UCLA
COMPUTATIONAL AND APPLIED MATHEMATICS

Performance of a Parallel Code for the Euler Equations on Hypercube Computers

Eric Barszcz
Tony F. Chan
Dennis C. Jespersen
Raymond S. Tuminaro

September 1989

CAM Report 89-26

Department of Mathematics
University of California, Los Angeles
Los Angeles, CA. 90024-1555

Performance of a Parallel Code for the Euler Equations on Hypercube Computers*

Eric Barszcz

NASA/Ames Research Center

Tony F. Chan†

Math Dept., U.C.L.A.

Dennis C. Jespersen

NASA/Ames Research Center

Raymond S. Tuminaro

Comp. Sci. Dept., Stanford U. and

RIACS/Ames Research Center

Abstract. We evaluate the performance of hypercubes on a computational fluid dynamics problem and consider the parallel environment issues that must be addressed, such as algorithm changes, implementation choices, programming effort and programming environment. Our evaluation focuses on a widely used fluid dynamics code, FLO52, written by Antony Jameson, which solves the two-dimensional steady Euler equations describing flow around an airfoil. We describe our code development experience, including interacting with the operating system, utilizing the message passing communication system, and code modifications necessary to increase parallel efficiency. Results from two hypercube parallel computers (a 16-node iPSC/2, and a 512-node NCUBE/ten) will be discussed and compared. In addition, we develop a mathematical model of the execution time as a function of several machine and algorithm parameters. This model accurately predicts the actual run times obtained and is used to explore the performance of the code in interesting but not yet physically realizable regions of the parameter space. Based on this model, predictions about future hypercubes are made.

Key Words. fluid dynamics, parallel computation, hypercube, Euler equations.

1. Introduction.

Motivated by the increasing computational demands of many scientific applications, an enormous research effort into parallel algorithms, hardware, and software has already been undertaken. Parallel machines, such as hypercubes,

* Funds for the support of this study have been allocated by NASA/Ames Research Center, Moffett Field, California under Interchange No. NCA2-233.

† This author was also supported by Dept. of Energy under contract DE-FG03-87ER25037 and by the ARO under contract DAAL03-88-K-0085. Part of this work was performed while the author was visiting RIACS/Ames Research Center.

have been commercially available for approximately five years with some second generation machines currently on the market. Now is an appropriate time to ask whether hypercubes can supply the additional computing power necessary to solve these computationally intensive problems.

To evaluate the potential of hypercubes on computational fluid dynamics (CFD) problems, we implemented an existing fluid dynamics code on three hypercube machines (a 32-node iPSC/1, a 16-node iPSC/2, and a 512-node NCUBE/ten). Other evaluations of parallel fluid dynamics codes can be found in [18], [5], [3], and [1]. We chose Antony Jameson's widely used FLO52 [10], which solves the two-dimensional steady Euler equations describing inviscid flow around an airfoil. FLO52 is representative of a large class of algorithms and codes for CFD problems. Its main computational kernel is a multi-stage Runge-Kutta integrator accelerated by a multigrid procedure. Using FLO52, we evaluate the performance of hypercubes, by comparing run times with results from a single processor of a Cray X-MP, and consider the parallel environment issues that must be addressed, such as algorithm changes, implementation choices, programming effort and programming environment.

Section 2 of the paper contains a description of the fluid dynamics in the FLO52 code with an introduction to multigrid acceleration. Sections 3 and 4 contain an abstract machine model for the hypercube machines and the parallelization of FLO52 based on this model. Section 5 has machine descriptions for the three hypercubes. Sections 6, 7, and 8 discuss coding experiences, code portability, and recommendations for improving the programming environment. Section 9 talks about the validation of the parallel version of the code. Sections 10 and 11 discusses optimizations and code performance. Section 12 contains a timing model and predictions based on it.

2. The Euler Code FLO52. An already existing flow code, FLO52, written by Antony Jameson [10], was chosen to study and port to the hypercube. This code was chosen for several reasons. It is well-known and various versions of it and its descendants are widely used in research and industrial applications throughout the world. It produces good results for problems in its domain of application (steady

inviscid flow around a two-dimensional body), giving solutions in which shocks are captured with no oscillations. It converges rapidly to steady state and executes rapidly on conventional supercomputer (uniprocessor) architectures. Finally, the code uses a multigrid algorithm to accelerate convergence, which makes it an interesting challenge for multiprocessing, as the small number of grid points on the coarsest meshes may make multiprocessing inefficient.

To study the steady Euler equations of inviscid flow, we begin with the unsteady time-dependent equations. The time-dependent two-dimensional Euler equations in conservation form may be written in integral form as

$$\frac{d}{dt}\iint w + \oint \mathbf{n}\cdot\mathbf{F} = 0. \tag{1}$$

This integral relation expresses conservation of mass, momentum, and energy. It is to hold for any region in the flow domain; $\mathbf{n}$ is the outward pointing normal on the boundary of the region. The variable w is the vector of unknowns

$$w = (\rho, \rho u, \rho v, \rho E)^T, \tag{2}$$

where ρ is density, u and v are velocity components directed along the x and y-axes, respectively, and E is total energy per unit mass. The function $\mathbf{F}$ is given by $\mathbf{F}(w) = (E(w), F(w))$ where

$$\begin{aligned} E(w) &= (\rho u, \rho u^2 + p, \rho uv, \rho uH)^T \\ F(w) &= (\rho v, \rho uv, \rho v^2 + p, \rho vH)^T. \end{aligned}$$

Here p is pressure and H is enthalpy; these are defined by

$$\begin{aligned} p &= (\gamma - 1)\rho[E - (u^2 + v^2)/2] \\ H &= E + p/\rho, \end{aligned}$$

where γ is the ratio of specific heats, for air taken to be the constant 1.4.

To produce a numerical method based on (1), the flow domain is divided into quadrilaterals, see Fig. (1).

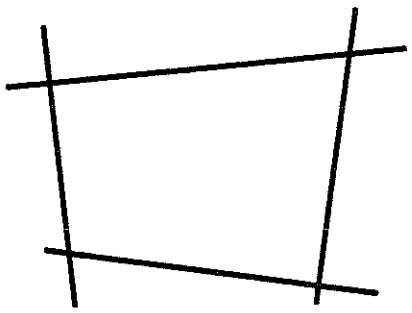

FIG. 1. *Typical Quadrilateral Cell*

On each quadrilateral of the domain, the double integral in (1) is approximated by the centroid rule and the line integral is approximated by the midpoint rule. This gives the approximation

$$(3)\quad \frac{d}{dt}(A_{ij}w_{ij}) + (\mathbf{n}\cdot\mathbf{F}|_{i+1/2,j} - \mathbf{n}\cdot\mathbf{F}|_{i-1/2,j}) + (\mathbf{n}\cdot\mathbf{F}|_{i,j+1/2} - \mathbf{n}\cdot\mathbf{F}|_{i,j-1/2}) = 0,$$

where A_{ij} is the area of cell (i,j) and $\mathbf{n}$ has been redefined to include a factor proportional to the length of the side. On a rectangular Cartesian mesh this reduces to central differencing. A dissipation term is added to (3) which is a blend of second and fourth-order differences. The rationale is that the fourth-order term prevents the oscillations that would occur in smooth regions of the flow (due to central differencing), while the second-order term, which is only non-negligible in steep gradient regions, serves to prevent overshoots and undershoots near shocks. The equation with dissipation terms has the form

$$\begin{aligned}\frac{d}{dt}(A_{ij}w_{ij}) &+ (\mathbf{n}\cdot\mathbf{F}|_{i+1/2,j} - \mathbf{n}\cdot\mathbf{F}|_{i-1/2,j}) \\ &+ (\mathbf{n}\cdot\mathbf{F}|_{i,j+1/2} - \mathbf{n}\cdot\mathbf{F}|_{i,j-1/2}) + (D_x + D_y)w_{ij} = 0,\end{aligned}$$

where

$$(4)\qquad D_x w_{ij} = -d_{i+1/2,j}w + d_{i-1/2,j}w.$$

The operator $d_{i+1/2,j}$ is defined as

$$d_{i+1/2,j}w = \frac{1}{2}\left(\frac{A_{ij}}{\Delta t_{ij}} + \frac{A_{i+1,j}}{\Delta t_{i+1,j}}\right)(\epsilon^{(2)}_{i+1/2,j}\Delta_x w_{ij} - \epsilon^{(4)}_{i+1/2,j}\Delta_x^3 w_{i-1,j}). \tag{5}$$

Here Δ_x is first-order differencing, $\Delta_x w_{ij} = w_{i+1,j} - w_{ij}$, and $\epsilon^{(2)}$ and $\epsilon^{(4)}$ are defined as follows. Let

$$\nu_{ij} = \frac{|p_{i+1,j} - 2p_{ij} + p_{i-1,j}|}{p_{i+1,j} + 2p_{ij} + p_{i-1,j}}, \tag{6}$$

let

$$\tilde{\nu}_{i+1/2,j} = \max(\nu_{ij}, \nu_{i+1,j}), \tag{7}$$

define

$$\tilde{\tilde{\nu}}_{i+1/2,j} = \max(\nu_{i-1,j}, \nu_{ij}, \nu_{i+1,j}, \nu_{i+2,j}), \tag{8}$$

and define

$$\begin{aligned} \epsilon^{(2)}_{i+1/2,j} &= \min(1/2, \kappa^{(2)}\tilde{\nu}_{i+1/2,j}) \\ \epsilon^{(4)}_{i+1/2,j} &= \max(0, \kappa^{(4)} - \alpha\tilde{\tilde{\nu}}_{i+1/2,j}). \end{aligned}$$

Here $\kappa^{(2)}$, $\kappa^{(4)}$, and α are parameters to be chosen. Typical values are $\kappa^{(2)} = 1$, $\kappa^{(4)} = 1/32$, and $\alpha = 2$. The dissipation term in the y direction is handled in a similar fashion. To summarize the spatial discretization, the convective transport terms are discretized using a nearest-neighbor operator, while the dissipation terms use information from two cells away in the north, south, east, and west directions.

The iterative method for steady-state problems is based on a time-marching method for the time-dependent equations (1). After the spatial discretization sketched in the previous paragraph, the equations form a system of ordinary differential equations

$$\frac{dw}{dt} + Hw + Pw = 0, \tag{9}$$

where H denotes the finite-difference operator corresponding to the differencing of the spatial derivatives in (1) and P denotes the finite-difference operator corresponding to

the artificial dissipation terms. A general multistage Runge-Kutta-like method for (9) can be written in the form

$$
\begin{aligned}
w^{(0)} &= w_n \\
w^{(k)} &= w^{(0)} - \Delta t \sum_{j=0}^{k-1} (\alpha_{kj} H w^{(j)} + \beta_{kj} P w^{(j)}), \quad 1 \le k \le m \\
w_{n+1} &= w^{(m)}.
\end{aligned} \tag{10}
$$

This starts from a numerical solution at step n and produces a solution at step $n+1$. The parameters are m, the number of stages; Δt, the time step; and $\{\alpha_{kj}\}$ and $\{\beta_{kj}\}$, coefficients chosen so that

$$
\sum_{j=0}^{k-1} \alpha_{kj} = \sum_{j=0}^{k-1} \beta_{kj}, \quad 1 \le k \le m. \tag{11}
$$

(This last restriction is sufficient to ensure that $w_{n+1} = w_n$ if and only if $(H+P)w_n = 0$, and hence any numerical steady state is independent of Δt.) If $m = 4$ and the only nonzero parameters for α_{kj} and β_{kj} are given by $\alpha_{10} = \beta_{10} = 1/4$, $\alpha_{21} = \beta_{21} = 1/3$, $\alpha_{32} = \beta_{32} = 1/2$, $\alpha_{43} = \beta_{43} = 1$, then the method would reduce to the classical four-stage Runge-Kutta scheme if the operators H and P were linear. Thus the method (10) is sometimes called a Runge-Kutta or modified Runge-Kutta method.

Another set of parameters is $m = 5$ and $\alpha_{10} = \beta_{10} = 1/4$, $\alpha_{21} = \beta_{21} = 1/6$, $\alpha_{32} = \beta_{31} = 3/8$, $\alpha_{43} = \beta_{41} = 1/2$, $\alpha_{54} = \beta_{51} = 1$. This five-stage method uses only two evaluations of the dissipative operator; it is desirable to decrease the number of evaluations of the dissipative operator because of the computational expense. (On a standard architecture one evaluation of the dissipation operator is about 25% more expensive than one evaluation of the hyperbolic operator.) This method also has a larger stability region than that of the method with the same coefficients that evaluates the dissipation term at every stage.

A convergence acceleration technique sometimes called residual smoothing or implicit residual averaging is used. This involves a slight modification of (10); instead of (10) one performs the iteration

$$
w^{(0)} = w_n
$$

$$(12) \qquad S(w^{(k)} - w^{(0)}) = -\Delta t \sum_{j=0}^{k-1} (\alpha_{kj} H w^{(j)} + \beta_{kj} P w^{(j)}), \quad 1 \le k \le m$$
$$w_{n+1} = w^{(m)}.$$

where S is the operator $S = (I - \epsilon_x \delta_{xx})(I - \epsilon_y \delta_{yy})$ applied to each component of w separately (here δ_{xx} is defined by $\delta_{xx} w_{rs} = w_{r+1,s} - 2w_{rs} + w_{r-1,s}$) and $\epsilon_x \ge 0$, $\epsilon_y \ge 0$ are constants. One can show that for $\epsilon_x > 0$, $\epsilon_y > 0$, the stability region of the iteration (12) is larger than that of (10), while the steady state remains unchanged. The solution of the linear system $S\Delta w = r$ in (12) vectorizes well on conventional supercomputer architectures because the matrices involved are simple tridiagonal Toeplitz matrices and the solution of the systems in one direction can be vectorized in the orthogonal direction. The implications of implicit residual averaging are more serious for parallel processing and a hypercube architecture, however, and will be discussed below.

Another convergence acceleration technique is "local time-stepping." This means that in (10) or (12) one uses a Δt which depends on the local cell; the value of Δt is usually chosen so that the (local) CFL number is a given constant. This technique is not impacted by the choice of architecture (conventional or multiprocessor).

