Performance of an Euler Code on Hypercubes*

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

March 1989

Abstract

This paper evaluates the performance of hypercube machines on a computational fluid dynamics problem. 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. In this paper, we give a description of FLO52, its hypercube mapping, and the code modifications to increase machine utilization. Results from two hypercube computers (a 16 node iPSC/2, and a 512 node NCUBE/ten) are presented 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. Predictions about future hypercubes are made using this timing model.

1 Introduction

An enormous research effort into parallel algorithms, hardware, and software has already been undertaken motivated by the need for more computing power within many scientific applications. Parallel machines, such as hypercubes, 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 in fact supply the additional computing power necessary to solve these problems.

To evaluate the potential performance of hypercubes on realistic 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). The fluid code was Antony Jameson's widely used FLO52 [1], which solves the two-dimensional steady Euler equations describing 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. In this paper, we give a description of FLO52, its hypercube mapping, and the code modifications necessary to improve machine utilization. Additionally, a performance evaluation is made based on runs from three machines (the 16 node Intel iPSC/2 located at Stanford University, the 512 node NCUBE/ten located at Caltech, and the Cray X-MP/48 located at NASA/Ames Research Center) and on a mathematical model of the execution time. The model defines the execution time as a function of several machine and algorithm parameters and accurately predicts the actual run times obtained. It is used to predict the performance of the code in interesting but not yet physically realizable regions of the parameter space.

2 The Euler Code FLO52

An already existing flow code, FLO52, written by Antony Jameson [1] was chosen to study the performance of hypercube machines. FLO52 is 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. In addition to its widespread use, FLO52 was chosen for study because of the multigrid acceleration used. In particular, the efficient parallelization of the algorithm's grid hierarchy (including grids with a small number of grids points), is non-trivial.

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

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

*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.

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. 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. For numerical stability, a dissipation term which is a blend of second and fourth-order differences is added. This gives the approximation

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

where 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.

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

$$\frac{dw}{dt} + Hw + Pw = 0, \tag{3}$$

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 (3) can be written in the form

$$\begin{aligned} w^{(0)} &= w_n \\ \text{for } k &= 1 \text{ to } m \\ w^{(k)} &= w^{(0)} - \Delta t \textstyle\sum_{j=0}^{k-1}(\alpha_{kj} H w^{(j)} + \beta_{kj} P w^{(j)}) \\ w_{n+1} &= w^{(m)}. \end{aligned} \tag{4}$$

This procedure 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

$$\sum_{j=0}^{k-1} \alpha_{kj} = \sum_{j=0}^{k-1} \beta_{kj}, \quad 1 \le k \le m. \tag{5}$$

(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.)

A convergence acceleration technique sometimes called residual smoothing or implicit residual averaging is used. This involves a slight modification of (4). In particular, the update for $w^{(k)}$ is now given by :

$$S(w^{(k)} - w^{(0)}) = -\Delta t \sum_{j=0}^{k-1}(\alpha_{kj} H w^{(j)} + \beta_{kj} P w^{(j)}), \tag{6}$$

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. The solution of the linear system $S\Delta w = r$ in (6) involves solving a series of independent tridiagonal Toeplitz matrices in the x and y directions.

Two other convergence acceleration techniques are used. One is "local time-stepping" where a different Δt is chosen for each cell according to a local CFL condition. Another is enthalpy damping [2] which takes advantage of the fact that the enthalpy is constant along streamlines [3].

The final acceleration technique to be discussed is the multigrid technique, which effects a substantial improvement in convergence. 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 fine grid solution. The coarse grid solution 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 ([5], [6]).

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 Parallelization of FLO52

Conceptually, converting the sequential FLO52 code to a hypercube version is relatively simple. Figure 1 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 which are then assigned to different processors. Each subdomain

has a "boundary layer" that contains values updated by other processors. Assignments are made using a binary reflected Gray code in two dimensions, as shown figure 1.

