Computing and Informatics

Technical Report
DL-TR-96-002

Towards Direct Simulation of Turbulent Combustion on the Cray T3D - Initial Thoughts and Impressions

D R Emerson and R S Cant

April 1996

COUNCIL FOR THE CENTRAL LABORATORY OF THE RESEARCH COUNCILS

Enquiries about copyright, reproduction and requests for additional copies of this report should be addressed to:

The Central Laboratory of the Research Councils
Chadwick Library
Daresbury Laboratory
Daresbury
Warrington
Cheshire
WA4 4AD
Tel: 01925 603397 Fax: 01925 603195
E-mail library@dl.ac.uk

ISSN 1362-0207

Towards Direct Simulation of Turbulent Combustion on the Cray T3D - Initial Thoughts and Impressions

D. R. Emerson[1] and R. S. Cant[2]

Abstract

This paper provides an initial assessment of a Cray T3D MPP system for an engineering application. A comparison of the performance of a Direct Numerical Simulation code on the Cray T3D and the Intel iPSC/860 is presented. The floating point performance of standard Fortran 77 indicates that there is room for improvement and details concerning necessary optimisations to improve the floating point performance of a conjugate gradient solver are given. The lack of a secondary cache clearly affects the cache efficiency when preconditioning is applied. In comparison to the Intel iPSC/860, the CRAY T3D shows a substantial improvement in performance and also offers a low-latency, high-bandwidth communications network.

---

1 D. R. Emerson is a Senior Scientific Officer at E.P.S.R.C., Daresbury Laboratory, Warrington WA4 4AD, U.K.

2 R. S. Cant is a lecturer in the Engineering Department, Cambridge, U.K.

Introduction

The dramatic rise in the peak floating point performance of "off-the-shelf" microprocessors based upon Reduced Instruction Set Computer (RISC) technology has recently led to many vendors offering Massively Parallel Processing (MPP) systems. In particular, after almost 20 years of providing supercomputers based upon tightly coupled, shared memory vector processors, Cray have entered the distributed memory computing market with the T3D. This is their "first generation" offering and is based upon the DEC Alpha microprocessor operating at 150 MHz. The Cray T3D has been designed to scale up to 2048 Processing Elements (PEs) and therefore has a peak performance of 300 Gflop/s. Cray have also set themselves an ambitious target of 1 Tflop/s peak performance by 1995/96 time frame and are looking to future advances in technology to provide a Tflop/s sustained performance by 1997/98 on real applications [1]. Whilst the macroarchitecture is to remain unchanged, the microarchitecture will embrace any developments that may occur.

The UK's T3D employs the DEC Alpha 21064 (EV4) microprocessor operating at 150 MHz and, when first installed, had 128 nodes each with 2 PEs, giving a total of 256 PEs. More recently, it has been upgraded to 256 nodes (512 PEs). It is IEEE compliant and supports 64–bit integer and 64–bit floating point arithmetic and by utilising advanced RISC technology, it can perform one floating point operation in one clock cycle. Each processor can therefore theoretically deliver 150 Mflop/s, making the peak performance of the machine 38.4 Gflop/s. Each DEC Alpha 21064 has 64 MBytes of local memory, giving a total memory for this configuration in excess of 16 GBytes. It has an 8 KByte data cache and an 8 KByte instruction cache but there is no secondary cache provided. The Cray T3D network topology has been designed as a 3D torus. In effect, it is a 3D grid with wrap-around connections. Although the architecture is based on MIMD technology, the memory is globally addressable

and each processor can have access to the memory of any other processor, although there is still some overhead associated with such communications. In principle, the Cray T3D macroarchitecture could therefore be viewed as a single virtual memory. However, the current mode of operation uses the Single Process Multiple Data (SPMD) style of programming and message passing is typically done using PVM although lower-level routines (with a correspondingly lower latency) can be employed. A high speed interconnection network, which supports bidirectional communications, can transfer data at up to 300 MBytes/s [1].

Running concurrently with the MPP procurement was the establishment of the UK High Performance Computing Initiative (UKHPCI). This was to put in place the foundations for exploiting the Cray T3D by the scientific and engineering community involved in large scale computing or the often referred to "Grand Challenge" areas of computing. One of these designated areas is combustion and, in particular, the Direct Numerical Simulation of turbulent Combustion (DNSC). In general, the procurement of this new machine is providing a fresh impetus for moving real engineering applications onto supercomputing systems and for moving onto parallel systems in general. The program developed for the DNSC calculations is known as ANGUS and was originally written by Stewart Cant. The parallel version of the code was developed by David Emerson to run on the 64 node Intel iPSC/860 hypercube at Daresbury Laboratory. Each i860 node is theoretically capable of delivering 60 Mflop/s and has 16 MBytes of local memory, giving a peak performance of 3.84 Gflop/s and a total memory of 1 GByte. However, one of the major underlying reasons for its parallel development was the knowledge that a new MPP system was to be procured which would be offering substantially more memory and computing power. The Cray T3D provides this necessary increase in resources offering an order of magnitude increase in both memory and peak performance. This paper therefore looks at the early performance of the T3D with a real application and compares it to the Intel iPSC/860 which, until recently, was

one the largest distributed memory MIMD machine available within the UK. All results were obtained in late 1994 and, although several upgrades have since taken place, none of these changes have affected the performances reported herein.

Motivation and Background for DNSC Calculations

Combustion in all its forms accounts for an overwhelmingly large proportion of the World's primary energy production. No other energy technology offers the convenience and flexibility of combustion in producing large amounts of energy exactly where and when it is needed. Modern combustion systems can be made highly efficient in terms of energy extraction from the primary fuel, but there is much less certainty about designing combustion systems which minimise the emission of harmful pollutants. Concern over emissions, and particularly over those from motor vehicle exhausts, has led to ever more stringent restrictions on allowable emission levels. Combustion engineers are finding that the time-honoured process of incremental design is no longer sufficient and a more fundamentally-based approach is necessary.