Yet another convergence acceleration technique is enthalpy damping [11]. This depends on the fact that for steady inviscid flow the enthalpy is constant along streamlines. Hence for flow from a reservoir with constant fluid properties, the enthalpy is everywhere equal to its freestream value H_∞. A sketch of the derivation of enthalpy damping is as follows (see [13] for more details). One can show that for irrotational homentropic flow one has the unsteady potential equation

$$(13) \qquad \phi_{tt} + 2u\phi_{xt} + 2v\phi_{yt} = (c^2 - u^2)\phi_{xx} - 2uv\phi_{xy} + (c^2 - v^2)\phi_{yy}$$

which can be transformed by a change of variables $\xi = x - ut$, $\eta = y - vt$ into the wave equation

$$(14) \qquad \phi_{tt} = c^2(\phi_{xx} + \phi_{yy}).$$

Solutions of this equation are undamped in time. A related equation [6] is the *telegraph equation*

$$\phi_{tt} + \alpha\phi_t = c^2(\phi_{xx} + \phi_{yy}). \tag{15}$$

For $\alpha > 0$ solutions of this equation decay as $t \to \infty$. The same procedure that derives the wave equation from the inviscid flow equations would derive the telegraph equation from the set of equations consisting of the inviscid flow equations with each equations augmented by the addition of a term proportional to $H - H_\infty$. Thus enthalpy damping consists of adding a term proportional to $H - H_\infty$ to each equation of (1); for example, in differential form the equation for conservation of mass becomes

$$\rho_t + (\rho u)_x + (\rho v)_y + \alpha\rho(H - H_\infty) = 0. \tag{16}$$

The final acceleration technique to be discussed is the multigrid technique. It is really multigrid which effects a substantial speedup in convergence for the flow code over the non-multigrid version of the code. The multigrid idea is based on using some relaxation scheme to smooth the error on a fine grid, then transferring the problem to a coarser grid and solving the transferred problem. A good approximate solution to the transferred problem can be obtained cheaply, since the coarse grid has only 1/4 the number of grid points (assuming two space dimensions) of the fine grid. The solution to the coarse grid problem is then used to correct the solution on the fine grid. The solution on the coarse grid is usually obtained by the same multigrid algorithm, so the size of the grids on which the problem is considered rapidly decreases. It is beyond the scope of this paper to consider all the motivation behind the origination and development of multigrid algorithms; we refer to the literature ([2], [17]). We will present a description of the multigrid algorithm used in the code, however.

To describe the multigrid algorithm it will be useful to consider a general time-dependent partial differential equation

$$\frac{\partial u}{\partial t} = Lu + f, \tag{17}$$

where L is a spatial operator (possibly nonlinear) and f is a forcing term. We assume we are given some sequence of grids $\mathcal{G}_0 \supset \mathcal{G}_1 \supset \mathcal{G}_2 \ldots$ On a structured quadrilateral mesh of the type we consider here, the grid $\mathcal{G}_{k+1}$ is constructed from $\mathcal{G}_k$ by omitting every other grid line in each direction. Functions discretized on grid $\mathcal{G}_k$ will be given a subscript k. The discretized version of the operator L on grid k will be written L_k. The partial differential equation (17) is discretized on the fine mesh $\mathcal{G}_0$ and some relaxation scheme is devised for the numerical solution of the discrete equations. We write one iteration of the relaxation scheme on grid k in the form

$$u_k \leftarrow M_k u_k + N_k f_k, \tag{18}$$

where M_k and N_k are (possibly nonlinear) discrete operators. It is assumed there are given operators $I_k^{k+1} : \mathcal{G}_k \rightarrow \mathcal{G}_{k+1}$ (fine-to-coarse transfer of solution), $\hat{I}_k^{k+1} : \mathcal{G}_k \rightarrow \mathcal{G}_{k+1}$ (fine-to-coarse transfer of residual), and $I_{k+1}^k : \mathcal{G}_{k+1} \rightarrow \mathcal{G}_k$ (coarse-to-fine transfer). After the relaxation step (18) on the fine mesh, one defines the forcing function on the coarse mesh via

$$f_{k+1} = \hat{I}_k^{k+1}(L_k u_k + f_k) - L_{k+1} I_k^{k+1} u_k. \tag{19}$$

(Note that even if $f_k = 0$, i.e., no forcing function on the fine mesh, we will in general have $f_{k+1} \neq 0$.) The initial guess on the coarse grid is defined by

$$u_{k+1} = I_k^{k+1} u_k. \tag{20}$$

After the solution on the coarse grid is improved (via one or more relaxation sweeps or via relaxation and transfer to a still coarser grid), the solution on the fine grid is corrected by

$$u_k \leftarrow u_k + I_{k+1}^k (u_{k+1} - I_k^{k+1} u_k). \tag{21}$$

This solution may be further improved by relaxation on the fine mesh. This multigrid algorithm is known as the "full approximation scheme" (FAS) algorithm.

The algorithm used in the code is the "sawtooth" algorithm. For two grids, this is defined by

$$\begin{aligned} u_0 &\leftarrow M_0 u_0 + N_0 f_0 \\ f_1 &= \hat{I}_0^1 (L_0 u_0 + f_0) - L_1 I_0^1 u_0 \\ u_1 &= I_0^1 u_0 \\ u_1 &\leftarrow M_1 u_1 + N_1 f_1 \\ u_0 &\leftarrow u_0 + I_1^0 (u_1 - I_0^1 u_0). \end{aligned}$$

For more than two grids, the sawtooth algorithm is recursively defined by replacing the step

$$u_1 \leftarrow M_1 u_1 + N_1 f_1 \tag{22}$$

with

$$u_1 = \text{result of sawtooth algorithm starting on } \mathcal{G}_1. \tag{23}$$

For a structured quadrilateral mesh the operator I_0^1 is defined by (see Fig. 2)

$$I_0^1(u_0)_{ij} = \frac{1}{A_{1,ij}} \left(A_0^{NE} u_0^{NE} + A_0^{NW} u_0^{NW} + A_0^{SW} u_0^{SW} + A_0^{SE} u_0^{SE} \right), \tag{24}$$

where A_1 denotes area on the coarse mesh and A_0 denotes area on the fine mesh. The operator $\hat{I}_0^1$ is defined as in (24) but without the area factors. Finally, the operator I_1^0 is defined via bilinear interpolation (see Fig. 3).

$$I_1^0(u_1)_{ij} = \frac{9}{16} u_1^{SW} + \frac{3}{16} \left(u_1^{NW} + u_1^{SE} \right) + \frac{1}{16} u_1^{NE}. \tag{25}$$

A good initial guess u_0 on the fine mesh $\mathcal{G}_0$ can be obtained by solving the numerical problem on the coarse mesh $\mathcal{G}_1$ and interpolating the resulting solution to the fine mesh. This is the "full multigrid" strategy, and is the approach taken in the code.

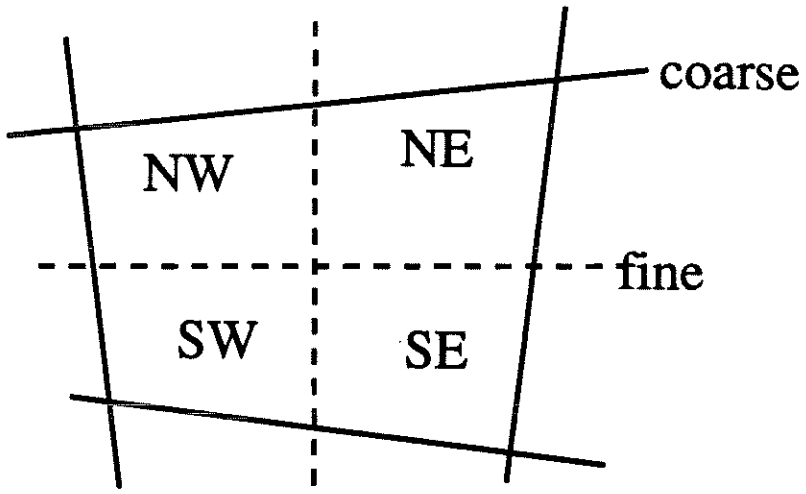

FIG. 2. *Fine-to-Coarse Interpolation*

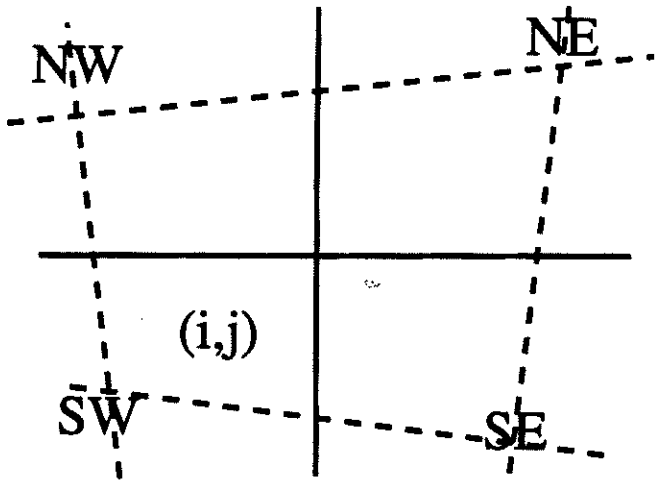

FIG. 3. *Coarse-to-Fine Interpolation*

The boundary conditions used were the following. At the airfoil surface, the normal velocity component is set to zero, and the tangential velocity component density are extrapolated from the cell immediately adjacent to the solid surface. The pressure is extrapolated via a pressure gradient which is computed via the normal momentum equation. The far field boundary conditions are based on locally one-dimensional characteristics and Riemann invariants. At each cell, the Riemann invariants are calculated for the one-dimensional flow problem normal to the outer boundary. If the normal velocity component points outward and the flow is subsonic, then the two outgoing characteristic variables are extrapolated to the exterior, the incoming characteristic variables is taken to have its free-stream value, and the velocity component tangential to the boundary is extrapolated from the interior. If the normal velocity component points inward and the flow is subsonic, then the two incoming characteristic variables are extrapolated from the exterior (at their free-stream values), the outgoing characteristic variable is extrapolated from the interior, and the tangential velocity is taken to have its free-stream value.

In summary, the algorithm under consideration consists of an explicit multistage relaxation method with local time-stepping, implicit residual averaging, and multigrid to accelerate convergence. The latter two techniques are especially significant when considering transporting the code to a parallel processing environment.

3. Abstract Machine Model. We consider a distributed memory hypercube architecture with explicit message passing as our basic abstract machine model. An N-dimensional hypercube computer contains 2^N CPU's with each CPU directly connected to N neighbors [15]. When the processor addresses are expressed in binary, the neighbors of a processor are determined by complementing a single bit of the address, a different bit for each neighbor. The diameter of the hypercube, the maximum number of stages required to send a message from one processor to any other, is N.

For example, a 4-dimensional hypercube contains sixteen (2^4) processors, each processor is connected to four neighbors, and at most four stages are required to pass a message from one processor to any other. Figure 4 shows a 4-dimensional

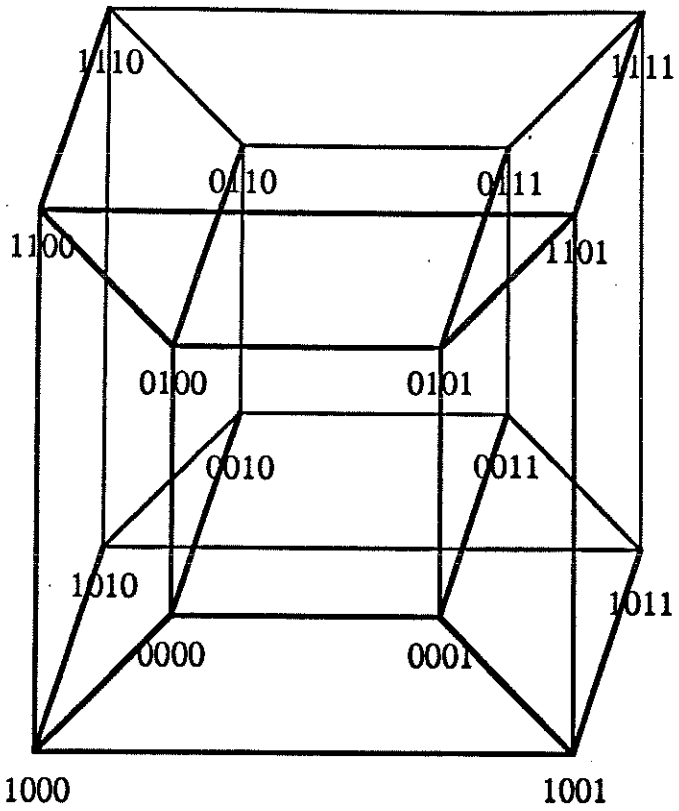

FIG. 4. *Interconnections for a 4-dimensional hypercube.*

hypercube where vertices represent processors and edges represent communication links. The processor addresses are given in binary.

The following is the list of assumptions about the abstract machine:

1. Processors are homogeneous;
2. Processors are connected in a hypercube architecture;
3. Memory is distributed evenly among processors;
4. Memory is locally addressed;
5. Computation is much faster than communication;
6. The machine is a multiple-instruction stream, multiple-data stream (MIMD) architecture;
7. Message buffer packing and unpacking must be done by the programmer;
8. Message destination processors are explicitly addressed;
9. Messages are forwarded automatically by intermediate processors;

10. A separate host processor controls all input/output.

Various aspects of the abstract machine model affect parallelization of the code. Having homogeneous processors implies data does not have to be moved for processing by a unique piece of hardware. With computation being much faster than communication, it is desirable to maximize the computation to communication ratio on each processor. Since communication is approximately proportional to the surface area and computation to the volume, of the local domain, cube and squares typically yield a better computation to communication ratio than strips or slabs. Also, nearest neighbor communication is preferred over multiple hop communication since it can be significantly cheaper.

4. Parallelization of FLO52. Conceptually, converting the sequential FLO52 code to a hypercube version is simple. Figure 5 shows a sample computational domain mapped onto a hypercube. The computational domain is an "O" grid that is logically equivalent to a rectangular grid. The logical domain is broken into equal sized subdomains. Each subdomain has a boundary layer that contains values updated by other processors. The subdomains are assigned one to a processor. Assignments are made using a two-dimensional binary reflected Gray code (see [4]), as shown in figure 5.

The code structure in the main body closely resembles the sequential version, with some reordering to decrease communication overhead. The algorithm is fully explicit except for the implicit residual averaging scheme. Nested loops in explicit sections of the code now operate on local subdomains instead of the whole domain. Typically, the computation/communication pattern for the nested loops is as follows. Each processor updates points in its subdomain (applying a local differencing operator) and then exchanges boundary values with the appropriate neighbor. If a boundary corresponds to a physical boundary, then boundary conditions may be evaluated.