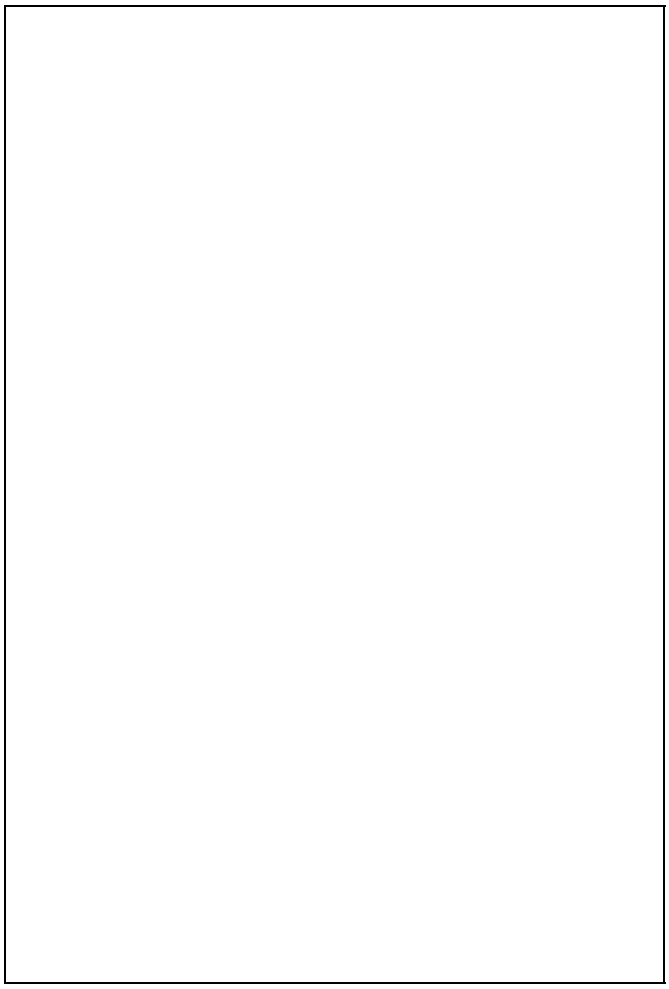

Figure 1: Computational domain mapped to hypercube.

The code structure in the main body of the computation closely resembles the sequential version, with the exception of some re-ordering of the computations to decrease communication overhead. The algorithm is fully explicit except for an implicit residual averaging scheme. The nested loops in the explicit sections now operate on the local subdomains instead of the whole domain. Typically the computation/communication pattern for the nested loops in each processor is as follows. Each processor updates points in its subdomain (applying a local differencing operator). Then each processor exchanges the updated boundary values with the appropriate neighbor. If a boundary corresponds to a physical boundary, then some boundary condition 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 grid points at a distance two). This larger stencil results in additional communication since information from points on the subdomain boundary and points adjacent to the boundary must be communicated to the neighbors. Note that if the coarsest grid during the multigrid cycle has a minimum of two points in each dimension per processor the dissipation operator can get needed values from nearest neighbors so, for the explicit part of the code, all communication is between nearest neighbors.

The implicit part of the code involves solving a series of tridiagonal matrices. Being implicit, most of the operations cannot be performed well in parallel. Many experiments were made with this element of the code. The experiments correspond to implementation changes as well as changing the algorithm. A discussion of this part of the code will be deferred to the performance section of the paper where the results of these experiments will be presented.

4 Optimization of Parallel FLO52

To quantify the bottlenecks that arise as a result of parallelization, we analyze the main algorithm modules of the code and describe several modifications to improve parallel efficiency. 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 (6).
2. Dissipation: The Pw term in (6).
3. Local Time Step: Compute local time step.
4. Residual Averaging: Solve the tri-diagonal systems of equations in (6).
5. Boundary Conditions
6. Enthalpy Damping
7. Time Advance: Combine the results of the above categories to form (6).
8. I/O
9. , 10. Projection and Interpolation: Multigrid operations for grid transfers.

Table 1 contains the run time, percentage of CPU time, and efficiency for each category running on one node and 16 nodes of the iPSC/2 for the initial version of the code. In our experiments efficiency is defined relative to a base number of processors as follows :

$$e_b(p) = \frac{bT(b)}{pT(p)} \tag{7}$$

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. For these cases a "relative" efficiency is calculated based on the smallest number of processors needed to run the problem. Unless otherwise stated, b is equal to 1. For the sake of comparison, all timing results in this section are given in seconds and obtained by executing with the same input parameters on the iPSC/2.

category\nodes	1	%	16	%	$e_1(16)$
Total	25027	100.0	2767	100.0	.57
Flux	6241.3	24.9	565.5	20.4	.69
Dissipation	4779.6	19.1	476.9	17.2	.63
Time Step	1044.2	4.1	71.2	2.6	.92
Residual Avg.	4233.3	16.9	963.3	34.8	.27
Boundary C's	146.6	0.6	27.1	1.0	.34
Enthalpy Damp	420.1	1.7	27.2	1.0	.96
Advance Soln.	3301.8	13.2	210.1	7.6	.98
Input/Output	147.6	0.6	39.2	1.4	.24
Project	4006.3	16.0	323.5	11.7	.77
Interpolate	705.9	2.8	62.5	2.3	.71

Table 1: Run times (sec.) for Initial Version on iPSC/2.

