks. This approach intuitively appeared to be reasonable. However. we were not able to calculate the computational cost on each computer by simply adding the elapsed time for computing each block. This was due to the execution of dependent parallel processes on the computer network. Fig. 6a shows the performance of six blocks on the slowest processor. In this case. when the block solvers start at the same time since all necessary interface information is already received from the neighboring blocks. Elapsed block solver time is the same when all six processors are running simultaneously. In Fig. 6b. the same information is presented for a fast processor. The elapsed block solver time depends when each block receives the required interface data to start block computations. We abandoned this approach. since we were not able to perform load balancing with a cost function based on elapsed computation time.

The second approach for determining computation cost function was based on finding a relationship in terms of CPU time. We also had to consider the number of concurrent processes on the system. All CPU bound user processes should be waiting for CPU time on the same CPU queue with equal priority on the UNIX system. The share of CPU time for all of the parallel processes of an application is then proportional to the percentage of number of processes for the application in comparison with all of the processes running on that computer. Therefore. the elapsed time used by a single block on a computer can be calculated by multiplying the sum of the measured CPU time for all the blocks by the percentage calculated above. When there are no extraneous processes. elapsed time of a single block is equal to the sum of CPU time measured for all the blocks on the same processor. Several UNIX commands were used to determine the number of total processes on a computer but results were not as expected (e.g. the total number of processes running on the computer was usually less than the known number of blocks on that computer). Based on many trials. we observed that. similar to the first case. when all processes on computers are mutually independent. this approach works well. When the processes are mutually dependent. this approach did not work due to the difficulty in measuring the number of total processes on each computer.

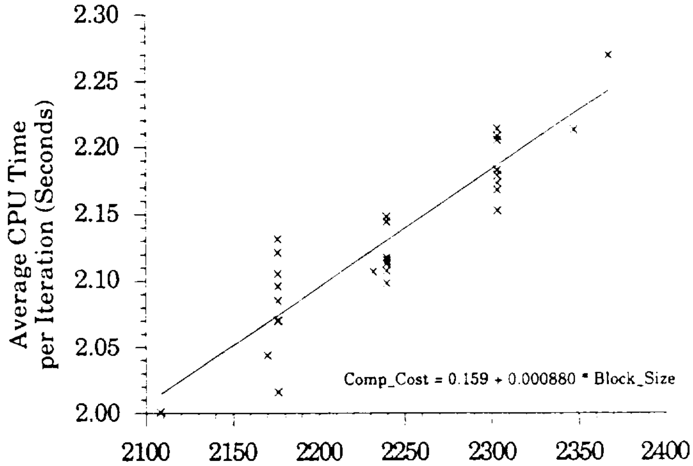


Fig. 5. Approximation of computation cost function from CPU time (65 000) node FLOW3P case on LACE).

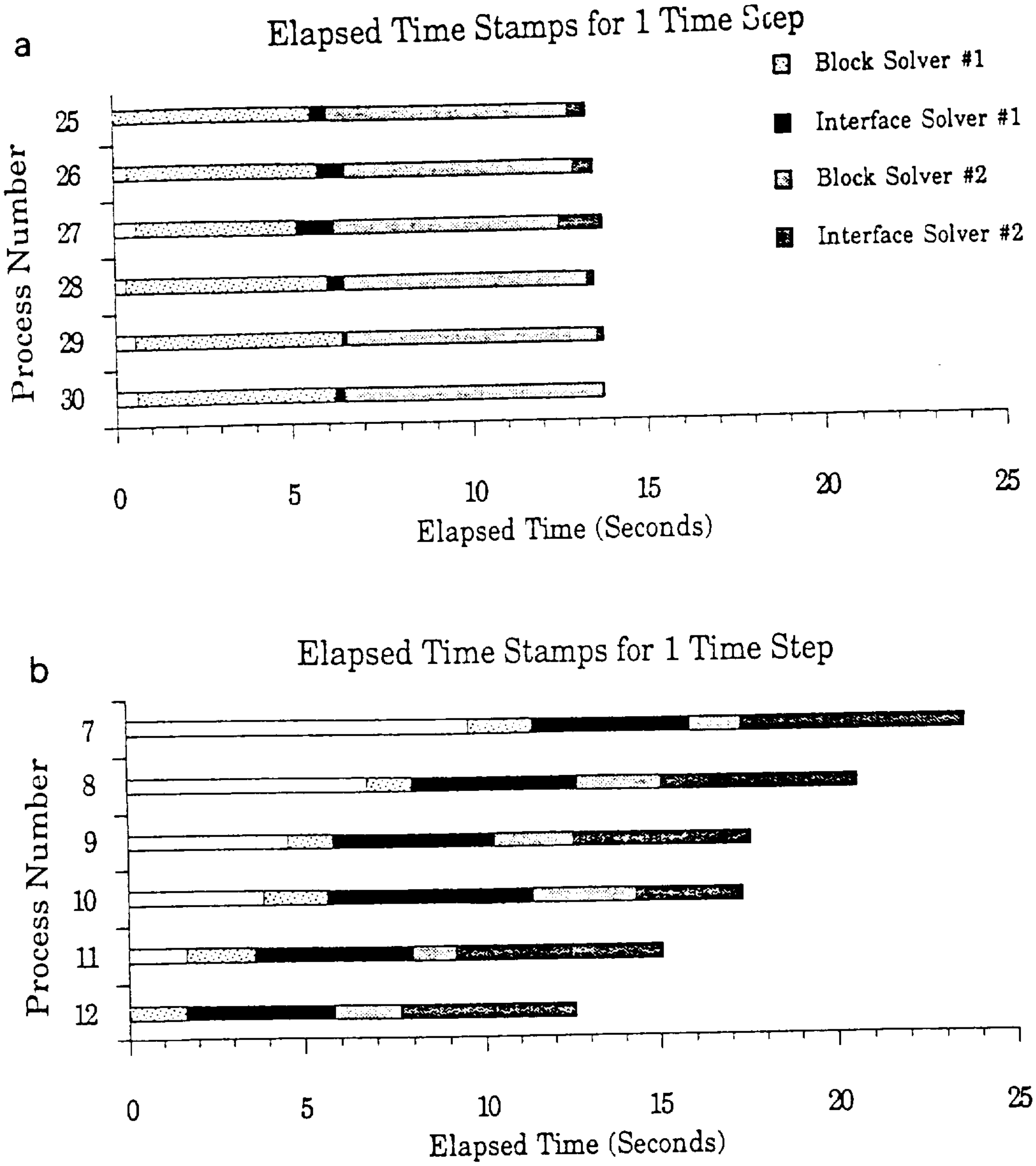


Fig. 6. Time history of two sets of blocks on (a) a heavily-loaded computer and (b) a lightly-loaded computer on LACE.

The third approach was based on the lessons learned from the above experiments. Since parallel computation of many blocks are inherently mutually dependent and since they cause problems in counting the number of processes, we exclude them in counting the number of processes on one computer. We only count the number of independent processes for each computer. A background process, Process_Tracker, which uses UNIX *ps* command, is initiated on each computer after each load re-distribution. Process_Trackers periodically (about 10 seconds in our experiment) count extraneous independent processes (total processes subtract by the number of parallel application processes) on each computer and provide the average number of extraneous processes for all the time steps from the most recent load distribution to the last time step. The total computation cost used by all blocks on a computer j, C^j, can be estimated by

$$C^j = \left(\sum (t^j_{i,\mathrm{CPU}})\right)(N^j_{\mathrm{extr}} + N^j_{\mathrm{app}})/N^j_{\mathrm{app}}$$

where $t^j_{i,\mathrm{CPU}}$ is the measured CPU time for block i on computer j per time step, N^j_{app} is the total number of blocks of a given parallel application on computer j, N^j_{extr} is the average number of extraneous processes on computer j. If there are no extraneous processes, the computation cost of a block becomes equal to the sum of CPU times of all the blocks on that computer. Since some block processes finish earlier than others on the same computer due to differences in block sizes. N'_{app} is calculated as the total number of grid points of all the blocks on computer j divided by the number of grid points of the largest block on computer j. Therefore, the dynamic computation cost function of a computer is the interpolation of the estimated elapsed computation time for all blocks on the computer.

We tested the computation cost function obtained by the third approach and used it as a basis for dynamic load balancing. The calculated computation time as calculated above was found to be an accurate estimate of the real life situation as it will be discussed in Section 4.

3.2. Finding dynamic communication cost function

The dynamic communication cost of each data block depends on the size of the interface information. Since the geometry of every data block is fixed, the total number of interface grid points of each data block is known. Due to the fact that the interface message size is a function of the number of interface grid points, the number of bytes of information to be sent by each interface grid point can be easily determined.

The communication cost is also a function of the speed of the communication network, and the amount of traffic on the computer network. We have tried to use the most recent communication cost measurement to predict the communication cost in the immediate future. One encountered problem was due to the fact that the system clocks of different computers may be quite different, which makes the timing recording for communication cost inaccurate. Since the user cannot adjust the system clock of all computers on the network, we adopted the following procedure to ensure the accuracy of the timing measurements.

(1) Find the difference between the clocks of all computers by sending a round trip message from computer a to computer b and back to a. The message is time stamped each time before it is sent. Let the transmission time for the round trip message be t_{round}, the clock difference between computer a and computer b can be calculated as

$$\text{clock}_{ab} = \text{stamp}_b - (\text{stamp}_a + t_{\text{round}}/2)\,.$$

(2) During each step, each process sends a message with a departure time stamp $t_{\text{departure}}$.

(3) The process which receives the message makes an arrival time stamp t_{arrival}.

(4) The communication cost between the two data blocks is the actual data transmission time t_{ab} which can be calculated by using the following equation

$$t_{ab} = t_{\text{arrival}} - t_{\text{departure}} - \text{clock}_{ab}$$

We experimented with the measurement of communication time on an Ethernet by sending the same message between two computers many times and observed that the measured communication time between two computers may vary over a large range (see Fig. 7). However, for a message under 2K bytes, the average of the measured elapsed communication time per time step on an Ethernet which was not highly loaded was found to be close to a constant. The results of this experiment is as expected since (i) the messages sent between computers through Ethernet are grouped in packets, and (ii) Ethernet assigns the priorities to messages randomly after a collision occurs.

At this time we should note that during our experiments on tightly distributed computational environmental (such as workstations using a common file server: IBM SP1 at Kingston and Cluster of RS/6000 workstations at NASA Lewis Research Center), the communication cost between blocks represented only a small percentage of the cost in terms of the total processing time (less than 2% of the total computation time). One explanation of this small communication cost is that the application is two dimensional so that the amount of interface data between blocks is small. The other explanation is that all nodes use a common file server and are in a local network. Therefore, we can almost ignore the communication costs in such an environment. However, in a more general loosely distributed computational environment (network of independent workstations) with a long communication distance (e.g. literally hundreds of miles away) and for three-dimensional parallel applications, one obviously cannot ignore communication costs.

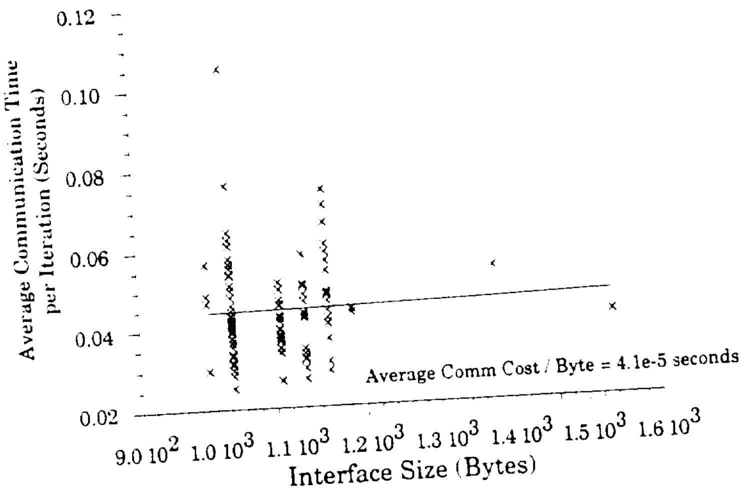


Fig. 7. Approximation of communication cost function (FLOW3P case with 6 blocks per computer; 5 computers used with no load).

3.3. Parallel CFD software environment

In our dynamic computer load balancing experiments, we used a three layer hierarchy built on top of the UNIX environment (see Fig. 8). The lowest layer is the application portable parallel library (APPL) developed at NASA Lewis [12]. APPL provides tools for portability on different distributed computers. The middle layer is a database management library (GPAR) specifically developed for the parallel computation of problems defined by computational grids [13]. GPAR is built on the top of APPL to support managing multi-block grid applications on parallel/distributed computers. GPAR supports structured or unstructured grids within blocks and support different types of block interfaces (matching, non-matching, overlapping, etc.). The highest layer is the CFD application programs. A CFD program FLOW3P has been used as a test bed for our load balancing experiments [5]. The flow chart of this program is shown in Fig. 2. The overall flowchart of the multi-block solver environment with different application programs is shown in Fig. 9. As can be seen from this figure, other portable parallel parallel communication libraries can also be utilized as well as other applications by using GPAR. Grid blocking capabilities and post processing of blocks are additional features of this environment.

The information flow relative to the dynamic computer load balancer is depicted in Fig. 10. The dynamic load balancer acts as a process controller for a given parallel job. Based on heuristic rules or the cost functions obtained in the past execution of the CFD program, the load balancer first distributes

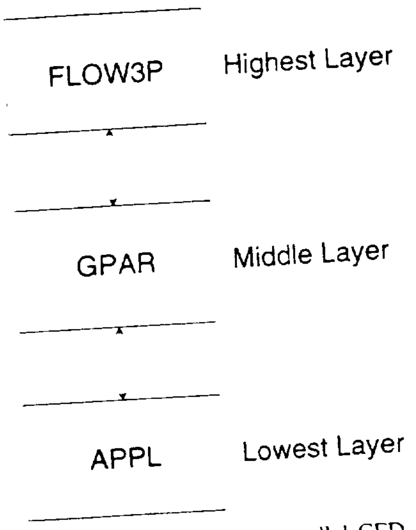


Fig. 8. Hierarchical environment for parallel CFD computations.

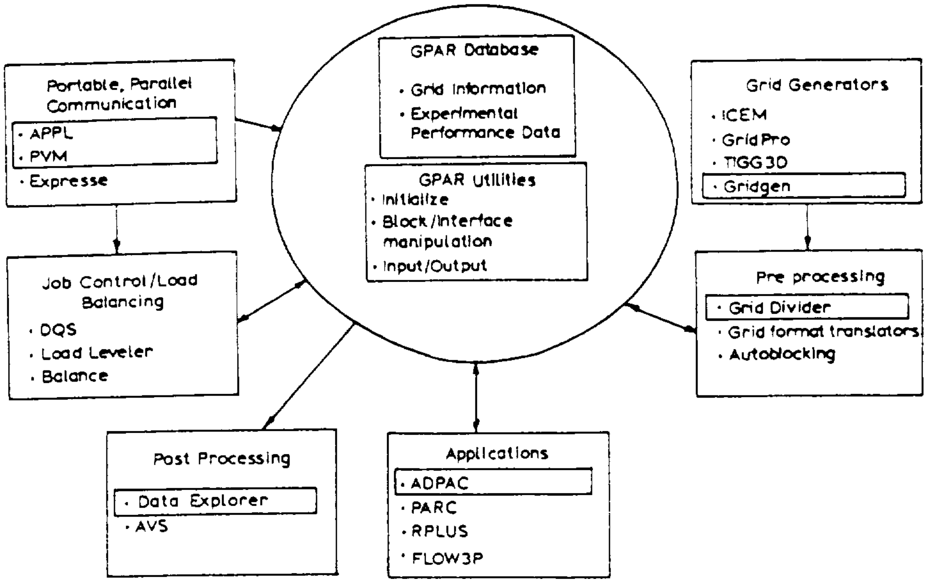


Fig. 9. Distribute multi-block solver tool kit.

block processes to computers through APPL. Then, after every n time steps, the load balancer collects (i) the average computation cost and communication cost of every process in the period from last load distribution to present time, (ii) the extraneous process information of every computer from Process-Trackers, (iii) old data block distribution from APPL, and (iv) data block information and interface data from GPAR. Based on the above information, the load balancer re-distributes the data blocks among the computers.

4. Examples

The following examples demonstrate the applicability of the dynamic load balancing method for solving parallel CFD problems in the distributed computing environment. The first example dem-

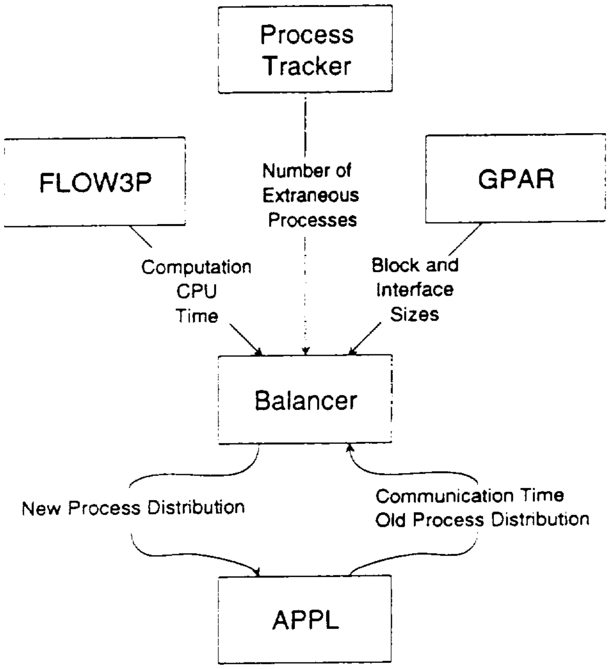


Fig. 10. Information flow for the dynamic load balancer.

onstrates the application of the dynamic load balancer in a controlled environment. In this case, we have the total control of the load distribution on all the computers. The computers used in the experiment were five nodes on an IBM SP1 at IBM Research Center at Kingston, New York. The parallel CFD application program FLOW3P was used in this example. We assumed little knowledge about the details of FLOW3P. Only several time-measuring instructions were added to FLOW3P around the block solver and the interface solver as shown in Fig. 2. The input CFD data was a C-grid with 65 000 nodes (see Fig. 11). This grid was divided into 30 blocks by a grid-dividing program developed for management of parallel grids. The topology of the blocks is depicted in Fig. 12. Numbers on the C-grid indicate block numbers. The sizes of data blocks are listed in Table 1. Thousands of time steps are usually required to obtain the final solution. After each time step, interface data are sent between adjacent blocks via Ethernet.

Based on the assumption that the computers are of the same speed, these thirty data blocks are initially distributed six per computer to five nodes on SP1 for parallel processing. In this experiment, the cost function used for load balancing did not include the communication cost. We forced the dynamic load balancer to rebalance load on the computer for every n time steps, where $n = 13$. Fig. 13 depicts timing for the application code execution in $5n$ time steps. The solid line represents the average elapsed time used for execution per time step. The dashed line represents the estimated time of execution per time step under balanced load distribution. The suggested load distribution at the end of every set of n time steps are listed in Table 2. The integers under each computer number in every n time steps are data block numbers. The extraneous load on each computer measured during each n time step (in terms of number of processes are shown by the floating point numbers) is listed below the suggested load distribution. During the first n time steps, only the processes of the application are loaded on these five computers and no extraneous processes are introduced. Since there was no extraneous load introduced during the n time steps, no load re-distribution was necessary at the end of n time steps. During the second set of n time steps, three independent extraneous processes which contains infinite loops were introduced to computer 1. Since computer 1 was slowed down, the computation time of the application per time step jumped up during the second set of n time steps. During this second set of n time steps, the dynamic load balancer detected the change in the computation cost function. In the beginning of the third set of n time steps, the dynamic load balancer

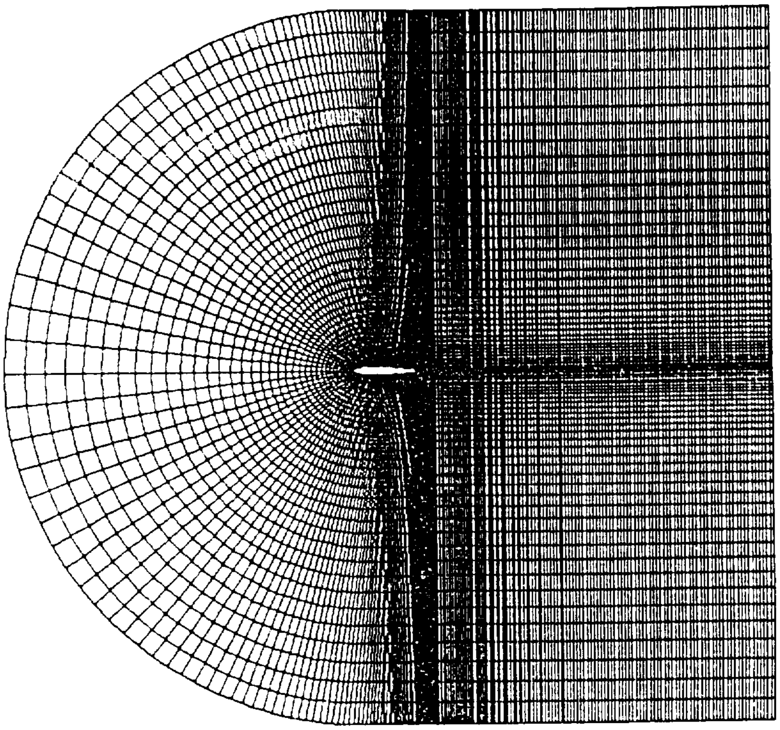


Fig. 11. FLOW3P C-grid with 65 000 nodes.

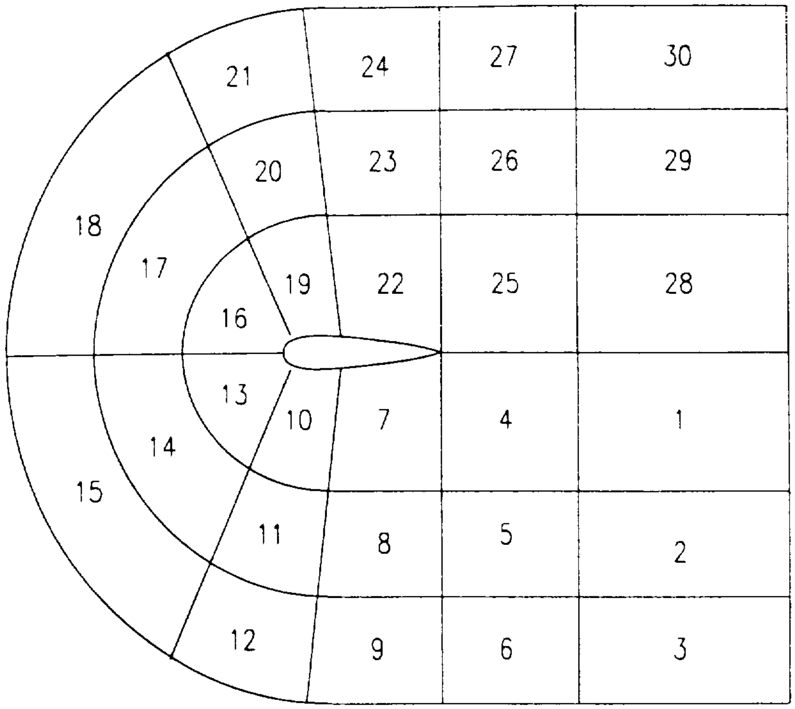


Fig. 12. Topological relationship for C-grid shown in Fig. 10.