It should be noted that most of the code corresponds to a local 5 or 9 point operator, but the fourth order dissipation operator requires a larger stencil (utilizing information from two cells away). This larger stencil results in additional communi-

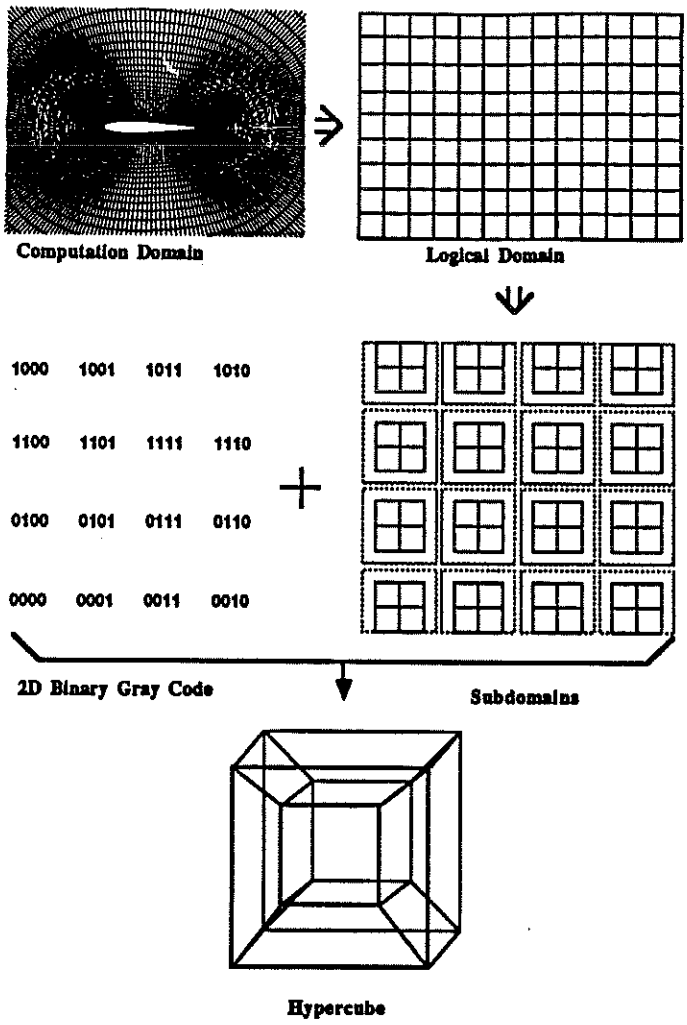

FIG. 5. *Computational domain mapped to hypercube.*

cation overhead since information from points adjacent to the boundary as well as the boundary must be communicated. Note that if the coarsest grid during the multigrid cycle has a minimum of two cells in each dimension per processor, all communication for the explicit portion of the code is nearest neighbor.

The implicit portion of the code involves solving a series of tridiagonal matrices. Having been designed for a vector uniprocessor, the algorithm used within FLO52 does not parallelize well. Many experiments are made with this element of the code. The experiments correspond to algorithm implementation changes as well as algorithm changes. We defer a discussion of the tridiagonal solvers to the performance section of the paper where the results of these experiments will be presented.

Other aspects of the code include an initialization phase and an output phase. The initialization phase entails mapping the physical domain onto the processors and sending relevant parameters (input values, neighbor information, etc.). Our mapping routine allows the user to specify the number of processors in each dimension. Thus, squares, rectangles, and strips can be tested.

The output phase of the code requires that the nodes send information to the host and then the host either prints to standard output or writes to a file. The host code contains a main loop which processes all incoming messages according to their type. Blocks of data transferred from the nodes to the host are preceded and followed by message types to indicate the beginning and end of a block. To maintain consistency, global synchronization is necessary because the Intel iPSC/1 host does not distinguish between message types and does not guarantee the order in which messages will be processed [16]. Global synchronization is accomplished using the Intel utility "gop". This utility uses a tree structure to collect information from a subcube. Its main purpose is to provide global operations such as max, min, sum, multiply, though it can be used as part of a global synchronizer as well. In addition, we tailored it to our specific needs by adding and extending the number of operations. For example, we use the extended "gop" utility to print the maximum residual and its location.

Overall, while parallelization of the main body of the code was non-trivial, writing the parallel code was not too difficult. Unfortunately, input and output operations between host and nodes do require substantial time to code and debug when compared to their serial counterparts.

5. Machine Descriptions. The code runs on three commercial hypercube machines. They are a 32-node Intel iPSC/1 [8], a 16-node Intel iPSC/2 [9], and a 512-node NCUBE/ten [14]. The original target machine was the Intel iPSC/1 located at RIACS, NASA/Ames Research Center. After an initial working code was developed, it was migrated to the Intel iPSC/2 located at Stanford University. Then using communication libraries developed by Michael Heath [7], the code was converted to run on the NCUBE/ten located at CalTech.

Each node of the Intel iPSC/1 contains an Intel 80286 chip as CPU, an Intel 80287 numeric co-processor, 512 Kbytes of dual-ported memory, and eight communication channels (seven for hypercube communication and one for global communication with the host). The host also uses the Intel 80286/80287 chip set, runs the Xenix operating system, and is connected to all of the processors via a global Ethernet channel. Communication in the iPSC/1 is packet-switched and cannot be performed on all channels in parallel. Also, every message arrival interrupts the CPU, even if the message is destined for another processor. Figure 6 shows the cost for node-to-node communication on the iPSC/1, iPSC/2, and NCUBE/ten in milliseconds for messages of various lengths. Figure 7 shows the cost for host-to-node communications on the iPSC/1 and iPSC/2 in milliseconds.

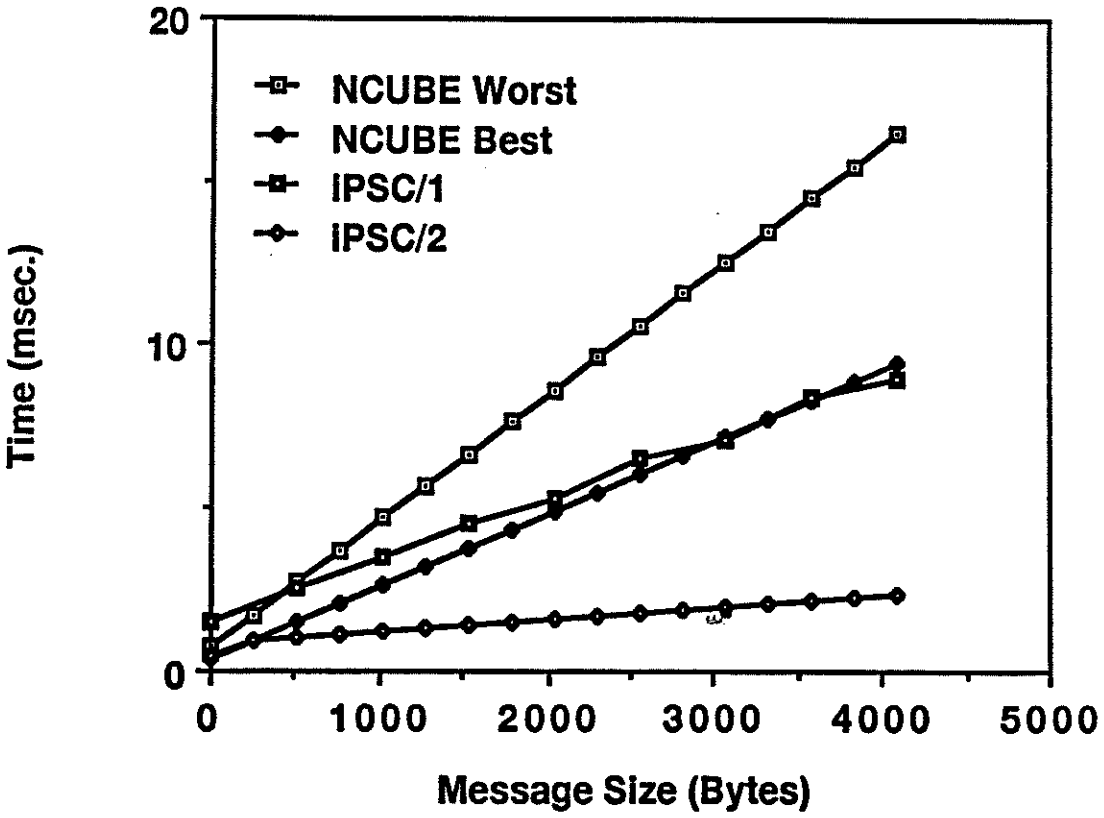

FIG. 6. *Node-to-Node Communication Cost.*

Each node of the Intel iPSC/2 contains an Intel 80386 CPU, an Intel 80387 numeric co-processor (rated approximately 500 KFLOPS), 4 megabytes of memory, a

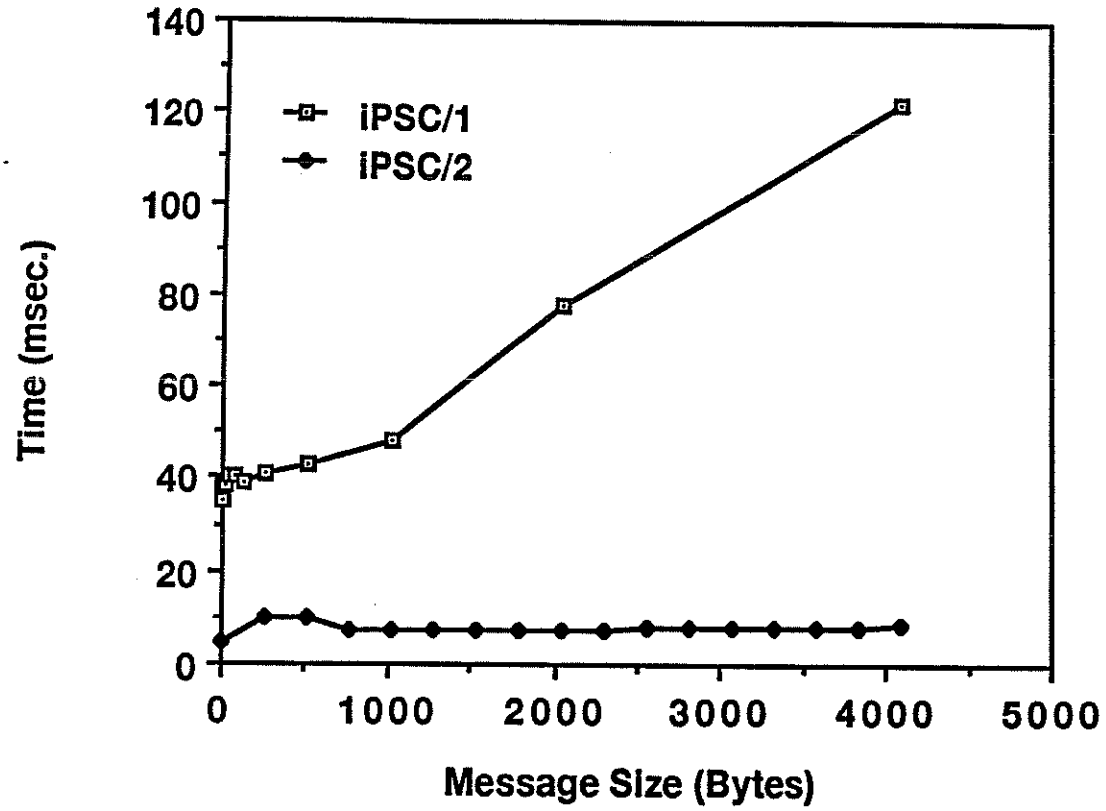

FIG. 7. *Host-to-Node Communication Cost.*

vector co-processor board rated at 6.7 MFLOPS, and eight communication channels (seven for hypercube communication and one for external I/O). The host also uses the Intel 80386/80387 chip set, runs a Unix V.3 based operating system, and connects to the nodes via the external I/O port on node zero. Communication on the iPSC/2 is circuit-switched using a proprietary chip and can be performed on all channels simultaneously. All routing and intermediate stages are handled in hardware and do not interrupt the CPU. Communication is 3 to 10 times faster than on the iPSC/1 depending on message size. The peak communication rate between two neighboring processors is 2.8 Mbytes per second.

Each node on the NCUBE contains a proprietary chip as its CPU (rated approximately 480 KFLOPS), 512 Kbytes of memory, and eleven full duplex communication channels (ten for hypercube communication and one for external I/O). The host uses the Intel 80286/80287 chip set, runs the Axis operating system (a Unix-style operating

system), and is contained on a board that fits into one of the I/O slots. Communication with a node is done by treating it as a device and performing I/O to that device. Communication is packet switched and can be performed on all channels simultaneously. There are 22 bit-serial I/O lines paired into 11 bidirectional channels of which ten channels are reserved for hypercube connections and one for external I/O. Each bit-serial line has a data transfer rate of about 1 Mbyte per second.

6. Coding Experiences. The initial target for the conversion of FLO52 was a version programmed in C to run on an Intel iPSC/1. C was chosen as the target language because it forced a complete rewrite of the Fortran code creating an in-depth understanding of the algorithm, and the translation process encouraged the new code to be written with the architecture in mind. In addition, the C compiler was more stable than the Fortran compiler at that time.

The task of developing FLO52 on the Intel iPSC/1, though conceptually easy, proved difficult. Many of the frustrations are similar to those associated with code development on serial machines. In particular, the original FLO52 is a sophisticated Cray-optimized code with few comments and many subtle features that are critical to convergence. Verifying the numerical properties of this code is time-consuming (even on a serial machine). By way of example, one bug in an earlier version of our code was not detectable until 200 iterations in a 300-iteration run. However, in addition to the usual debugging difficulties, there is an additional level of complexity associated with a parallel code: printing results, synchronization problems, message passing problems, and determining which processors are executing incorrect code. In general, a good strategy is first to produce a working algorithm on one processor before attempting to execute the code on multiple processors. This tends to separate problems associated with numerics and those associated with communication. Another strategy is to partition the domain so only one dimension is split across multiple processors. This allows the separation of communication problems by dimension.

Unfortunately, even with good programming techniques, code development on the iPSC and the NCUBE is a time-consuming exercise. We list a few aspects of the

iPSC environment that we found particularly frustrating.

1. A significant amount of time is wasted waiting for cube initialization on the iPSC/1 (2 minutes per initialization). Unfortunately, the cube must be frequently re-initialized when a program terminates abnormally (i.e., while debugging a program).
2. The C compiler is slow and not always successful. In particular, it takes approximately 20 minutes to compile the entire code on the Intel iPSC/1 as compared to about 5 minutes on iPSC/2 and 2 minutes on a Sun 3/50. In addition, the compilation frequently fails due to segmentation faults.
3. The node memory is barely sufficient for realistic applications. Specifically, the iPSC/1 nodes contain only 512 Kbytes per processor. Approximately 200K of this memory is taken by the operating system. For our application we could only fit a 32 by 32 grid in each processor for a simple version of our code. This is partly due to the many variables that must be stored per grid point.
4. The iPSC environment has almost no debugging tools. Besides the lack of a debugger, there is no easy mechanism to directly print intermediate results from the nodes. To print from the nodes, the developer has two options. The first involves sending a message from the node to the host so that the host will print the information. This procedure requires modification of both the host and the node program and is itself prone to error. Many times the user finds himself debugging the message passing sequence so that he can see intermediate results. The second alternative is to use the "syslog" feature. This allows a node to send a string that eventually appears in a logfile. This requires the user to first store his results in a string and then invoke the primitive. In general, it is difficult to determine when the actual syslog was invoked as opposed to when it appears in the logfile.

So as not to discourage the prospective iPSC programmer, we should note that most of the above difficulties appear far more frequently on the iPSC/1 than on the

iPSC/2. In addition, the user does have the option of doing his code development and debugging on a simulator (which can be run on another machine). This helps overcome many of the limitations mentioned above. For example, it is possible to print out messages directly from the simulated nodes. Unfortunately, the use of the simulator is limited to small cases, due to long execution times for large simulations, and many problems do not arise until one runs on the actual hardware.