Many designers of combustion systems ranging from simple domestic gas burners through spark-ignition petrol engines to power-station boilers now have access to computer-simulation packages based on the principles of Computational Fluid Dynamics (CFD). These have proved immensely useful in visualising the reacting flow field in various combustion-system geometries. Nevertheless, the flow field in practical combustors is inevitably turbulent, and it is necessary to devise mathematical models to represent the processes of turbulent transport, reaction and heat release. The need for such models arises from the nature of turbulence: a turbulent flow contains three-dimensional motion on all length scales from the largest, constrained only by the geometry of the system, down to the smallest, which is controlled by the viscosity of the fluid. Even in a small petrol engine the range of scales can be two or three orders of magnitude and this means that the resolution necessary to

compute such flows in detail is well beyond the reach of present-day computers. Therefore, current practice is to make use of equations which have been averaged to eliminate any small-scale activity. Information lost in the averaging process is restored through the use of turbulence and combustion modelling. Clearly, the construction and validation of such models requires data from some outside source. Traditionally this has been the role of experiment, but there are many quantities of interest in turbulent combustion which are simply not accessible to current experimental techniques despite recent progress in laser diagnostic methods. Fortunately, recent advances in computer power have made it possible to compute simple flames in highly idealised geometries without recourse to modelling. This is Direct Numerical Simulation (DNS), and the technique has proved very useful over a number of years in non-reacting turbulent flow. Compared with experiment, DNS suffers from severe limitations of geometry and problem size, but the major advantage of DNS over experiment is that all quantities of interest are accessible.

Description of DNSC Program

The purpose of the present project is to carry out DNS of turbulent premixed combustion in order to generate statistical data in support of modelling. The equations to be solved are the Navier-Stokes equations for fluid flow, augmented by two additional equations each describing the transport of a single scalar variable and together specifying the thermochemical state of the system in the presence of differential diffusion effects. Thus, in total there are six partial differential equations to be solved: a continuity equation;

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho u_k}{\partial x_k} = 0 \tag{1}$$

three momentum equations which, using tensor notation can be compactly written as follows;

$$\frac{\partial \rho u_i}{\partial t} + \frac{\partial}{\partial x_k}(\rho u_k u_i) = -\frac{\partial p}{\partial x_i} + \frac{1}{Re}\frac{\partial}{\partial x_k}\left[\frac{\partial u_i}{\partial x_k} + \frac{\partial u_k}{\partial x_i} - \frac{2}{3}\delta_{ik}\frac{\partial u_j}{\partial x_j}\right] \tag{2}$$

a scalar equation representing temperature (T);

$$\frac{\partial \rho T}{\partial t} + \frac{\partial}{\partial x_k}(\rho u_k T) = \frac{1}{RePr}\frac{\partial}{\partial x_k}\left\{\frac{\partial T}{\partial x_k}\right\} + HW_c \tag{3}$$

and a scalar equation representing a reaction progress variable (c);

$$\frac{\partial \rho c}{\partial t} + \frac{\partial}{\partial x_k}(\rho u_k c) = \frac{1}{RePrLe}\frac{\partial}{\partial x_k}\left\{\frac{\partial c}{\partial x_k}\right\} + W_c \tag{4}$$

These equations have been non-dimensionalised with respect to characteristic length and time scales, characteristic density and pressure, and the unburned to adiabatic temperature difference. The three non-dimensional numbers which appear are the Reynolds number, Re, the Prandtl number, Pr and the Lewis number, Le, which are given by:

$$Re = \frac{\rho_r u_r L}{\mu_r} \;\; ; \;\; Pr = \frac{\mu_r c_p}{\lambda} \;\; ; \;\; Le = \frac{\lambda}{\rho_r D c_p} \tag{5}$$

where the subscript *r* represents some reference condition. A low Mach number approximation is made in order to decouple the density (ρ) from the pressure (p) and thus avoid the need to resolve acoustic length and time scales in the simulation. Thus the ideal gas equation of state in non-dimensional form becomes:

$$\rho = \left[1 + \frac{\alpha}{1-\alpha}T\right]^{-1} \tag{6}$$

Note that no turbulence modelling assumptions are made anywhere in this set of equations, which remain exact in both laminar and turbulent flow at low Mach number.

In order to simulate non-reacting flow it is sufficient to solve only the first four of the above differential equations, with the density assumed constant. In order to simulate combustion it is necessary to solve at least the first five differential equations with a specified chemical reaction rate. In order to take account of unequal diffusion of heat and mass all six differential equations must be solved with the Lewis number Le specified different from unity. A great deal of useful data has been obtained from combustion simulations carried

out in this manner at constant density, but for full realism the density (and the molecular transport properties) must be allowed to vary with the temperature.

It is necessary to specify the chemical reaction rate which arises in both of the scalar equations. Realistic chemical reaction mechanisms are extremely long and complicated and their numerical analysis is a major task in itself. At this point, some approximation is inevitable, since limitations of computer capacity will not allow the treatment of both turbulence and chemistry in full detail. The purpose of the present investigation is to examine the fluid-mechanical implications of combustion in a turbulent environment, and the inner workings of the chemistry are of secondary importance. Thus, the preferred approach is to assume that the chemical reaction of interest is a one-step irreversible reaction governed by Arrhenius kinetics. The rate expression is then:

$$W_c = B\rho(1-c)exp\left[-\frac{\beta(1-T)}{1-\alpha(1-T)}\right] \tag{7}$$

where the exponential dependence of the reaction rate on temperature provides for realistic behaviour of the flame.

Discretisation of the equations is carried out using standard second-order central differences on a three-dimensional grid of unit cells. The velocity nodes are located at the face-centres of each cell, giving a staggered-grid arrangement that conserves kinetic energy as well as mass and momentum [2]. This is a low-order discretisation by comparison with much previous DNS work where pseudo-spectral or high-order finite-difference methods have been used [3,4]. However, the use of low-order discretisation in the present work is justified physically by the high level of natural diffusion in the combustion problem. Time advancement is by a modified explicit second-order Adams-Bashforth procedure incorporating the solution of a Poisson equation for the pressure.

