UCLA
COMPUTATIONAL AND APPLIED MATHEMATICS

# Performance of an Euler Code on Hypercubes

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

March 1989

CAM Report 89-12

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

# 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.

| *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.

in this section are given in seconds and obtained by executing with the same input parameters on the 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 $256 \times 64$ 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

| *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.

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 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 $256 \times 128$ 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.

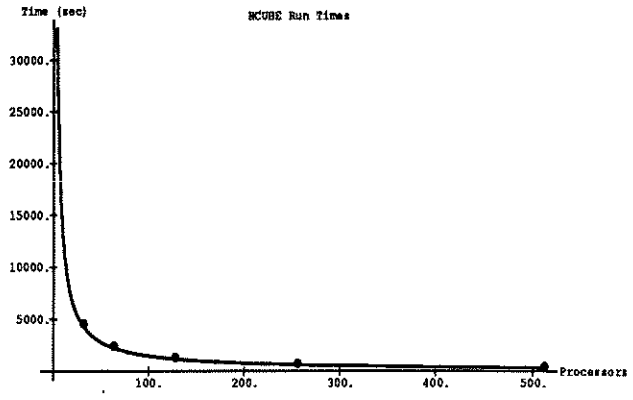


Figure 2: Comparison of model's predicted time and run times obtained on the NCUBE for a 256 × 128 grid.

## 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, $\alpha$ is the startup cost to send a message, and $\beta$ 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 ($\alpha$ and $\beta$ ) 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.

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.

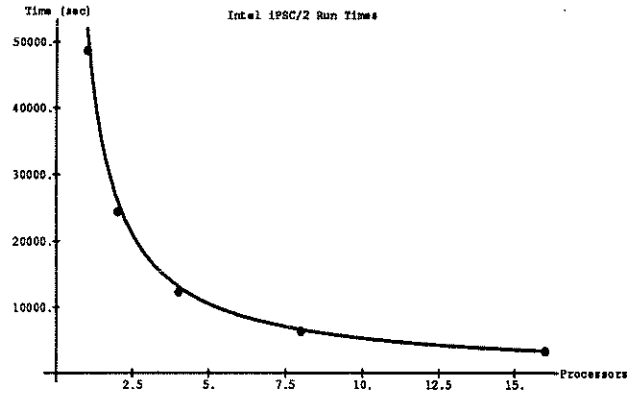


Figure 3: Comparison of model's predicted time and run times obtained on the iPSC/2 for a 256 × 128 grid.

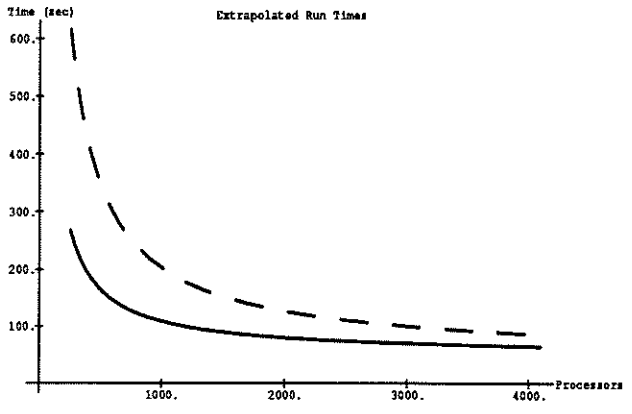


Figure 4: Predicted run time vs no. of processors for Intel (solid) and NCUBE (dashed).

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 function of available processors for a $256 \times 128$ 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 $256 \times 128$ 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 $256 \times 128$ 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 pre-

dict 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.