7. Conversion from iPSC to NCUBE. In this section, we remark on the task of converting an iPSC/1 code to an NCUBE machine. The two machines are almost identical in terms of message passing primitives (with slightly different syntax). Machine-independent routines (communication and cube initialization) for both the iPSC and NCUBE have been written by Michael Heath of Oak Ridge National Laboratory. We rewrote our iPSC code to take advantage of these routines. Then, using the NCUBE library, ported the code to the NCUBE. Unfortunately, there were compatability problems. We list a few of the changes necessary to port the code:

1. Complicated timing macros would not compile on the NCUBE. To avoid changing the code, macro preprocessing was done on another machine and the output sent to the NCUBE for compilation.
2. Integers on the nodes of the NCUBE have a different size than the host. This problem can be overcome by changing all the integers on the host to long integer.
3. Certain input formats cause the host program to crash.
4. The order of some statements (during initialization) does not matter on the iPSC, but is important for the NCUBE. This machine dependency cannot be easily hidden by library routines.
5. At most four files can be open on the NCUBE at one time, and host to node communication uses one of these files.

In general, we found that it is not possible to simply port an Intel iPSC code to an NCUBE, even when using a set of machine-independent routines. However with some effort (a week or two), it was possible to get the code working.

8. Recommendations. Most of the difficulties mentioned in the previous two sections are not inherent to the hypercube structure and can be corrected with a better programming environment. We suggest a few additions to the programming environment that would greatly facilitate the programming effort.

1. Basic debugging tools. Supporting the C library function "printf" on the nodes would ease the burden of printing intermediate results. A message tracing utility would simplify finding communication errors. By message tracing, we mean some facility for determining which nodes sent particular messages, which nodes received particular messages, and which nodes are waiting to receive messages. In addition, profiling utilities to detect execution bottlenecks should be provided.
2. Elimination of the host program. The host program is used for input, output, and setting up the computation. It would simplify matters if everything could be done from one program.
3. Primitives to support common parallel processing tasks. These include automatic processor mapping routines. That is, the user specifies an array to distribute over the hypercube. The routine determines the subarrays and corresponding processors available that minimize communication. Other primitives would support nearest-neighbor communications to maintain the array. At a higher level, routines would automatically perform the necessary communication and computation for the application of a local differencing operator to all points in a distributed array.
4. Numerical libraries. Basic linear algebra routines (such as those in LINPACK and EISPACK) should also be supported. For FLO52, a routine to perform the solution of a tridiagonal matrix could have been utilized.

9. Validation of Numerical Results. Our discussion of the numerical results is somewhat brief as they are similar to those obtained using Jameson's FLO52 code (see [10]). All implementations of the parallel code discussed in this paper converge to the same solution. Figure 8 shows the density field for a transonic flow problem -

angle of attack equal to 1.25 degrees and a Mach number of 0.8 on a 256 × 64 grid after 200 multigrid iterations. The corresponding pressure coefficient plots are shown

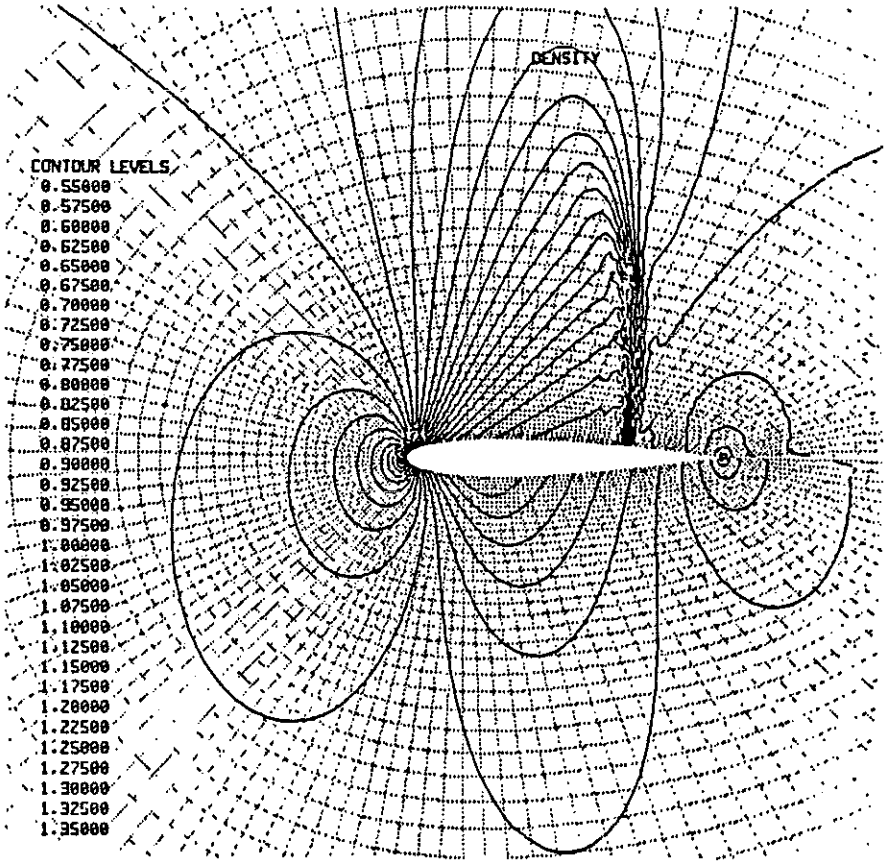

FIG. 8. *Density Field Obtained with Parallel FLO52 on transonic flow problem.*

in figure 9. Both plots are typical solutions to this problem and are identical to those produced by Jameson's FLO52. We compare the convergence rates of our initial code with that of Jameson's code in figure 10.

It is clear from this figure that the convergence behavior of the two codes is similar. Differences between the two codes are due to some minor simplifications in the grid transfer operations.

10. Optimization of Parallel FLO52. To locate bottlenecks and inefficiencies, timing information for each routine is collected from all processors and averaged. This information is then grouped into ten categories that represent the main algorithmic modules of the code. Their performance is analyzed and several modifications to

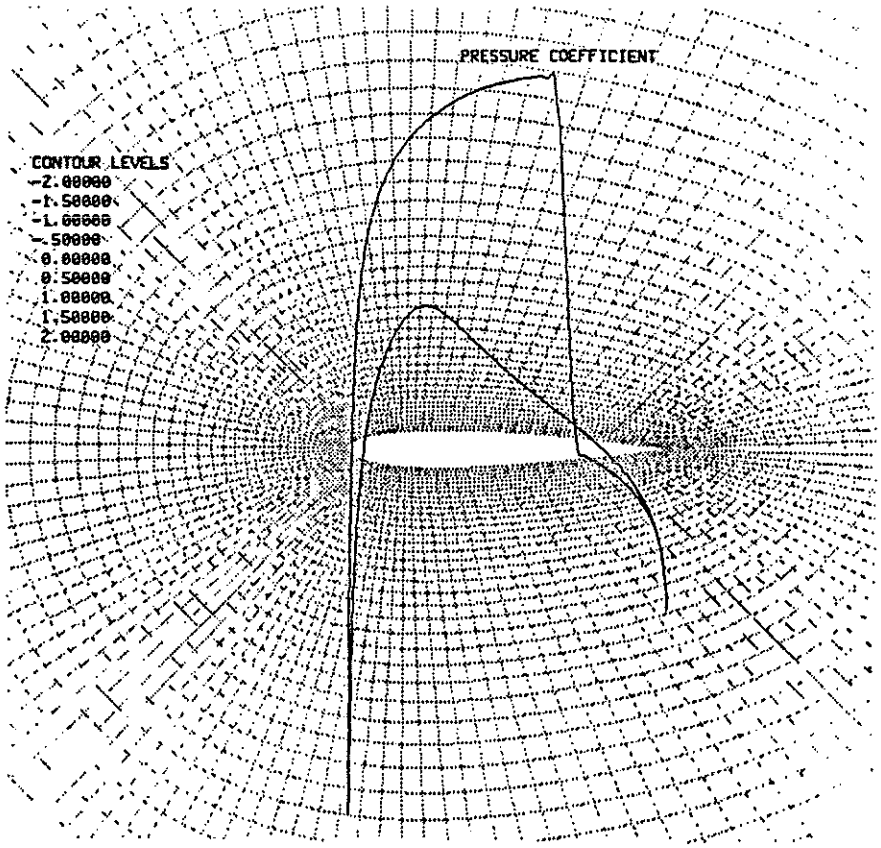

FIG. 9. *Pressure Coefficient Plot Obtained with Parallel FLO52 on transonic flow problem.*

improve parallel efficiency are described. We first describe our algorithmic subdivisions for the timing analysis.

Modules of the code are grouped into the ten categories described below.

1. Flux Calculations: The Hw term in (12).
2. Dissipation: The Pw term in (12).
3. Local Time Step: Compute local time step.
4. Residual Averaging: Solve the tri-diagonal systems of equations in (12).
5. Boundary Conditions
6. Enthalpy Damping
7. Time Advance: Combine the results of the above categories to form (12).
8. I/O
9. , 10. Projection and Interpolation: Multigrid operations for grid transfers.

For the sake of comparison, all timing results in this section are given in seconds

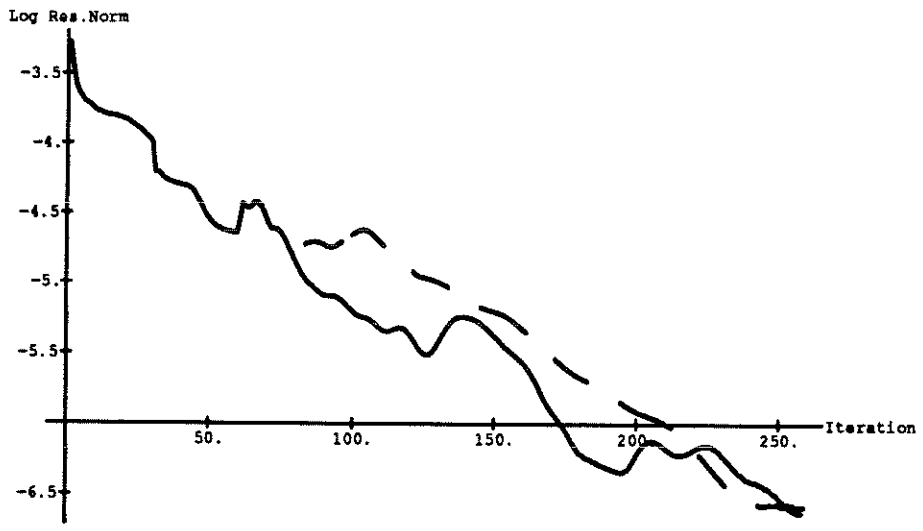

FIG. 10. *Convergence Comparison Between our naive code (dashed) and Jameson's code (solid) on transonic flow problem.*

and were executed on the iPSC/2 using identical input parameters. All runs use a three level sawtooth multigrid method with one Runge-Kutta pre-relaxation sweep; 30 multigrid iterations on each of the two coarser grids and 200 iterations on the finest grid. The algorithm is applied to a transonic flow problem with an angle of attack of 1.25 degrees and a Mach number of 0.8. The finest grid in the multigrid procedure is a 256×64 mesh.

One of the measures used in determining how well an algorithm is matched to a machine is efficiency. Unfortunately, not all problems can run on a single processor because of memory limitations. So, we define efficiency relative to the smallest number of processors that can execute a given problem. This "relative" efficiency is defined as follows:

$$e_b(p) = \frac{bT(b)}{pT(p)} \tag{26}$$

where p is the number of processors whose efficiency we are calculating, b is base number of processors, and $T(n)$ is the run time on n processors. The case where b equals 1 is the standard definition of parallel efficiency. Other values of b arise when the problem is too large to run on a single processor. Unless otherwise stated, b is

categories	1 node	2 nodes	4 nodes	8 nodes	16 nodes	$e_1(16)$
Total	25027	13734	7987	4460	2767	.57
Flux	6241.3	3170.6	1659.9	1005.8	565.5	.69
Dissipation	4779.6	2497.7	1306.7	722.8	476.9	.63
Time Step	1044.2	530.3	267.4	135.5	71.2	.92
Input/Output	147.6	98.1	64.4	48.4	39.2	.24
Residual Avg.	4233.3	2906.5	2257.4	1352.2	963.3	.27
Boundary C's	146.6	275.2	290.2	84.6	27.1	.34
Enthalpy Damp.	420.1	210.1	104.7	53.0	27.2	.96
Advance Soln.	3301.8	1650.4	821.7	413.5	210.1	.98
Project	4006.3	2032.2	1025.4	537.4	323.5	.77
Interpolate	705.9	363.3	188.8	106.5	62.5	.71

TABLE 1

Run times (seconds) of version 1 of Parallel FLO52 on iPSC/2 for a 256 x 64 grid.

equal to 1.

We begin our discussion with version 1. Version 1 closely resembles our first parallel FLO52 code. Tables 1 and 2 contain the run times and percentage of overall CPU time for each catagory. Recall, all results are for the iPSC/2.

Unfortunately, the efficiency is only 57% on the 16 node system. Still, a 57% efficiency corresponds to a speedup of better than 9. It should be noted that efficiency is a function of grid size and number of processors. If more processors are used (and the grid size is kept fixed), then the efficiency will decrease. On the other hand, if the grid size increases and the number of processors is held constant, efficiency will increase (see figure 11). For now we only state that the 57% efficiency is unsatisfactory, and the following versions attempt to improve this.

A detailed inspection of the timings reveals three areas where poor machine utilization significantly degrades the overall run time. These areas correspond to the flux, dissipation, and residual averaging routines as shown in tables 1 and 2.

categories	1 node	16 nodes
Flux	24.9	20.4
Dissipation	19.1	17.2
Time Step	4.1	2.6
Input/Output	.6	1.4
Residual Avg.	16.9	34.8
Boundary C's	.6	1.0
Enthalpy Damp.	1.7	1.0
Advance Soln.	13.2	7.6
Project	16.0	11.7
Interpolate	2.8	2.3

TABLE 2

Individual Percentages of run times for version 1 of Parallel FLO52 on iPSC/2.