The usefulness of any direct numerical simulation depends *inter alia* on the turbulence Reynolds number that can be attained. In the present case the limit of resolution is set by

the need to maintain an adequate representation of a laminar flame. A minimum of about ten computational cells is required, and in order to remain in a regime of combustion of technological interest it is necessary to set the size of the smallest scale of turbulence to be comparable with this figure. It can be shown that the attainable Reynolds number is given by:

$$Re = \left(\frac{N}{n}\right)^{4/3} \tag{8}$$

where N is the size of one side of the computational domain and n is the required size of the smallest scale of turbulence, both expressed as a number of computational cells. Thus, at a small-scale resolution of ten cells the Reynolds number is about 30 for a grid size of 128^3. This represents the current state-of-the-art in combustion DNS. Increased grid size increases the Reynolds number to about 70 for a grid size of 256^3 and about 120 for the maximum grid size of 368^3 which appears to be achievable on the T3D. The last figure is becoming comparable with the turbulence Reynolds numbers attained in simple laboratory burner experiments.

The class of problem to be addressed using the DNSC concerns turbulent premixed flame propagation into an oncoming stream of reactants. A planar laminar flame is positioned near the centre of the computational domain perpendicular to the streamwise direction. It is supplied with fresh reactants by means of a turbulent inflow condition and burned products are exhausted by means of an outflow boundary condition. The spanwise boundary conditions are periodic. This problem class offers the possibility of achieving statistical stationarity and of collecting genuinely time-averaged statistics. For development purposes a simpler problem will be tackled involving two laminar flames initialised back-to-back. This problem is time-dependent, but avoids the need to specify a turbulent inflow condition and yields double the statistical sample size for flame-related quantities [5,6].

The pressure solver utilises a conjugate gradient method with a Modified ILU (MILU) preconditioner [7]. As with many CFD algorithms, the resulting matrix, A, is both sparse

and symmetric. In this case, A is heptadiagonal and the periodic boundary conditions also mean that the matrix is singular. The pseudo-code can be written as:

$$
\begin{aligned}
& x_0 = 0 \; ; \; r_0 = b - Ax_0 \\
& p_{-1} = 0 \; ; \; \beta_{-1} = 0 \\
& solve \;\; \omega_0 \;\; from \;\; P\omega_0 = r_0 \\
& \rho_0 = (r_0, w_0) \\
& For \;\; i = 0, 1, 2, ... \\
& \quad p_i = \omega_i + \beta_{i-1} p_{i-1} \\
& \quad q_i = Ap_i \\
& \quad \alpha_i = \frac{\rho_i}{(p_i, q_i)} \\
& \quad x_{i+1} = x_i + \alpha_i p_i \\
& \quad r_{i+1} = r_i - \alpha_i q_i \\
& \quad If \;\; ||r_{i+1}||_2 < \varepsilon \;\; stop \\
& \quad \omega_{i+1} = P^{-1} r_{i+1} \\
& \quad \rho_{i+1} = (r_{i+1}, \omega_{i+1}) \\
& \quad \beta_i = \frac{\rho_{i+1}}{\rho_i} \\
& end
\end{aligned}
\tag{9}
$$

where P represents the preconditioning matrix. If P = I, the identity matrix, there is no preconditioning, but in general, P represents some approximation to the inverse of A. In this case, the MILU preconditioner is used and the diagonal is evaluated only once at the initialisation phase of the program. The parallel preconditioning is done locally (there is no fill in).

Parallel Implementation of ANGUS

The grid partitioning strategy employed for ANGUS is quite typical of many domain decomposition techniques used in parallel CFD. Due to the finite difference stencil it is necessary to introduce "halo" or "ghost" cells at the interface boundaries. This then allows derivatives to be determined at these regions. The halo cells are then updated as required and this is illustrated in figure 1. The difference stencil employed in ANGUS necessitates the inclusion of only one halo cell. In other applications, it may be necessary to accommodate two halo cells. The data transfer routines used are written in a general way and this can be accomplished by simply setting a parameter NHALO = 1 or 2.

The code uses a staggered grid approach, with the pressure located at the cell centre. Due to this staggering, it is necessary to transfer data from diagonally adjacent neighbours as illustrated in figure 1. This can be done in two ways: (i) by transferring a single element to the point where it is required; or (ii) by utilising the fact that the data has already been transferred to the domain's nearest neighbour. In the first approach, the logic can get considerably more involved whereas in the latter approach it necessitates only the inclusion of two additional halo cells in the data transfer. This contributes only a minor additional overhead and is far easier to implement and maintain. Figure 1 also shows the typical communications required between processors, including the periodic data transfer.

Message passing was initially done using standard PVM. However, whilst PVM is good for general purpose computing and portability, there are several other options on the T3D which give enhanced performance and reduced latency. The PVMFPSEND and PVMFPRECV routines in release 3.3 combine the packing and sending of data, something which actually took three calls previously. This, of course, presumes the data is already packed and ready to send. This was already the case with ANGUS because the Intel does not have a packing routine and to minimise the number of calls to CSEND, it can be

advantageous to pack several arrays into a long vector and then transfer the data. The latency overhead is therefore only incurred once at the expense of taking slightly longer to deliver the message. The modification of the data transfer routines to use the PSEND/PRECV calls was therefore trivial. Figures which indicate the relative performance of the different message passing facilities on the Cray T3D are given in table 1. These values were obtained with a simple pingpong style test and differ slightly from those quoted by Oed [1] and the Intel results are taken from [8].

Table 1

Relative performance of message passing on the Cray T3D

Library	Latency (μs)	Bandwidth (MBytes/s)
PVM	50 - 70	40 - 60
PSEND/PRECV[3]	32 (< 4096 Bytes)	34
	222 (> 4096 Bytes)	26
shmem_put	6[a]	120
shmem_get	6[a]	60
Intel	76 (< 100 Bytes)	2.4
	205 (> 100 Bytes)	2.8

(a) Cray have actually obtained latencies of about 1 – 2 μs [1].