Table 1
Number of nodes in each data block

Block number	Block size	Block number	Block size	Block number	Block size
1	2304	11	2304	21	2176
2	2304	12	2176	22	2240
3	2176	13	2240	23	2304
4	2304	14	2304	24	2176
5	2304	15	2176	25	2240
6	2176	16	2176	26	2304
7	2240	17	2232	27	2176
8	2304	18	2108	28	2240
9	2176	19	2240	29	2304
10	2240	20	2304	30	2176

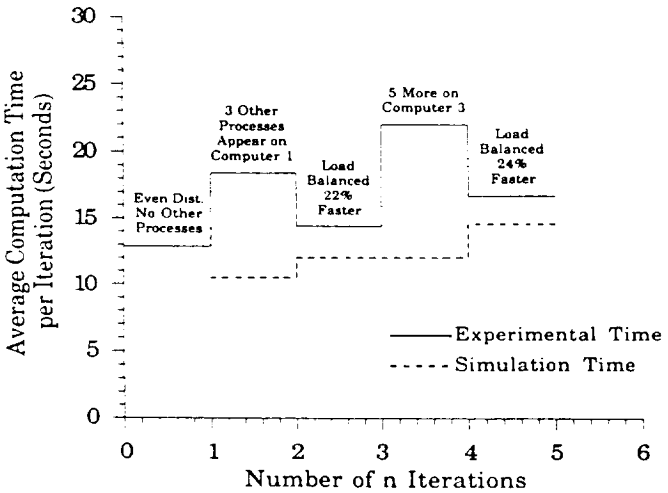


Fig. 13. Timing result using FLOW3P on SP1 in a controlled environment with varying load on 5 computers.

Table 2
The load distribution of 5 nodes of SP1 for every *n* time steps

Iteration	Computer 1	Computer 2	Computer 3	Computer 4	Computer 5	Experimental time	Simulation time
1*n*	1, 2, 3, 4, 5, 6	7, 8, 9, 10, 11, 12	13, 14, 15, 16, 17, 18	19, 20, 21, 22, 23, 24	25, 26, 27, 28, 29, 30	12.2	10.5
	0.0	0.0	0.0	0.0	0.0		
2*n*	2, 4, 5	1, 7, 8, 10, 11, 12	6, 13, 14, 15, 16, 17, 18	3, 19, 20, 21, 22, 23, 24	9, 25, 26, 27, 28, 29, 30	17.7	12.0
	3.0	0.0	0.0	0.0	0.0		
3*n*	2, 4, 5	1, 7, 8, 10, 11, 12	6, 13, 14, 15, 16, 17, 18	3, 19, 20, 21, 22, 23, 24	9, 25, 26, 27, 28, 29, 30	13.8	12.0
	3.0	0.0	0.0	0.0	0.0		
4*n*	2, 4, 5, 18	1, 7, 8, 10, 11, 12, 13, 14	6, 16, 17	3, 15, 19, 20, 21, 22, 23, 24	9, 25, 26, 27, 28, 29, 30	21.8	14.6
	3.0	0.0	5.0	0.0	0.0		
5*n*	2, 4, 5, 18	1, 7, 8, 10, 11, 12, 13 14	6, 16, 17	3, 15, 19, 20, 21, 22, 23, 24	9, 25, 26, 27, 28, 29, 30	16.3	14.0
	3.0	0.0	5.0	0.0	0.0		

removed three blocks from computer 1 and distributed these three blocks to the other computers. Since there were no new processes introduced in the third *n* time steps, we can see that the load re-distribution reduced the computation time by 22% compared to the second set of *n* time steps. During the fourth set of *n* time steps, we introduced another five extraneous load processes to computer 3 so that the processing time for the application jumped up again. After load re-distribution at the end of 4*n* time steps, the computation time was reduced 24% in the fifth set of *n* time steps.

The second example demonstrates the same experiment, except (i) six IBM RS/6000 Model 560 computers at NASA's Lewis Research Center in Cleveland, Ohio were used, and (ii) the computational environmental was an uncontrolled multi-user environment. This cluster of RS/6000 computers was also connected by an Ethernet and a common file server. In this experiment, the communication costs, although it is small, were included in the cost function used for load balancing. The load distribution was rebalanced among the computers in every *n* ($n = 13$) time steps. Fig. 14 depicts timing for the application code's execution under such conditions during 5*n* time steps. Table 3 describes the suggested re-distribution of 30 data blocks to the computers at the end of each *n* time steps. During the first *n* time steps, the application program's processes were evenly distributed on these six computers since we did not have information to do load balancing. Since there were changes of extraneous loads during every *n* time steps, the computation load was re-distributed at the end of every set of *n* time steps as depicted by the dashed line. Similar to the first example, three extraneous processes after *n* and five extraneous processes after 3*n* time steps were introduced to compare the effects of the uncontrolled environment.

5. Discussions

We have described the encouraging progress on dynamic load balancing for parallel CFD problems. Many new questions surfaced which warrants further investigations.

(1) The dynamic load balancer described in this paper is designed for reducing the computer processing time for CFD problems on workstation clusters where the only other computation processes are single processes. It would be interesting to test if two or more parallel applications

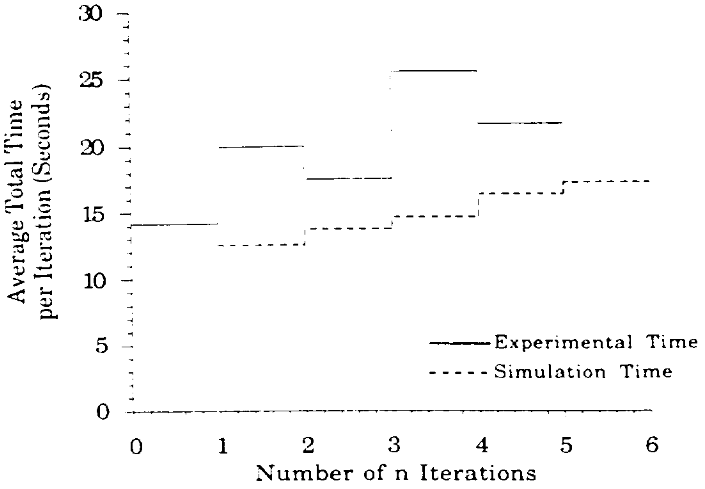


Fig. 14. Timing result using FLOW3P on LACE (RS/6000 workstations) in a multi-user environment with varying load on 6 computers.

are concurrently executed using the developed load balancer on the same computer network. A racing for computing power may occur between two load balancers, which may diminish the effect of load balancing. Some rules and regulations may be needed to coordinate load balancing for multiple parallel programs.

(2) The above dynamic load balancer assumes that the parallel computer network does not have a system load balancing ability. Ideally, the developed scheme should complement 'a parallel operating system'. By making use of the intimate knowledge of the blocked data utilized by the specific CFD application, the developed scheme can provide guidance to such a global load balancing scheme. On the other hand, rules and regulations have to be placed to avoid conflict.

(3) The efficiency of parallelization decreases when the ratio of number of blocks to the number of computers approaches to one in a multi-user environment. This is due to the fact that, in this

Table 3
The load distribution on 6 nodes of LACE for every *n* time steps

Iteration	Computer 1	Computer 2	Computer 3	Computer 4	Computer 5	Computer 6	Experimental time	Simulation time
1*n*	1, 2, 3, 4, 5	6, 7, 8, 9, 10	11, 12, 13, 14, 15	16, 17, 18, 19, 20	21, 22, 23, 24, 25	26, 27, 28, 29, 30	14.1	12.5
	0.4	0.4	0.3	0.1	0.2	0.1		
2*n*	1, 2, 4, 5, 15	6, 7, 8, 9, 10	11, 13	14, 16, 17, 18, 19, 20	3, 21, 22, 23, 24, 25	12, 26, 27, 28, 29, 30	18.9	13.7
	0.0	0.5	3.6	0.1	0.2	0.0		
3*n*	1, 2, 4, 5, 15, 27	6, 7, 8, 9, 10, 24	11, 13	14, 16, 17, 18, 19, 20	3, 21, 22, 23, 25	12, 26, 28, 29, 30	17.5	14.6
	0.2	0.1	3.6	0.1	0.7	0.3		
4*n*	1, 2, 4, 15, 27, 28	7, 8, 9, 10, 24, 26	5, 11, 13	14, 16, 17 18, 19, 20, 30	3, 21, 22, 23, 25, 29	6, 12	24.4	16.3
	0.2	0.1	3.5	0.1	0.1	5.7		
5*n*	1, 2, 4, 15, 27, 28	7, 8, 9, 10, 24, 26	5, 11, 13	14, 16, 17, 19, 20, 30	3, 18, 21, 22, 23, 25, 29	6, 12	21.5	17.2
	0.3	0.4	3.7	0.7	0.1	5.5		

case, the share of parallel processes in one computer changes significantly due to the addition or deletion of extraneous processes. Adding another process on a computer will slow down the parallel process on that computer and hence slow down the parallel computations on all computers. However, the load balancer does not have much choice to re-distribute the parallel application load.

(4) We cannot allocate too many block processes on one computer. The number of processes on each computer is limited by the operating system. To avoid this problem, we combine several data blocks into a single one.

(5) At this time the re-distribution of the load is done intermittently. It is assumed that the processes come to a stop before restarting the algorithm. One can improve the efficiency of such a scheme.

6. Conclusions

A methodology for dynamic load balancing of parallel CFD applications was developed. This method enables the use of networked, multi-user workstations for solving large CFD problems. It is not designed for any particular code but as a general tool to automate load balancing any given computer network. The developed procedure provides a near optimal solution in terms of computation time. Experiments demonstrate the practicality of the proposed dynamic load balancing method. In this study, it is demonstrated that (i) dynamic computer load balancing can significantly increase the speed of distributed computations in a multi-user environment, (ii) computational speed of computers is the main factor to determine the solution time, (iii) the communication cost for parallel applications in a tightly distributed computation environment can be reduced to a relatively small portion of total solution cost by properly managing the data, (iv) the effective computation speed of a computer in a multi-user mode to a parallel user can be measured if the number of processes on that computer and (v) the measurement of communication speed on Ethernet network becomes unreliable as the network becomes crowded. The developed procedure can be extended to accomodate adaptive algorithms in terms of grids or solution schemes as well as complex computer networks.

Acknowledgment

This research has been sponsored by NASA Lewis Research Center, Computational Technologies Branch under Grant No. NAG3-1246. Computer support by IBM Research Center, Kingston, New York is also gratefully acknowledged.

References

[1] T.A. Egolf, Connection Machine utilization and experience at the United Technologies Research Center, Presented at the Fourth SIAM Conference on Parallel Processing for Scientific Computing, Chicago, 10–13 December, 1989.

[2] T.F. Chan, Y. Saad and M. Schultz, Solving elliptic partial differential equations on hypercubes, in: M. Heath, ed., Proc. First Conf. on Hypercube Multiprocessors, Knoxville, TN (SIAM, Philadelphia, August, 1985) 196–210.

[3] T. Tezduyar, S. Aliabadi, M. Behr, A. Johnson and S. Mittal, Parallel finite-element computation of 3D flows, IEEE Computer (October, 1993) 27–36.

[4] T.P. Green, R. Pennington and D. Reynolds, Distributed Queuing System Version 2.1 Release Notes, Supercomputer Computations Research Institute, Florida State University, 1993.

[5] H.U. Akay, A. Ecer and W.B. Kemle, A parallel explicit solver for unsteady compressible flows, in: K.G. Reinsch et al., eds., Parallel Computational Fluid Dynamics '91 (Elsevier Science Publishers, The Netherlands) 1–16.

[6] D. Williams, Performance of Dynamic Load Balancing Algorithms for Unstructured Grid Calculations, CalTech Report, C3P913, 1990.

[7] H. Simon, Partitioning of unstructured problems for parallel processing, NASA Ames Tech Report, RNR-91-008, 1991.

[8] R. Lohner, R. Ramamurti and D. Martin, A Parallelizable Load Balancing Algorithm, Proc. 31st Aerospace Sciences Meeting & Exhibit, Reno, Nevada, 11–14 January, 1993.

[9] H.U. Akay, R. Blech, A. Ecer, D. Ercoskun, B. Kemle, A. Quealy and A. Williams, A database management system for parallel processing of CFD algorithms, in: R.B. Pelz et al., eds., Parallel Computational Fluid Dynamics '92 (Elsevier Science Publishers, The Netherlands, 1993).

[10] Y.P. Chien, F. Carpenter, A. Ecer and H.U. Akay, Computer load balancing for parallel computation of fluid dynamics problems, Parallel Computational Fluid Dynamics 93, Paris, France, May, 1993.

[11] T.H. Cormen, C.E. Leiserson and R.L. Rivest, Introduction to Algorithms (The MIT Press, Cambridge, Massachusetts, 1989).

[12] A. Quealy, G.L. Cole and R.A. Blech, Portable programming on parallel/networked computers using application portable library (APPL), NASA Technical Memorandum, 106238, July, 1993.

[13] A. Ecer, H.U. Akay, W.B. Kemle, H. Wang, D. Ercoskun and E.J. Hall, Parallel computation of fluid dynamics problems, Comput. Methods Appl. Mech. Engrg. 112 (1994) 91–108.

Parallel CFD '96, Capri, Italy, 1996.

Communication Cost Function for Parallel CFD Using Variable Time Stepping Algorithms

Y.P. Chien, S. Secer, A. Ecer and H.U. Akay

Computational Fluid Dynamics Laboratory
Purdue School of Engineering and Technology, IUPUI
Indianapolis, Indiana 46202, USA

ABSTRACT

Network of workstations are widely used for parallel computational fluid dynamics (CFD). A unique problem in parallel CFD is load balancing. We have been studying dynamic load balancing for parallel CFD on a heterogeneous and multi-user environment for several years. Our approach is to cut the problem domain into n blocks and distribute the blocks among m processors, where $m < n$. Computer load is balanced by distributing blocks among computers such that the maximum elapsed execution time of all blocks is minimized. Our load balancing uses optimization algorithms based on the computation and communication cost functions. The cost functions developed previously were under the assumption that all blocks are computed using the same time-steps [1]. Recent CFD algorithm development demonstrates that variable time-stepping approach can significantly reduce the computation and communication requirements. Variable time-stepping algorithms allow different time-step sizes to be selected independently in different subdomains (blocks) [2]. Therefore, uniform time step assumption used in our previous cost function is not valid for variable time stepping algorithms. In this paper, we describe a new communication cost function for parallel CFD using variable time stepping algorithms. The experiments demonstrate that the proposed communication cost function is reasonably accurate.

1. INTRODUCTION

Recent parallel CFD algorithm development demonstrates that variable time-stepping approach can significantly reduce the computation and communication time requirement [2, 3]. The variable time-stepping algorithm means that different sizes of time-steps can be selected independently in different subdomains (blocks) for each time-step. In a parallel solution environment (where a block-based parallelization is applied) the time-steps of blocks may be different from each other and change dynamically. In order to compare the communication time in different block interfaces, a reference time step, basic time-step, is defined as the minimum time-step that can be chosen by all blocks. We assumed that the time-steps chosen by all blocks are integer multiples of the basic time-step.

Three different situations are considered in block-based variable time-stepping algorithms:

· *All blocks choose the same fixed time-step (fix time-stepping algorithm)*: all blocks solves, sends and receives messages every basic time-step.

· *Blocks choose their own time-steps independent of each other*: Blocks will solve and exchange information when they reach to their own time-steps chosen independently. In this case, blocks and interfaces choose the same time-step.

· *Interfaces choose their own time-steps independent of the blocks*: In this case, interfaces will send information to the neighboring interfaces when they reach to their interface time-steps.

While the flexibility of choosing different time-steps throughout blocks and interfaces eliminates many unnecessary computation and communication, it complicated the tasks of load balancing. However, the complication does not affect the load balancing algorithm but rather require a new cost functions derivation. Since the number of time steps executed in blocks and interfaces are different, the communication per time step does not reflect the true load on the computers. In this paper we will discuss how to find the communication cost for variable time stepping algorithms.

We have derived equations in [1] to provide the communication cost of sending one CFD interface message between two CFD processes whether they are on the same machine or different machines. To find the communication cost for a parallel CFD on different computers, we represent this communication cost by *elap_comm*[i][j], where j is the block that sends the message and i is the block that receives the message. We will use *elap_comm*[i][j] as the starting point to estimate the communication cost in CFD programs in this paper.

2. COMMUNICATION COST FUNCTION FOR VARIABLE TIME STEPPING INTERFACE

The notations used for deriving the communication cost function of variable time stepping interface throughout this paper are listed as follows:

- N: number of total basic time-steps in the program execution.
- *elap_comp*[i][m]: average elapsed computation time of block i on computer m per time-step.
- *num_ts*[i] : number of time-steps executed by block i during the entire execution.
- *btspts_comp*[i]: the average number of basic time-steps per time-step for block i.
- *elap_comm*[i][j]: average communication time from the interface of block j to the interface of block i per time-step.
- *tot_elap_comm*[i][j]: total elapsed communication time from the interface of block j to the interface of block i for the entire execution.
- *tot_waiting_time*[i][j]: total waiting time from the interface of block j to the interface of block i or the entire execution.
- *tot_comm_wait*[i][j]: total elapsed communication and waiting time from the interface of block j to the interface of block i for the entire execution.
- *comm_wait_bts*[i][j]: average elapsed communication and waiting time between the interfaces of blocks i and j per basic time-step.
- *btspts_comm*[i]: the average number of basic time-steps per time-step for the interface of block i.

The communication cost function for a variable time-stepping CFD algorithm is derived as follows. It is assumed that there are p computers or processes and there are k CFD blocks on computer m. Block i has n neighbors which are numbered from 1 to n. In order to estimate the communication cost for a variable time-stepping CFD algorithm, we need to know $num\_ts[i]$, where $num\_ts[i]$ is:

$$num\_ts[i] = N / btspts\_comp[i] \tag{1}$$

Step 1: Find the *elap_comm*[*i*][*j*] for each neighbor block pairs.

Elap_comm[*i*][*j*] is the communication cost for one interface connection between the interfaces of blocks i and j per time-step. This has been presented in Parallel CFD'96 conference.

Step 2: Find the total communication time for every neighbor block pairs during the CFD execution.

$$tot\_elap\_comm[i][j] = elap\_comm[i][j] * (N / btspts\_comm[i]) \tag{2}$$

Step 3: Find the total elapsed communication and waiting time for every neighbor block pairs.

This is the most important step in estimating the total communication time. This value will be calculated for each block with its neighbors. The total elapsed communication time and waiting time between two blocks is composed of two terms (equation 3).

$$tot\_comm\_wait[i][j] = tot\_elap\_comm[i][j] + tot\_waiting\_time[i][j] \tag{3}$$

The second term in equation 3 is the *waiting_time*, which is introduced by the variable time-stepping algorithm. For two neighbor blocks i and j, if $num\_ts[i]$ is bigger than $num\_ts[j]$ then block i will experience more computation. During the entire CFD execution, block i will compute (*num_ts*[*i*]-*num_ts*[*j*]) more basic time-steps then block j (block i and block j are called slow and fast blocks, respectively). Therefore, block j will reach to its interface time-step earlier then block i and will wait for block i to reach to a basic time-step equal to or bigger than block j current basic time-step. The equation for *total_waiting_time* for fast block j (due to block i) on computer m is:

$$tot\_waiting\_time[j][i] = elap\_comp[i][m] * (num\_ts[i] - num\_ts[j]) \tag{4}$$

However, slow block i will not experience any waiting time, since when block i reaches its interface time-step block j interface would have already sent its message to block i interface. Therefore, for slow block i the total elapsed communication and waiting time is:

$$tot\_comm\_wait[i][j] = tot\_elap\_comm[i][j] \tag{5}$$

and for fast block j the total elapsed communication and waiting time is:

$$tot\_comm\_wait[j][i] = tot\_elap\_comm[j][i] + tot\_waiting\_time[j][i] \qquad (6)$$

Step 4: Find the average elapsed communication time for interfaces from blocks i and j per basic time-step.

$$comm\_wait\_bts[i][j] = tot\_elap\_comm[i][j] / N \qquad (7)$$

Simplification of the cost function for fixed time-step CFD algorithms
If the CFD code uses the fixed time-stepping algorithm then since for all blocks $num\_ts[i] = num\_ts[j] = N$ and $btspts\_comm[i] = 1$ for all interfaces. Therefore, equations 5 and 6 would be simplified as:

$$tot\_comm\_wait[i][j] = elap\_comm[i][j] * N \qquad (8)$$

Substituting equation 8 into 7, equation 7 can be simplified as:

$$comm\_wait\_bts[i][j] = elap\_comm[i][j] \qquad (9)$$

3. ACCURACY OF THE COMMUNICATION COST FUNCTION

A variable time-stepping CFD code, PARC3D, was run on five processors of an IBM SP computer in the test case. APPL message passing library [4] is used for parallel programming execution. In PARC3D code, the computation is handled by block solver and communication is managed by the interface solver. The block solver and interface solver in each process choose their own time-steps during the program execution. The CFD data is divided into 16 similar sized blocks.

Four load balancing cycles are executed in this test. In each of the load balancing cycle, the elapsed execution time was measured. Initially the number of processes are evenly distributed among computers. A cost function is derived based on the time measurement. The elapsed execution time estimated using this cost function is compared with the measured time (see Figure 1). This derived cost function is used by the load balancing algorithm [5] to provide a balanced load distribution. The elapsed execution time predicted by the cost function for the new load distribution was compared with the actual measured execution time for the new load distribution (see Figure 2). This test case shows that the derived cost function closely describe the time used for parallel CFD execution. Based on the cost function, the load balancing algorithm does suggest a better load distribution.

4. CONCLUSION

Communication cost function for variable interface time stepping CFD algorithm is developed. The new communication cost function has been tested on IBM SP computer. The test case showed that the cost function can predict the execution time of variable time step parallel CFD code.

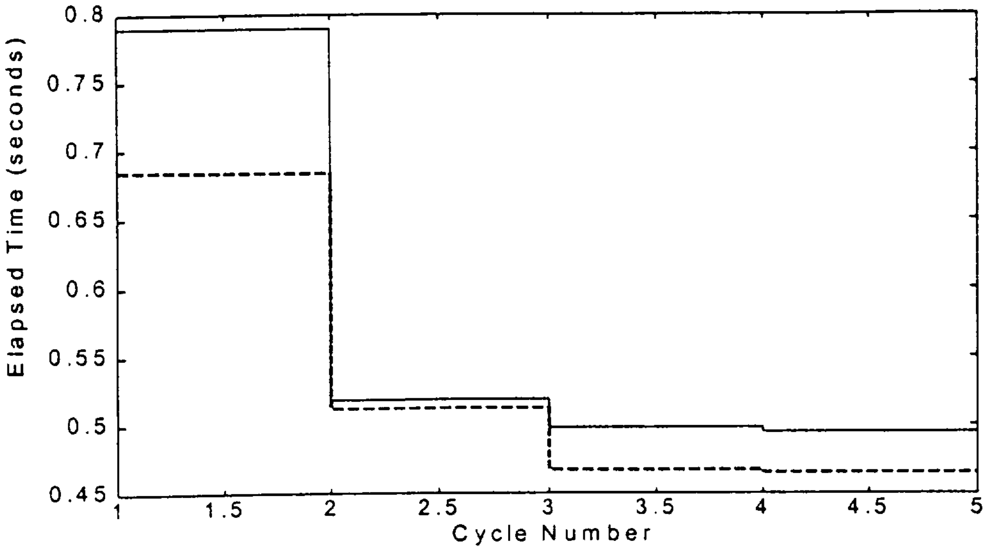


Figure 1. The comparison of measured execution time and the time obtained from the cost function. The solid line is the measured elapsed time. The dashed line is the elapsed time of generated by the cost function.