To improve efficiency, the flux and dissipation routines were rewritten to reduce the number of messages. For the flux routine this was quite simple. The old routine sent 5 separate messages for each boundary of the subdomain. These messages corresponded to the unknowns: density, x momentum, y momentum, enthalpy, and pressure. The new version simply packs these values together into one message and sees an improvement from 69% to 85% in efficiency. In the case of the dissipation routine, reducing the number of messages required a significant re-ordering of the computations and additional memory. In particular, the older version sends one message for each cell on a border. The new version packs all of the cells on a border into one message buffer requiring a significant increase in buffer space. In addition to buffer packing, overlapping of computation and communication was implemented in the dissipation routine. This is accomplished by sending the shared boundary values to neighbors and then updating the interior of the subdomain. When boundary values arrive from the neighbors, the borders of the subdomain are updated. Between the buffer packing and the overlapping of computation and communication, efficiency for

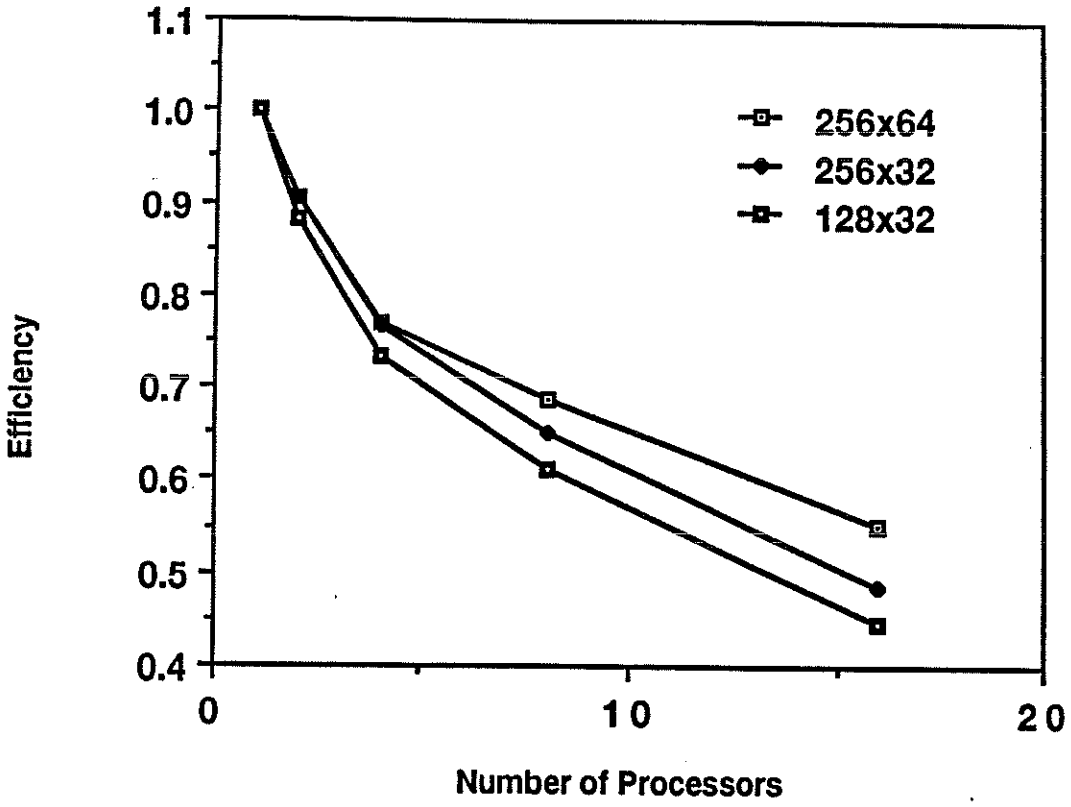

FIG. 11. *Efficiency of version 1 on the iPSC/2.*

the dissipation routines went from 63% to 93% and the overall efficiency from 57% to 61% on the 16 node system. Table 3 contains run times obtained on the iPSC/2 for version 2 with these changes.

While the efficiency is still not spectacular, it does represent a real improvement over the initial 57% efficiency. This implies attention must be paid to system parameters when coding applications for a parallel machine. On a system without a large communication startup cost, one would expect that reducing the number of messages would not make a substantial difference in efficiency.

Version 3 attempts to improve the execution time for the tridiagonal matrix solves. In the tridiagonal solves of the original FLO52 algorithm, information travels across the whole computational domain in both the x and y directions. This has the effect of creating waves of information that pass from left to right and right to left in the x direction, then bottom to top and top to bottom in the y direction. Initially, nodes

categories	1 node	2 nodes	4 nodes	8 nodes	16 nodes	$e_1(16)$
Total	24694	13391	7728	4213	2499	.61
Flux	6240.1	3140.3	1632.2	965.7	528.5	.74
Dissipation	4539.2	2277.2	1143.9	579.5	306.6	.93
Time Step	1043.8	530.2	267.1	135.5	71.7	.91
Input/Output	145.8	91.0	66.5	49.2	40.4	.23
Residual Avg.	4233.3	2906.8	2253.8	1351.3	971.6	.27
Boundary C's	146.8	275.0	289.9	84.0	26.5	.34
Enthalpy Damp.	420.2	209.9	104.8	52.6	26.5	.99
Time Step	3302.2	1651.0	822.1	412.1	207.8	.99
Project	3916.7	1950.4	964.1	484.0	263.1	.93
Interpolate	705.8	358.8	183.5	99.4	56.1	.79

TABLE 3

Run times (seconds) of version 2 of Parallel FLO52 on iPSC/2 for a 256×64 *grid.*

processed their whole sub-domain before packing interior boundary values into buffers. This minimized the number of messages. However, the efficiency is only 27% because only one node in a processor row or column is active at any instant.

By not packing values and sending a message for every row when moving in the x direction and every column when moving in the y direction, the efficiency jumps to 43%, the overall efficiency jumps from 61% to 75% with a corresponding decrease of 450 seconds in run time (on 16 nodes). This is because a diagonal wave of computation develops moving across the computational domain allowing processors to start doing some work before the first processor has finished.

One of the key points to learn is that minimizing communication is not always desirable. In the case of the tridiagonal solves, minimizing the number of messages actually slowed the computation down.

Despite the improvement, the tridiagonal solves are still quite costly (see Table 4). In particular, for large numbers of processors the low efficiency of the tridiagonal

categories	1 node	2 nodes	4 nodes	8 nodes	16 nodes	$e_1(16)$
Total	24772.	12609	6561	3507	2058	.75
Flux	6241.9	3142.1	1586.6	840.5	456.6	.85
Dissipation	4540.1	2277.9	1144.0	579.3	299.0	.95
Time Step	1044.1	530.2	267.2	135.5	71.7	.91
Input/Output	146.3	91.4	66.3	48.0	39.3	.23
Residual Avg.	4306.5	2295.6	1336.2	812.8	627.3	.43
Boundary C's	147.1	86.2	63.2	28.4	13.7	.67
Enthalpy Damp.	420.1	210.0	104.7	52.5	26.4	.99
Time Advance	3302.4	1650.9	821.8	411.8	206.9	.99
Project	3917.7	1966.6	986.9	498.8	261.3	.94
Interpolate	705.6	358.5	183.6	99.6	55.6	.79

TABLE 4

Run times (seconds) of version 3 of Parallel FLO52 on iPSC/2 for a 256×64.

solve seriously affects the run time of the whole algorithm. Based on this observation, version 4 eliminates the tridiagonal solves and replaces them with an explicit residual averaging scheme.

The role of the tridiagonal matrix solve is to smooth the residual and enhance stability. This smoothing is critical to the algorithm's rapid convergence. However, the smoothing can also be accomplished with an explicit local averaging algorithm. One iteration of our new smoothing algorithm defines the new residual at a point (x, y) by averaging its value with the average of the four neighboring points (Jacobi relaxation). That is, for one Jacobi sweep, we replace the central equation in (12) with

$$w^{(k)} - w^{(0)} = \tilde{S}(-\Delta t \sum_{j=0}^{k-1}(\alpha_{kj} H w^{(j)} + \beta_{kj} P w^{(j)}), \quad 1 \le k \le m, \tag{27}$$

where $\tilde{S}$ is the operator defined by

$$(\tilde{S}r)_{i,j} = \frac{1}{2} r_{i,j} + \frac{1}{8}(r_{i+1,j} + r_{i,j+1} + r_{i-1,j} + r_{i,j-1}). \tag{28}$$

We note that this explicit Jacobi smoothing has been used by Jameson in his codes

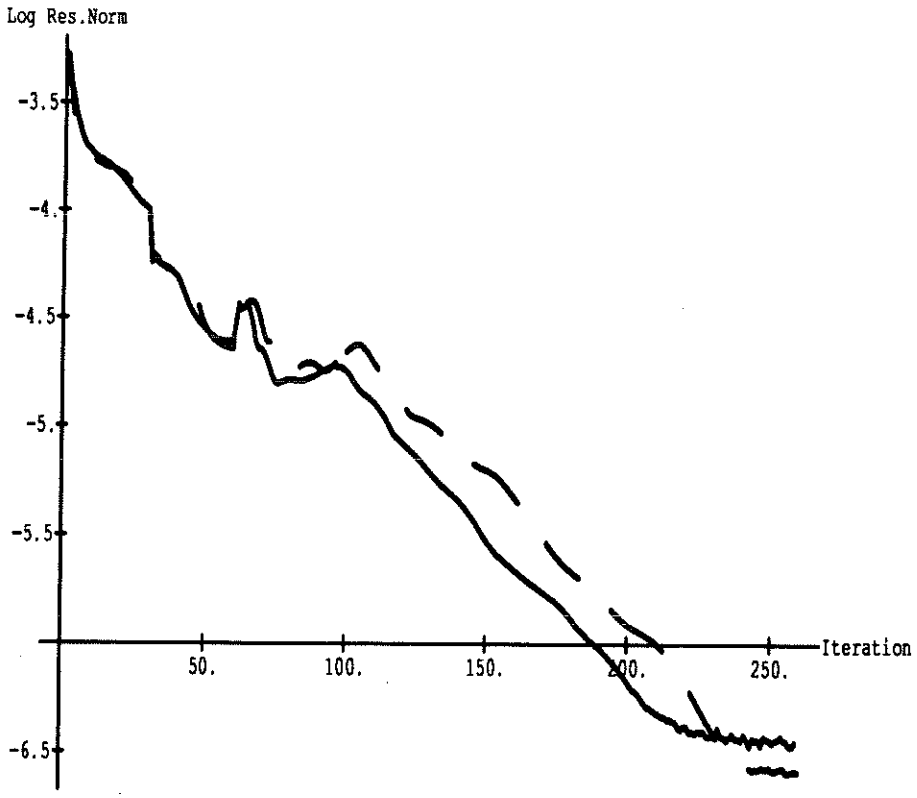

FIG. 12. *Overall convergence using explicit (solid) vs. implicit (dashed) residual averaging on a* 256×64 *grid.*

for solving flow problems on unstructured meshes [12]. Version 4 utilizes two sweeps of this Jacobi iteration in place of the tridiagonal solves. Of course we must not only check the run times of version 4 (given in Table 5), but also the convergence behavior as we have altered the numerical properties of the algorithm. Figure 12 compares the residual of the density in the L_2 norm for versions 3 and 4. We simply state that the new smoothing algorithm yields a better overall convergence rate than the tridiagonal solves on the limited class of problems that we tried. On a single processor, the new algorithm takes approximately the same time to execute a single iteration as the tridiagonal scheme. However as more processors are used, the new smoothing algorithm is faster per iteration than the tridiagonal scheme and the residual averaging efficiency rises from 43% to 78% on 16 processors of the iPSC/2 (Table 5).

In studying Table 5, we make a number of observations about the parallel implementation of FLO52. The first observation is that the residual averaging segment

categories	1 node	2 nodes	4 nodes	8 nodes	16 nodes	$e_1(16)$
Total	24393	12281	6218	3229	1725.	.88
Flux	6240.0	3140.4	1584.9	828.2	439.7	.85
Dissipation	4538.0	2276.4	1143.9	579.4	298.0	.95
Time Step	1044.0	530.1	267.1	135.5	71.2	.91
Input/Output	146.3	91.4	68.3	54.5	40.1	.22
Residual Avg.	3933.5	1976.6	1009.2	543.2	316.4	.78
Boundary C's	146.4	80.2	46.0	24.7	12.2	.67
Enthalpy Damp.	420.3	209.9	104.7	52.4	26.5	.99
Time Advance	3303.1	1651.4	822.4	411.4	207.5	.99
Project	3915.9	1966.1	987.9	500.3	259.4	.94
Interpolate	705.7	358.7	183.4	99.6	55.1	.79

TABLE 5

Run times (seconds) of the final version of parallel flo52 on ipsc/2 for a 256×64 *mesh.*

of the code dropped from 963 seconds for version 1 to 316 seconds for version 4 and that the overall time dropped from 2767 seconds to 1725 seconds. A second observation is that the input/output requirements for the code do not seem to significantly impact performance on the iPSC/2. A third observation is that some of the routines associated with explicit operators are still inefficient. In particular, the residual averaging scheme is the most inefficient major code segment. Figure 13 illustrates this graphically by comparing the total execution time of the residual averaging routine with the time it spends on communication. As the number of processors increases, the average number of internal boundaries per processor grows, causing the number of messages exchanged per processor to increase. Thus even though the messages are shorter, the increase in the number of messages causes the growth in communication time. This still does not explain why the residual averaging routine spends a greater percentage of time communicating than the other explicit routines. In fact, in examining the other explicit routines (flux, dissipation, interpolation, and projection), it is

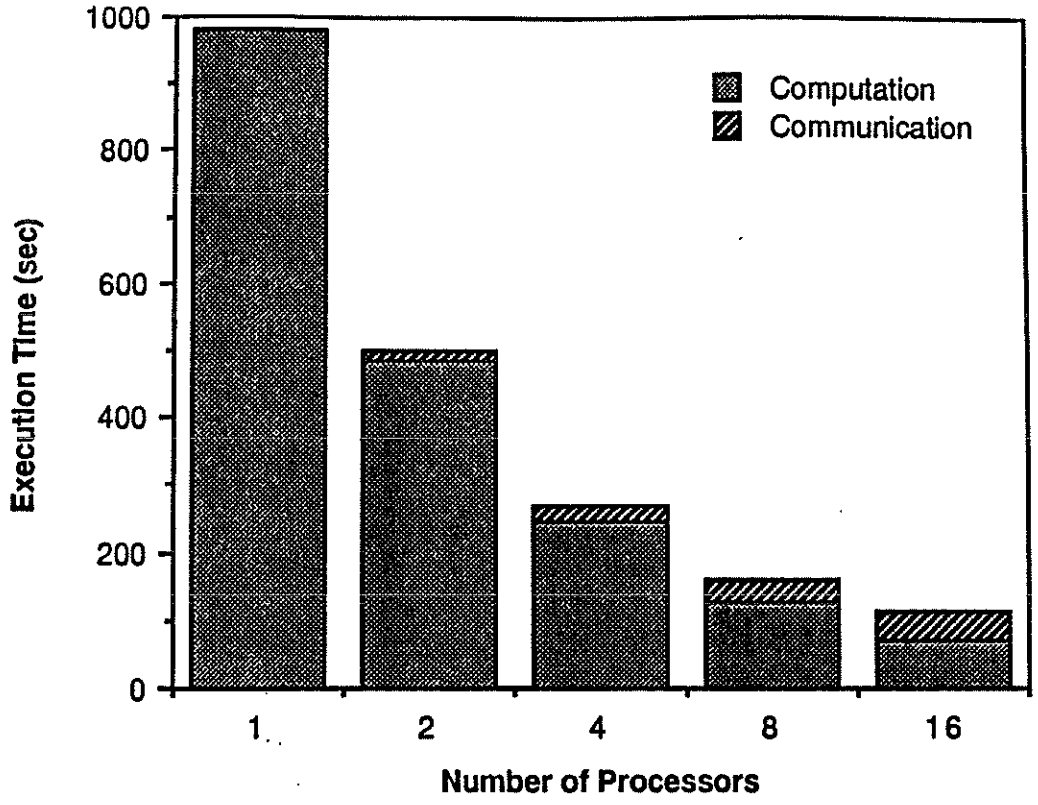

FIG. 13. *Residual Averaging Routine for* 128 x 32 *grid.*

clear that the amount of communication is roughly the same (per call) as the residual averaging routine. The reason for the lower efficiency is related to the amount of computational work associated with each operator: the ratio of computation to communication determines the efficiency. The more complex operators (flux, etc.) which require a greater amount of computational work per cell run more efficiently than simpler operators, such as explicit residual averaging.