As can be seen from Table 1, there is a much lower latency involved with the new routines. One of the disadvantages of going to the lower level routines, apart from some nontrivial difficulties in implementing them, is that portability is lost. Another feature of using the low level routines is that the data must be in a static memory location. It is therefore necessary to define a common storage area in the relevant routine. Another problem, in particular with the shmem_put routine, is that cache coherency is not maintained. This places an onus on the programmer to ensure that the program is synchronised at the appropriate point and that the cache is flushed to avoid overwriting a recently updated variable with old data in cache. A

[3] These are the values obtained with PVM_DATA_MAX = 4096 Bytes

simple test was done without the cache flushing routine and a latency of 2 μs was obtained. The results in Table 1 were obtained by using the *shmem_udcflush()* routine to flush the cache but a faster routine can be employed whereby only a cache line is flushed. However, in practice, this was generally found to be less reliable that flushing the entire cache.

Early Results and Necessary Optimisations

From the outset, Cray have provided some extremely useful tools for analysing a code's performance. Extensive use has been made of *Apprentice* for profiling and measuring the performance of ANGUS. From timings on the Intel, it was already known that the pressure solver was the most compute-intensive part of the calculation. Indeed, the pressure solver takes in excess of 95% of the computing time at each time step. This was confirmed with Apprentice and it also indicated the relatively low floating point performance of the routine, which was coded in standard Fortran 77. As previously discussed, the pressure solver uses a preconditioned conjugate gradient method which, apart from the preconditioner, consists of essentially dot products and saxpy-type operations. It is straightforward to code and has a unit stride, which should provide good access to data in cache. However, the relatively small cache, and the lack of a secondary cache, appear to play a significant part in the relatively poor floating point performance of standard Fortran 77. This feature appears to be typical in RISC based microprocessors [9] and is, in part, related to the small cache but it is also compounded by the fact that access to main memory takes many additional clock cycles; typically factors of 3 – 5 are involved. In this particular case, the absence of the secondary cache may also be important.

Due to the domination of the pressure routine, early efforts have concentrated on trying to optimise that section of the code. As previously stated, the conjugate gradient method used is dominated by dot products and saxpy-type operations and these are therefore ideal candidates for incorporating into the Basic Linear Algebra Subroutines (BLAS). In the spirit

of the correct reporting of results advocated by David Bailey [10,11], I will provide sufficient information for the reader to understand what was involved. The time spent in converting the existing code to utilise the Level 1 BLAS routines, wherever they could be incorporated, was a few hours, including all relevant testing of each routine. In other words, it was a straightforward exercise. In one section it was necessary to introduce an additional routine which would zero all the halo elements to allow a dot product to be performed over all computational cells (including the halo region). The performance of the saxpy routine was measured on the Cray T3D and a comparison with the Intel is also included in figures 2 and 3. At the time of writing, the Cray T3D was using UNICOS-MAX 1.1.0.2 and the cf77_6.1 compiler. The Intel was using the Portland Group compiler (if77 with the –O4 –Mvect option) and running NX/2. The Intel BLAS routines are provided by Kuck and Associates.

The code fragment used to generate these results was:

```
parameter (imax=32,alpha=2.0)
dimension x(imax),y(imax)


t1 = rtc()
call saxpy(imax,alpha,x,1,y,1)
t2 = rtc()
```

The value of imax was varied from 2^5 (32 elements) to 2^{19} (524288 elements), the maximum memory available on the Intel. For the Intel, all variables were made 64–bit and the saxpy routine was correspondingly replaced by a daxpy routine. The call to the timing routine made use of the Real Time Clock (RTC) on the Cray T3D and DCLOCK on the Intel. The overhead associated with the appropriate clock call was also taken into account. The number of floating point operations required for a saxpy operation is two and the total number of

flop/s can therefore be determined from:

$$r = \frac{2 * imax}{time} \tag{10}$$

In standard Fortran 77, short vector lengths, which can be contained entirely within the cache, perform very well. However, the limited (and relatively immature) compiler technology on the T3D shows that the performance drops off to approximately 8 Mflop/s when the vector length begins to exceed the cache size. This performance can be improved by performing some loop unrolling but this was considered unnecessary for these tests. The benefit of converting the Fortran code to BLAS is clearly demonstrated in figure 3 where the saxpy loop asymptotes to approximately 30 Mflop/s. This represents a substantial improvement over the Cray Fortran and what was possible to achieve on the Intel, where the Fortran compiler is much more mature. This indicates a potential of 7 Gflop/s can be achieved on suitably large vectors using all 256 processors of the Cray T3D. In practice, the pressure solver in ANGUS has actually achieved 7 Gflop/s on a test run using a 368^3 global grid size. This performance was measured using Apprentice and did not include the MILU preconditioner.

From figure 3, it appears that there is only minimal improvement in performance for short vector lengths in comparison to the Intel. However, for vector lengths $\geq$ 128, the floating point performance increases from approximately 10 Mflop/s, the maximum attained on the iPSC/860 (about 17% of its peak performance), to about 30 Mflop/s on the T3D (about 20% of its peak performance). We therefore obtain about a factor of three in the raw performance of this type of library routine. As previously stated, this performance can be obtained on each PE (i.e. at this stage in the compiler evolution, the BLAS routines are not parallel) and therefore represents a potential 12–fold increase in computational power over the Intel iPSC/860.

The utilisation of the BLAS incurs a startup penalty. From simple tests with the above code, a startup cost of approximately 9 μs was obtained, which is in excess of 1000 clock

cycles. However, the accuracy of this figure is unknown due to the difficulty in timing very short vector lengths and any errors that may have been introduced by the extrapolation procedure. Figure 4 shows the performance of ANGUS for a relatively small (64^3) problem plotted against $\log_2(N_p)$. It should be noted that this problem could only fit onto 2 T3D processors and the time represents how long it takes to complete five timesteps and a small amount of I/O. It does not include the initialisation phase which, for production runs, will be a very small percentage of the total execution time. This figure reveals some interesting features concerning the performance of the T3D. In both cases, the conjugate gradient routine does not particularly benefit from preconditioning, even though the number of iterations required to achieve a converged solution is reduced from approximately 280 to 180. It is a cache-based problem caused by the large strides between the required elements of the MILU preconditioner. Moreover, the vector length decreases as the number of PEs increases and this allows better cache utilisation for the Fortran coded routine. As would be expected, the startup cost incurred by calling the BLAS will also affect the performance as the vector length decreases and the performance on 16 or more processors is on a par with the standard Fortran 77 code.