category\nodes	1	%	16	%	$e_1(16)$
Total	24393	100.0	1725	100.0	.88
Flux	6240.0	25.6	439.7	25.5	.85
Dissipation	4538.0	18.6	298.0	17.3	.95
Time Step	1044.0	4.3	71.2	4.1	.91
Residual Avg.	3933.5	16.1	316.4	18.3	.78
Boundary C's	146.4	0.6	12.2	0.7	.67
Enthalpy Damp	420.3	1.7	26.5	1.5	.99
Time Advance	3303.1	13.5	207.5	12.0	.99
Input/Output	146.3	0.6	40.1	2.3	.22
Project	3915.9	16.1	259.4	15.0	.94
Interpolate	705.7	2.9	55.1	3.2	.79

Table 2: Run times (sec.) for Final Version on iPSC/2.

Each run uses a three level sawtooth multigrid method with one Runge-Kutta pre-relaxation sweep; 30 multigrid iterations are used on each of the two coarser grids to get an initial approximation on the next finer grid, and 200 iterations are used on the finest grid. The algorithm is applied to a transonic flow problem with an angle of attack equal to 1.25 degrees and a Mach number of 0.8. The finest grid in the multigrid procedure is a 256×64 mesh.

We can see from table 1, the efficiency is only 57% on the 16 node system. Additionally, we expect this efficiency to degrade further as more processors are used. Still, a 57% efficiency corresponds to a speedup of better than 9 on the 16 node machine. We should note that the code tests whether messages need to be sent along each dimension so no messages are sent when the code is running on a single processor. This yields a truer efficiency than if a processor sends messages to itself.

A detailed inspection of the timings reveals three areas where the poor machine utilization significantly degrades the overall run time. These areas correspond to the flux, dissipation, and residual averaging routines. We now describe modifications to our original implementations of these subelements.

For both the flux and dissipation routines, we rewrote the code to reduce the number of messages. For the flux routine this was quite simple. In particular, the old routine used to send 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 to be sent and sees an improvement in efficiency from 69% to 85%. Unfortunately reducing the number of messages in the dissipation routine required a significant re-ordering of the computations and additional memory. In particular, the older version sends one message for each corresponding cell on a border. The new version sends just one message for each border. 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 the dissipation routines went from 63% to 95% and the overall efficiency from 57% to 61% on the 16 node system.

We next describe our modifications to the tridiagonal matrix solves. During the tridiagonal solves, 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, and then bottom to top and top to bottom in the y direction. Initially, all interior boundary values were packed into buffers to minimize the number of messages. This yielded an efficiency of 27% for residual averaging. Processors on the far side from the start of a wave have to wait until all processors in between have computed on their whole sub-domain.

By not packing values and by 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%. This is because a diagonal wave develops moving across the computational domain allowing the processors on the far side from the start to do some work before the first processor has completed all computation on its sub-domain.

Despite the improvements, the tridiagonal solves are still quite costly. In particular, for large numbers of processors the low efficiency of the tridiagonal solve can seriously affect the run time of the whole algorithm. Based on this observation, we eliminated the tridiagonal solves by replacing them with an explicit residual averaging scheme. The role of the tridiagonal matrix solve is to smooth the residual. This smoothing is quite 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). We utilized two iterations of this new smoothing algorithm instead of the tridiagonal solves. Of course we must not only check the run times using the explicit smoothing, but also the convergence behavior as we have altered the algorithm. We simply state that the new smoothing algorithm yields a better overall convergence rate than with 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

grid\nodes	1	2	4	8	16
128x32	6172	3142	1622	888	516
	1.0	.98	.95	.87	.75
256x32	12345	6220	3159	1685	941
	1.0	.99	.98	.92	.82
256x64	24353	12246	6180	3196	1704
	1.0	.99	.99	.95	.89
256x128	48450	24318	12250	6217	3216
	1.0	.996	.989	.97	.94

Table 3: Run times (top row) and efficiencies (bottom row) for FLO52 on iPSC/2 (excluding input/output).

grid \ nodes	4	8	16	32	64	128	256	512
128x32	4445	2305	1251	691	416			
	1.0	.96	.89	.80	.67			
256x32		4456	2310	1255	694	422		
		1.0	.96	.89	.80	.66		
256x64			4512	2345	1285	710	439	
			1.0	.96	.88	.79	.64	
256x128				4562	2401	1305	741	454
				1.0	.95	.87	.77	.63

Table 4: Run times (top row) and efficiencies (bottom row) for FLO52 on NCUBE (excluding input/output).