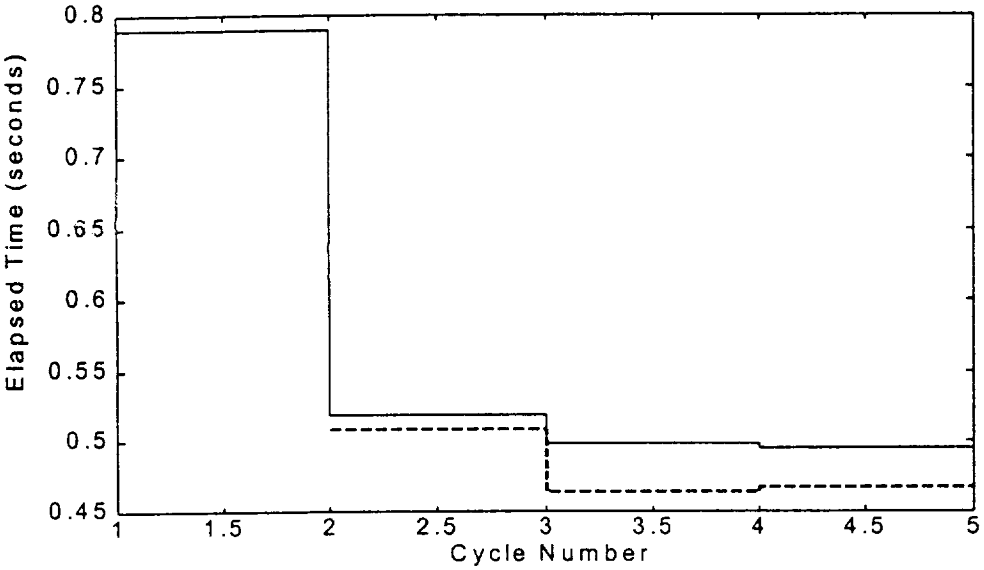


Figure 2. The comparison of predicted execution time to the measured execution time. The solid line is the measured elapsed time. The dashed line is the elapsed time of predicted by the cost function.

REFERENCES

1. Y.P. Chien, A. Ecer, H.U. Akay and S. Secer, "Communication Cost Function for Parallel CFD in a Heterogeneous Environment Using Ethernet," *Parallel Computational Fluid Dynamics: Algorithms and Results Using Advanced Computers*, Edited by P. Schiano et al., 1997, Elsevier Science, pp. 1-10.

2. H.U. Akay, A. Ecer and A.B. Acikmese, "Variable Time-Stepping Strategies for Explicit and Parallel Solution of Unsteady Viscous and Inviscid Compressible Flows," *Parallel Computational Fluid Dynamics: Algorithms and Results Using Advanced Computers*, Edited by P. Schiano et al., 1997, Elsevier Science, pp. 328-335.

3. R.D. Williams, J. Hauser and R. Winkelmann, "Efficient Convergence Acceleration for a Parallel CFD Code," *Parallel Computational Fluid Dynamics: Algorithms and Results Using Advanced Computers*, Edited by P. Schiano, et al., 1997, Elsevier Science, pp. 437-444.

4. A. Quealy, G.L. Cole and R.A. Blech, "Portable Programming on Parallel/Networked Computers Using the Application Parallel Library APPL", *NASA Technical Memorandum 106238*, Lewis Research Center, Cleveland, Ohio, USA, 1993.

5. Y.P. Chien, A. Ecer, F. Carpenter and H.U. Akay, "Computer Load Balancing for Parallel Computation of Fluid Dynamics Problems", *Computer Methods in Applied Mechanics and Engineering*, Vol. 120, 1995, pp. 119-130.

Parallel Computational Fluid Dynamics:
Algorithms and Results Using Advanced Computers
P. Schiano, A. Ecer, J. Periaux and N. Satofuka (Editors)

Communication Cost Function for Parallel CFD in a Heterogeneous Environment Using Ethernet

Y. P. Chien, A. Ecer, H. U. Akay and S. Secer

Computational Fluid Dynamics Laboratory
Purdue School of Engineering and Technology, IUPUI
Indianapolis, IN 46202, USA

In order to increase the efficiency of parallel CFD algorithms, a special domain decomposition approach is adopted which divides the problem domain into a number of blocks that are more than the number of computers. Each block is associated with a set of interfaces. Each block and interface is assigned with a block solver and an interface solver, respectively. A software library was previously developed to support this approach [1]. To improve the efficiency of parallel CFD using this approach, a load balancing algorithm [2, 3] was also developed. The load balancing algorithm requires the availability of a computation cost function and a communication cost function to describe the speed of the computers and networks for parallel CFD. In this paper, derivation of a new communication cost function for parallel computing in a heterogeneous network environment using Ethernet with TCP/IP protocol is presented. A practical real time procedure for obtaining the communication cost function during the execution of parallel CFD is described. This procedure supports dynamic computer load balancing of parallel codes. The experimental results show that the predicted elapsed times derived from our computation and communication cost functions are very close to the measured elapsed times.

1. INTRODUCTION

Solving computational fluid dynamics problems requires computers of very fast computation speed and large memory space. As the computation speed and memory size of computers increase, larger CFD problems need to be solved as well. Parallel and distributed computing are considered as practical ways of satisfying the computation requirement of parallel CFD algorithms. It is well known in the parallel computing community that the speed gain of parallel computations diminishes as the scale of parallelization increases. It is also well known that the causes of the diminished return of further parallelization are due to the load imbalance among parallel processors and the communication overhead between parallel and distributed processes.

Computer load balancing for parallel CFD is especially important when many processors are involved. Most domain decomposition based parallel approaches divide the problem domain into a number of subdomains that are the same as the number of computers used in parallel execution. Load balancing is achieved by changing the number of grid points in the subdomains. One assumption used in these approaches is that the relative computation speeds of the parallel and distributed computers and the effective communication speeds of the network are known. However, this assumption is valid when the parallel systems are homogeneous and are used in a single user mode. Since using homogeneous parallel computers in a single user mode is expensive, it is desirable to use many readily available heterogeneous networked workstations and supercomputers for parallel CFD. Besides, many supercomputers developed recently (e.g., IBM SP, Cray T3D, Silicon Graphics Galaxy, etc.)

can be considered as a set of connected high-end multi-user workstations with a special interconnection network. If a load balancing algorithm can be developed for networked heterogeneous workstations, it can be used in homogeneous environments too.

We have been studying dynamic load balancing for parallel CFD on a heterogeneous and multi-user environment for three years. Our approach is to cut the problem domain into n blocks and distribute the blocks among m processors, where $m < n$. Computer load is balanced by a proper distribution of blocks among the computers [2, 3]. In our study, we faced three issues. The first is to find a fast optimization algorithm for dynamic load balancing. The second is to determine the effective computation speed of all computers in a multi-user environment. The third is to find the effective communication speed of computer networks used for the parallel CFD. The solutions of the first two issues for a network of single CPU computers have been previously treated with success [1, 2] and a software package DLB was developed to generate these solutions. Being tested with several parallel CFD programs for many CPU bound cases, DLB demonstrated significant efficiency improvements especially in the cases that human intuition for load balancing was limited. However, we have not been able to use DLB for communication bound parallel CFD problems due to the lack of a good communication cost function until recently. In this paper, we illustrate practical means of determining a reliable communication cost function for a Ethernet network.

The paper is organized as follows. Section 1 is the general introduction of the effect of communication to the parallel CFD. Section 2 discusses how to determine the communication cost function for a Ethernet network and describes how to incorporate this cost function into the dynamic load balancing. Section 3 presents some experimental results. The last section concludes the paper.

2. DETERMINING A COMMUNICATION COST FUNCTION

By analyzing the time used for all processes in a parallel CFD, we found that the total elapsed time can be divided into three categories: the computation time, the communication time, and the waiting time. Load balancing can be used to minimize the communication time between computers and to minimize the waiting time of all processors. In other words, load balancing is to keep all computers busy and to reduce the cost of data exchange between computers. In order to balance the computer load, a cost function is needed. Our approach for predicting the future computation and communication cost functions is to derive them based on the immediate past computation and communication costs. We have developed algorithms to measure the total elapsed time and the computation time and derived the computation cost function [1]. Here, we describe how to find the communication time and derive a communication cost function. Since Ethernet network is a most widely used computer network, we concentrated our study on finding the communication cost function for Ethernet networks.

The measurement for the communication time on an Ethernet network during parallel CFD is a difficult problem due to the random nature of message passing and collision handling protocol. Since parallel CFD codes generate large amount of data for communication which affect the network load, the communication speed information during the execution is needed for load balancing. Although some specialized programs exist for monitoring the network load, it is difficult to use them only during the execution of parallel calculations. Therefore, an approach is developed to measure the *communication time* during parallel computations and to derive the *communication cost function* based on this measurement.

2.1 Measuring the Communication Time Between Processors

The criteria needed for measuring the communication speed of the computer network used for parallel calculations are rather unique. The measurement should reflect the communication speed during the parallel CFD execution and should have minimal perturbation to the load of the computers and network used for parallel CFD. In order to satisfy these requirements, we developed a communication tracking parallel program, CTRACK. Since the parallel CFD creates a lot of communication which affects the network load, CTRACK periodically records the communication speed by sending a message between every pair of computers used during the CFD execution. To prevent adding additional load to the computers and network, CTRACK sends at most one message through the network at any given moment. The idea for the measurement of communication time between two computers is straight forward. The source computer sends a time stamped message to the target computer. Immediately after the target computer receives the message, it attaches a new time stamp to the message. Then the communication time for sending the message is the difference between the two successive time stamps. However, many issues need to be considered in order to understand and utilize this measurement information. By investigating many measurements of communication speed of computer networks, the following observations were found to be significant for analyzing the measured communication time.

Observation 1: *Different computers have different clocks*. Since the two time stamps for measuring the communication time are taken on different computers, the clock difference between two computers must be known. A surprising fact is that the clock difference between computers can be significant. Even the clock difference of different processors on the same IBM/SP system can be in the order of milliseconds. Table 1 shows an example of the clock differences among three RS6000s and four processors of an IBM/SP. Since the clock differences can contribute to large measurement errors, the communication time c_{ba} for sending a message from computer a to computer b is modified as follows:

$$c_{ba} = t_2 - t_1 + \Delta t_{ba} \tag{1}$$

where t_1 is the time stamp on the message issued by computer a,
t_2 is the time stamp on the message issued by computer b, and
Δt_{ba} is the clock difference between computer a and computer b.

Table 1. Clock differences between three RS6000 (RS6K) and four nodes of an IBM/SP on the same local network in seconds.

	RS6K 1	RS6K 2	RS6K 3	node 4	node 5	node 6	node 7
RS6K 1	0	-0.0037	-0.0021	-2.0344	-2.0349	-2.0349	-2.0348
RS6K 2		0	0.0016	-2.0332	-2.0331	-2.0334	-2.0334
RS6K 3			0	-2.0348	-2.0346	-2.0348	-2.0346
node 4				0	-0.0001	-0.0003	-0.0001
node 5					0	0.0001	-0.0006
node 6						0	-0.0005
node 7							0

Since the clock differences between computers are constant, they need to be measured only once. The condition for using this procedure is that there are no loads on the computers and the network. The following is the procedure adopted for determining the clock difference between two computers:

Step 1. The computer a sends computer b a short message attached with a time stamp t_1.

Step 2. Immediately after receiving the message, the computer b attaches a new time stamp t_2 to the message and sends the message back to the computer a.

Step 3. Immediately after receiving the returned message from the computer *b*, the computer *a* time stamps the message with t_3.

Step 4. The clock difference, Δt_{ba}, between computer *b* and computer *a* is calculated from:

$$\Delta t_{ba} = (t_1 - t_2) + 0.5(t_3 - t_1) \tag{2}$$

Observation 2: *Communication time is stepwise linear with the size of the message.* According to the IEEE standard 802.3, the message on Ethernet is sent in packets. The maximum number of data in each packet is 1,500 bytes [4]. By measuring the communication time for messages of various sizes, we also observed that the communication time is stepwise linear to the size of the message. This fact can be used for deriving of the communication cost, *C*, of a large message based on the communication cost of other message:

$$C = K_s A \tag{3}$$

where $A = c_{ba} = t_2 - t_1 + \Delta t_{ba}$ for a message of one packet size and K_s is the number of packets.

Observation 3: *The communication cost for sending messages between two processes on the same processor cannot be neglected.* Contrary to our earlier assumption that the communication between processes on the same computer takes negligible time, we observed that this communication cost can be as high as about one third of the communication time for sending the same message between two different computers.

2.2 The Effect of the Load on the Computer to the Communication Time

After incorporating the three aforementioned facts into the communication cost function and using it in the load balancing algorithm described in [2], the predicted communication time was still found to be far from the actual measured communication time. Therefore, other factors that affect the communication time were investigated.

Observation 4: T*he communication time for sending a message between processes on the same computer is a function of the number of the load on the computer.* Based on the measurements of communication time for sending the same message between two processes on the same computer under various computational loads, the communication cost function can be approximated by the following linear function (Figure 1):

$$C = K_s(A + K_p L) \tag{4}$$

where, *C* is the communication time for sending a message between two processes on a computer, K_s is the number of packets used for sending the message, *A* is the communication time when there is no load on the computer in terms of seconds, K_p is the load factor in terms of seconds per process (load) on the computer, and *L* is the number of processes (load) on the computer.

It should be noted that the CPU bound loads on a computer give different linear functions than the I/O bound loads.

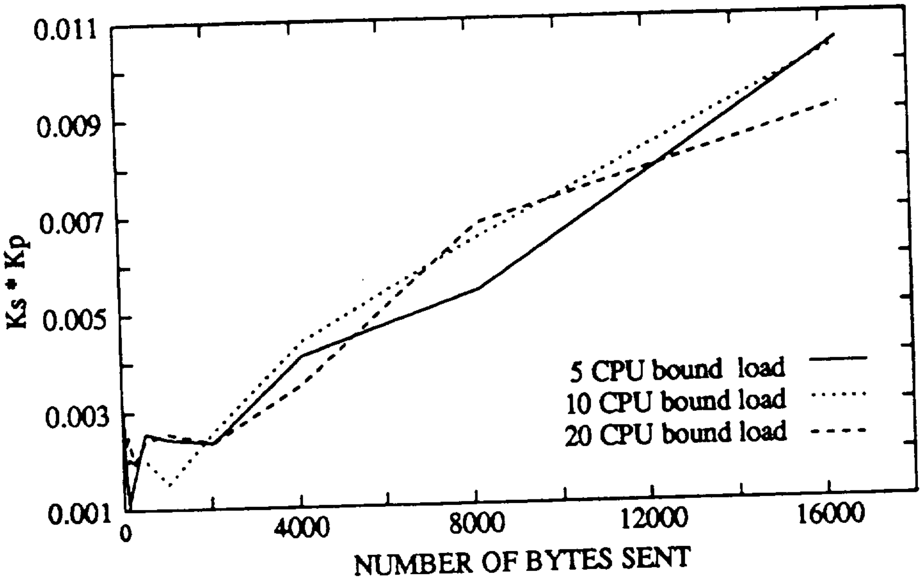


Figure 1. Load factor versus the message size with different CPU bound loads on the computer.

Observation 5: *The communication time between processes on different computers is affected by both the loads of the source computer and destination computer.* To study the effect of computer loads to the communication time, computers of different speeds were assigned as the sender and the receiver of the message in the measurement of communication time. Table 2 shows the communication time of sending 32-byte messages under various loads on both the sender computer and the receiver computer. Both the sender and the receiver computers are IBM RS6000s but the CPU speed of the sender is twice faster than that of the receiver. The measurement result is an average of 700 trials. The unit of the numbers in the table is milliseconds. In this particular case, the load of the receiver computer affects the communication time the most. However, in other cases, such as more load on a slower sender and less load on a faster receiver, the load of the sender computer controls the communication time. This can be explained by the design of the UNIX system [5]. In a multi-user and multi-tasking computer, the CPU is shared by all processes on the computer. The operating system assigns a time quantum to each task in the process round robin queue. Therefore, the more load and the slower CPU on a computer will cause slower response to the message by the computer.

Table 2. Effect of the loads on sender and receiver computers to the communication time.

Number of CPU bound loads on the receiver	Number of CPU bound load on the sender						
	0	1	2	3	4	5	6
0	2.8	2.5	2.5	3.1	3.0	2.6	2.6
1	3.0	3.0	3.3	4.2	4.0	3.4	2.7
2	4.8	5.2	5.2	5.5	6.3	5.6	4.5
3	6.2	6.2	6.6	6.4	7.0	7.2	7.1
4	7.5	7.8	7.5	6.1	7.4	7.7	8.1
5	8.3	8.0	8.5	7.2	8.6	7.8	7.9
6	7.8	8.6	8.6	7.9	6.6	6.4	6.3

2.3 Derived Communication Cost Function

Based on the above observations, communication cost functions for sending messages on the same computer and to different computers are developed.

2.3.1 For sending messages between processes on the same computer

Communication time between two processes on the same computer is:

$$K_s\left(A + K_{cpu}L_{cpu} + K_{io}L_{io}\right) \tag{5}$$

where K_s is the number of packets used by the message,
A is the communication time for sending one packet between computers,
K_{cpu} is the time quantum for CPU bound processes,
L_{cpu} is the average number of CPU bound processes,
K_{io} is the time quantum for I/O bound processes, and
L_{io} is the average number of I/O bound processes.

This communication cost function can accurately predict the communication time between computers in a testing environment in which the type of processes are known. However, it is difficult to determine K_{io} and L_{io} during practical parallel computations since whether a process is CPU bound or I/O bound is unknown. Depending on the input and load distribution, a parallel program can be CPU bound in one case and I/O bound in another case. To solve this problem, it is assumed that any process that is not our parallel CFD process as the CPU bound process (we also call them extraneous processes). A program PTRACK has been developed for finding the number of extraneous processes during parallel CFD executions [2]. It is also assumed that parallel CFD for a given input is a fixed combination of CPU bound and I/O bound processes. Therefore, the communication cost function between two processes on the same computer can be rewritten as:

$$K_s(A + K_{cpu}L_{cpu} + K_{cfd}L_{cfd}) \tag{6}$$

where K_{cfd} is the time quantum for parallel CFD processes and L_{cfd} is the average number of parallel CFD processes.

The coefficients K_s and L_{cfd} can be obtained or calculated from the CFD data input, and L_{cpu} can be measured [2]. The coefficient A, K_{cpu} and K_{cfd} can be derived by measuring the communication cost under different L_{cpu} and L_{cfd}. Since K_{cpu} and K_{cfd} are independent of the computer network, they need to be derived only once. The coefficient A reflects the network load so that it is measured repeatedly during parallel CFD. It should be noted that due to the random nature of the computer and network loads, and due to the collisions in the Ethernet, the reliable measurement value should be the mean of many measurement samples.

2.3.2 For sending messages between processes on different computers

The approach used for deriving the communication cost function for sending messages between processes on different computers is similar to that on the same computer. However, the communication cost function is affected by the number and type of loads on both the sending and receiving computers (as described in Observation 5). Therefore, the communication cost function between two processes on different computers can be approximated from:

$$K_s(A+F)+\Delta t_{ab} \tag{7}$$

where, K_s is the number of packets needed for the message,

A is the communication time for one packet in the network,

Δt_{ab} is the clock difference between computers a and b, and

F is a function of the loads on the sending and receiving computers.

The function F can be derived accurately only when the load matrix as shown in Table 2 is available for computers a and b. However, generating the load matrix is a time consuming process which is not suitable for real-time dynamic load balancing. Based on the observation of the load matrices of many pairs of computers, F is approximated by:

$$F = max\left\{\left(Ka_{cpu}La_{cpu} + Ka_{cfd}La_{cfd}\right),\left(Kb_{cpu}Lb_{cpu} + Kb_{cfd}Lb_{cfd}\right)\right\} \tag{8}$$

where

La_{cpu} is the average number of extraneous CPU bound processes in the sending computer a,

Ka_{cpu} is the time quantum for extraneous CPU bound processes in the sending computer a,

La_{cfd} is the average number of CFD processes in the sending computer a,

Ka_{cfd} is the time quantum for CFD processes in the sending computer a,

Lb_{cpu} is the average number of extraneous CPU bound processes in the receiving computer b,

Kb_{cpu} is the time quantum for extraneous CPU bound processes in the receiving computer b,

Lb_{cfd} is the average number of CFD processes in the receiving computer b,

Kb_{cfd} is the time quantum for CFD processes in the receiving computer b.

The procedure for finding this communication cost function is as follows:

Step 1. Find Δt_{ab} using the procedure for determining the clock difference between two computers described in section 2.1.

Step 2. Let computer a to be the message sender and b to be the message receiver. Measure the communication cost without parallel CFD load on both sender and receiver computers.

Step 3. Measure the communication cost after adding several CPU bound loads to the receiver computer. Since K_s and all L are known, Kb_{cpu} can be derived based on the results of steps 2 to 4.

Step 4. Measure the communication cost after adding several CFD loads to the receiver computer. Since K_s and all L are known, Kb_{cfd} can be derived based on the results of steps 1 to 4.

Step 5. Change the role of sender and receiver and repeat steps 2 to 4 to generate Ka_{cpu} and Ka_{cfd}

3. EXPERIMENTAL RESULTS

3.1 Evaluation of the Communication Cost Function

The communication cost function is used to predict the elapsed processing time of a parallel CFD with various data input and various number of computers. Table 3 summarizes the results on an IBM/SP system. The column for "# of blocks" in the table defines the number of solution blocks used in each case. The column for "ratio of comp/comm" describes the ratio of measured elapsed computation time to the measured elapsed communication time. The column for "% error" is calculated as:

$$\%\text{error} = \left|\left(\frac{\text{measured elapsed time} - \text{predicted elapsed time}}{\text{measured elapsed time}}\right)\right|$$

Table 3. Performance of cost function with different data input.

Case	# of Blocks	# of Hosts	Average block grid points	Ratio of comp/comm time	Measured elapsed time	Predicted elapsed time	% Error
1	5	3	11000	1.43	0.085	0.086	1.2
2	5	3	40000	2.45	0.262	0.249	5
3	10	3	22000	0.77	0.233	0.263	12
4	12	5	19000	0.186	0.299	0.360	20
5	15	5	15000	0.088	0.410	0.290	29
6	15	3	15000	1.00	0.234	0.235	0.5
7	20	5	12000	0.064	0.449	0.336	25

The majority of the cases in the experiment have unreasonably high communication costs. These cases were chosen for demonstrating the accuracy of the communication cost function in rather unfavorable conditions. The ratio of the measured elapsed computation time to the measured elapsed communication time is determined by the sizes of blocks, the number of computers used and the topology of the blocks. As depicted in the table, the communication cost function gives fairly accurate prediction of elapsed execution time when the communication cost is comparable to or little more than the computation time. When the weight of communication is several times of that of the computation time, the cost function becomes inaccurate. However, this situation does not usually occur in practical applications with very large size blocks.

3.2 Dynamic Load Balancing Using the Communication Cost Function

The following experiment demonstrates the applicability of the communication cost function. Three IBM RS6000 computers were used in the experiments. The CPU speeds of the first two RS6000s are similar. The CPU speed of RS6000 #3 is about one half of that of the other RS6000s. In order to make communication a dominant factor in parallel computations, a small case with 54,400 grid points was executed on three computers. The CFD problem is divided into 5 blocks of similar sizes. In this arrangement, the communication time used in the program execution is comparable to the computation time even when the load is balanced. Initially, the load is distributed to the computers as follows:

RS6000 #1	RS6000 #2	RS6000 #3
block 1	block 2	blocks 3, 4, 5

Using the communication cost function described in the previous sections and the computation cost function described in [2], the load balancing algorithm [3] predicted that the elapsed execution time would be 0.372 seconds per time step. The measured actual elapsed execution time of this distribution was 0.367 seconds (Figure 2).