Finally, we suggest modifications that should further improve the performance. One possibility is to send messages in parallel. Specifically, the typical communication/computation pattern involves each processor sending and receiving information on four boundaries and then performing the computation. The communication time could be significantly reduced if the four messages are sent in parallel or at least partially overlapped (asynchronous versus synchronous communication). Another modification is to overlap more of the communication and computation. That is, send all boundary information, perform the computation in the interior, and then

grid\nodes	1	2	4	8	16
128 × 32	6172	3142	1622	888	516
	1.0	.98	.95	.87	.75
256 × 32	12345	6220	3159	1685	941
	1.0	.99	.98	.92	.82
256 × 64	24353	12246	6180	3196	1704
	1.0	.99	.99	.95	.89
256 × 128	48450	24318	12250	6217	3216
	1.0	.996	.989	.97	.94

TABLE 6

Run times (top row) and efficiencies (bottom row) for FLO52 on iPSC/2 (excluding input/output).

receive the boundary information that has (hopefully) already arrived. This is in fact implemented in the dissipation routine but is not implemented for any other routines.

In summary, our optimizations increased the efficiency of the code from 57% to 88% and reduced the run time from 2767 seconds to 1725 seconds. While most of these changes were not difficult to implement, our results imply that both algorithm and implementation issues can greatly affect the performance of a code on a parallel machines.

11. Machine Performance Comparisons. In this section we compare the performances of the iPSC/2, the NCUBE and one processor of a Cray X-MP. Our comments in this section are intended to quantify the performance of the existing hypercubes (relative to each other and the Cray) as well as to lead into the next section, which discusses the potential capabilities of hypercube machines.

We first consider the run times and efficiencies of the Intel iPSC/2 and the NCUBE systems. Tables 6 and 7 show both the run time and efficiency of the parallel FLO52 code (excluding input/output times) for a variety of mesh sizes and number of processors. These efficiencies are measured with respect to the leftmost entry in each row (i.e., speed up is measured against the smallest number of processors available for

grid\nodes	4	8	16	32	64	128	256	512
128 × 32	4445	2305	1251	691	416			
	1.0	.96	.89	.80	.67			
256 × 32		4456	2310	1255	694	422		
		1.0	.96	.89	.80	.66		
256 × 64			4512	2345	1285	710	439	
			1.0	.96	.88	.79	.64	
256 × 128				4562	2401	1305	741	454
				1.0	.95	.87	.77	.63

TABLE 7

Run times (top row) and efficiencies (bottom row) for FLO52 on NCUBE (excluding input/output).

a given problem). From the tables, we can see that with our largest grid (256 × 128) the code completed in 454 seconds on a 512-NCUBE and 3,216 seconds on a 16-node iPSC/2. By comparison, a 1 processor Cray X-MP can perform the same calculation in 169 seconds. Of course, these comparisons are not only machine dependent but grid size dependent as the size of the grid effects the efficiency (machine utilization) on the hypercubes. In general, for the iPSC/2 the efficiencies (and run times) are fairly constant for the same number of grid points per processor. This can be seen by comparing the efficiencies along the diagonals of table 6. For this code, an efficiency of 85% is obtained when there are approximately 1000 points per processor. The NCUBE exhibits similar behavior where approximately 300 points per processor is necessary to achieve an 85% efficiency. Overall, on the largest grid the iPSC/2 yields a speedup of 15 going from 1 to 16 processors and the NCUBE yields a speedup of 10.1 going from 32 to 512 processors. While neither of the hypercubes outperformed the Cray X-MP, the NCUBE is capable of computing within a factor of 3 of the Cray. Given the lower costs of the hypercubes and the somewhat primitive nature of both the hardware and software, this performance is impressive. In the next section, we consider the performance of future hypercube machines.

categories	4 nodes		8 nodes		16 nodes		$e_4(16)$	
	Intel	NCUBE	Intel	NCUBE	Intel	NCUBE	Intel	NCUBE
Total	1627.5	4512.5	892.5	2458.5	518.5	1336.0	.78	.84
Flux	414.1	1015.7	232.4	531.1	133.6	295.3	.77	.86
Dissipation	291.5	782.0	151.9	399.1	82.1	207.4	.89	.94
Time Step	68.0	157.2	35.6	79.6	20.3	42.1	.84	.93
Input/Output	18.8	105.5	15.7	177.5	11.3	105.8	.42	.25
Residual Avg.	271.3	910.1	162.0	477.1	112.0	270.4	.61	.84
Boundary C's	26.6	65.6	14.5	38.3	7.4	19.0	.89	.86
Enthalpy Damp.	26.2	69.9	13.1	35.0	6.7	17.6	.98	.99
Time Step	205.7	557.1	103.6	279.4	53.3	141.6	.96	.98
Project	253.2	687.3	132.4	352.6	72.8	185.2	.87	.93
Interpolate	52.1	162.1	31.3	88.8	19.0	51.6	.69	.79

TABLE 8

Run times using the same number of processors on a 128×32 *grid.*

It is difficult to compare the performance of the NCUBE and Intel machines, as they are configured with different numbers of processors. Table 8 compares the performance of the two machines with the same number of processors. We see that the Intel (without vector boards) outperforms the NCUBE when the number of processors is the same (the Intel is roughly 2.8 times faster). Based on separate timing tests (not shown here), the Intel processors are approximately 3 times faster computationally and 1.7 times faster for node-to-node communication involving small messages. Even though the NCUBE has slower computation and communication, it is more efficient than the Intel iPSC/2 with the same number of processors because the computation to communication ratio is better. Unfortunately for the NCUBE (not the iPSC/2), input/output time (requiring node to host communication) represents a significant bottleneck (see table 8). In fact when a large number of processors are used, the input/output operations dominate the total execution time (see figure 14). This may

be due to a number of factors including the following: the host processor is shared with other users, the slow host to node communication on the NCUBE, and the slow host processor. The extent to which this limits the performance depends somewhat on the application. For example, if several runs are made with the same input parameters (e.g., same airfoil with different angles of attack) then this bottleneck may not be so significant. It should also be noted that we did not attempt to optimize the input/output operations for the NCUBE and we probably could have improved this time if optimization was attempted. However, considering that FLO52's input/output requirements are somewhat minimal, this stage represents a disproportionate fraction of the total time.

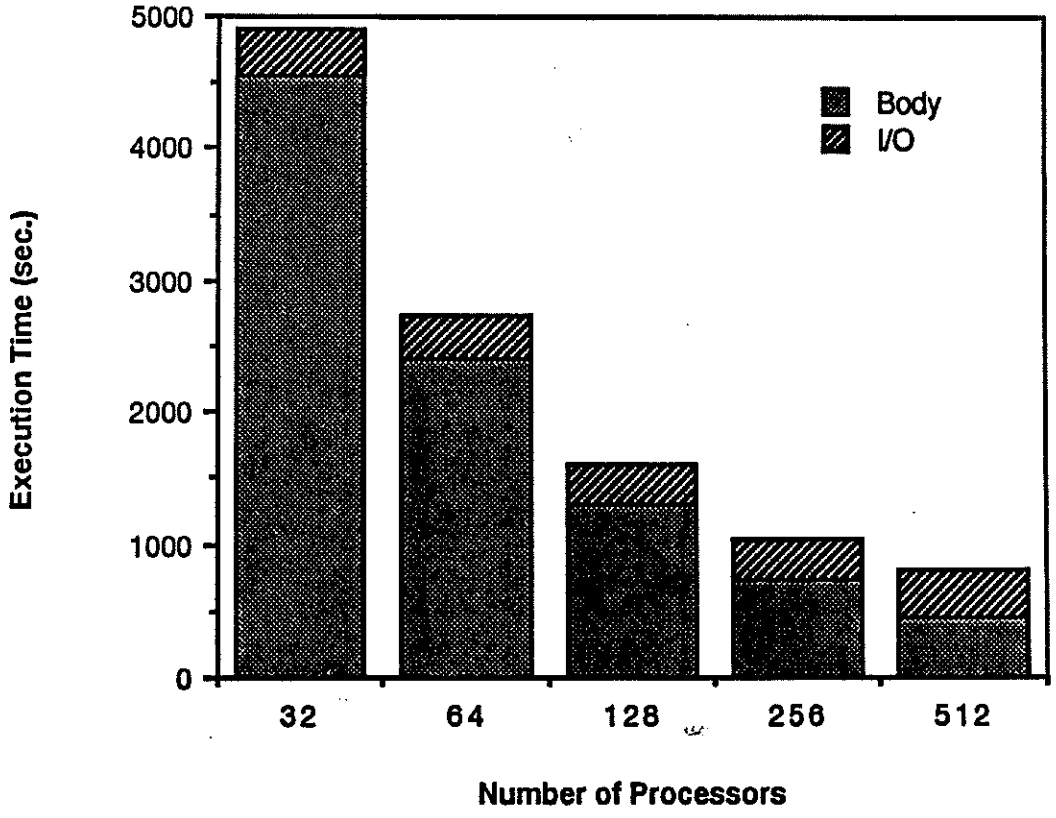

FIG. 14. *Input/Output vs. total run time for* 256 x 128 *Grid on NCUBE.*

It is not fair to conclude on the basis of Table 8 that the IPSC/2 is a faster machine than the NCUBE. While the processors on the iPSC/2 are faster than the NCUBE processors, NCUBE systems are typically configured with more processors than iPSC/2 systems. The most important comparison is performance versus machine

cost, but we do not use this criterion here. Instead, we measure the number of processors necessary on each machine to achieve a given run time. Figure 15 illustrates these results for a 128×32 grid. From this figure, we conclude that it takes approximately 3.3 times more NCUBE processors than iPSC/2 processors to achieve the same run time. If we scale the number of NCUBE processors by 3.27 and compare it with the Intel iPSC/2 results, we see that the results are almost identical (see figure 16). As we already noted, the NCUBE is more efficient than the Intel iPSC/2 with the same number of processors. However, since it takes more NCUBE processors to achieve the same run time as the Intel, there is not much difference in the efficiencies to achieve a particular run time. Thus, we can conclude that an iPSC/2 and an NCUBE with roughly 3.3 times as many processors as the iPSC/2 are approximately equivalent.

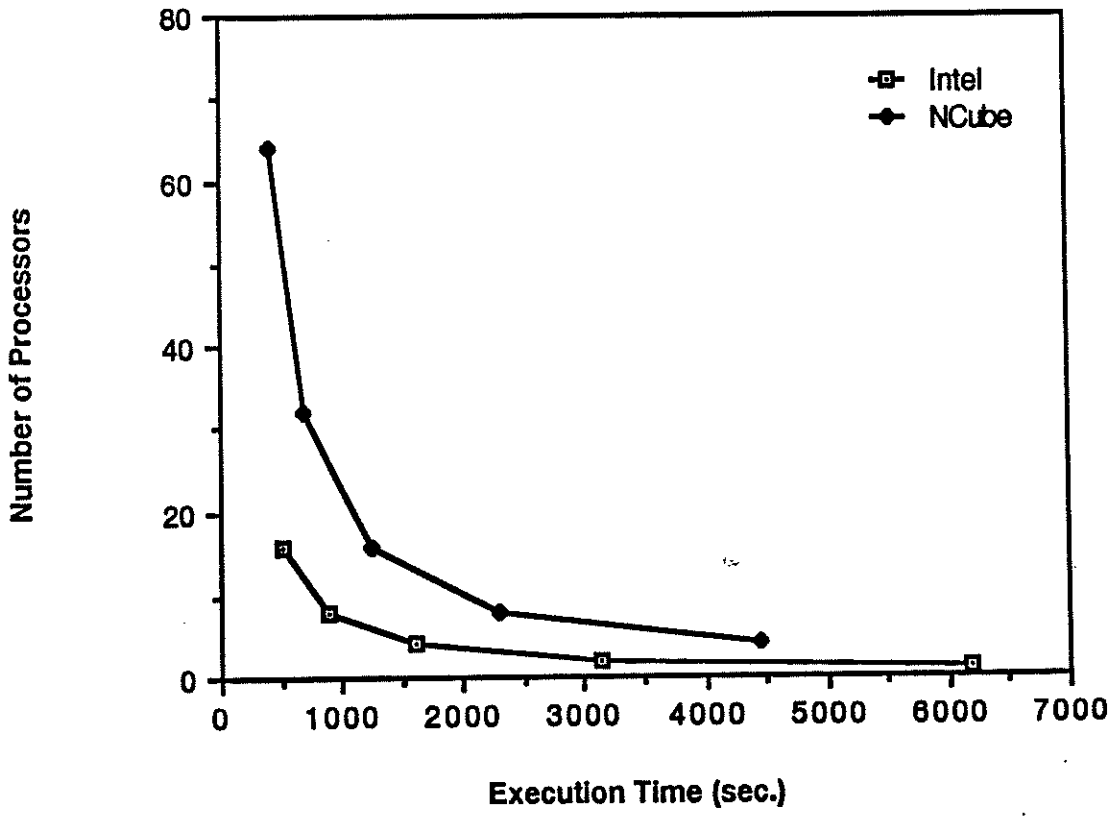

FIG. 15. *Processors necessary to achieve a given run time on a* 128×32 *grid.*

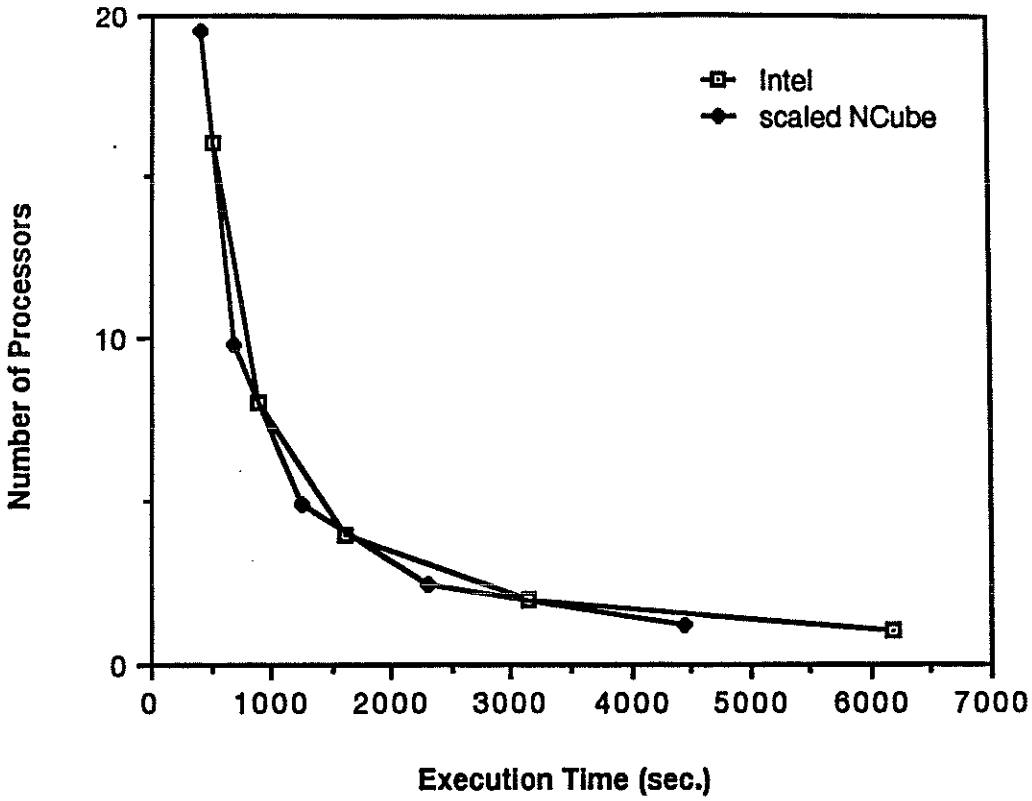

FIG. 16. *Scaled NCUBE processors to achieve a given run time on a* 128×32 *grid.*

12. Timing Model. In this section, we analyze the run time and efficiency of the FLO52 algorithm as a function of communication speed, computation rate, number of processors, and number of grid points. A model for the execution time of FLO52 is developed to carry out the analysis. This model allows us to explore the potential performance of future hypercube systems by considering not yet realizable machine parameters. Comparisons between future hypercubes and traditional supercomputers are made.