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
I/O	18.8	105.5	15.7	177.5	11.3	105.8	.42	.25
Res. Avg.	271.3	910.1	162.0	477.1	112.0	270.4	.61	.84
Bound. C's	26.6	65.6	14.5	38.3	7.4	19.0	.89	.86
Enthalpy	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 5: Run times and efficiencies using the same number of processors on a 128×32 grid.

residual averaging efficiency rises from 43% to 78% on 16 processors of the iPSC/2.

Table 2 shows the run times for our final version with all the modifications mentioned in this section. We should note that input/output which we thought would degrade the parallel performance, did not. The parallel efficiency for the input/output time is quite poor, however the percentage of the total run time is still insignificant on the 16 node system. The main figure to to notice is the overall improvement from the original version shown in table 1. In particular, the total run time dropped from 2767 seconds to 1725 seconds on the 16 node system. This corresponds to a rise in efficiency from 57% to 88%. Unfortunately, this implies that it is important to consider the computer architecture when designing and implementing even the small sub-sections of an algorithm.

5 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 these existing machines as well as to lead into the next section which discusses the potential capabilities of hypercube machines.

The overall machine performance and efficiency is dependent on the number of processors and the size of the grids. Thus to quantify the performance, we must study the execution times obtained with a variety of grid sizes and over a range of processors. Unfortunately due to memory limitations on the NCUBE, we are somewhat restricted in the cases that we consider here. For the most part the grids used for the timings are realistic and practical for this fluid problem.

We first consider the run times and efficiencies of the Intel iPSC/2 and the NCUBE systems. Tables 3 and 4 show both the run time and efficiency of the parallel FLO52 code (excluding input/output) 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 a given problem). Unfortunately, the input/output operations on the NCUBE so severely lengthened the total run time that we excluded them from these tables. The Intel iPSC/2 yields a speedup of 15.0 running from 1 to 16 processors, and the NCUBE yields a speedup of 10.1 running from 32 to 512 processors on the 256×128 grid. The decrease in efficiency for the larger grids on the NCUBE is actually caused by the input/output stage (even though the times for the input/output routines are omitted). This is because some processors are waiting in their computation stage to receive messages from processors that are still performing input/output operations. In general, on the Intel iPSC/2 the efficiency of the FLO52 code is greater than 85% when there are at least 1024 grid points per processor. We estimate that the NCUBE requires about 1/3 as many points per processor as the iPSC/2 to exceed 85% efficiency. Furthermore, one processor of the Cray X-MP can perform the same calculation in 169 seconds as compared with 454 seconds on a 512 NCUBE system and 3216 seconds on a 16 node iPSC/2 (excluding input/output). While neither of these systems outperformed a Cray X-MP, the NCUBE is capable of computing within a factor of 3 of the Cray for this code.

It is difficult to compare the performance of the NCUBE and Intel machines as they are configured with different numbers of processors. Table 5 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 by a factor of 2.8. Based on 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.

It is unfair to conclude on the basis of Table 5 that the IPSC/2 is a faster machine than the NCUBE. While the computation and communication on the iPSC/2 are faster than the NCUBE, NCUBE systems are typically configured with more processors than iPSC/2 systems. We omit the detailed timings and simply state that from our results, it takes about 3.3 times more NCUBE processors to achieve the same run time as the Intel iPSC/2.

6 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.

The FLO52 execution time model is created using the Mathematica symbolic manipulation package [16]. 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 associated with 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 the number of numerical operations averaged over all processors. Values for operation and communication costs are based on separate timing tests.

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.

$$\begin{aligned} T(\alpha,\beta,N,ops,P) \approx \\ & ops[6663 + 479636\sqrt{\tfrac{N}{P}} + 297036\tfrac{N}{P}] \\ & +\alpha[142563 + \log_2(P)]+ \\ & \beta[34328 + 366391\sqrt{\tfrac{N}{P}} + 3080.2\log_2(P)] \end{aligned} \tag{8}$$

where ops is the number of floating point operations, α is the startup cost to send a message, and β is the communication rate. 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 and use N to denote the total number of grid points. Additionally, the assumption that $N \geq 16P$ is made (only for the asymptotic analysis). This avoids the situation where the number of processors exceeds the number of grid points on the coarsest grid. In this case, there will be many idle processors when computing on the coarser grids.

Equation (8) reveals a number of interesting aspects about the run time. One important property of this code is that asymptotically it attains a perfect speedup. Specifically if we fix P, and let N approach infinity, then T becomes proportional to $\frac{N}{P}$. This implies that the efficiency approaches one. Equation (9) extracts the basic elements from (8) that govern the behavior of the execution time:

$$\begin{aligned} T(cc,N,ops,P) \approx cc[d_1 + d_2\sqrt{\tfrac{N}{P}} + \log_2(P)]+ \\ ops[d_3 + d_4\sqrt{\tfrac{N}{P}} + d_5\tfrac{N}{P}]. \end{aligned} \tag{9}$$

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 (9) 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 term is produced by the global communication operations necessary for norm calculations. For modest sizes of P ($P < 10000$), 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.