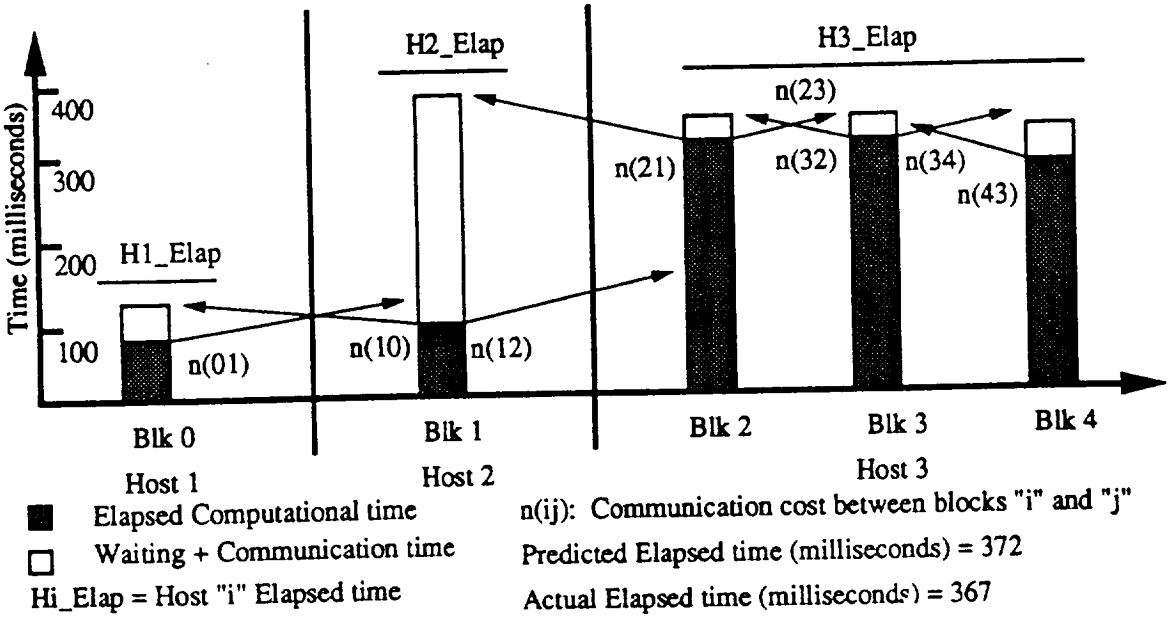


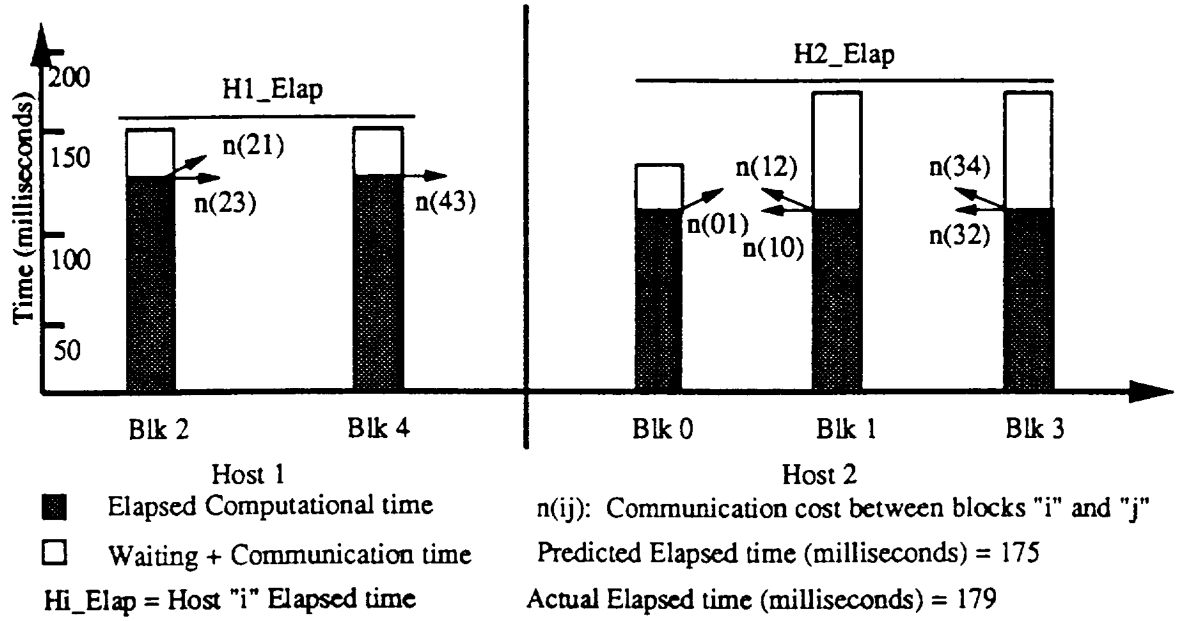
Figure 2. Computation, communication and the waiting time in one iteration before DLB.

Based on the information obtained in this execution, the load balance program suggested the following distribution:

RS6000 #1	RS6000 #2	RS6000 #3
blocks 1, 2, 5	blocks 3, 4	

This suggested distribution shows that parallelization to more than two computers actually increases the execution time. The suggestion also agrees with the fact that RS6000 #3 is a slower computer. The load balancing program predicted that the elapsed execution time for this distribution is 0.175 seconds per time step. The measured actual elapsed execution time for this load distribution is 0.179 seconds per time step (Figure 3). This experiment demonstrates that the communication cost function is fairly accurate.

The development of a communication cost function relies on the accurate measurement of the communication time. Due to the random nature of the Ethernet and TCP/IP, one time measurement is mostly unreliable. Therefore, all measurements are repeated several hundred times (as time permits) concurrently with the parallel CFD execution. The result is the mean of all these repeated measurements. Since the parallel CFD executions usually run for hours, there is usually enough time to take the communication time measurement repeatedly without adding noticeable load to the computers and the network.



Predicted Elapsed time (milliseconds) = 175

Actual Elapsed time (milliseconds) = 179

Figure 3. Computation, communication and the waiting time in one iteration after DLB.

4. CONCLUSIONS

The communication time for parallel CFD is a function of not only the computer network but also the loads on the computers which send and receive the message. A communication cost function is developed based on these observations. A software package is also developed to automatically derive the communication cost function for Ethernet network and TCP/IP protocol.

ACKNOWLEDGMENT

This research was funded by the NASA Lewis Research Center, Computational Technologies Branch, under grant: NAG3-1577.

REFERENCES

1. Akay, H.U., Blech, R., Ecer, A., Ercoskun, D., Kemle, B., Quealy, A. and Williams, A. (1993), 'A Database Management System for Parallel Processing of CFD Algorithms,' *Parallel Computational Fluid Dynamics '92*, Edited by R.B Pelz, et al., Elsevier Science Publishers, The Netherlands, pp. 9-23.
2. Chien, Y. P., Carpenter, F., Ecer, A. and Akay, H.U., "Load Balancing for Parallel Computation of Fluid Dynamics Problems," *Computer Methods in Applied Mechanics and Engineering*, Vol. 120, 1995, pp. 119-130.
3. Chien, Y. P., Ecer, A., Akay, H.U. and Carpenter, F., " Dynamic Load Balancing on Network of Workstations for Solving Computational Fluid Dynamics Problems," *Computer Methods in Applied Mechanics and Engineering*, Vol. 119, 1994, pp. 17-33.
4. Held, G., *Ethernet Networks* (Second Edition), John Wiley & Sons, New York, 1996.
5. Leach, R.J., *Advanced Topics in Unix*, John Wiley & Sons, Inc., New York, New York, 1994

AIAA 97-0027

Efficient Parallel Communication Schemes for Explicit Solvers of NPARC Codes

N. Gopalaswamy, A. Ecer, H.U. Akay and Y.P. Chien
Computational Fluid Dynamics Laboratory
Purdue School of Engineering and Technology, IUPUI
Indianapolis, IN

35th Aerospace Sciences Meeting & Exhibit

January 6-10, 1997 / Reno, NV

EFFICIENT PARALLEL COMMUNICATION SCHEMES FOR EXPLICIT SOLVERS OF NPARC CODES

N. Gopalaswamy, A. Ecer, H.U. Akay and Y.P. Chien

Computational Fluid Dynamics Laboratory
Purdue School of Engineering and Technology, IUPUI
Indianapolis, Indiana 46202

Abstract

A scheme for improving the efficiency of communications for the parallel computation of Euler equations is presented. PARC code is employed as an example for analyzing the flow through a supersonic inlet. The flowfield is divided into subregions called "blocks." The parallel computation of the problem normally requires communication between each block after each time-step of an explicit Runge-Kutta integration scheme. In the developed procedure, the boundary conditions are frozen for $k = 10 - 20$ time-steps and blocks are integrated in time without communicating with each other during this period. When the boundary conditions are updated, an oscillatory error is introduced into the solution with a fundamental period of $4k$ time-steps, which is then filtered in time. As a result, the communication cost of parallel computing is significantly reduced. Examples for steady and unsteady flows are presented to demonstrate the applicability of the developed procedure.

Introduction

During the parallelization of explicit schemes, the efficiency of the communication plays a critical role. Especially for a structured grid, one can develop explicit schemes where computational cost is small in comparison with the communication cost. In the present paper the PARC code with an explicit Runge-Kutta scheme is chosen as the parallel numerical algorithm to be studied.[1] Parallelization of this code has already been discussed in a previous paper.[2] It is based on a block-based structure of the data where the solution domain is divided into many subdomains or "blocks". The global solution is obtained by integrating the equations for each block separately. The blocks are interconnected to each other through an overlapping region or "interface," by one grid point. The solution scheme marches in time while exchanging boundary values of each block at each time-step. Figure 1 summarizes the arrangement for the case of two neighboring blocks.[2] The numerical integration of the grid points are conducted inside a block solver. The block solver updates an interface solver at intervals which then communicates with the interface solver of the neighboring block. Each interface solver also updates its block after receiving information from its neighbor. As can be seen from this figure, each block and its interface solvers are on the same processor. The time intervals for sending and receiving information between the blocks and interfaces can be different, and can be chosen based on the local conditions.[3] The distribution of the blocks among a given number of processors can be optimized by distributing the blocks according to their computation and communication requirements.[2,4]

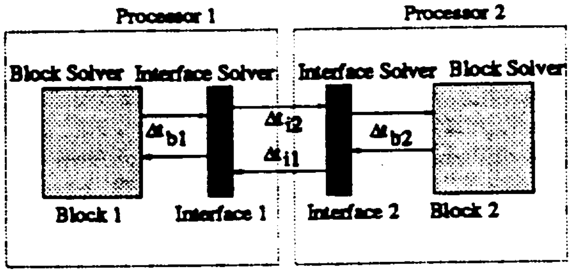


Figure 1: Blocks and Interfaces

In Reference 2, based on the local stability conditions, the time intervals for communicating between the blocks and interfaces, as defined in Figure 1 were selected. The resulting system was then load balanced and considerable efficiency improvements were obtained, specifically by reducing the communication cost. In the present paper, a brief summary of this procedure and the parallelization tools are provided. A further attempt to reduce the communication cost is presented here since the stability requirements for explicit schemes can be quite stringent. Specifically, the communication time interval is further increased, exceeding the limit suggested by the local stability conditions at the interface. By not updating the boundary of a block at required time intervals, the solution becomes discontinuous between the blocks. An error is introduced at each boundary which produces high frequency spatial oscillations inside each block. Based on the study of this error, a filtering scheme is developed, which corrects the boundary conditions and eliminates the high frequency noise. By employing this scheme, one can reduce the communication cost by 90% yet maintain the same accuracy. The numerical results presented in this paper demonstrate applications for both steady and unsteady flows.

Background Information

For the parallelization of the NPARC code, several tools were utilized. A brief summary of these tools and the employed Runge-Kutta algorithm are summarized here.

GPAR - A Grid Oriented Database for Parallel Computing

GPAR[5] was developed specifically for data management of block structured CFD algorithms. It involves two data sets: blocks and interfaces. The grids in each block or interface can be either structured or unstructured. In addition, interfaces can be matching, unmatching, overlapping or nonoverlapping. These parameters, once defined by the application programmer, can then be used by GPAR to handle the low level requests between the processors. Two primary low level message passing libraries are utilized: APPL, developed by NASA LeRC and PVM. The relationship between the components is illustrated by Figure 2.

Explicit Runge-Kutta Algorithm and Stability

The governing Euler equations for inviscid flow are cast in the following conservation form:

$$\frac{\partial Q}{\partial t} + \frac{\partial F_j}{\partial X_j} = 0 \qquad (1)$$

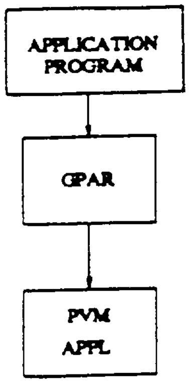


Figure 2: Relationship of GPAR with the application program

where $Q = (\rho, \rho u, \rho v, \rho w, \rho E)^T$, and F_j are the inviscid flux vectors. These equations are transformed into computational coordinates and are solved in strong conservation form by the NPARC code. Additional source terms appear on the right hand side of Equation 1 for axisymmetric flows. The NPARC code can solve the Euler equations either with an implicit Beam-Warming algorithm, or an explicit multi-stage, Runge-Kutta scheme. In the present paper, a three-stage variant of the Runge-Kutta scheme is considered. The Euler equations are cast in semi-discretized form as follows:

$$\frac{dQ}{dt} = A_j \cdot F_j = \mathrm{RHS} \qquad (2)$$

where A is the space discretization operator operating on the vector of conservation variables Q. Central differencing is used for the discretization of the spatial domain. The three-stage Runge-Kutta scheme used can be written as follows:

$$\begin{aligned} Q(0) &= Q^n \\ Q(1) &= Q(0) + 0.6\Delta t\ \mathrm{RHS}(0) \\ Q(2) &= Q(1) + 0.6\Delta t\ \mathrm{RHS}(1) \\ Q(3) &= Q(2) + \Delta t\ \mathrm{RHS}(2) \\ Q^{n+1} &= Q(3) \end{aligned} \qquad (3)$$

where Δt is the time-step used for the temporal integration. A linearized stability analysis for the one-dimensional Euler equations in conservation form discretized as defined in Equation 3 yields the following CFL stability criterion:

$$c = \frac{(u+a)\Delta t}{\Delta x} \leq 1.8 \qquad (4)$$

where c is the Courant number in equation 4. The amplification factor $G(z)$ can be defined in terms of the characteristic polynomial obtained from the linearized stability analysis.

$$
\begin{aligned}
|G| &= \left| \frac{Q^{n+1}}{Q_n} \right| \\
G &= 1 - z + 0.6z^2 - 0.36z^3 \qquad (5) \\
z &= Ic \sin\theta
\end{aligned}
$$

where θ is the phase angle obtained from a Fourier decomposition in the frequency domain. The region near $\theta = 0$ corresponds to the low frequency region, while the region near $\theta = \pi$ corresponds to the high frequencies. The highest frequency resolvable by the mesh corresponds to a wavelength of $2\Delta x$. Figure 3 contains a plot of the amplification factor G. It can be seen from the figure that the amplification factor G is approximately equal to unity for both high and low frequency ranges. This implies that the low and high frequency spatial waves are not damped by the three-stage Runge-Kutta scheme. Artificial viscosity is normally introduced to damp the high frequency oscillations.

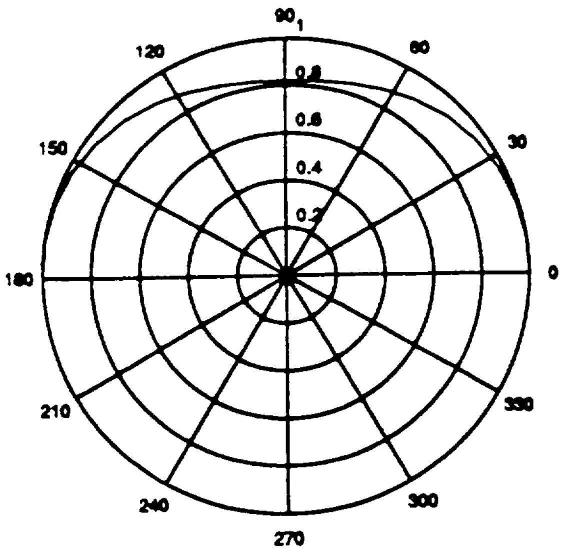


Figure 3: $|G|$ for $0 < \theta < \pi$, and $c = 0.9$

Variable Time-Stepping

For improving the efficiency of the numerical integration of the Euler equations, a variable time-stepping procedure was implemented for each block and interface.[2] For each block, by checking the CFL condition for all the grid points inside the block, a time-step was chosen to ensure stability as shown below.

$$
\Delta t = \frac{c}{Max_i \left[|U_k| + a \left| K_k^i \right| \right]} \qquad (6)
$$

where k is the coordinate direction. Similarly, a time-step was chosen for an interface based on the stability of the grid points on that interface. For supersonic points, the interface communicated only in one direction. For subsonic points, an interface communicated in both directions but at different intervals.

$$
\begin{aligned}
\overrightarrow{\Delta t_{i,k}} &= \frac{\Delta x_k}{u_k + a} \\
\overleftarrow{\Delta t_{i,k}} &= \frac{\Delta x_k}{a - u_k}, \quad (a - u_k) > 0 \qquad (7)
\end{aligned}
$$

From the above stability requirements, the time-step for each block and interface was defined as an integer multiple of a basic time-step. For steady flows, where time-accuracy is not required, a local time-step is defined from the CFL condition for each node individually.

Test Cases

Two test cases are chosen to investigate the effect of the reduced communications. These cases were also employed in the previous study of the NPARC code.[2]

- Steady Flow: An axisymmetric mixed-compression VDC (Variable Diameter Centerbody) inlet is considered under a supersonic inflow of M=2.5 and a subsonic compressor face outflow Mach number M=0.3.[6] The 2D version of the NPARC code has an option to handle axisymmetric flow also. The reference inlet pressure is 117.8 lb/ft^2, and the reference inlet temperature is 395 Rankine. The cowl-tip radius of the inlet, Rc=18.61 inches is used to non-dimensionalize the lengths. The grid for this inlet consists of approximately 4500 nodes, and is divided into 15 blocks, all of approximately equal size as shown in Figure 4. A steady state solution is sought using local time-stepping for all nodes in each block with a uniform Courant number of 0.9 for all nodes. The solution is plotted in the form of density at the midpoint of each interface for all blocks every iteration. This test case is chosen as an example of a small problem where communication cost is large in comparison with the computation cost.

- Unsteady Flow: The same grid illustrated in Figure 4 is used to study the response to a sinusoidal temperature perturbation applied at the inlet section. The amplitude of the perturbation is 5% and the frequency is 100 Hz. The density variation is plotted at one of the subsonic interfaces downstream of the normal shock. The steady state solution is obtained first and then the temperature perturbation is applied. The reference pressure and temperature are 117.8 lb/ft^2 and 395 Rankine respectively. Variable time-stepping is used inside each block, as described in equation 6.

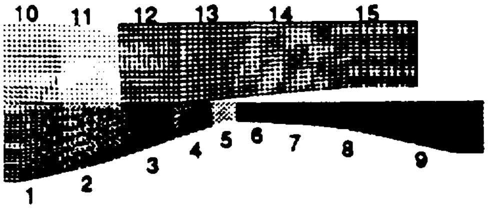


Figure 4: Axisymmetric Case with 15 Blocks

Reduced Communications for Explicit Schemes

By using variable time-stepping considerable improvements in efficiency were obtained.[2] In order to further reduce the communication cost dictated by the stability conditions, one can further increase the interval for updating the interfaces. An experiment was conducted as follows: after grid points on each block and interface have chosen their own time-step, based on local stability conditions, the boundary conditions were frozen for 10 time-steps. This led to both spatial and temporal oscillations. The magnitude of these oscillations was negligible for supersonic interfaces but significant for subsonic interfaces. In Figure 5, the variation of density with respect to time is plotted at the subsonic interface between blocks 8 and 9 in Figure 4 for the steady flow test case. The solution is stable inside each block, since the time-step chosen for integration satisfies the local stability condition for the grid points inside the block. However, the solution is polluted by a high frequency noise emanating from the discontinuity introduced on the boundary. A frequency decomposition of the signal in Figure 6 shows that the high frequency oscillations are associated with distinct frequencies. They corresponded to a time period of:

$$T = 4k\Delta t \qquad (8)$$

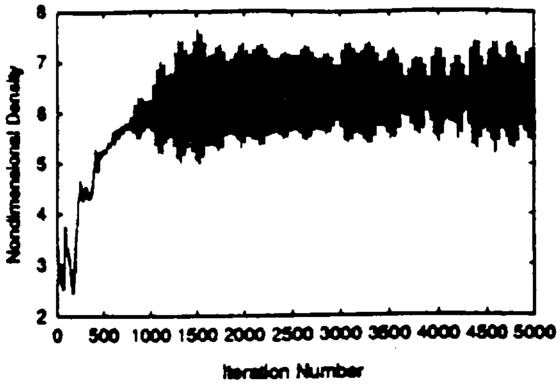


Figure 5: Density variation for $k = 10$

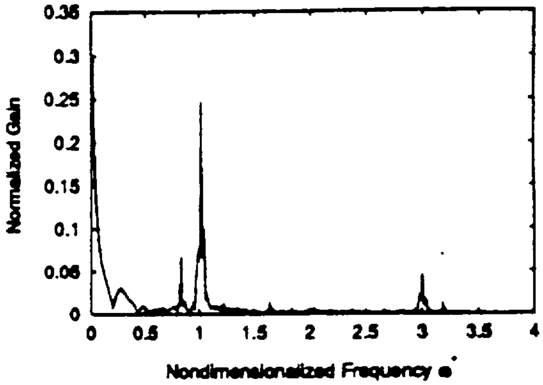


Figure 6: Frequency response of density variation for $k = 10$

and its multiples, where k is the communication interval. The frequency in Figure 6 is nondimensionalized as follows:

$$\omega^* = \omega\left(\frac{2k\Delta t}{\pi}\right) \qquad (9)$$

The same behavior was observed when k was increased to 20, as shown in Figures 7 and 8, although there are many more peaks observed in the frequency spectrum. This is due to the fact the frequencies excited by the communication errors are much lower than the previous case and interact with the correct solution. This point will be further discussed below.

Figure 9 shows the spatial oscillations developing inside a block due to the error introduced by freezing the boundary conditions for $k = 20$. The frequency decomposition of the signal in Figure 9 is shown in Figure 10, which indicates a significant oscillation with a wavelength of $2\Delta x$ near the boundary.

In the following, the source of the above errors introduced by reducing the communications is discussed, and a filtering technique is utilized to elimi-

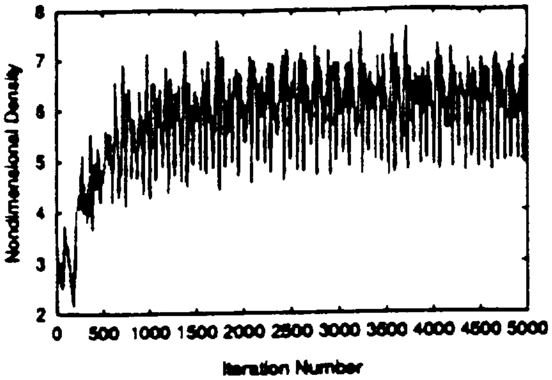


Figure 7: Density variation for $k = 20$

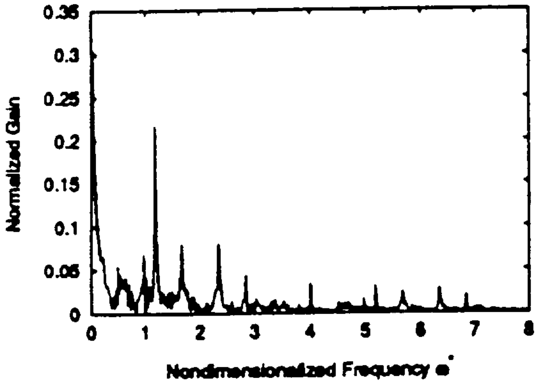


Figure 8: Frequency response of density variation for $k = 20$

nate the oscillations while maintaining the accurate solution.

Error Analysis

An investigation was carried out to explore the origin of the above oscillations. The following simple model was defined to study the problem.

For the case of two blocks shown in Figure 11, the flow is assumed to be one-dimensional from left to right and subsonic. The interfaces belonging to Block I and Block II overlap each other only by one grid point. Since the flow is subsonic, two waves propagate information downstream with speeds u and $u + a$ while one wave propagates information upstream with a speed of $u - a$. u is the fluid velocity while a is the acoustic speed for the fluid.