The FLO52 execution time model was created using the *Mathematica* symbolic manipulation package [19]. For each major subroutine in FLO52, there is a corresponding function in the *Mathematica* program. Instead of computing the numerical operators, these functions compute the number of floating point operations in the corresponding subroutine. For the asymptotic analysis, these functions return the maximum number of operations any processor executes. For the performance modeling,

the functions return predicted times for the subroutines averaged over all processors. Below we illustrate a few of the *Mathematica* functions:

$$\begin{aligned}
\text{flux}[x_, y_, P_] &:= 35xy \text{ adds} + (12x + 11y) \text{ adds} + \\
&\quad 15xy \text{ mults} + (7x + 8y) \text{ mults} \\
\text{enthalpy}[x_, y_, P_] &:= 7xy \text{ adds} + 11xy \text{ mults} + 3xy \text{ divides} + \\
&\quad \text{copy} + 2 \text{ adds} + \text{divides} \\
\text{rk}[x_, y_, i_, j_, P_] &:= \text{step}[x, y, P] + \text{saveold}[x, y, P] + \text{fiveresi}[x, y, i, j, P] + \\
&\quad 5(\text{commbdry}[x, y, P] + \text{take}[x, y, P]) + \text{enthalpy}[x, y, P] + \\
&\quad \text{If}[i == j, 0, \text{rescal}[x, y, P] + \text{gop}[P] + \text{farfield}[x, y, P], P]
\end{aligned}$$

The above functions correspond to flux calculation, enthalpy damping, and a Runge-Kutta iteration. The variables x and y are the number of grid points per processor in each direction, P is the number of processors, and i and j are algorithm parameters. The *Mathematica* functions may have calls to other functions (including recursive calls). The variables corresponding to floating point operations (adds, mults, etc.) are assigned appropriate weights. For the asymptotic analysis, they are assigned in units of floating point operations (ops), where 1 op corresponds to a single add. For the performance modeling, the floating point operations are assigned values determined from separate timing experiments. A sample operation count for the *rk* function expanded by the *Mathematica* package yields:

$$\begin{aligned}
rk(x, y, 1, 1, P) &= 207\alpha + \beta(40 + 379x + 396y) + \\
&\quad ops(5.1 + 459.5x + 153y + 911.1xy)
\end{aligned} \tag{29}$$

where we assume that the cost, $c(n)$, to send and receive a message of length n can be modeled by

$$c(n) = \alpha + \beta n. \tag{30}$$

In general the xy terms in (29) correspond to operations on the interior of a processor's sub-domain. The separate x and y terms correspond to operations associated with the boundary of these sub-domains. Using these *Mathematica* functions we produce an overall execution time for the multigrid algorithm. The final execution time

expression is somewhat complicated. We give a simplified expression for a 3 level multigrid algorithm using 30 sawtooth iterations on the two coarsest grid levels and 200 sawtooth iterations on the fine grid:

$$T(\alpha,\beta,N,ops,P) \approx ops[6663 + 479636\sqrt{\tfrac{N}{P}} + 297036\tfrac{N}{P}] + \alpha[142563 + \log_2(P)] + \beta[34328 + 366391\sqrt{\tfrac{N}{P}} + 3080.2\log_2(P)] \quad (31)$$

To produce this expression, we assume that the number of grid points in the x direction is four times the number in the y direction, $N \geq 16P$, and $P > 8$, where N is the total number of grid points.

We simplify (31) by extracting the basic elements that govern the behavior of the execution time:

$$(32) \qquad T(cc,N,ops,P) \approx cc[d_1 + d_2\sqrt{\tfrac{N}{P}} + \log_2(P)] + ops[d_3 + d_4\sqrt{\tfrac{N}{P}} + d_5\tfrac{N}{P}].$$

The d_i's are constants, and the communication costs (α and β) are approximated by one variable, cc. The $\sqrt{\frac{N}{P}}$ terms in (32) correspond to operations on the boundary of the processor sub-domains. The $\frac{N}{P}$ term corresponds to computation in the interior and the $\log_2(P)$ is produced by the global communication operations necessary for norm calculations. For $P < 10,000$, these global operations do not dominate the overall run time. Notice that with the exception of the $\log(P)$ terms, all occurrences of P and N are together as $\frac{N}{P}$. This implies that for a modest number of processors the important quantity is the number of grid points per processor. Additionally, equation (32) illustrates that a perfect speedup can be obtained for large enough grids. That is, if we fix P, and let N approach infinity, then T becomes proportional to $\frac{N}{P}$. This implies that the efficiency approaches one.

Before utilizing the model, we verify its validity by comparing the predicted run times with those obtained on the NCUBE and Intel iPSC. It is worth mentioning that our initial comparisons resulted in substantial improvements in the program as the model revealed inefficiencies in the original coding. Figure 17 illustrates the close correlation between the model's predicted time and the actual run time for the NCUBE. Similar results (not shown here) were obtained for the iPSC/2.

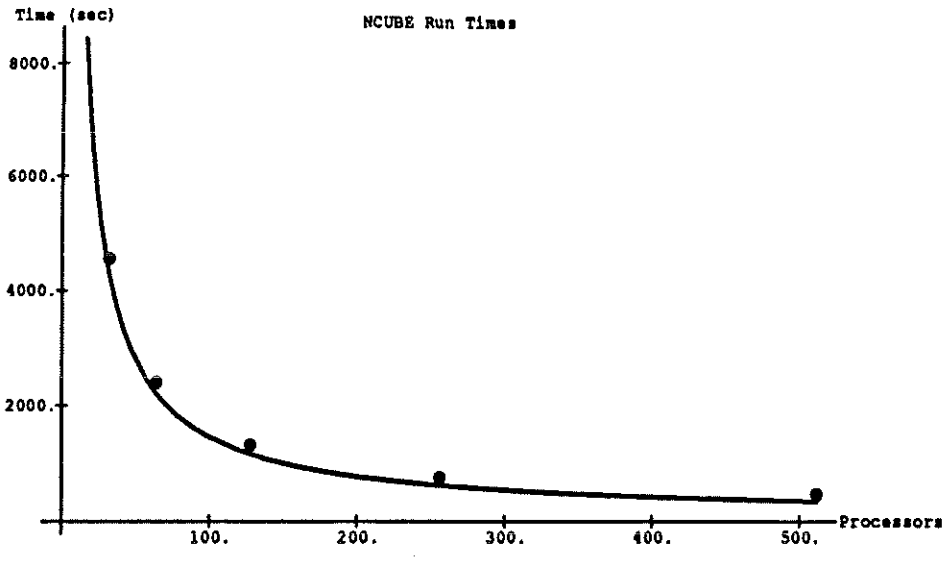

FIG. 17. *Comparison of model's predicted time and run times obtained on the NCUBE for a* 256×128 *grid.*

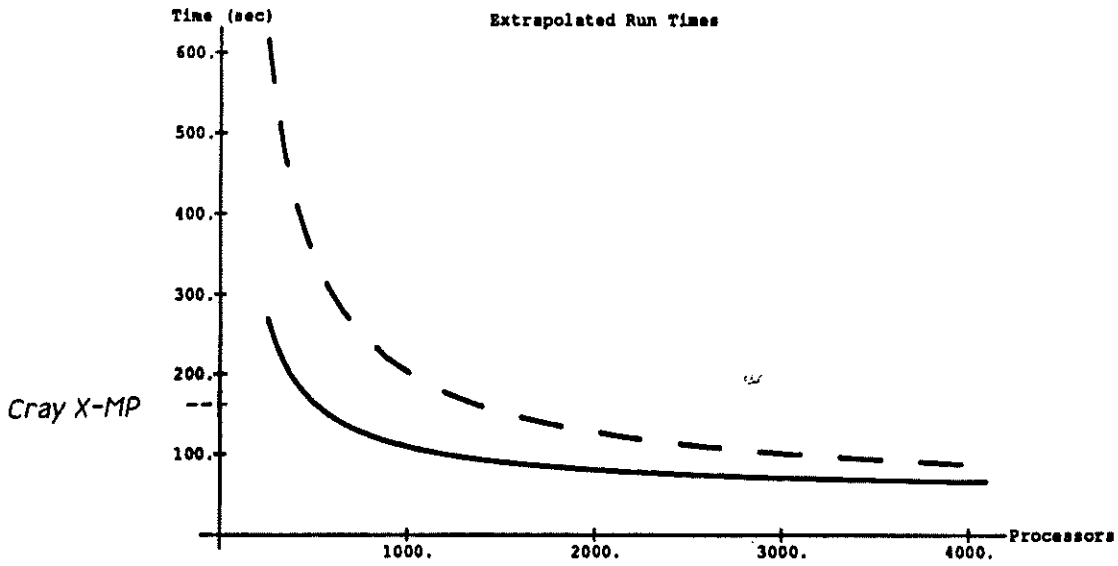

FIG. 18. *Predicted run time vs no. of processors for Intel iPSC/2 (solid) and NCUBE (dashed).*

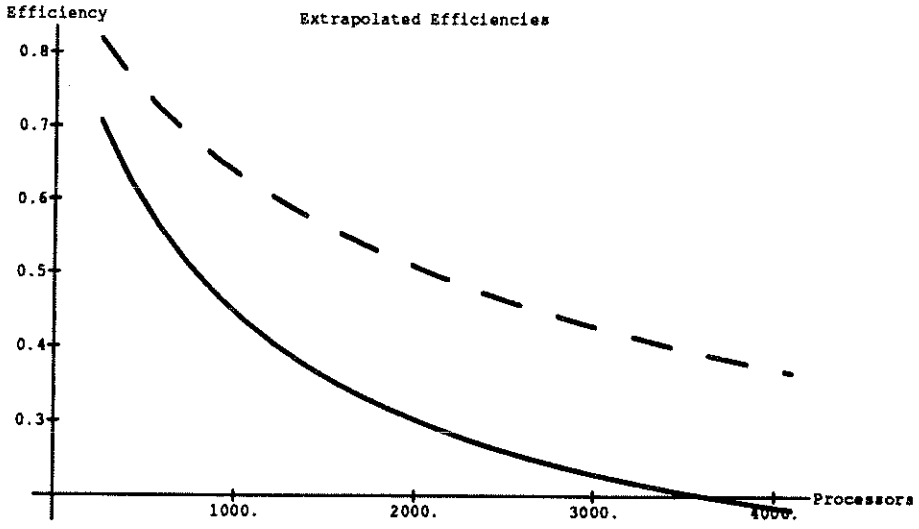

FIG. 19. *Predicted efficiency vs no. of processors for Intel iPSC/2 (solid) and NCUBE (dashed).*

We now use the model to illustrate the potential of hypercube computers. Specifically, we compare the predicted times of the FLO52 model with the actual run times of a Cray X-MP (using one processor) located at NASA/Ames Research Center. It should be kept in mind when reading these comparisons that the results are not only machine parameter dependent but also grid size dependent. For our experiments in this section, we use primarily grid sizes that are appropriate for this problem.

Before using the model, we recall from the previous section that the execution time of the flow code on the NCUBE was about three times the execution time of the code on the Cray X-MP. Given that neither the NCUBE nor the iPSC/2 outperformed the Cray, it is logical to ask what machine parameters are required to match (or surpass) the Cray X-MP. To do this, we can use the model to predict the performance of future, currently non-existent, machines. Since the parameter space is vast, it is necessary to understand how different parameters interact with each other and ultimately reflect the total run time. For the most part, there are three possibilities for improving a hypercube's performance: improve communication speeds, improve computation rates, or use more processors. However, when extrapolating these parameters it is necessary to understand how they influence machine utilization. For example, figures 18 and 19 consider the predicted run time and efficiency of the iPSC/2 and NCUBE

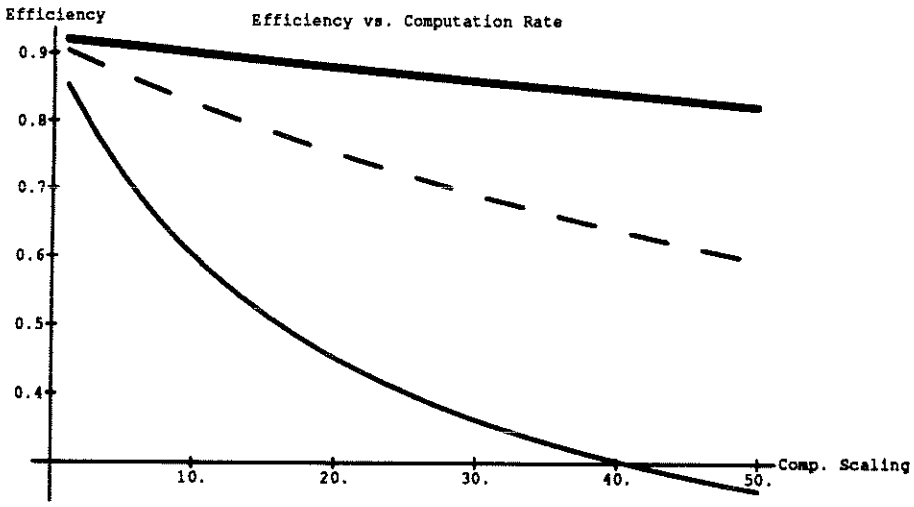

FIG. 20. *Drop in efficiency as computation rate increases. Thin line is for 6554 grid points, thick line is for 163,840 points and dashed line is for 32,768 points. 'Comp. Scaling' factor refers to increase over iPSC/2 computation rate.*

as a function of available processors for a 256×128 grid. In effect, this reduces the amount of computation per processor that must be performed. As expected, the run time decreases when more processors are used. Unfortunately, many processors are needed to match a Cray X-MP (700 for an iPSC/2 and 1500 for an NCUBE) due to the large drop in efficiency. Of course, the efficiencies would be higher if a grid with more cells were used (as is typically the case for 3-dimensional problems).