How does this compare to the Intel ? As with all of these problems, it is difficult to assess because of many factors. For example, the problem described could only fit onto 2 T3D processors but needed 8 i860 processors. The Intel has global summation routines (`gdsum`) but the T3D, as yet, does not have any such routines although they are expected with the next compiler release. An indication of the relative performance of each machine on this problem is given in Table 2 but some care must be taken when interpreting these results. The BLAS performance on the Intel is not particularly effective when compared to the standard F77 results, even though this problem has a long vector length on the Intel. This is an indication of a more "mature" technology where the Mvect option allows the

use of an instruction which bypasses the cache and gives a much better performance with long vector lengths. It should also be noted that the Intel version is not identical to the CRAY version, although they are quite close. It was found for the BLAS version that loop unrolling was beneficial for the matrix-vector product on the T3D but not particularly useful on the Intel. Again, this could be an indication that the technology on the Intel is in a more mature state. Furthermore, no loop unrolling was employed for the Fortran 77 version of the conjugate gradient solver on either machine because other work, such as dot products, was carried out during the matrix-vector operation.

The results in Table 2 were obtained by performing a fixed number of iterations in the conjugate gradient solver. This was done because the number of iterations necessary to obtain a converged solution with the parallel preconditioner varied slightly at each timestep. Therefore, to give a clearer indication of where the time was being spent, the number of iterations was fixed to 300 for the solver without preconditioning and to 150 when the preconditioner was being applied. This is in broad agreement with what is observed in practice where the number of iterations are reduced by approximately a factor of two.

The detailed timing analysis of each individual loop section reveals some interesting answers. These timings represent the total times of each section after 5 complete timesteps. The higher bandwidth and significantly lower latency of the CRAY T3D gives very substantial improvement over the Intel for the global summation and message passing. The CRAY T3D summation routine used a binary tree approach with the low level shared memory facility for message passing. At the time of writing, no intrinsic routine was available and it would be reasonable to expect an improvement in these figures if an intrinsic routine were provided. As indicated previously, the poor cache utilisation of the preconditioner causes a significant degradation in performance and even though only half the number of iterations are completed, there is no real saving in elapsed time. It should be noted that even when the preconditioner

is not applied, the subroutine still performs some work in copying one array into another. On the CRAY T3D, this was done using the BLAS `scopy` routine. However, this was not found to be beneficial on the Intel and was performed using standard Fortran and this accounts for the difference in time for the preconditioning with code types 2 and 4 on the T3D.

Table 2

Performance of Intel iPSC/860 and Cray T3D[4] on 8 Processors for a 64^3 Problem

CRAY T3D	Code Type (time in seconds)			
Routine	1	2	3	4
global summation	0.2	0.3	0.2	0.3
message passing	4.0	8.0	4.0	8.0
preconditioning	23.8	2.65	23.8	5.15
q = Ap	7.86	14.66	10.38	21.0
BLAS/Fortran	12.13	24.24	21.63	44.0
Total Time	47.99	49.85	60.01	78.45
Intel iPSC/860	Code Type (time in seconds)			
Routine	1	2	3	4
global summation	10.85	20.78	10.41	20.63
message passing	38.93	77.23	39.17	78.26
preconditioning	50.94	13.62	48.64	13.60
q = Ap	43.27	86.39	25.33	50.65
BLAS/Fortran	36.87	73.67	29.37	58.64
Total Time	180.86	271.69	152.92	221.78

The preconditioner and matrix-vector product, q = Ap, are both executed in standard Fortran 77. The times for these sections in Table 2 therefore give an indication of how real sections of code perform on the two machines. If the results obtained using the code types 3 and 4 are analysed (these sections of code are identical on both machines whereas the matrix-vector loop for the BLAS routines was unrolled on the CRAY T3D) it can be seen

[4] Code Type 1 = BLAS + preconditioning, 2 = BLAS without preconditioning, 3 = F77 + preconditioning, 4 = F77 without preconditioning.

that the typical improvement in performance is about 2.5. If the BLAS/Fortran sections are compared (these sections involve the saxpy and dot product operations) it is clear that the Cray routines are significantly better. However, the Fortran performance on the Intel is better than its Level 1 BLAS and the real improvement is again about 2.5.

Table 3

Performance of the Cray T3D[5] on 64 Processors for a 128^3 Problem

CRAY T3D	Code Type (time in seconds)			
Routine	1	2	3	4
global summation	0.47	1.01	0.46	0.87
message passing	10.48	22.81	10.47	22.79
preconditioning	36.39	4.35	36.32	8.53
q = Ap	11.89	24.32	15.98	34.42
BLAS/Fortran	18.45	40.21	33.44	71.73
Total Time	77.68	92.70	96.67	138.34

Whilst the preceding analysis does give some comparison between the Intel and T3D, it is not sufficient to presume that preconditioning is not always effective. Its success can frequently depend upon the partitioning used and the problem size. Many of the problems under consideration for the T3D are far too large for the Intel and the behaviour of the solver has to be investigated to determine the optimum approach. In the foregoing example, the computational domain was partitioned into $2\times2\times2$ subdomains. Table 3 shows the performance for a problem too large for the Intel and was run on 64 PEs on the CRAY T3D. The problem size was 128^3, the smallest problem of practical interest for the DNS calculations and the computational grid was partitioned into $4\times4\times4$ subdomains. The timings are again after 5 complete timesteps. For this example, the code took, on average, 497 iterations to converge and when preconditioning was applied, this was reduced to 228

[5] See Table 2 for definition.

iterations. It is clear in this case that the type 1 approach, involving Level 1 BLAS and preconditioning, performs the best and represents a 25% reduction in time in comparison to the standard Fortran 77 coding with preconditioning.