During the parallel computation, point 1 serves as the downstream boundary condition for Block I, and point 3 serves as the upstream boundary condition for Block II. Points 2 and 4 are computed as interior points of Blocks I and II respectively. During the time-integration, the solution values computed

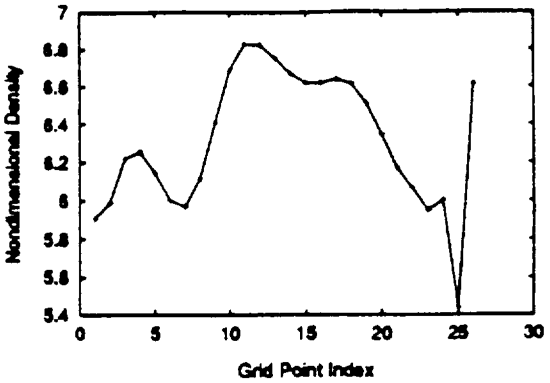


Figure 9: Instantaneous density variation inside a block for $k = 20$

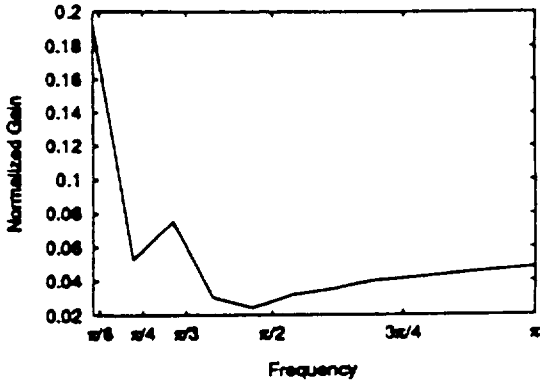


Figure 10: Spatial frequency distribution of density variation inside a block for $k = 20$

at point 2 overwrite the previous values at point 3, every time communication between the interfaces takes place. Simultaneously, the values computed at point 4 overwrite the previous boundary condition at point 1.

If the communication is halted for a specified interval, then the time-integration in Blocks I and II proceed with the boundary condition remaining frozen at the values received during the past communication step. Hence an error is introduced into the time-integration procedure in both blocks. If the semi-discretized Euler equations can be expressed as follows:

$$\frac{dQ}{dt} = A \cdot Q \quad , \tag{10}$$

for a linearized operator $A \cdot$, the error obeys the same difference equation as the solution. Hence, if we call the error X, the following relation is valid:

$$\frac{dX}{dt} = A \cdot X \tag{11}$$

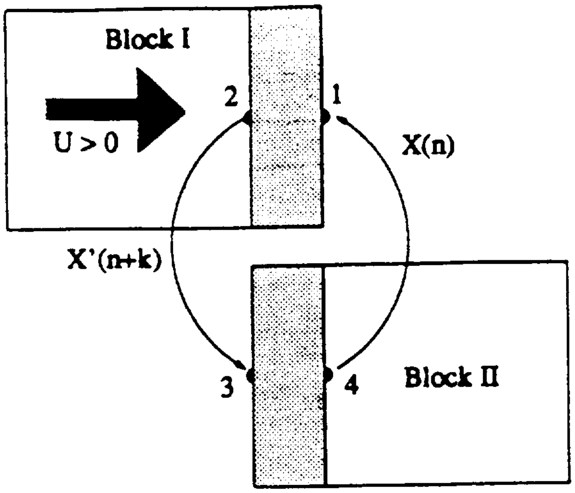


Figure 11: Model of Feedback System

One can trace the propagation of the error through this model. Assume that $X_1(n)$ is an error in the boundary condition, first introduced at a time-step n, at point 1. If the boundary conditions are held fixed for k time-steps, this error will propagate upstream in Block I to point 2. Since the error also obeys the same discretized equation as the solution for a linear operator, the error will be modified by the time it propagates to point 2 to become $X_2(n+k)$ after k time-steps. When communication occurs at this instant, $X_3(n+k)$ is replaced by $X_2(n+k)$. Over the next k steps. the error at point 3 propagates to point 4 and also gets modified by the integration process to become $X_4(n+2k)$. Thus, when communication now occurs at $n+2k$, $X_1(n+2k)$ becomes equal to $X_4(n+2k)$ and this process repeats itself. This can be summarized with the following set of expressions:

$$\begin{aligned} X_2(n+k) &= f_1 \cdot X_1(n) \quad , \\ X_3(n+k) &= X_2(n+k) \quad , \\ X_4(n+2k) &= f_2 \cdot X_3(n+k) \quad , \\ X_1(n+2k) &= X_4(n+2k) \quad , \\ X_1(n+2k) &= f_1 \cdot f_2 \cdot X_1(n) \end{aligned} \tag{12}$$

where $f_1\cdot$ and $f_2\cdot$ are operators representing the integration process inside each block. The last expression in equation (12) provides a relationship between the error introduced at time-step n and $n+2k$. It will be shown in the following section that spatial oscillations produce a negative feedback which can be approximated with the following relationship:

$$f_1 \cdot f_2 \cdot \approx -1 \tag{13}$$

Based on this approximation, one can describe the oscillations in time at a boundary point by the following relationship:

$$X_1(n+2k) = -X_1(n) \tag{14}$$

Taking a Z-Transform of the above relation leads to:

$$\begin{aligned} z^{2k}X(z) &= -X(z) \\ (1+z^{2k})X(z) &= 0 \\ z^{2k} &= -1 \end{aligned} \tag{15}$$

The solution of the above equation provides $2k\theta = 2m\pi + \pi, m = 0, 1, 2, 3....$ where $z = re^{I\theta}$. The fundamental solution is $2k\theta = \pi$, corresponding to $m = 0$. Hence the fundamental frequency of oscillations corresponds to a period of $T = 4k\Delta t$.

Origin of the Negative Feedback

As suggested in Equation 13, the net effect of the two operators $f_1\cdot$ and $f_2\cdot$ leads to a system with negative feedback. In order to understand this behavior one has to study the difference representation of the employed three-stage explicit Runge-Kutta scheme. For a wave traveling downstream with a wave speed of $u+a$, one can write a difference equation as follows:

$$\frac{dQ}{dt} = (u+a)\frac{Q_{i-1} - Q_{i+1}}{2\Delta x} \tag{16}$$

where $(u+a)$ is a constant. The explicit Runge-Kutta difference representation yields:

$$\begin{aligned} Q_i^{n+1} &= a_5 Q_{i-3}^n - a_2 Q_{i-2}^n - a_4 Q_{i-1}^n \\ &\quad +(1+2a_2)Q_i^n \\ &\quad +a_4 Q_{i+1}^n + a_2 Q_{i+2}^n + a_5 Q_{i+3}^n \end{aligned} \tag{17}$$

where

$$\begin{aligned} a_2 &= -0.15c^2 \\ a_4 &= 0.135c^3 - 0.5c \\ a_5 &= 0.045c^3 \\ c &= \frac{(u+a)\Delta t}{\Delta x} \end{aligned} \tag{18}$$

The difference equation (17) is then modified near the boundaries and cast in matrix form as follows:

$$\begin{aligned} \{\Delta Q\} &= \{Q_i^{n+1} - Q_i^n\} \; ; \; i = 2, 3, ..., N-1 \\ \{\Delta Q\} &= [B]\{Q_i^n\} + \{B\}'Q_L + \{B\}''Q_R \end{aligned} \tag{19}$$

where $\{\Delta Q\}$ denotes the vector of unknowns, N is the total number of grid points, and Q_L , Q_R are the left and right boundary conditions respectively. The vectors $\{B\}'$ and $\{B\}''$ have the following structure:

$$\{B\}' = \left\{ \begin{array}{c} a_1 \\ -a_2 \\ a_5 \\ \vdots \end{array} \right\} \qquad (20)$$

$$\{B\}'' = \left\{ \begin{array}{c} \vdots \\ -a_5 \\ -a_2 \\ -a_1 \end{array} \right\} \qquad (21)$$

In the above, $a_1 = 0.5c - 0.045c^3$; furthermore, a new variable is defined as $a_3 = 0.09c^3 - 0.5c$. One can also express the matrix $[B]$ in the following form:

$$[B] = \qquad (22)$$

$$\begin{bmatrix} a_2 & a_3 & -a_2 & -a_5 & . & . & . & . & . & . \\ -a_3 & 2a_2 & a_4 & -a_2 & -a_5 & . & . & . & . & . \\ -a_2 & -a_4 & 2a_2 & a_4 & -a_2 & -a_5 & . & . & . & . \\ a_5 & -a_2 & -a_4 & 2a_2 & a_4 & -a_2 & -a_5 & . & . & . \\ . & a_5 & -a_2 & -a_4 & 2a_2 & a_4 & -a_2 & -a_5 & . & . \\ & & & & \vdots & & & & & \\ . & . & . & a_5 & -a_2 & -a_4 & 2a_2 & a_4 & -a_2 & -a_5 \\ . & . & . & . & a_5 & -a_2 & -a_4 & 2a_2 & a_4 & -a_2 \\ . & . & . & . & . & a_5 & -a_2 & -a_4 & 2a_2 & a_3 \\ . & . & . & . & . & . & a_5 & -a_2 & -a_3 & a_2 \end{bmatrix}$$

An eigenvalue-eigenvector decomposition of the matrix $[B]$ shows that the scheme is stable since the real part of all the eigenvalues is negative except for one which is equal to zero. The zero eigenvalue corresponds to the highest frequency spatial oscillation with a wavelength of $\lambda = 2\Delta x$ and hence there is no damping for these high frequency spatial waves. This behavior was also observed for Euler equations from the linearized stability analysis described in Figure 3. The introduction of an error Q_L on the left boundary excites the low frequency waves which are convected with little damping for positive c. Thus, one can state that $f_1 \cdot \approx 1$. On the other hand the introduction of an error Q_R at the right boundary excites the $2\Delta x$ wave again with no damping. This results in $f_2 \cdot \approx -1$.

The behavior described above can be illustrated by a simple one-dimensional example as shown in Figure 12. An error of unit magnitude was applied at both boundaries of a block, and c=0.9 was chosen to advance for $n = 30$ time steps. It can be seen from Equations 20 and 21 for $c = 0.9$, all three non-zero entries in $\{B\}'$ are positive, while the signs of the three non-zero entries in $\{B\}''$ alternate. Therefore, a disturbance applied on the left boundary gets convected downstream with little damping as expected. On the other hand, the one applied on the right boundary produces a high frequency oscillation of wavelength $2\Delta x$ which travels upstream again without being damped by the difference scheme.

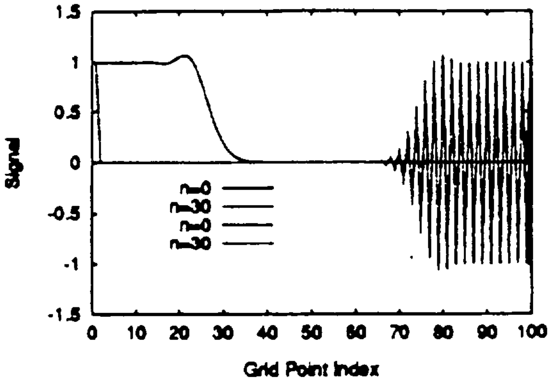


Figure 12: Convection of a disturbance applied on the two boundaries of a block by the Runge-Kutta Scheme

In the above model equation, no artificial viscosity was introduced. In the solution of Euler equations, when an error is introduced on the boundary of a subsonic block, after waiting a reasonable number of time-steps, one can expect that it will appear at one grid point downstream of the boundary with approximately the same magnitude. On the other hand, the same error will appear at one point upstream of the boundary with a negative sign. This behavior is distinctly observed for subsonic flows in the solution of the Euler equations. For supersonic flows, waves traveling upstream are damped by adding numerical viscosity; thus, the feedback and resulting oscillations are negligible.

At this point, one can also comment on the differences observed in the frequency response of the density variation for $k = 10$ and $k = 20$ as shown Figures 6 and 8. If the communication is delayed too long, there is a coupling between the waves originating from different boundaries as well as waves reflecting back. Thus, for $k = 20$, one observes a more complicated frequency response.

Filtering of the Oscillatory Signals

From the discussion of the previous sections, it can be deduced that the freezing of the boundary conditions introduces a high frequency error into the solution with a distinct period of $4k\Delta t$. Since the frequency of the noise is known, one can design a low-pass filter to eliminate the high frequency noise and allow the solution to pass through. To design a simple filter, a moving average was employed. As described in Figure 11 the computed solutions at points 2 and 4 are filtered as follows:

$$\begin{aligned}\overline{Q_2^n} &= \frac{1}{4}\sum_{j=0}^{j=3} Q_2^{n-j\times k} \\ \overline{Q_4^n} &= \frac{1}{4}\sum_{j=0}^{j=3} Q_4^{n-j\times k} \qquad (23)\end{aligned}$$

where k is the communication interval. The right hand side of Equation 23 involves the raw data calculated at every k time-steps at points 2 and 4. Intermediate time-steps are not utilized in filtering. The left hand side defines the filtered value of the boundary condition which is communicated to the neighboring block. This operation corresponds to applying an Finite Impulse Response (FIR) filter, where its z-transform can be expressed as follows:

$$b = \frac{z^{-1} + 2z^{-2} + 3z^{-3} + 4z^{-4} + 3z^{-5} + 2z^{-6} + z^{-7}}{16} \qquad (24)$$

which is always stable.

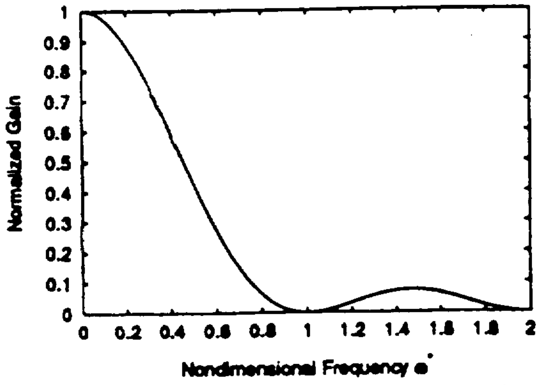


Figure 13: Frequency response of the FIR filter

In Figure 13, the normalized gain of the filter is plotted against non-dimensionalized frequency ω^*. As can be seen from this figure, the selected filter provides zero gain for a non-dimensionalized frequency of unity or $T = 4k\Delta t$.

Results

Steady Flow

For the steady flow test case described previously, a base case solution is obtained initially by communicating every time-step. Then, first, the communication is frozen for 10 steps and the resulting solution is filtered at every communication step before being sent to the neighboring interface. Local time-stepping is used inside each block for both the base case and the case with the filter. Figure 14 shows the density variation at the mid-point of the subsonic interface in block 9 in Figure 4. As can be seen from this figure, the same steady state solution is reached after 5000 time-steps for both cases. There are some differences in the transient behavior of the solution. Figure 15 shows the frequency spectrum of the same density for both solutions. One can observe that the solutions are accurate within a certain frequency range. Second, communication is frozen for 20 steps and the solution is again filtered before communication. Figure 16 shows the resulting density variation for the same subsonic interface, and Figure 17 shows the frequency distribution.

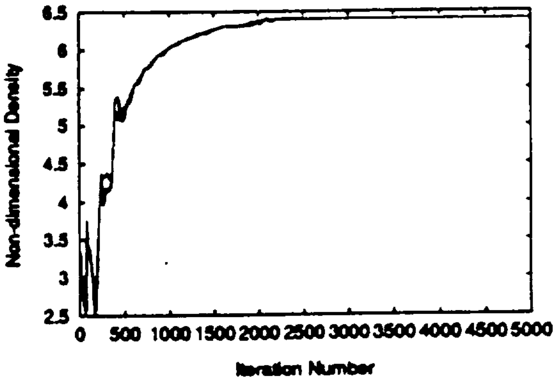


Figure 14: Comparion of solutions with filtering ($k = 10$) and the base case

This steady flow case is solved on varying number of processors to compare the savings in elapsed time due to the reduced communication. Two systems were used to solve the cases; i) an IBM SP2 tower with 32 processors using a fast communication network (HPN) located at Poughkeepsie, New York, and ii) an IBM SP1 tower with 16 processors using an ethernet based communication network located at NASA LeRC. The timings obtained are presented in the form of speedup and efficiency which

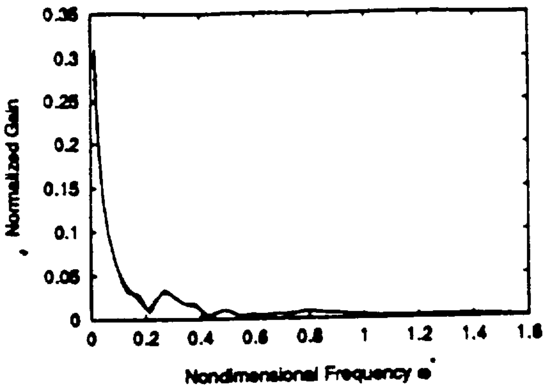


Figure 15: Comparison of the frequency response of solution with filtering ($k = 10$) and the base case

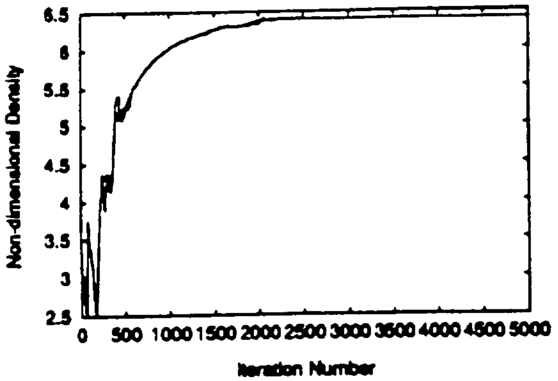


Figure 16: Comparion of solutions with filtering ($k = 20$) and the base case

can be defined as follows:

$$\text{Speedup} = \frac{\text{Elapsed Time with 1 Processor (k=1)}}{\text{Elapsed Time with m Processors}} \tag{25}$$

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{m}} \tag{26}$$

Figures 18 and 19 shows the speedup and efficiency for the steady case for $k = 10$ with filtering and the base case on both types of networks. As can be seen from these figures, a high level of efficiency is maintained, even when a small problem with 4,592 grid points is running over 6-12 processors. Speedup is improved considerably, since the communication cost is reduced by 90-95%. The efficiency improvement is significant, mainly due to the relative importance of the communication cost for the base case. It is also observed that for a slow network like ethernet, communication dominates the total elapsed time for the computation of the problem, and hence dramatic improvements are obtained in the speedup and efficiency when communication is reduced by 90%.

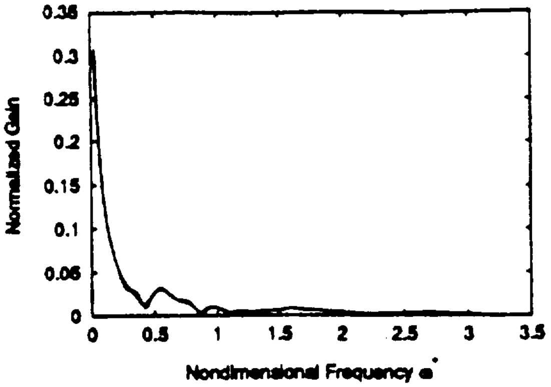


Figure 17: Comparison of the frequency response of solution with filtering ($k = 20$) and the base case

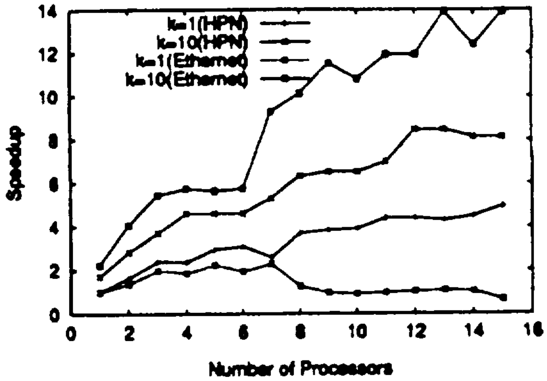


Figure 18: Comparison of speedup with filtering ($k = 10$) and the base case

Unsteady Flow

An unsteady flow test case is chosen as described previously. A base case was run by communicating every time-step to obtain an accurate solution. Between 400-1800 time-steps were employed to integrate over one period of the oscillation for different blocks. Figure 20 illustrates the variation of density at the midpoint of the subsonic interface of block 9 in Figure 4 for $k = 10$ and $k = 20$ without filtering. For $k = 10$, unsteady response is quite accurate. In this case, it was observed that one can freeze the boundaries for $k = 10$, and obtain reasonably accurate solutions even without filtering as shown in Figure 20. This may be due to the fact that the time-steps for the unsteady flow test case are much smaller than those for the steady flow case. In the same figure, it can be seen that communicating every 20 time-steps introduces an error which eventually causes the solution to diverge. For this case filtering can be used to eliminate the error and recover the wave of frequency 100 Hz. Figure 21 illustrates the variation of density at the same location in block

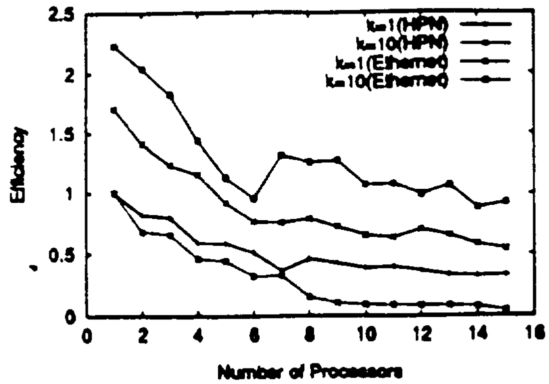


Figure 19: Comparison of efficiency with filtering ($k = 10$) and the base case

9 for two filtered cases with $k = 10$ and $k = 20$ in comparison with the base case. As can be seen from Figure 21, the filtering introduces a slight lag for $k = 20$. The design of another filter may eliminate the lag observed in the $k = 20$ case. Also, for the same case, inaccuracies are observed which are associated with the startup transients.

The frequency spectrum of the solution for the base case and for two filtered cases with $k = 10$ and 20 are shown in Figures 22 and 23. There is very little difference between the frequency content of these three solutions.

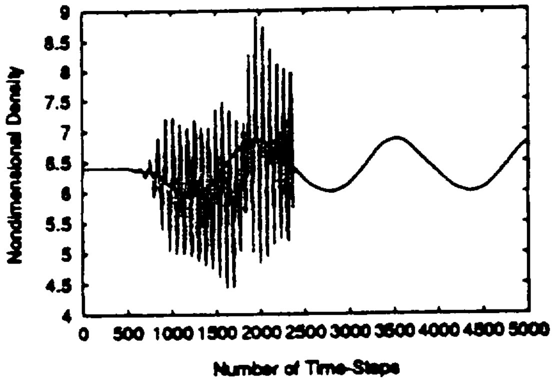


Figure 20: Comparison of the solution for $k = 10, 20$ without filtering

Figures 24 and 25 show the speedup and efficiency for the computation of the unsteady flow case on varying number of processors for both types of networks. Again it can be seen from the figures that reduction of the communication by 95% contributes to a significant improvement in the speedup and efficiency. However, the improvement in speedup and efficiency is not as high as compared to the steady flow case. This is because of the difference in the time-stepping schemes between the two cases. In the steady flow case, local time-stepping is used which means for $k = 10$, communication takes place every 10 computation steps for all interfaces. In the unsteady flow case, each block picks a certain time-step which can be different from other blocks. Hence, for $k = 20$, the number of computation steps before communication occurs can vary from 4 to 20 for various blocks. This can cause communication bottlenecks which could account for the lower efficiency improvements when compared to the steady flow case.

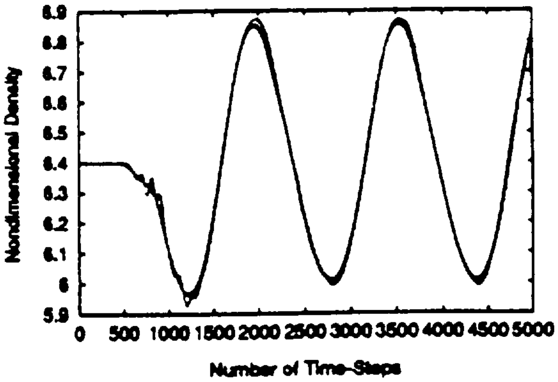


Figure 21: Comparison of the solution for the base case and $k = 10, 20$ with filtering

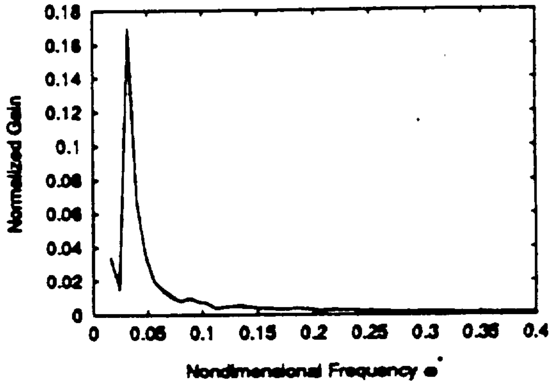


Figure 22: Frequency response of solutions for the base case and $k = 10$ with filtering

Conclusions

In this paper, a filtering procedure is presented to improve the efficiency of parallel computation of Euler equations using an explicit scheme. It is demonstrated that, in terms of obtaining an accurate solution, the time-step chosen by the stability condition for each block may be too restrictive. One can reduce the communication between the blocks by 90-95% and still obtain an accurate solution.