Another possibility is to reduce the run time by using faster processors or perhaps vector boards. To model this, we introduce scaling factors for the computation and communication rates. By definition, a scaling factor of k implies an improvement by a factor of k over the iPSC/2. As figure 20 illustrates, an increase in computation rate without a corresponding increase in communication speed also results in a substantial under-utilization of the machine. That is, while the run time continues to decrease, it is ultimately limited by communication speeds.

If we want high efficiencies, reductions in computation time per processor should also be accompanied by improvements in communication time. We summarize the situation in figure 21 where we fix the number of processors and the communica-

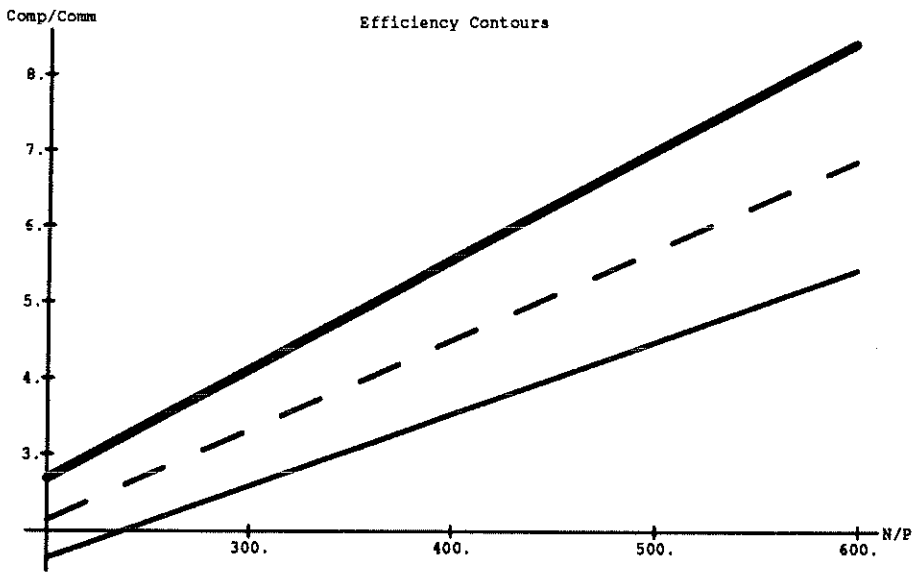

FIG. 21. *Contour lines of efficiency varying the grid points per processor and the computation/communication ratio. The thick, dashed, and thin lines correspond to efficiencies of 70%, 73%, and 76% respectively.*

tion speeds, and vary the computation rate as well as the number of grid points. The computation/communication ratio on the y axis is given based on the iPSC/2's ratio. That is, a ratio of 5 implies that this hypothetical machine has a computation/communication ratio that is 5 times larger than an Intel iPSC/2. Our intention in the figure is to highlight the dependence of the efficiency on both the computation/communication ratio and on the number of grid points per processor. That is, the efficiency contours in figure 21 are primarily dependent on these ratios as opposed to the individual quantities (when $P < 10,000$). Thus, if one has a target efficiency, figure 21 indicates the relationship between the computation/communication ratio and the number of grid points per processor required to achieve this target.

Suppose, for example, we choose 76% as a target efficiency for solving the FLO52 problem on a 256×128 grid using a 64 processor machine. This corresponds to 512 grid points per processor. From figure 21 we deduce that the computation to communication ratio should be approximately 4.5 times greater than on the current iPSC/2 machine. Using our model, it is now possible to determine communication and computation speeds necessary for a machine to match the Cray subject to the

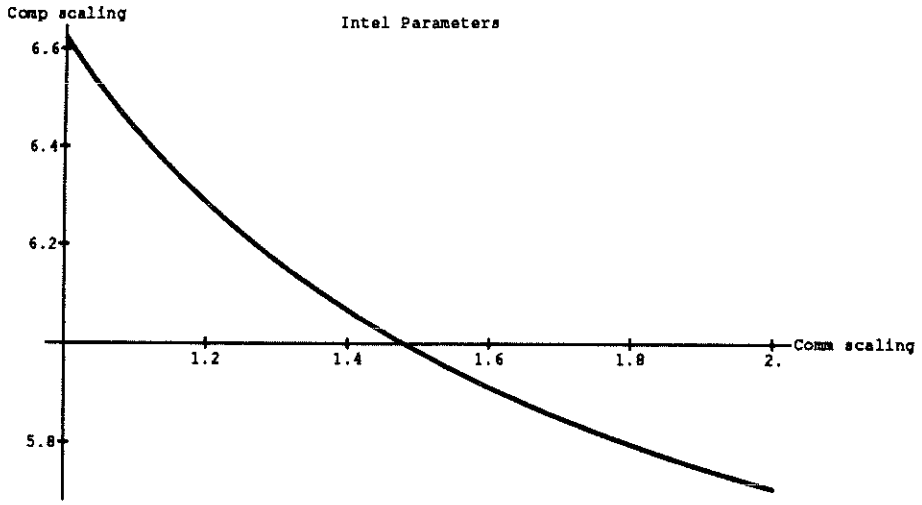

FIG. 22. *Increase in performance necessary for Intel to compete with Cray X-MP on a* 256×128 *grid.*

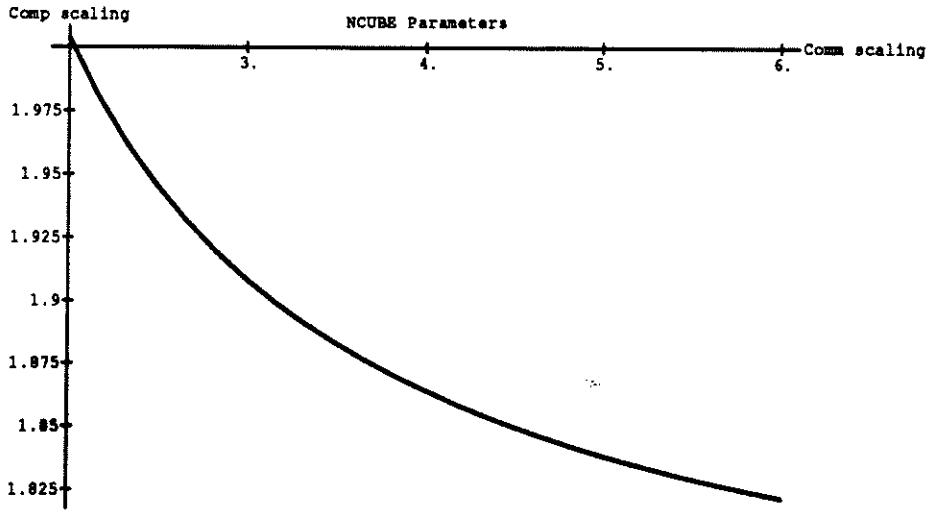

FIG. 23. *Increase in performance necessary for NCUBE to compete with Cray X-MP on a* 256×128 *grid.*

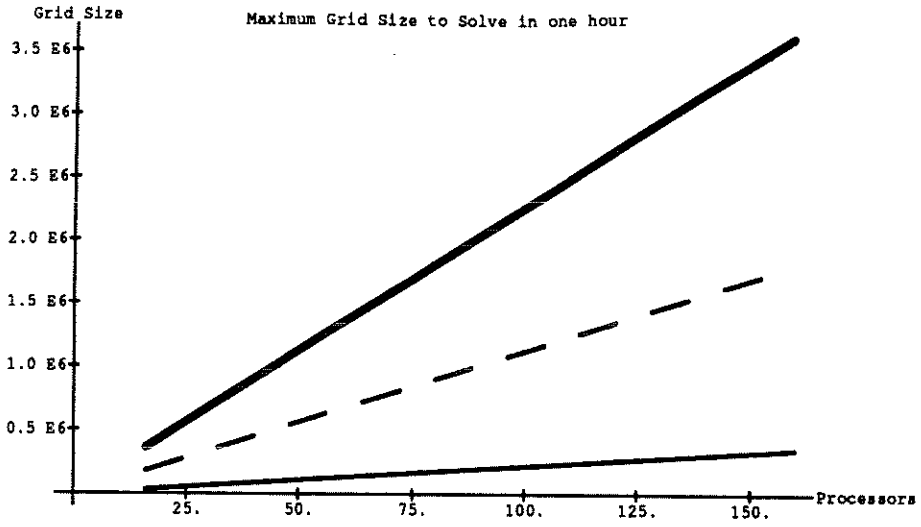

FIG. 24. *Largest grid that 3 level FLO52 algorithm can complete in one hour. The thick, dashed and thin lines corresponds to a systems with communication speeds of 10, 10, and 1 times faster than the iPSC/2 and computation speeds of 10, 5, and 1 times faster than the iPSC/2.*

4.5 ratio. Specifically, a machine with communication speeds 1.4 times faster than the iPSC/2 and computation rate 6.1 times faster can solve this problem in the same time as a Cray X-MP. Figure 22 shows the necessary computation and communication parameters to solve our problem in the same time as a Cray X-MP. We can repeat this procedure for a 512 node NCUBE system. Specifically, figure 23 shows the same information for the NCUBE where, for example, increasing both the communication and the computation by a factor of 2 matches the Cray's performance. Considering the primitive state of these current machines, it does not seem unreasonable that these kinds of increases can be realized in the next generation machines.

While most of the plots in this section have considered run times for a fixed problem size, it may be argued that a better measure is to determine the largest problem that can be solved in a fixed amount of time. We conclude this section, with a plot (figure 24) that considers the largest size problem that can be solved in one hour on a hypercube-like system.

13. Summary. Based on our FLO52 experience, it is clear that hypercube machines can supply the high computation rates necessary for CFD applications. Un-

fortunately, this performance comes at considerable programming expense. To write efficient code, close attention must be paid to the algorithm/architecture interaction and the programming environment can accentuate the difficulties.

In general, parallelization (based on domain decomposition) is straightforward. In fact, the main body of the parallel FLO52 code closely resembles the serial version, with the exception of the residual averaging. Of course, domain decomposition is more complicated if unstructured grids are used. And, the parallelization process requires more effort if a greater percentage of the algorithm is implicit.

Even though FLO52 is largely explicit, a substantial effort was required to produce efficient code. Much of the time was spent coding and debugging input/output routines. Additional time was lost to peculiarities of the operating systems (non-robust compilers, frequent crashing of machine, etc.). Still more time was spent in optimization. Even the conversion from the iPSC to the NCUBE was time-consuming.

While it was difficult to produce an efficient code for these hypercubes, the overall computing performance is promising. In our experiments, the 16-node iPSC/2 ran within a factor of 20 of the Cray X-MP without using vector boards and the 512-node NCUBE came within a factor of 3. Additionally, with the help of a timing model, we can predict the performance of future hypercube systems. A 64-node iPSC/2 machine with processors that are 5.5 times faster computationally and 2.5 times faster communication-wise will yield similar performance to a one processor Cray X-MP. A 512-node NCUBE system with 2 times faster communication speeds and 2 times faster computation rate will also produce performance similar to the Cray X-MP.

REFERENCES

[1] Agarwal, *Development of a Navier-Stokes Code on a Hypercube and a Connection Machine*, presented at *Fourth Conference on Hypercubes, Concurrent Computers and Applications*, Monterey, CA, 1989.

[2] A. Brandt, *Guide to Multigrid Development*, in *Multigrid Methods*, W. Hackbusch and U. Trottenberg, eds., Lecture Notes in Mathematics 960, Springer-Verlag, Berlin, 1982.

[3] D. A. Calahan, *Performance Analysis and Projections for a Massively-Parallel Navier-Stokes*

Implementation, presented at *Fourth Conference on Hypercubes, Concurrent Computers and Applications*, Monterey, CA, 1989.

[4] T. Chan, Y. Saad, *Multigrid algorithms on the hypercube multiprocessor*, IEEE Trans. Conput., C-35, pp. 969-977.

[5] G. Chesshire and A. Jameson, *FLO87 on the iPSC/2: A Parallel Multigrid Solver for the Euler Equations*, presented at *Fourth Conference on Hypercubes, Concurrent Computers and Applications*, Monterey, CA, 1989.

[6] R. Courant and D. Hilbert, *Methods of Mathematical Physics, vol. 2*, Interscience Publishers, New York, 1962.

[7] Personal Communication with Michael Heath, Oak Ridge National Laboratory, 1988.

[8] *iPSC User's Guide*, Version 3, Intel Scientific Computers, Beaverton, Oregon, 1985.

[9] *iPSC/2 User's Guide*, Version 1, Intel Scientific Computers, Beaverton, Oregon, 1987.

[10] A. Jameson, *Solution of the Euler Equations for Two Dimensional Transonic Flow by a Multigrid Method, Appl. Math. and Comp.* 13:327–355 (1983).

[11] A. Jameson, W. Schmidt, and E. Turkel, *Numerical Solution of the Euler Equations by Finite Volume Methods Using Runge-Kutta Time Stepping, AIAA Paper 81-1259*, AIAA 14th Fluid and Plasma Dynamics Conference, Palo Alto, 1981.

[12] A. Jameson and T. J. Baker, *Improvements to the Aircraft Euler Method, AIAA Paper 87-0452*, AIAA 25th Aerospace Sciences Meeting, Reno, NV, 1987.

[13] D. Jespersen, *Enthalpy Damping for the Steady Euler Equations, Appl. Numerical Mathematics* 1:417–432 (1985).

[14] *NCUBE Users Manual*, Version 2.1, NCUBE Corporation, Beaverton, Oregon, 1987.

[15] Y. Saad and M. Schultz, *Data Communication in Hypercubes, Report YALEU/DCS/RR-428*, Yale University, 1985.

[16] Personal Communication with Dave Scott, Intel Scientific Computers, 1987.

[17] S. Spekreijse, *Multigrid Solution of the Steady Euler Equations*, CWI Tract 46, Center for Mathematics and Computer Science, Amsterdam, The Netherlands, 1988.

[18] D. Tylavsky, *Assessment of Inherent Parallelism in Explicit CFD Codes, NASA Ames Research Report*, Moffet Field, CA, 1987.

[19] S. Wolfram, *Mathematica, A System For Doing Mathematics By Computer*, Addison-Wesley, Reading, Massachusetts, 1988.