Summary and Conclusions

An assessment of the CRAY T3D has been provided. Whilst there is room for improvement with the floating point performance of standard Fortran 77, it is possible to improve the performance by the utilisation of the Level 1 BLAS in solvers such as conjugate gradient algorithms. However, preconditioning does present a problem for efficient cache utilisation and the lack of a secondary cache clearly plays a role. In comparison to the Intel iPSC/860, the CRAY T3D shows a substantial improvement in performance. This is particularly true in the communications network where low-level routines provide a low latency and a high bandwidth. These features are clearly important for iterative schemes, such as the conjugate gradient algorithm, where global sums are performed frequently and significant amounts of interface data need to be transferred.

Acknowledgments

The authors would like to express their gratitude to EPCC support staff for their help and advice in porting ANGUS to the Cray T3D.

REFERENCES

[1] W. Oed, "The Cray Research Massively Parallel Processor System CRAY T3D", Cray Internal Publication.

[2] F. H. Harlow and J. E. Welch, Phys. Fluids 8, 2182-2189, 1965.

[3] S. A. Orszag, Phys. Fluids supplement II, 250-257, 1969.

[4] S. Lele, J. Comput. Phys. 103, 16-43, 1992.

[5] R. S. Cant, C. J. Rutland and A. Trouvé, Proc. 1990 CTR Summer Program, pp271-279, Center for Turbulence Research, Stanford University, 1990.

[6] C. J. Rutland and A. Trouvé, Combustion and Flame 94, 41-57, 1993.

[7] T. F. Chan and C-C. J. Kuo, "Parallel Elliptic Preconditioners: Fourier Analysis and Performance on the Connection Machine", Computer Physics Communications, Vol. 53, 1989, pp 237–252.

[8] J. Helin and R. Berrendorf, "Analyzing the Performance of Message Passing Hypercubes: A Study with the Intel iPSC/860", ACM International Conference on Supercomputing, Cologne, Germany, 1991.

[9] D. Bailey, "RISC Processors and Scientific Computing", NAS Technical Report RNR-93–004, 1993.

[10]D. Bailey, "Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers", Supercomputing Review, August, 1991, pp 54–55.

[11]D. Bailey, "Misleading Performance in the Supercomputing Field", Supercomputing 92 Proceedings, Published by IEEE, held at Minneapolis, November 16–20, 1992, pp 155–158.