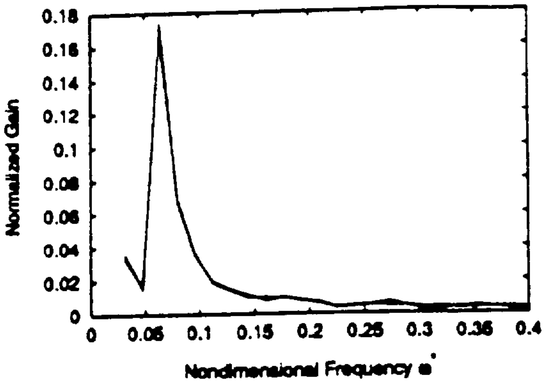


Figure 23: Frequency response of solutions for the base case and $k = 20$ with filtering

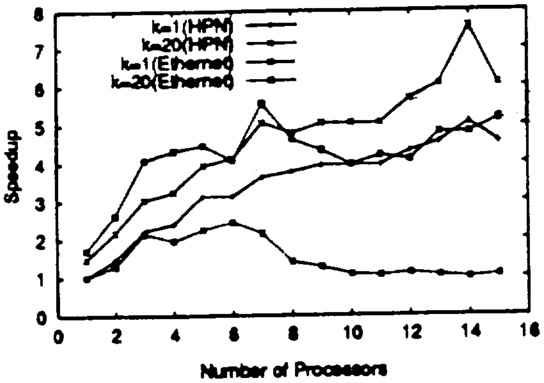


Figure 24: Comparison of speedup with filtering ($k = 20$) and the base case

The filtering procedure coupled with the variable time-stepping schemes enables efficient utilization of the parallel algorithm for both steady and unsteady flows.

It is illustrated that one can communicate with neighboring blocks only when necessary and improve efficiency. Heterogeneity of the flow-field and the computer systems is exploited for this purpose. Study of the interface conditions in the frequency domain provides insight into the problem. Similar filters can be developed for schemes other than Runge-Kutta schemes.

Acknowledgements

This research was funded by the NASA Lewis Research Center under NAG3-1577. The authors appreciate the support provided by Rich Blech, Gary Cole and Joongkee Chung of Computational Technologies Branch of NASA LeRC.

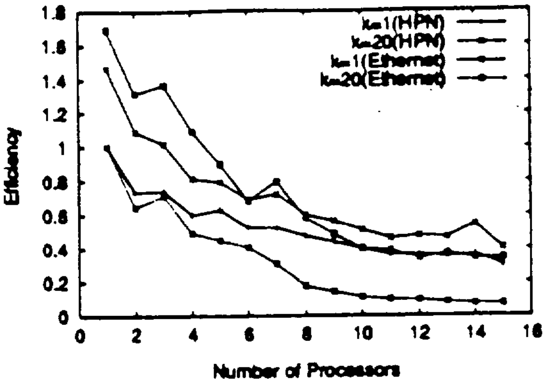


Figure 25: Comparison of efficiency with filtering ($k = 20$) and the base case

References

1. G.K. Cooper and J.R. Sirbaugh, "The PARC Code: Theory and Usage," *Arnold Engineering Development Center TR-89-15,* 1989.

2. N. Gopalaswamy, H.U. Akay, A. Ecer and Y.P. Chien, "Parallelization and Dynamic Load Balancing of NPARC Codes," AIAA Paper No. 96-3302, July 1-3, Lake Buena Vista, FL, 1996.

3. N. Gopalaswamy, Y.P. Chien, A. Ecer, H.U. Akay, R.A. Blech and G.L. Cole, "An Investigation of Load Balancing Strategies for CFD Applications on Parallel Computers," Parallel CFD '95, June 26 - 29, 1995, Pasadena, California, U.S.A.

4. Y.P. Chien, A. Ecer, H.U. Akay, F. Carpenter and R.A. Blech, "Dynamic Load Balancing on a Network of Workstations for Solving Computational Fluid Dynamics Problems," Computer Methods in Applied Mechanics and Engineering, vol. 199, pp. 17-33, 1994.

5. H.U. Akay, R.A. Blech, A. Ecer, D. Ercoskun, B. Kemle, A. Quealy and A. Williams, "A Database Management System for Parallel Processing of CFD Algorithms," Parallel CFD '92, Edited by R.B. Pelz, et al., Elsevier, Amsterdam, pp. 9-23, 1993.

6. J.K. Chung, "Numerical Solution of a Mixed Compression Supersonic Inlet Flow," AIAA Paper No. 940583, 32nd Aerospace Sciences Meeting, Reno, Nevada, 1994.

Filtering Techniques in Parallel Computing

A. Ecer, H.U. Akay and N. Gopalaswamy

CFD Laboratory
Purdue School of Engineering and Technology, IUPUI
Indianapolis, Indiana 46202

1. INTRODUCTION

Our current research efforts are aimed at improving the efficiency of computing on parallel computers. In working with MIMD machines, we have chosen the path of domain decomposition as a basis for parallel computing. The problem to be solved over a given domain is parallelized by means of dividing the domain into many sub-domains, called blocks, and solving the governing equations over these blocks. The blocks are connected to each other through the inter-block boundaries, called interfaces. These blocks can then be allocated to certain processors in the parallel computing environment, and the solution of the problem over the entire domain will be achieved by solving the governing equations over each block in parallel (Akay 1993, Chien 1994, Gopalaswamy 1995, 1996).

Many sub-problems, one for each block, are solved in parallel while they have to communicate in terms of boundary conditions specified at their interfaces. Such an approach can be simplified by assuming that all the blocks are of equal size and require identical computational effort, and that all the processors have identical computation and communication resources. In such a case, one would perform identical computations on each processor and after exchanging messages synchronously proceed towards a solution in a parallel fashion. Such an approach of homogeneous parallel computing may not be very efficient. First, the available computer resources may be heterogeneous. Second, many large problems which require parallel computing are rather complex and cannot be defined as a homogeneous problem. For the case of fluid mechanics problems, each flow region (block) may require a different level of grid refinement, solution strategy and computational effort. Therefore, we expect that the assumption of homogeneity is too restrictive. Although it simplifies the parallelization process, it produces inefficiencies.

Our efforts during the last two years are aimed towards developing schemes which are optimum locally in each flow region. We chose to employ filtering as a way to determine the accuracy and stability conditions for each block and improve the efficiency of computations. Implementation of filtering techniques for improving communications between the blocks is discussed in Gopalaswamy (1997). In this paper, we discuss the utilization of filtering for increasing the efficiency of computations inside each block as related to the accuracy and stability of a given numerical scheme.

2. FILTERING OF BLOCK BASED SOLUTIONS

In using a given numerical scheme, one can improve the efficiency of computations by studying the spectral behavior of the solution. Multi-grid techniques employ coarse grids to act as filters for the solution on fine grids and yet speed-up the rate of convergence. Here, we will be applying classical filtering techniques.

The first problem to be studied is the stability of computations inside a block. For each block, the stability condition as specified by the Courant number calculated for all grid points dictates the time step for the block. For obtaining a steady state, one can sacrifice time accuracy and choose a local time step for each grid point. For explicit schemes, the time step chosen by the stability limit is too restrictive. On the other hand, it is known that for many schemes the stability limit can be a function of the spatial wavelength of the Fourier components of the solution. If one can filter some of the high frequency components of the solution, the time step can be increased and the efficiency of the algorithm can be improved, as it will be discussed below for two different schemes. It is also observed that since the discretization errors are larger for the high frequency components of the solution, filtering may not destroy the accuracy of the solution. One then filters the high frequency components of the output in space before proceeding with the next time step. If the objective is to calculate a steady state solution, once the solution converges to a steady state by using a large time step, the filter can be reduced or removed and integration may continue with a smaller time step. This is similar to the implementation of the multi-grid method, when the filter is switched on and off to obtain an accurate solution with faster convergence.

2.1. FILTERING FOR ACCURACY

The following convection-diffusion equation is studied as a one-dimensional example, for the purposes of investigating the usefulness of a block-based filter to improve stability conditions:

$$w_t + uw_x = \alpha w_{xx} \tag{1}$$

First a difference equation is obtained by a forward difference approximation of the time-derivative, upwind differencing for the first order space derivative, and centered differencing for the second order space derivative. The resulting difference equation can be written as follows:

$$\frac{w_i^{n+1} - w_i^n}{\Delta t} + u\frac{w_i^n - w_{i-1}^n}{\Delta x} = \alpha\frac{w_{i+1}^n - 2w_i^n + w_{i-1}^n}{(\Delta x)^2} \tag{2}$$

A von Neumann type of analysis leads to the following expression for the single-step amplification factor:

$$\frac{w_i^{n+1}}{w_i^n} = G = (1 - c - 2d) + (c + 2d)\cos\theta - Ic\sin\theta \tag{3}$$

where $c = u\frac{\Delta t}{\Delta x}$ and $d = \alpha\frac{\Delta t}{(\Delta x)^2}$. The equation for G is that for an ellipse centered at $1 - c - 2d$ with a major axis of $c + 2d$ and a minor axis of c when drawn on the complex plane. Figure 1 shows a sketch of G.

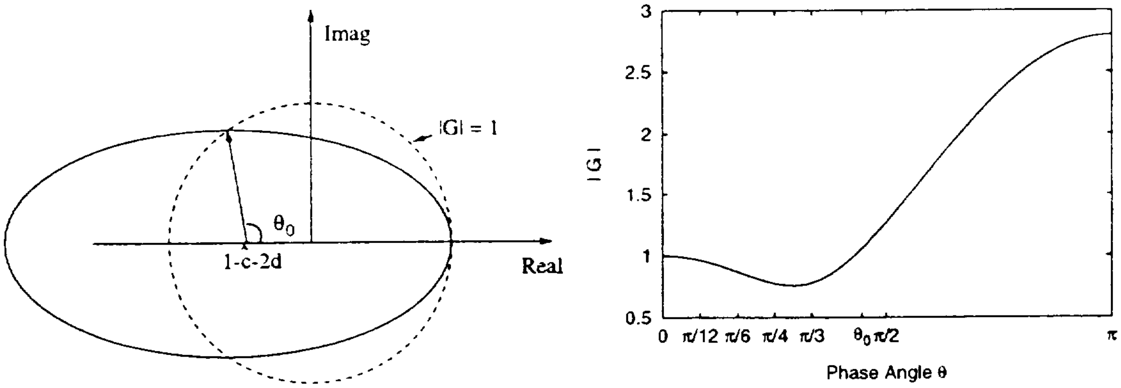


Figure 1: Representation of the function G in the complex plane.

If we define a wavenumber $k_x = \frac{\theta}{\Delta x}$ over a uniform grid, then we can see that all wave numbers are stable when the ellipse lies inside the unit circle on the complex plane. The three stability conditions are:

- $c + 2d < 1$; implies that the center of the ellipse will lie on the positive real axis.
- $c < 1$; implies that the minor axis of the ellipse is less than the radius of the unit circle.

- $c^2 < c + 2d$; implies that the curvature of the ellipse is smaller than the curvature of the unit circle close to $\theta = 0$.

The first condition combined with the second is the most restrictive. The last condition is also important since it directly affects the low frequency waves ($k_x \approx 0$). As can be seen from the figure, violating the first two conditions allows the ellipse to grow on the negative real axis as well as the imaginary axis. Only waves up to a value of θ_o are stable, for other values of θ, the amplification factor G lies outside the unit circle and hence grow with every time-step. Since instability is caused by the high-frequency waves, they can be filtered out. However it is important to preserve the low-frequency waves, and hence the last condition must always be satisfied.

The following example problem was studied in order to obtain a better understanding of the above phenomenon. A sinusoidal signal was convected and diffused in a large domain Ω with a constant speed $u = 0.02$, and $\Omega \approx [-5, 5]$, with $\Delta x = 0.02$ as shown in Figure 2. The diffusion constant α was varied in order to yield different diffusion numbers d.

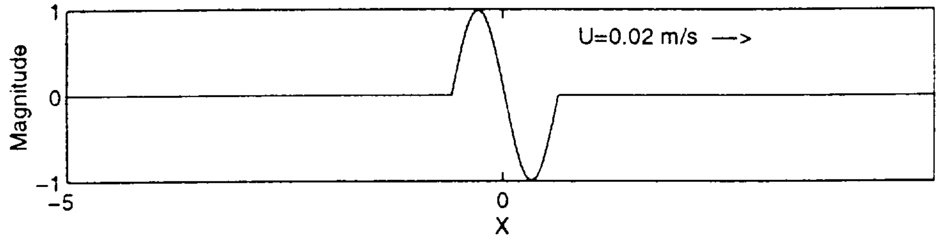


Figure 2: Sinusoidal signal convection and diffusion.

The value of $\cos\theta_o$ is plotted in Figure 3 for various c and d. This can be used to find the cut-off frequency above which waves become unstable.

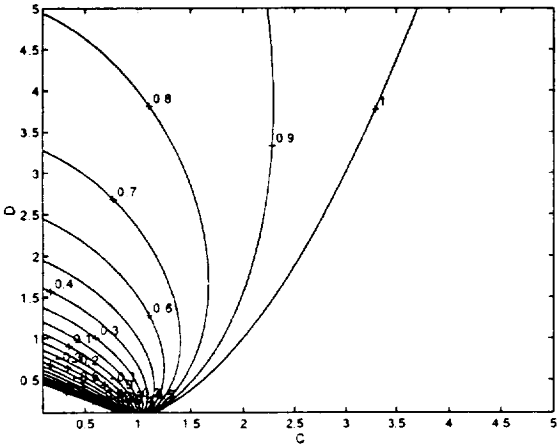


Figure 3: Values of $\cos\theta_o > \frac{c^2-d}{2d(c+d)} + \frac{2d-1}{2d}$.

A low-pass filter was designed to filter out high frequency waves. A combination of $c = 1.01$ and $d = 0.03$, which corresponds to a $\theta_0 = 78^o$, 1.37 rad, was chosen to advance the solution for 50 time steps. The sinusoidal signal is convected over a distance $50\Delta x = 1.0$ and simultaneously diffused. For these conditions, the system slightly exceeds the stability limit and one can still integrate the equations for a short period. In Figure 4, the solution after 50 time-steps with and without filtering is shown. It can be seen that the high frequency error has been damped out by the filter yielding a stable solution. The frequency content of both solutions is displayed next in Figure 5 where the error shows up in the high frequency region. The filter transfer function is also plotted in the same figure, yielding zero gain for the high frequency region.

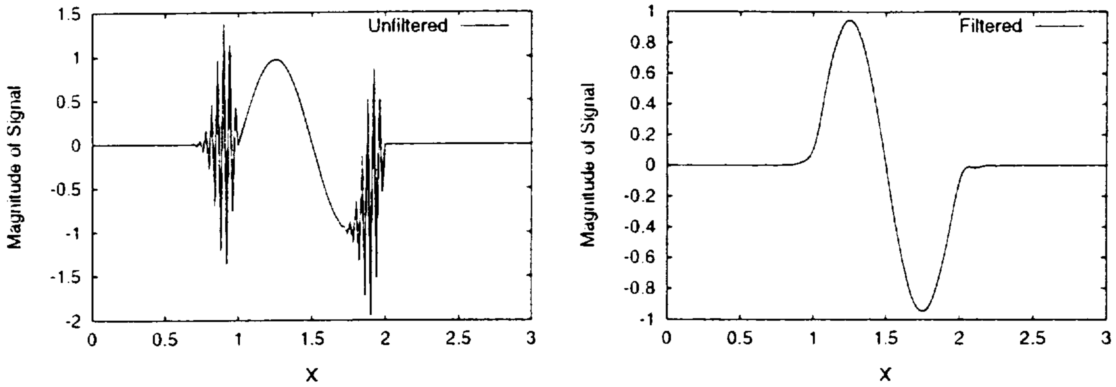


Figure 4: Computed solution after 50 time steps for c=1.01 and d=0.03.

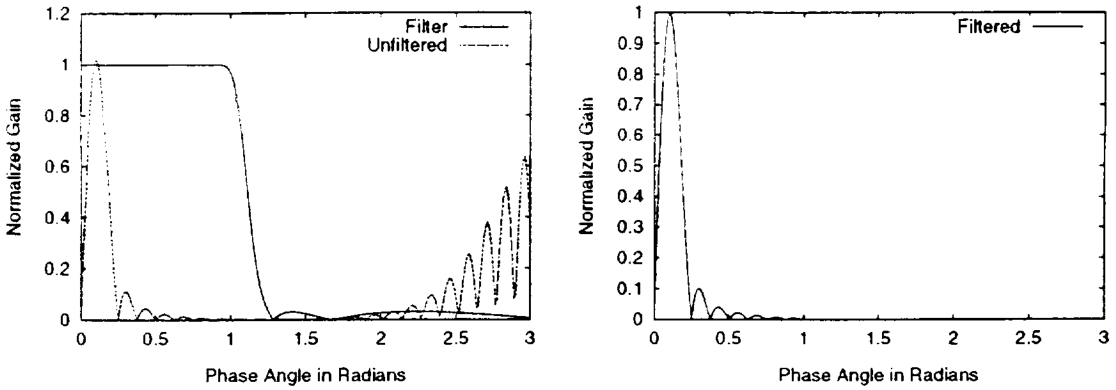


Figure 5: Frequency response of the solutions.

2.2. FILTERING FOR ACCURACY AND STABILITY

Filtering techniques can be employed for improving both the accuracy and stability of a numerical scheme simultaneously. When the time-step is increased, one has to control the level of accuracy. A multistage Runge-Kutta method was considered as a second example. The one-dimensional convection equation was used to study the behavior of this scheme.

$$w_t + uw_x = 0 \tag{4}$$

$$\frac{dw_i}{dt} = u\frac{w_{i-1} - w_{i+1}}{2\Delta x} = RHS \tag{5}$$

The spatial derivative is first discretized with centered differences and the Runge-Kutta method is applied as a separate time-integration of the semi-discretized equation above. A three-stage variant of the Runge-Kutta method leads to the following set of equations:

$$\begin{aligned} w_i(0) &= w_i^n \\ w_i(1) &= w_i(0) + 0.6\Delta t RHS(0) \\ w_i(2) &= w_i(0) + 0.6\Delta t RHS(1) \\ w_i(3) &= w_i(0) + \Delta t RHS(2) \\ w_i^{n+1} &= w_i(3) \end{aligned} \tag{6}$$

where Δt is the time-step used for the temporal integration. Assuming u is a constant, a von Neumann type of stability analysis leads to the following stability criterion:

$$c\sin\theta = \frac{u\Delta t}{\Delta x} \leq 1.8 \tag{7}$$

where θ is the phase angle resulting from a Fourier decomposition of the solution in the spatial frequency domain. Correspondingly, the stability polynomial becomes:

$$G(z) = 1 - z + 0.6z^2 - 0.36z^3 \tag{8}$$

where $z = Ic\sin\theta$. In Figure 6 a plot of the stability polynomial is shown for two Courant numbers $c = 0.9$ and $c = 3.0$.

From the curves it can be seen that the amplification factor $|G|$ is close to unity for both very low frequency ($\theta \approx 0$) and very high frequency ($\theta \approx \pi$) waves. The highest frequency (π) corresponds to a wavelength of $2\Delta x$. The stability polynomial indicates that waves with a wavelength of $4\Delta x$ are the ones to become unstable first.

Utilizing a higher Courant number improves the efficiency of a numerical algorithm, but most of the high frequency waves are unstable for higher Courant numbers. However, using a filtering technique to identify and correct the unstable waves allows one to convect a group of waves at higher Courant numbers. If we assume that c=0.9 provides an accurate solution, we have to design a digital filter to convert G(3.0) to G(0.9) to obtain the same level of accuracy. We would like to construct a G for a Courant number of 3 which approximates the G of $c = 0.9$, i.e. $G(3.0) \approx G^{3.34}(0.9)$. Figure 7 illustrates the accurate transfer function, G(0.9) and the desired transfer functon for the filter, GF=$G^{3.34}(0.9)/G(3.0)$, where G(3.0) is the transfer function for the Runge-Kutta operator with $c = 3.0$. Also shown in this figure are the transfer function for the derived digital filter, GFD, and the combined transfer function of the Runge-Kutta operator ($c = 3.0$) with the filter, GFC=GFD*G(3.0). As can be seen from this figure GFD represents a low-pass FIR+IIR type filter.

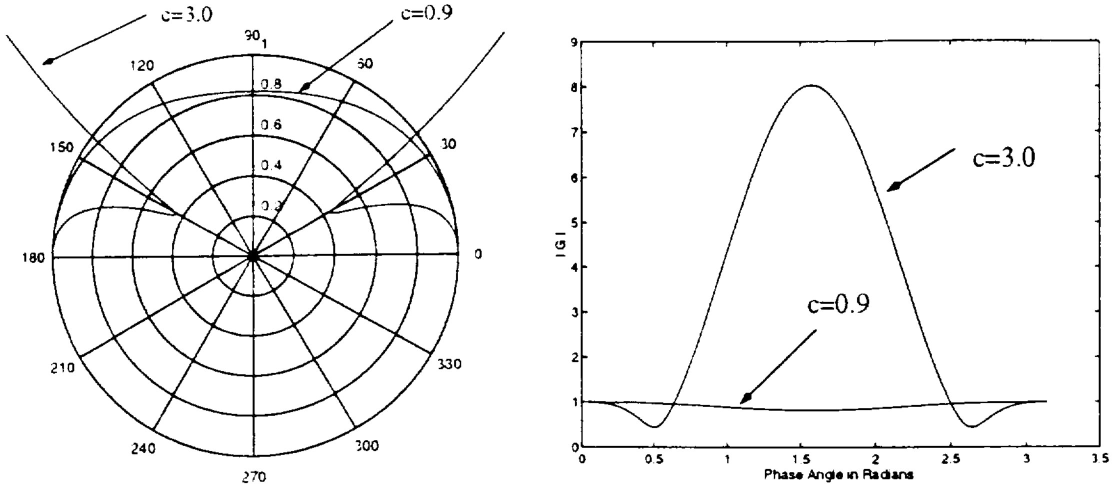


Figure 6: $|G|$ for $0 < \theta < \pi$ and $c = 0.9, 3.0$.

The one-dimensional convection equation is employed to convect the sine wave by using the 3-stage Runge-Kutta scheme at a Courant number $c = 3.0$ following the approach outlined above. The solution of the problem is plotted in Figure 8 at different time-steps. The developed digital filter enables the convection of the waves without diffusion and without violating the conditions for stability.

3. CONCLUSIONS

As discussed in this paper, efficiency of a given solution scheme can be improved by filtering. Our intent is to employ filters locally for each block in parallel computations. We can both monitor and control the speed and accuracy of the computations inside each block by the proposed scheme.

REFERENCES

Akay H.U., Blech R.A., Ecer A., Ercoskun D., Kemle B., Quealy A. and Williams A. (1993) A Database Management System for Parallel Processing of CFD Algorithms, *Parallel CFD '92*, Edited by R.B. Pelz, et al., Elsevier, Amsterdam, pp. 9-23.

Chien Y.P, Ecer A., Akay H.U., Carpenter F. and Blech R.A. (1994) Dynamic Load Balancing on a Network of Workstations for Solving Computational Fluid Dynamics Problems, *Computer Methods in Applied Mechanics and Engineering*, vol. 199, pp. 17-33.

Gopalaswamy N., Chien Y.P., Ecer A., Akay H.U., Blech R.A. and Cole G.L. (1995) An Investigation of Load Balancing Strategies for CFD Applications on Parallel Computers, *Parallel Computational Fluid Dynamics*, Edited by A. Ecer et al., Elsevier, Amsterdam, pp. 703-710.