Figure 2: Comparison of model's predicted time and run times obtained on the NCUBE for a 256×128 grid.

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 worthwhile mentioning that our initial comparisons resulted in substantial improvements in the program as the model revealed inefficiencies in the original coding. Figures 2 and 3 illustrate the close correlation between the model's predicted time (solid lines) and the actual run times (dots). Additional comparisons (not shown here) between the predicted and actual run times of the major FLO52 subroutines lead us to conclude that the model accurately reflects the behavior of the code.

In evaluating the NCUBE and iPSC/2 as supercomputers, it seems appropriate to use the model to compare the run times with that of a Cray X-MP. In particular, we consider NCUBE and iPSC/2 systems with many processors. Figure 4 plot the predicted run time for the iPSC/2 as a

Figure 3: Comparison of model's predicted time and run times obtained on the iPSC/2 for a 256 × 128 grid.

Figure 4: Predicted run time vs no. of processors for Intel (solid) and NCUBE (dashed).

function of available processors for a 256×128 grid. As expected, the run time decreases when more processors are used. Theoretically, the iPSC/2 with 700 processors and the NCUBE with 1500 processors exceed the performance of a Cray X-MP. The main reason for this disappointing result is that the efficiencies are extremely low with many processors. In the 2000 processor range for the NCUBE, almost all the time is spent on communication. Perhaps, however, it is not logical to just extrapolate the number of processors. We can also consider increases in communication speeds and computation rates. Specifically, what values must the machine parameters (computation rate, communication speed, and number of processors) take to achieve approximately the same run time as the Cray X-MP for our 256 × 128 grid problem. Of course, there is a whole three-dimensional space of parameters that satisfies the above criteria. From this space we give an example for the iPSC/2 and the NCUBE/ten. A 64 node iPSC/2 can solve the 256 × 128 problem in approximately the same time as one processor of a Cray X-MP if the communication speeds of the iPSC/2 are twice as fast and its computation rate 5 times faster. The 512 node NCUBE can also compete with a Cray on this problem if its communication speeds are 5 times faster and its computation rate is twice as fast. Considering the somewhat primitive state of these machines, these improvements in computation and communication are not unreasonable.

7 Summary

Based on our FLO52 experience, it is clear that hypercube machines can supply the high flop rates necessary for CFD applications. Unfortunately, this performance comes with considerable programming inconvenience. While some of this is due to the complexity of distributed memory parallel programming, the primitive environments of the current machines make this task considerably more difficult.

Parallelization of FLO52 (based on domain decomposition) is straight forward. In fact, the main body of the code closely resembles the serial version with the exception of the residual averaging. Nevertheless, a substantial effort was required to produce working implementations. Much of this time was spent debugging, and coding input/output. Additional time was lost due to peculiarities of the operating systems (less than robust compilers, frequent crashing of machine, etc.). Still more time had to be spent optimizing the code for the machine to obtain efficient utilization. Even the conversion from the iPSC to the NCUBE was time consuming.

While it is difficult to produce an efficiently running code on these hypercubes, the overall computing performance is promising. In our experiments, the 16 node iPSC/2 runs within a factor of 20 of a single processor Cray X-MP and the 512 node NCUBE within a factor of 3. Additionally, with the help of a timing model, we predict the performance of future hypercube systems. Specifically, a 64 node iPSC/2 machine with 5 times faster computation and 2 times faster communication will yield similar performance to a one processor Cray X-MP. A 512 node NCUBE system with 5 times faster communication

and 2 times faster computation will also produce similar performance to the Cray X-MP.

References

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

[2] 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.

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

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

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

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

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

[8] T. Chan, Y. Saad, and M. Schultz, *Solving Elliptic Partial Differential Equations on Hypercubes*, *Proceedings of the First Conference on Hypercube Multiprocessors*, Knoxville, TN, August 1985, M. Heath (ed), SIAM, Philadelphia, 1986, pp. 196-210.

[9] J. Gustafson, G. Montry, and R. Benner, *Development of Parallel Methods for a 1024-Processor Hypercube*, *SIAM Journal on Scientific and Statistical Computing*, Vol. 9, No. 4, July 1988, pp. 609-638.

[10] Personal Communication with Mike Heath, Oakridge National Lab, 1988.

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

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

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

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

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

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