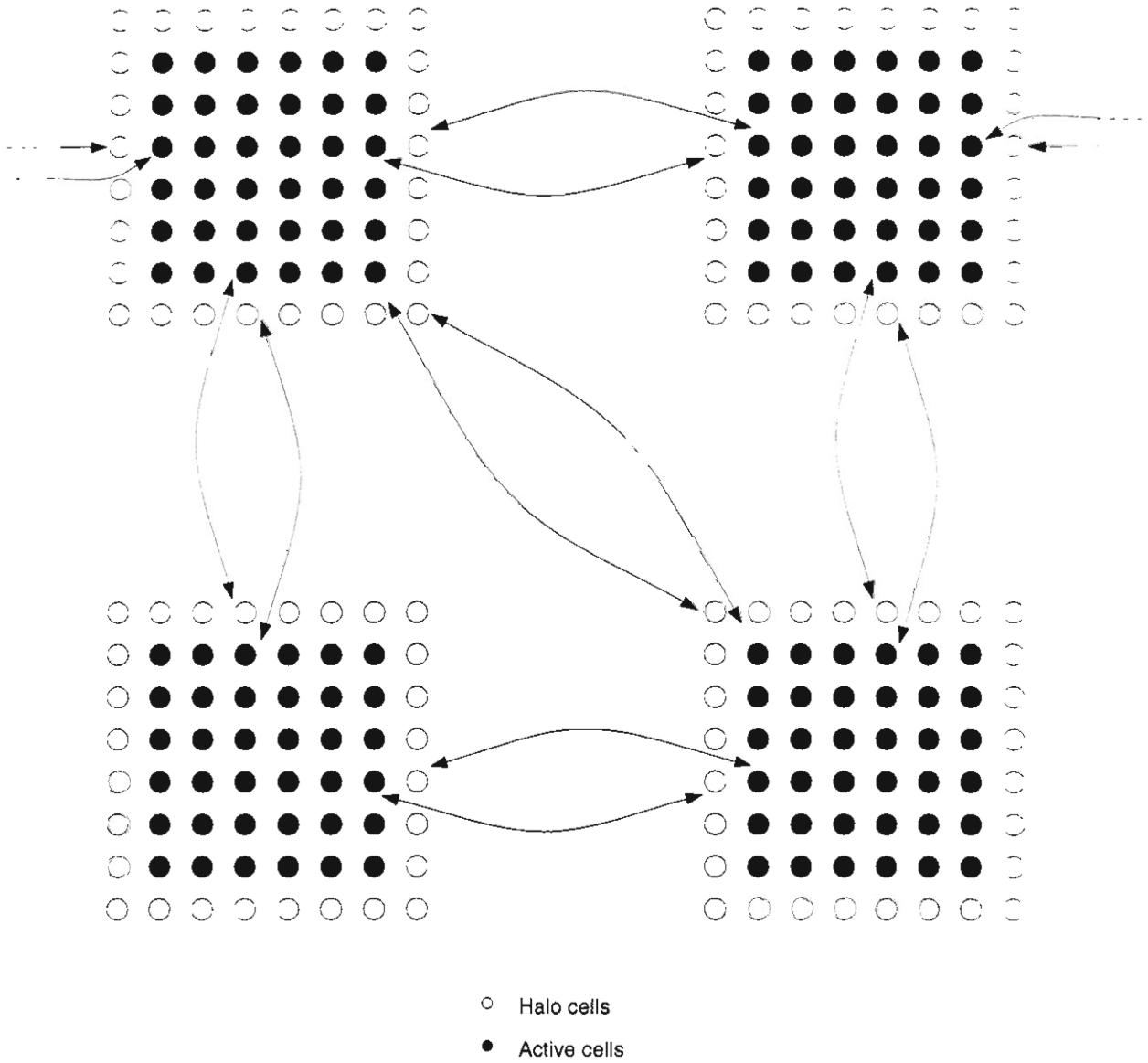

Figure 1 Communications Strategy for ANGUS

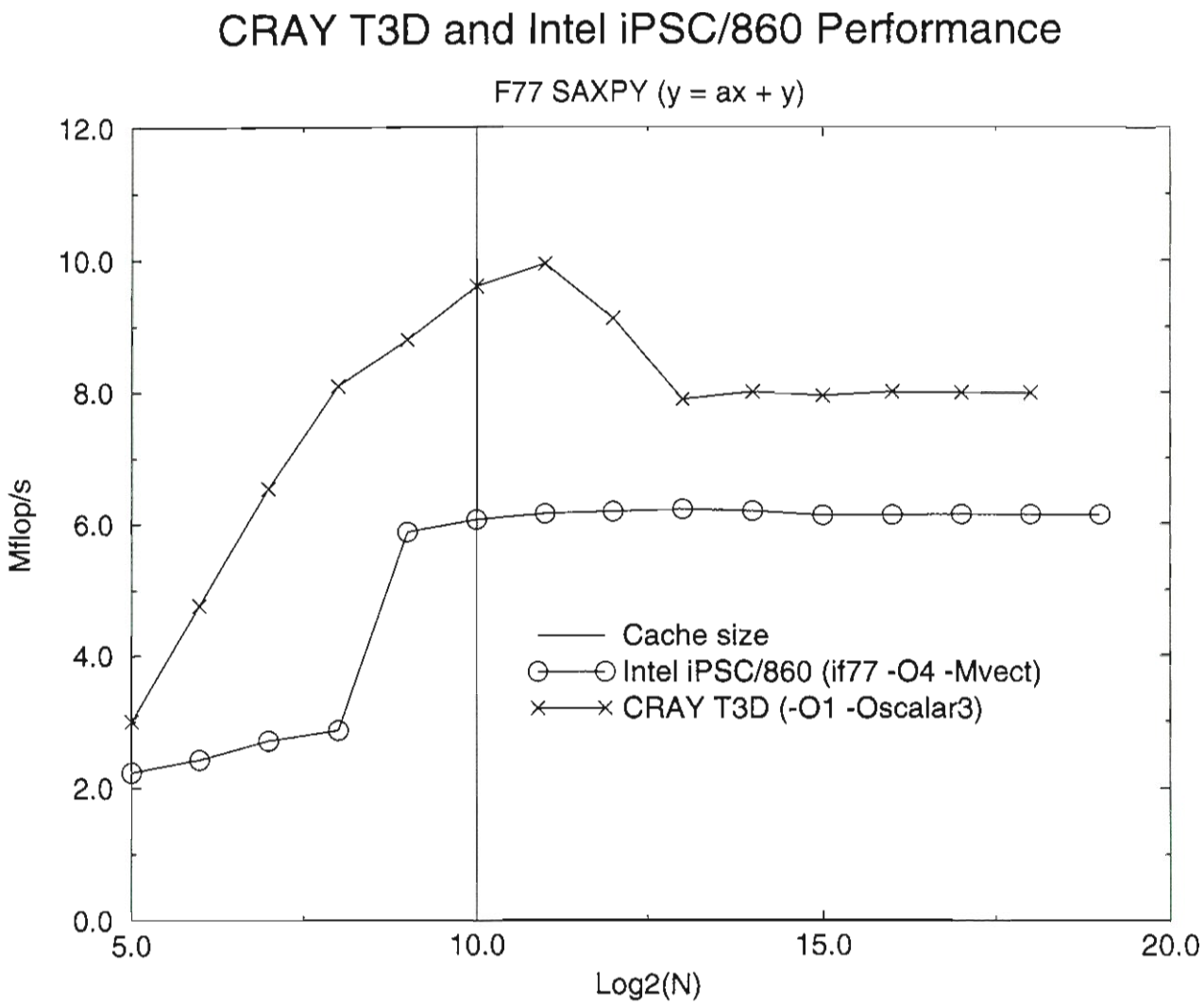

Figure 2 Fortran 77 Saxpy Performance on CRAY T3D and Intel iPSC/860