Gopalaswamy, N., Akay H.U., Ecer A. and Chien Y.P. (1996) Parallelization and Dynamic Load Balancing of NPARC Codes, *AIAA Paper No. 96-3302, 32nd AIAA/ASME/SAE/ASEE Joint Propulsion Conference*, Lake Buena Vista, FL.

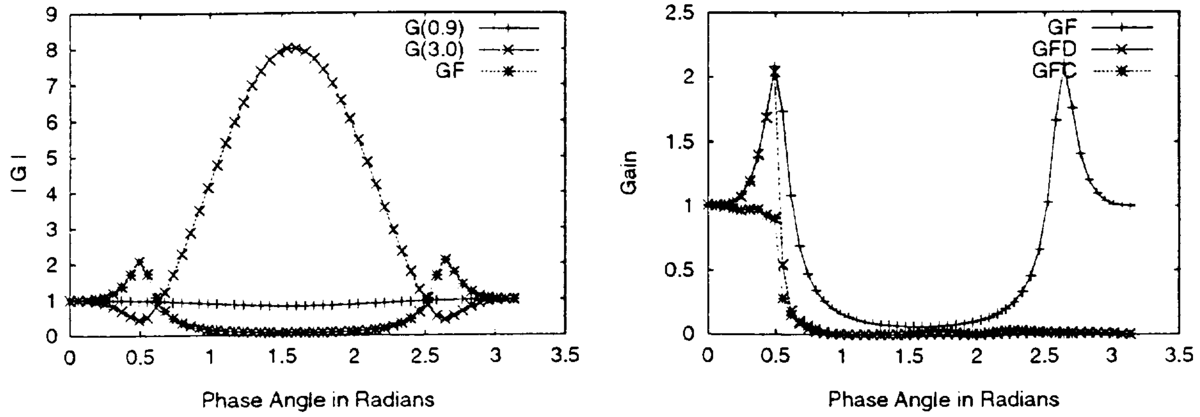


Figure 7: Transfer functions for the Runge-Kutta operators and the filter.

Gopalaswamy N., Ecer A., Akay H. U. and Chien Y.P. (1997) Efficient Parallel Communication Schemes for Explicit Solvers of NPARC Codes, *AIAA Paper No. 97-0027, 35th Aerospace Sciences Meeting*, Reno, Nevada.

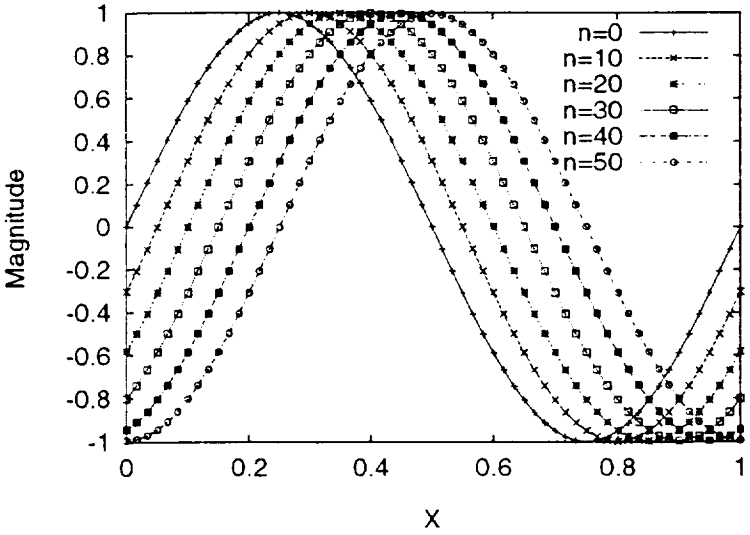


Figure 8: Solution after 50 steps with filtering and $c = 3.0$.

AIAA 98-0616

Digital Filtering Techniques for Parallel Computation of Explicit Schemes

A. Ecer, N. Gopalaswamy, H.U. Akay and Y.P. Chien
Computational Fluid Dynamics Laboratory
Purdue School of Engineering and Technology, IUPUI
Indianapolis, IN

36th Aerospace Sciences Meeting & Exhibit

January 12–15, 1998 / Reno, NV

For permission to copy or republish, contact the American Institute of Aeronautics and Astronautics
1801 Alexander Bell Drive, Suite 500, Reston, Virginia 20191-4344

DIGITAL FILTERING TECHNIQUES FOR PARALLEL COMPUTATION OF EXPLICIT SCHEMES

A. Ecer, N. Gopalaswamy, H.U. Akay and Y.P. Chien

Computational Fluid Dynamics Laboratory
Purdue School of Engineering and Technology, IUPUI
Indianapolis, Indiana 46202

Abstract

A computational technique is presented for designing a filter to improve the computational efficiency of a numerical scheme. For an explicit scheme, the integration time step is increased, causing several waves to become unstable. These waves are filtered without disturbing the accuracy of the solution and the accuracy of the remaining waves are controlled. The scheme is applied to the solution of the Euler equations by using the NPARC code.

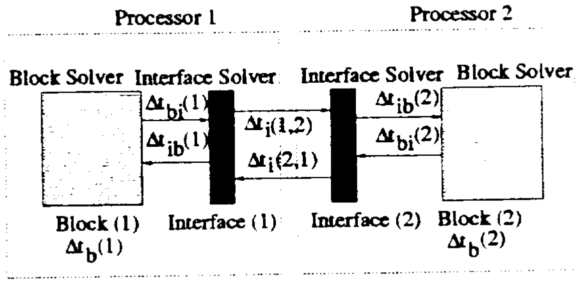


Figure 1: Parallel execution with block and interface solvers.

Introduction

For the solution of complex flow problems, implementation of a computational algorithm requires several important choices. First, a computational grid is generated which reflects the local complexity of the flow with appropriate grid refinement. Then, the computational scheme is adjusted for accuracy and efficiency for the problem in hand based on previous experience. The content of numerical viscosity is usually tested and the time step of integration is prescribed for each problem. In this paper, the utilization of digital filtering techniques is described for treatment of such accuracy and efficiency problems.

The flow problem is defined in a block-structured fashion.[1,2] The flow field is divided into sub-domains called "blocks" which are connected at "interfaces." The algorithm employed to calculate the flow field inside each block is called the "block solver." The accuracy and efficiency of the numerical scheme is defined locally for each block solver. The communication between the block solvers are handled by "interface solvers." This approach is suitable for parallel computing where available computer resources are assigned to each block solver as required by the complexity of the flow in that region.[3] Figure 1 shows a schematic of the relationship between the block and interface solvers in a parallel environment. The time step in each block is denoted by Δt_b, the time step for communication from the parent block to interface is denoted by Δt_{bi}, the time step for communication from an interface to its parent block is denoted by Δt_{ib} and the time step for communication between interfaces is given by Δt_i.

In this paper, the developed techniques were implemented to explicit schemes. Explicit schemes are known to have restrictions on the time step of integration based on the CFL stability condition. As one studies this condition carefully, it states that the system is stable for waves of all possible frequencies on a given grid. On the other hand, it is known that the high frequency waves are not accurately represented by a given difference scheme. Thus, the CFL condition implies that these waves will be numerically integrated even though they may not be accurate.

In the developed scheme, the CFL condition is relaxed. The time step is increased such that the stability of only certain low frequency waves are maintained. The unstable high frequency waves are filtered. As a result of this procedure the efficiency of the computations are increased by obtaining stable

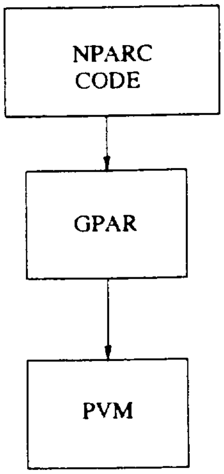


Figure 2: Parallelization with GPAR

solutions at higher time steps without losing accuracy.

The block-filtering scheme is defined for each individual block. A spatial filter is employed inside each block. This scheme replicates some of the functions of multi-grid schemes. In this case, only a single grid is utilized. Also, the choice of the filter is related quantitatively to the spectral contents of the solution. At each time step, after the filtering operation, there is a mismatch at the interfaces for the boundary conditions for each block. This error is also filtered by using a previously developed temporal interface filter.[4] Since needs for accuracy will be different for steady state versus time-accurate solutions one can filter more waves and use a larger time step if time-accuracy is of no concern.

The NPARC code[5] was utilized to demonstrate the developed filtering procedure. This code was parallelized by using some parallelization tools (GPAR, DLB) developed previously.[1,2] PVM[6] is used as a low level message passing library to handle parallel communication and execution. Figure 2 illustrates the relationship between the three components of the parallelized application program. An explicit three-stage Runge-Kutta time stepping scheme was selected. For the chosen two- and three-dimensional inlet problems the CFL limit of C = 1.0 was observed for both steady and time-accurate problems. This limit was then extended to higher Courant numbers by using the developed filtering scheme.

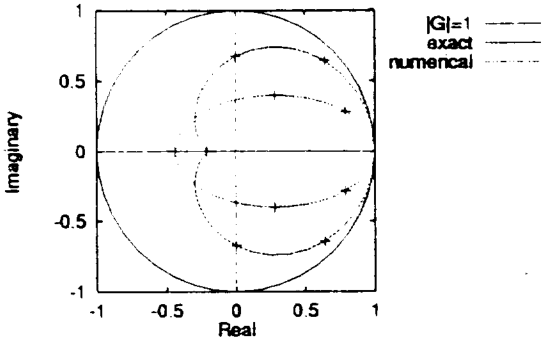


Figure 3a: Amplification factors for c=0.4, d=0.16

Accuracy and Stability of Explicit Schemes

To describe the basics of the developed scheme, the following one-dimensional convection-diffusion equation is considered first:

$$w_t + uw_x = \alpha w_{xx} \tag{1}$$

By using forward differencing in time, upwind and central differencing in space, one can write the following difference equation:

$$\frac{w_i^{n+1} - w_i^n}{\Delta t} + u\frac{w_i^n - w_{i-1}^n}{\Delta x} = \alpha\frac{w_{i+1}^n - 2w_i^n + w_{i-1}^n}{(\Delta x)^2} \tag{2}$$

A von Neumann type of analysis leads to the following expression for the single-step amplification factor:

$$\frac{w_i^{n+1}}{w_i^n} = G = (1 - c - 2d + (c + 2d)\cos\theta_x - Ic\sin\theta_x) \tag{3}$$

where $c = u\frac{\Delta t}{\Delta x}$ and $d = \alpha\frac{\Delta t}{(\Delta x)^2}$ and θ_x is the phase angle in space. The equation for G is that for an ellipse centered at $1 - c - 2d$ with a major axis of $c + 2d$ and a minor axis of c when drawn on the complex plane. Figures 3a and 3b show a sketch of G for two combinations of c and d.[7] The scheme is stable for the first case. The value of θ_x for which the scheme becomes unstable is approximately 120 degrees for the second case.

We can see that G is stable for all phase angles θ_x when the ellipse lies inside the unit circle on the complex plane. The three stability conditions are:

1. $c + 2d < 1$; implies that the center of the ellipse will lie on the positive real axis.

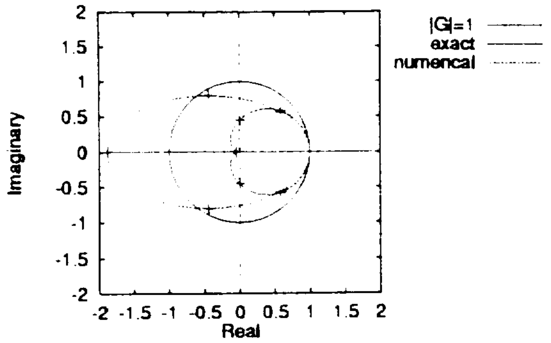


Figure 3b: Amplification factors for c=0.8, d=0.32

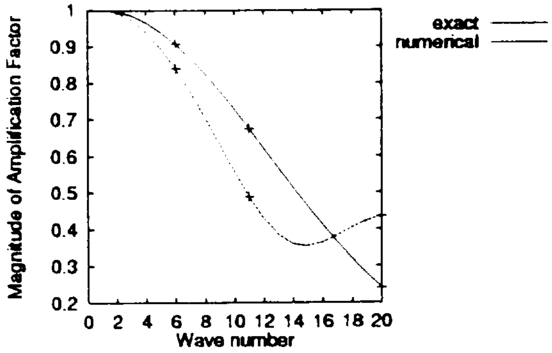


Figure 4a: Spectral decomposition of amplification factors for c=0.4, d=0.16

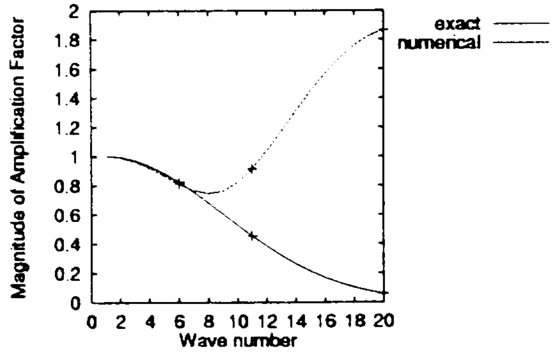


Figure 4b: Spectral decomposition of amplification factors for c=0.8, d=0.32

2. $c < 1$; implies that the minor axis of the ellipse is less than the radius of the unit circle.

3. $c^2 < c + 2d$; implies that the curvature of the ellipse is smaller than the curvature of the unit circle close to $\theta_x = 0$.

The first condition is the most restrictive one. The last condition is also important since it directly affects the stability of the low frequency waves ($\theta_x \approx 0$). As can be seen from Figure 3b, violating the first 2 conditions allows the ellipse to grow on the negative real axis as well as the imaginary axis. For the second case, only waves up to a value of $\theta_x \approx 120$ degrees are stable, for other values of θ_x, the amplification factor G lies outside the unit circle and hence grow with every time-step. If one can filter these high frequency waves, it is possible to obtain stable and accurate solutions at such Courant numbers. However, it is important to preserve the low-frequency waves, and hence the last condition must always be satisfied.

If one considers the accuracy of the convection-diffusion equation Eq. (2), the spectrum of the differential equation in Eq. (2) can be compared to that of the difference equation as follows:

$$G_{exact} = e^{-\alpha\omega_x^2\Delta t}\left(\cos(\omega_x u\Delta t) - I\sin(\omega_x u\Delta t)\right) \tag{4}$$

$$G_{num} = (1 - c - 2d + (c + 2d)\cos\theta_x - Ic\sin\theta_x) \tag{5}$$

where, $\omega_x = \theta_x/\Delta x$.

In Figures 3a and 3b the plus (+) symbols denote the amplification factors corresponding to wavelengths of $2\Delta x, 4\Delta x$, and $8\Delta x$. Since the difference equation has no imaginary components, the amplification factor is symmetric about the real axis. As can be seen from the figures, even for low Courant numbers, accuracy of high frequency components of the solution is not very high. Figures 4a and 4b show the magnitudes of the amplification factors for both cases. Even though the magnitude responses of the exact and numerical schemes are close, their phase is different even for low frequency waves.

Also, steady state solutions of the two equations can be compared in a similar manner. Figure 5 shows the steady state solution of the convection diffusion equation for boundary conditions 0 and 1 at each end of a domain of length 1.0, as computed from Eqs. (1) and (2). The spectral decomposition of the error is shown in Figure 6. The accuracy of the steady state solution is also dominated by the low frequency waves. A similar spectral decomposition of the three-stage Runge-Kutta scheme for the one-dimensional, inviscid, convection equation,

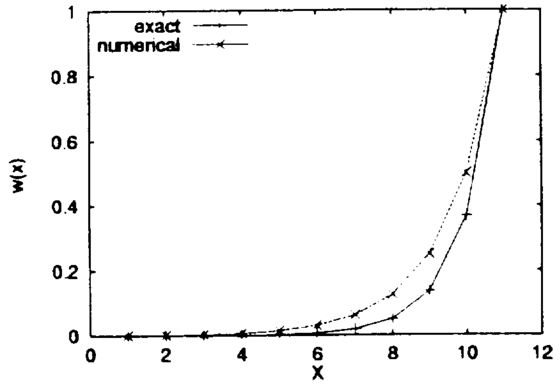


Figure 5: Spatial exact and numerical solutions

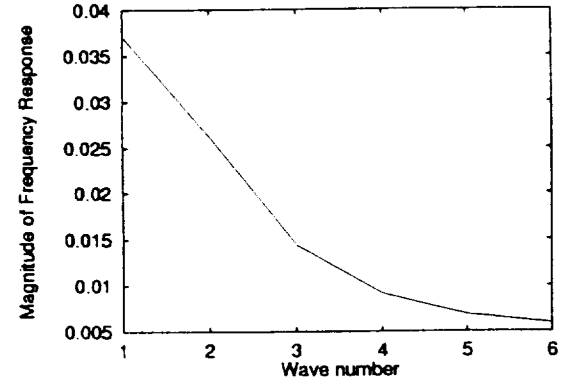


Figure 6: Spectral decomposition of error

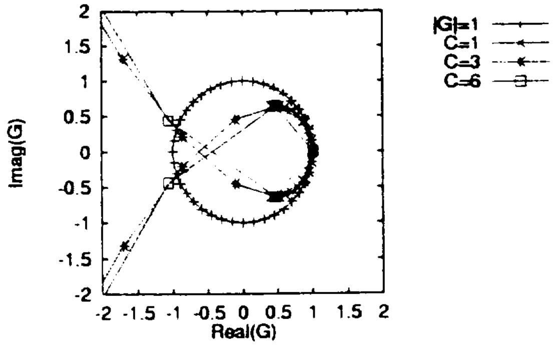


Figure 7a: Spectral decomposition of the amplification factor of 3-stage R-K scheme

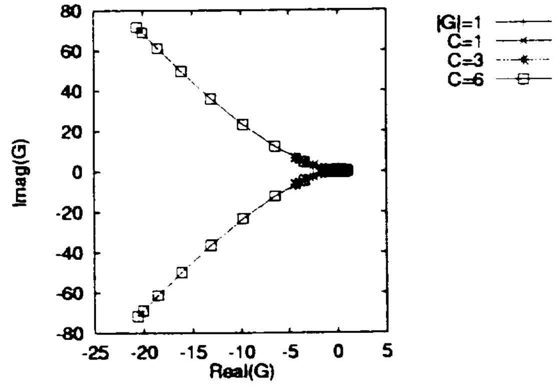


Figure 7b: Spectral decomposition of the amplification factor of 3-stage R-K scheme

($\alpha = 0$), as obtained through a von Neumann stability analysis, is shown in Figure 7a for different Courant numbers. The amplification factor for the exact solution for the convection equation is the unit circle. For C=1, the scheme is stable for all waves, the numerical errors are maximum for waves of $4\Delta x$ or a phase angle of 90 degrees. This scheme is time accurate for the low frequencies. For high Courant numbers, the scheme is stable for only a range of low frequencies. For $C = 6$, the magnification factor increases considerably as shown in Figure 7b. Here the objective is to filter such high frequency components of this solution and obtain stable results.

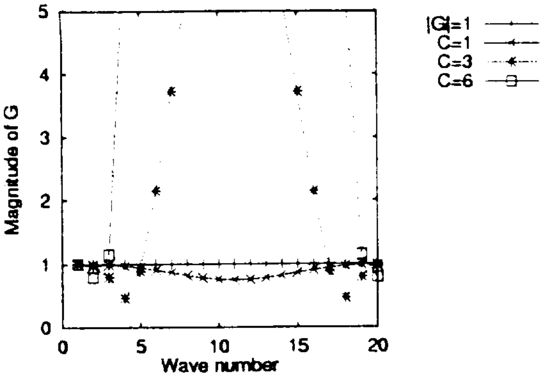


Figure 7c: Spectral decomposition of the amplification factor of 3-stage R-K scheme

Numerical Integration of Euler Equations

The derivations will be restricted to the axisymmetric form of Euler equations, dropping the viscous terms of the Navier-Stokes equations for brevity.

$$\frac{\partial \bar{Q}}{\partial t} + \frac{\partial \bar{E}}{\partial x} + \frac{\partial \bar{F}}{\partial y} + \bar{H} = 0 \qquad (6)$$

$$\bar{Q} = \frac{Q}{J} \quad ; \quad \bar{E} = \frac{1}{J}(\xi_x E + \xi_y F) \qquad (7)$$
$$\bar{H} = \frac{H}{J} \quad ; \quad \bar{F} = \frac{1}{J}(\eta_x E + \eta_y F)$$

$$Q = \left\{ \begin{array}{c} \rho \\ \rho u \\ \rho v \\ \rho e \end{array} \right\} ; \; H = \frac{1}{y} \left\{ \begin{array}{c} \rho v \\ \rho v u \\ \rho v^2 + P \\ (\rho e_t + P) v \end{array} \right\} \qquad (8)$$

$$E = \left\{ \begin{array}{c} \rho u \\ \rho u^2 + P \\ \rho u v \\ (\rho e_t + P) u \end{array} \right\} ; \; F = \left\{ \begin{array}{c} \rho v \\ \rho v u \\ \rho v^2 + P \\ (\rho e_t + P) v \end{array} \right\} \qquad (9)$$

where J is the Jacobian of coordinate transformation. The three-stage Runge Kutta time-stepping scheme is written as follows:

$$\frac{\partial \bar{Q}_{i,j}}{\partial t} = \text{RHS} \qquad (10)$$

$$\text{RHS} = \frac{\bar{E}_{i-1,j} - \bar{E}_{i+1,j}}{2\Delta\xi} + \frac{\bar{F}_{i,j-1} - \bar{F}_{i,j+1}}{2\Delta\eta} - \bar{H}_{i,j} \qquad (11)$$

$$\begin{aligned} \bar{Q}_{i,j}(1) &= \bar{Q}_{i,j}(n) + 0.6\Delta t \text{ RHS}(n) \\ \bar{Q}_{i,j}(2) &= \bar{Q}_{i,j}(n) + 0.6\Delta t \text{ RHS}(1) \qquad (12) \\ \bar{Q}_{i,j}(n+1) &= \bar{Q}_{i,j}(n) + \Delta t \text{ RHS}(2) \end{aligned}$$

where central discretization is used for evaluating the source term in Equation (11) for each coordinate direction and n denotes the time level or iteration level.

Stability of the Runge-Kutta Scheme

For the purposes of a linearized stability analysis, the inviscid fluxes along the coordinate directions are transformed according to the following relationships:

$$\frac{\partial \bar{E}}{\partial \xi} = \frac{\partial \bar{E}}{\partial \bar{Q}} \frac{\partial \bar{Q}}{\partial \xi} = A \frac{\partial \bar{Q}}{\partial \xi} \qquad (13)$$

$$\frac{\partial \bar{F}}{\partial \eta} = \frac{\partial \bar{F}}{\partial \bar{Q}} \frac{\partial \bar{Q}}{\partial \eta} = B \frac{\partial \bar{Q}}{\partial \eta} \qquad (14)$$

For the purposes of the stability analysis, the source term $\bar{H}_{i,j}$ is neglected. Expanding the solution in a Fourier series assuming periodic boundary conditions yields:

$$\bar{Q}_{i,j} = \sum_{l=0}^{N_\xi - 1} \sum_{m=0}^{N_\eta - 1} \hat{Q}_{lm}(t) e^{I\theta_{\xi,l} i} e^{I\theta_{\eta,m} j} \qquad (15)$$

where $\hat{Q}$ is the amplitude of a particular harmonic and N_ξ, N_η are the number of grid points in the ξ and η coordinate directions respectively. Considering the stability of a single harmonic, the amplification matrix G of the harmonic can be obtained as:

$$\begin{aligned} G(\theta_\xi, \theta_\eta) &= \frac{\hat{Q}^{n+1}}{\hat{Q}^n} \\ &= \left(Y + \Delta t N + 0.6\Delta t^2 N^2 + 0.36\Delta t^3 N^3\right) \\ N &= -I\left(A^n \sin(\theta_\xi) + B^n \sin(\theta_\eta)\right) \qquad (16) \end{aligned}$$

where, Y is the identity matrix, and θ_ξ and θ_η are the spatial phase angles in the ξ and η coordinate directions respectively. Matrix N is a function of the local Mach number, flow direction and the grid dimensions.