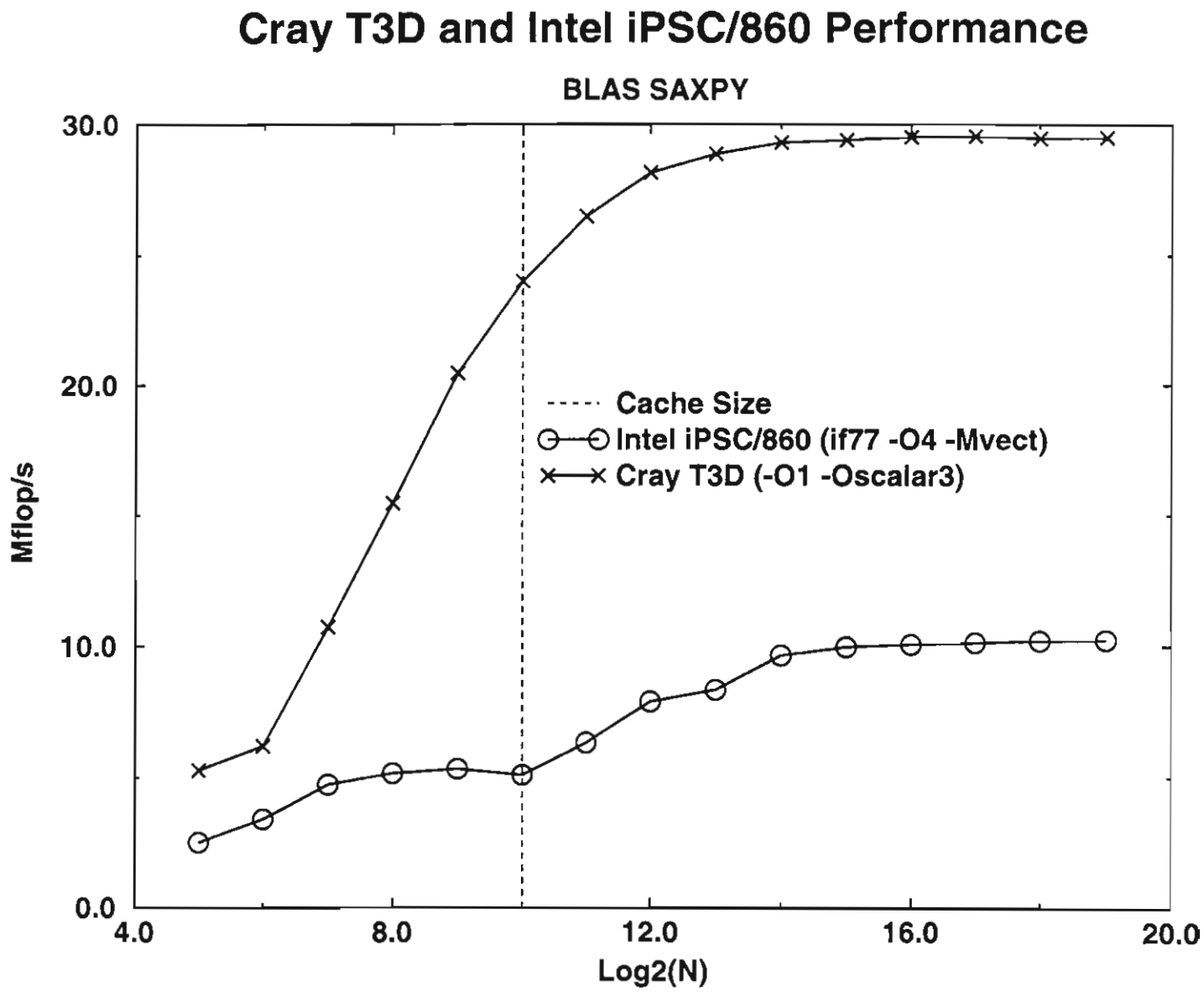

Figure 3 BLAS Saxpy Performance on CRAY T3D and Intel iPSC/860

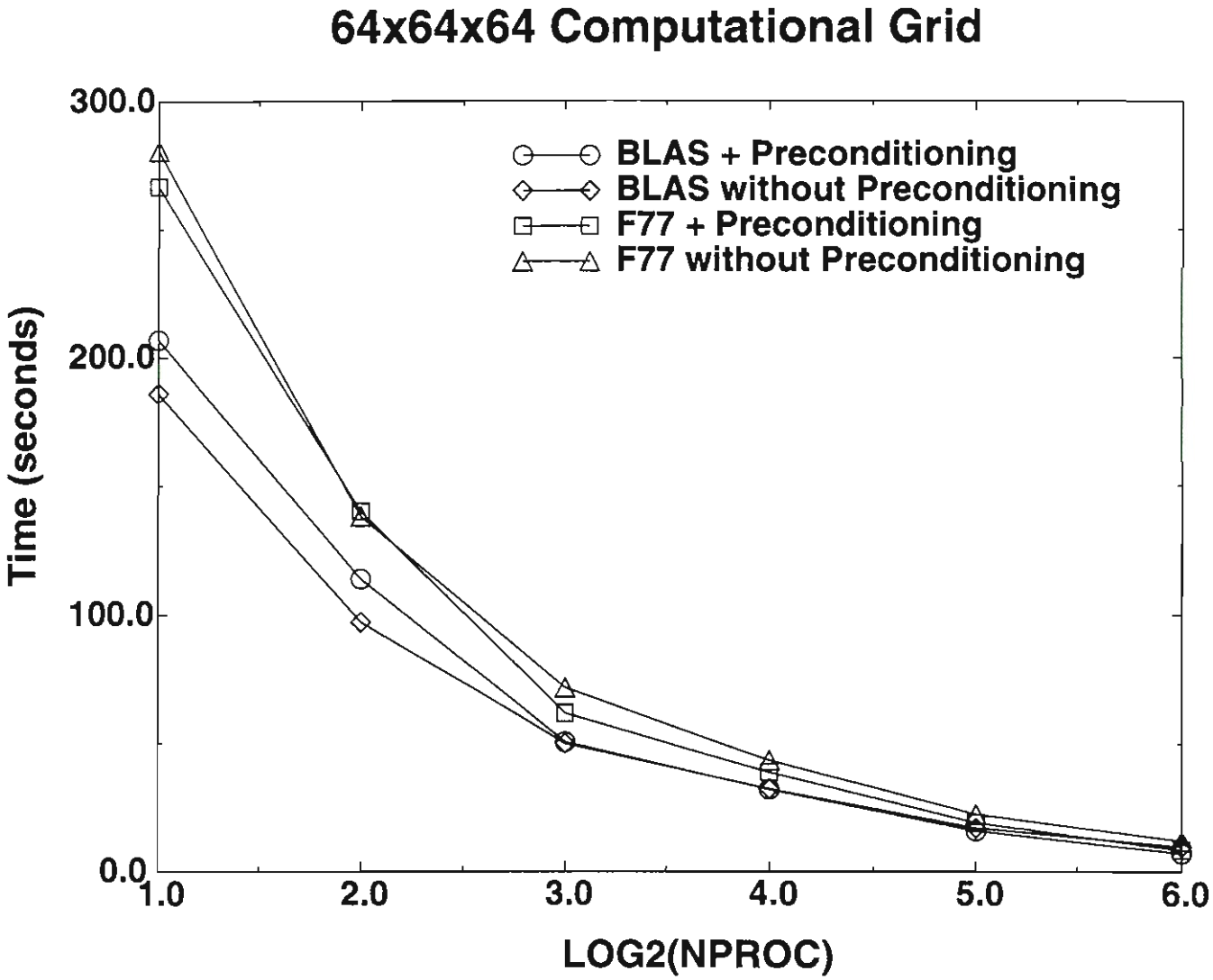

Figure 4 Performance of a Conjugate Gradient Solver on the CRAY T3D