Design of a Block Filter

If one assumes that the numerical integration of the Euler equations with a Courant number $C =$

1 for the three-stage Runge-Kutta scheme provides an accurate and stable solution, the objectives in designing a filter can be summarized as follows:

- accuracy problem: the filter should provide accurate solution for the low frequency waves for $C = p, p > 1$.
- stability problem: the filter should stabilize or in this case eliminate the unstable, high frequency portion of the solution with Courant number $C = p$, with $p > 1$. These objectives are achieved by filtering the residual vector after each numerical integration step.

Accuracy Problem

The accuracy problem is treated by comparing the two solutions obtained by different time steps. The change in a specific harmonic of the residual, when integrated by $C = 1$ and after p time steps, can be written as follows:

$$\hat{Q}^{n+p}_{C=1} - \hat{Q}^n = \left((G_{C=1})^p - Y\right)\hat{Q}^n \qquad (17)$$

On the other hand, the change in a harmonic of residuals after one integration time step with $C = p$, is equal to,

$$\hat{Q}^{n+p}_{C=p} - \hat{Q}^n = \left(G_{C=p} - Y\right)\hat{Q}^n \qquad (18)$$

We can define a filter which will equate these two residuals.

$$F * \left(\hat{Q}^{n+p}_{C=p} - \hat{Q}^n\right) = \hat{Q}^{n+p}_{C=1} - \hat{Q}^n \qquad (19)$$

Thus, by multiplying the $C = p$ residuals with this filter, we can obtain accurate solutions for all waves if such a filter can be designed.

The filter matrix is defined by the following expression:

$$\begin{aligned} F &= \left((G_{C=1})^p - Y\right) * \left(G_{C=p} - Y\right)^{-1} \\ &= F(M, \phi, \theta_\xi, \theta_\eta) \qquad (20) \end{aligned}$$

The filter matrix F is a 4x4 complex matrix whose elements are a function of the Mach number M, flow angle with respect to the coordinate directions (ξ, η) and the phase angles in each coordinate direction θ_ξ and θ_η. It should be noted that, the solution $\hat{Q}^{n+p}_{C=p}$ obtained with Courant number p is unstable, and the filtering operation should in theory produce a stable solution for all frequencies.

Stability Problem

For a linear problem away from boundaries, the filter, as defined in Eq. (20), may provide a stable and accurate computation of all waves. However, for non-linear problems, it becomes very difficult to design a filter which can stabilize all low and high frequency waves and still provide accuracy. For $C = 6$, Figure 7b illustrates that certain waves become highly unstable. In this case, rather than obtaining an accurate solution for these waves, a more practical approach of filtering these waves is proposed. The filter matrix F is further modified by multiplying it with a low pass filter which damps out the high frequency components of the solution including all unstable waves. The filter is designed to provide an accurate solution only for the remaining low frequency waves.

An eigenvalue-eigenvector decomposition of the amplification matrix G yields the following CFL condition for the spectral radius of G:

$$|\lambda_{\max}(G)| < 1 \qquad (21)$$

$$C_{\max} = \Delta t \left(U \sin(\theta_\xi) + V \sin(\theta_\eta) + a\sqrt{(\xi_x^2 + \xi_y^2)\sin(\theta_\xi) + (\eta_x^2 + \eta_y^2)\sin(\theta_\eta)}\right) \le 1.8 \qquad (22)$$

where, $U = \xi_x u + \xi_y v, V = \eta_x u + \eta_y v$ and a is the speed of sound.

If N_ξ and N_η are the total number of grid points in the ξ and η directions respectively, the number of low frequency waves to keep, n_ξ and n_η are chosen such that for $\theta_\xi = 2\pi\frac{n_\xi}{N_\xi}$, and $\theta_\eta = 2\pi\frac{n_\eta}{N_\eta}$, the above stability condition is satisfied. The final filter is defined as:

$$\begin{aligned} F^* &= F_{lp} F\,, \qquad (23) \\ F_{lp} &= \Omega Y \qquad (24) \end{aligned}$$

where, Ω is unity for all θ_ξ, θ_η for which the scheme is stable, and zero for all other waves for $\hat{Q}^{n+p}_{C=p}$.

FFT Implementation

After approximately every n (e.g. n=100), computational steps, the filter matrix $F(\theta_\xi, \theta_\eta, M, \phi)$ is evaluated for each block by computing an average Mach number and flow angle ϕ in the block. The numerical implementation of the filter matrix is of size $F(4, 4, jmax, kmax)$ where $jmax$ and $kmax$ are the total number of grid points in the ξ and η coordinate

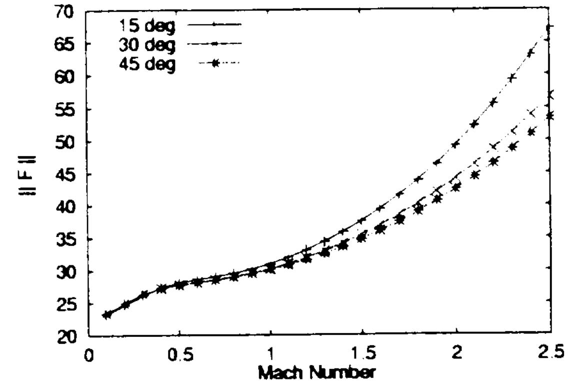


Figure 8: Matrix norm of F for various Mach numbers and flow angles

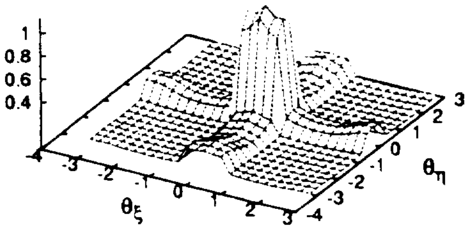


Figure 9: Magnitude of F^*_{44} for $\phi = 10$ degrees

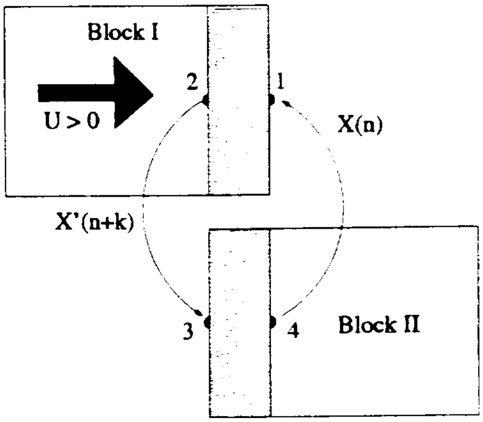


Figure 10: Feedback Model

directions respectively. The two-dimensional FFT of the unstable residual obtained after a complete Runge-Kutta cycle with a Courant number of $p > 1$ is computed using a separate subroutine.[8] Next a matrix vector product of the unstable residual with the filter matrix is carried out for each phase angle θ_ξ and θ_η. Finally, the complex coefficients obtained are damped further for the values of θ_ξ and θ_η by multiplying those coefficients with a very small number (≈ 0.01). The final coefficients are then used for the inverse FFT to yield the filtered residual in the spatial domain. The filtered residual is then added to the solution $\bar{Q}^n$ to yield the stable and accurate solution for $\bar{Q}^{n+p}_{C=p}$. Figure 8 shows the computed matrix norm for a 14x21 grid block for a range of Mach numbers between 0.1 to 2.5 and a range of flow angles $0 < \phi < \pi/2$. From this figure it can be seen that the filter matrix is more sensitive for supersonic Mach numbers compared to subsonic Mach numbers. A frequency response of the F^*_{44} element for $\phi = 10$ degrees and for the same 14x21 grid block is shown in Figure 9. As can be seen from this figure, high frequency waves are filtered out.

Interface Filtering In Time

During the parallel computation of the flow problem, the difference equation is integrated in time for all the grid points of each block.[2,3] The solution values at the interfaces are held fixed during a Runge-Kutta cycle, and information is exchanged after proceeding one time-step. For small Courant numbers $C \leq 1$, this freezing of boundary conditions at the interfaces produces negligible oscillations in the solution inside the blocks. However, for larger Courant numbers, e.g. 3 and 6, these oscillations are reinforced by a feedback system originating from the Runge-Kutta scheme applied at the boundary points.[4] Also, one of the assumptions of the block filtering procedure is that the solution inside the block is periodic. The Fourier decomposition is inaccurate near the boundaries if the non-periodicity is strong, which produces discontinuities in the residuals across the interfaces when information is exchanged. As discussed previously,[4] an error is introduced at the interfaces which then forms a negative feedback system. As an example, consider the pair of blocks in Figure 10. If the semi-discretized Euler equations can be expressed as follows:

$$\frac{dQ}{dt} = A \cdot Q \tag{25}$$

For a linearized operator $A\cdot$, the error obeys the same difference equation as the solution. Let us call the error introduced due to the non-periodicity of the solution across the interfaces as X.

$$\frac{dX}{dt} = A \cdot X \qquad (26)$$

If $X_1(n)$ is an error in the boundary condition, first introduced at a time-step n, at point 1, this error will propagate upstream in Block I to point 2 at the next communication interval. Assume it propagates to point 2 to become $X_2(n+1)$ before the next exchange. When communication occurs at this instant, $X_3(n+1)$ is replaced by $X_2(n+1)$. The error at point 3 propagates to point 4 to become $X_4(n+2)$.

$$\begin{aligned} X_2(n+1) &= f_1 \cdot X_1(n) \quad , \\ X_3(n+1) &= X_2(n+1) \quad , \\ X_4(n+2) &= f_2 \cdot X_3(n+1) \quad , \\ X_1(n+2) &= X_4(n+2) \quad , \\ X_1(n+2) &= f_1 \cdot f_2 \cdot X_1(n) \end{aligned} \qquad (27)$$

where $f_1 \cdot$ and $f_2 \cdot$ are operators representing the integration process inside the block. The last expression in equation (27) provides a relationship between the error introduced at time-step n and $n+2$. It was shown[4] that during the numerical integration, the introduced error leads to a negative feedback which can be approximated with the following relationship:

$$f_1 \cdot f_2 \cdot \approx -1 \qquad (28)$$

Based on this approximation, one can describe the oscillations in time at a boundary point by the following relationship:

$$X_1(n+2) = -X_1(n) \qquad (29)$$

Taking a Z-Transform of the above relation leads to:

$$\begin{aligned} z^2 X(z) &= -X(z), \\ z^2 &= -1 \end{aligned} \qquad (30)$$

The solution of the above equation provides $2\theta_t = 2m\pi + \pi, m = 0, 1, 2, 3....$ where $z = re^{I\theta_t}$. The fundamental solution is $2\theta_t = \pi$, corresponding to $m = 0$. Hence the fundamental frequency of oscillations corresponds to a period of $T = 4\Delta t$. The filter developed previously[4] can thus be applied to this signal to yield zero gain for this wavelength. The interface filter is developed for an interval corresponding to p where $C = p > 1$ is the Courant number used inside the blocks. The solution at the interfaces is sampled every communication step, which is equal to p, and filtered based on averaging of the solution stored for the current communication step and the previous 3 communication steps. The filter is of type FIR and its Z-transform looks like:

$$B(z) = \frac{z^{-1} + 2z^{-2} + 3z^{-3} + 4z^{-4} + 3z^{-5} + 2z^{-6} + z^{-7}}{16} \qquad (31)$$

Test Cases

The following two test cases were considered:

1. An axisymmetric mixed-compression VDC (Variable Diameter Centerbody) inlet is considered under a subsonic inflow of M=0.3 and a subsonic compressor face outflow Mach number M=0.4. The inlet geometry supplied[9] was modified by increasing the throat area to permit subsonic unchoked flow throughout the inlet. The 2D version of the NPARC code has an option to handle axisymmetric flow also. The reference inlet pressure is 117.8 lb/ft^2, and the reference inlet temperature is 395 Rankine. The cowl-tip radius of the inlet, Rc=18.61 inches is used to non-dimensionalize the lengths. The grid for this inlet consists of approximately 4500 nodes, and is divided into 15 blocks, all of approximately equal size as shown in Figure 11. First a steady-state solution is sought using local time-stepping for all nodes in each block with a uniform Courant number of 1.0 for all nodes. Then a Courant number of 3.0 is used for all nodes and block and interface filtering is switched on to obtain a stable steady state solution.

2. The same geometry as defined in test case 1 is chosen, except the grid is refined 3 times in the flow direction. Refined grid increases the number of the stable waves and allows accurate solutions even when the Courant number is increased to 6. The resulting refined grid is shown in Figure 12. Also, the inlet Mach number is increased to 0.5 and the exit Mach number is fixed at 0.6. This was done to study the behavior of the filter for a different Mach number and also to achieve convergence to steady state in the same number of iterations as that for test case 1. The same blocking strategy as in test case 1 was used.

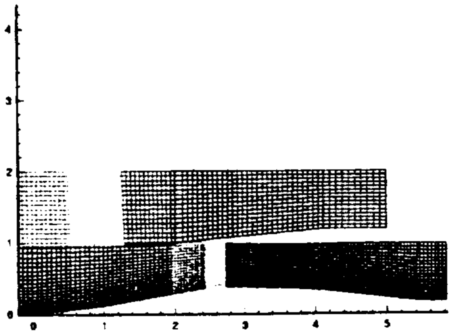


Figure 11: Grid for axisymmetric engine inlet with 15 blocks (test case 1)

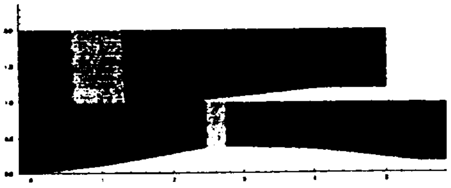


Figure 12: Grid for axisymmetric engine inlet with 15 blocks (test case 2)

Results

The test cases were run on an IBM SP2 parallel supercomputer located in Poughkeepsie, New York. The communication subsystem used by the SP2 is HPN (High Performance Network) using a switched Fast Ethernet. Up to 15 of the available 16 processors on the SP2 were used for the current study.

As described in test case 1, first a steady state solution is obtained for the prescribed geometry and flow conditions. Next, the Courant number was increased to 3.0 for all nodes in each block. A filter matrix as defined in Eq. (20) was recalculated every 100 steps. Only 4 out of 21 waves were kept as defined in Eq. (24). Also an interface filter designed for a Courant number of 3.0 was used to damp oscillations near the boundaries. The solutions obtained are plotted in the form of the nondimensional density variation at the midpoint of each block. Figures 13–14 show a comparison of the solution obtained with the two Courant numbers. The iteration number for the case with Courant number 3.0 in the figures have been normalized to those for Courant number 1.0., i.e., the iteration number for a Courant number of 3.0 is scaled by 3. The solution components because they have been damped out by the block and interface filter. However, the final steady state solution reached with both Courant numbers is the same, and hence it is not necessary to integrate the high frequency components if only a steady state solution is desired. The above procedure is similar to a multigrid scheme where high frequency waves are filtered by using a coarse grid. In this case only a single grid is utilized. The number of waves to be kept is determined based on stability and accuracy conditions. The basic Runge-Kutta algorithm is not modified; only a filtering algorithm is added to modify the solution at each time-step. Finally, for each block a different filter is designed based on local flow conditions and grid size.

The Courant number is increased to 6.0 as described in test case 2 with the refined grid. Block and interface filtering is switched on to damp the high frequency oscillations (for wave numbers grater than 4), arising from the instability of the explicit Runge-Kutta scheme for this Courant number. The solution is again plotted in the form of the nondimensional density variation with iteration number normalized to a Courant number of 1.0. From Figures 15–16 it can be seen that as in the results for test case 1, the high frequency components are absent from the solution for C=6.0. However, the steady state solution obtained with a Courant num-

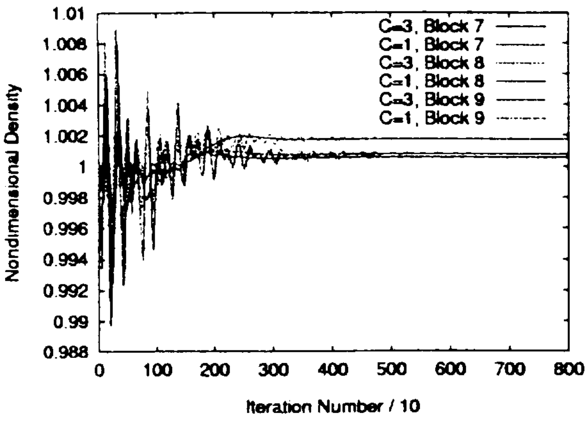


Figure 13: Density variation at midpoint of blocks 7-9

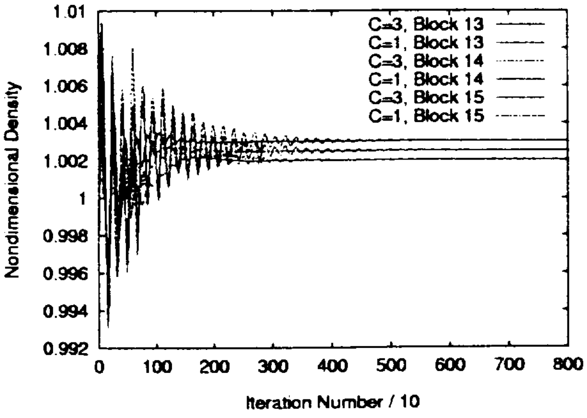


Figure 14: Density variation at midpoint of blocks 13-15

ber of 6.0 is the same as the steady state solution obtained with a Courant number of 1.0.

Finally, to provide an idea of the expected improvements in the parallel speedup and efficiency from the above filtering techniques, Figures 17 and 18 show the speedup and efficiency obtained for the two test cases with the IBM SP2 parallel supercomputer. Speedup and efficiency for these cases are defined as follows:

$$\text{Speedup} = \frac{\text{Elapsed Time with C} = 1.0 \text{ on 1 Processor}}{\text{Elapsed Time on n Processors}} \tag{32}$$

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{n}} \tag{33}$$

From Figures 17 and 18 it can be seen that very high parallel speedup and efficiency can be achieved with the implementation of the filtering techniques,

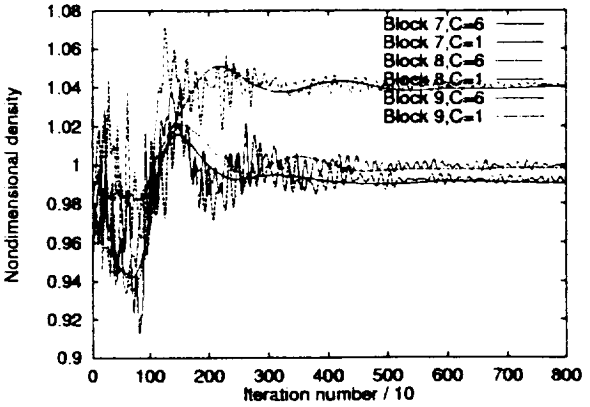


Figure 15: Density variation at midpoint of blocks 7-9

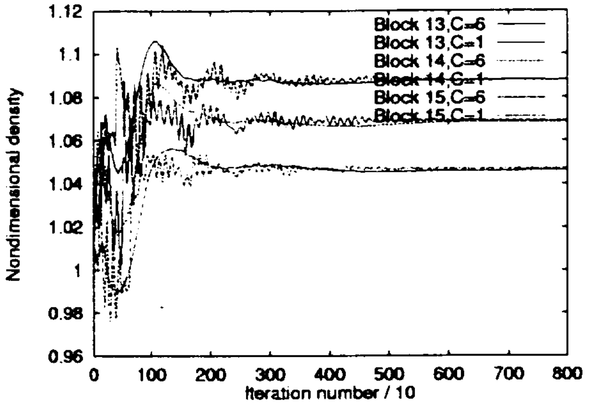


Figure 16: Density variation at midpoint of blocks 13-15

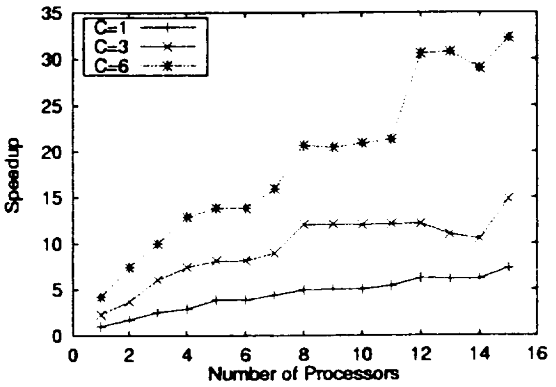


Figure 17: Speedup for test cases 1 and 2

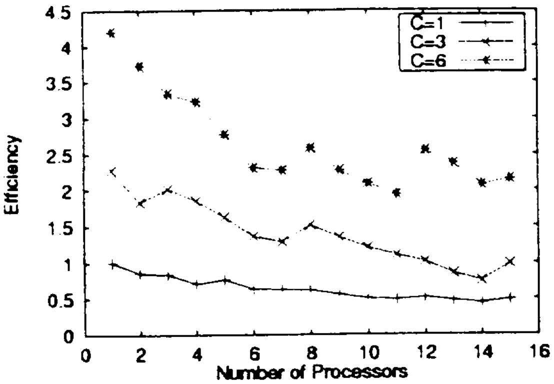


Figure 18: Efficiency for test cases 1 and 2

respectively. An effective speedup of 2.2 and 4.3 is achieved on the single processor case with Courant numbers of 3 and 6, respectively. Hence the efficiency of the filtering procedure is estimated to be about 70%. It is conceivable that with more efficient FFT algorithms or with grid dimensions which are a power of 2, this overhead may be reduced considerably yielding even greater parallel speedup and efficiency.

Conclusions

The proposed filtering techniques are aimed at improving the efficiency of a numerical scheme by selecting the information to be computed. The aim is to calculate the accurate portion of the solution and filter the inaccurate part which in fact increases the computational cost. The design of the filter can be automated based on the calculated initial results. The scheme provides the same benefits of the multigrid technique, yet it is adaptive to the problem and works on a single grid. One can design filters for both implicit and explicit schemes without modifying the original algorithm.

Acknowledgments

This research was funded by the NASA Lewis Research Center under NAG3-1577. The authors appreciate the support provided by the Computational Technologies Branch of NASA LeRC.

References

1. Akay, H.U., Blech, R.A., Ecer, A., Ercoskun, D., Kemle, B., Quealy, A., and Williams, A., "A Database Management System for Parallel Processing of CFD Algorithms," *Parallel CFD '92*, Edited by Pelz, A.B., et al., Elsevier, Amsterdam, 1993, pp. 9-23.

2. Chien, Y.P., Ecer, A., Akay, H.U., Carpenter, F., and Blech, R.A., "Dynamic Load Balancing on a Network of Workstations for Solving Computational Fluid Dynamics Problems," *Computer Methods in Applied Mechanics and Engineering*, vol. 119, 1994, pp. 17-33.

3. Gopalaswamy, N., Akay, H.U., Ecer, A., and Chien, Y.P., "Parallelization and Dynamic Load Balancing of NPARC Codes," AIAA Paper No. 96-3302, July 1-3, Lake Buena Vista, FL, 1996.

4. Gopalaswamy N., Ecer A., Akay H. U., and Chien Y.P., "Efficient Parallel Communication Schemes for Explicit Solvers of NPARC Codes," AIAA Paper No. 97-0027, Reno, January 1997.

5. Cooper, G.K., and Sirbaugh, J.R., "The PARC Code: Theory and Usage," Arnold Engineering Development Center, TR-89-15, 1989.

6. Geist, G.A., Beguelin, A.L., Dongarra, J.J., Jiang, W., Manchek, R., and Sunderam, V., "PVM 3 User's Guide and Reference Manual," *Oak Ridge National Laboratory ORNL/TM-12187*, 1993.

7. Roache, P.J., "Computational Fluid Dynamics," Hermosa Publishers, Albuquerque, New Mexico, 1976.

8. Singleton, R.C., "Multivariate Complex Fast Fourier Transform," Fortran66 Source from http://www.netlib.org/go/fft.f, 1968.

9. Chung, J., "Numerical Solution of a Mixed Compression Supersonic Inlet Flow," AIAA Paper No. 94-0583, Reno, January 1994.