The communication performance of the Cray T3D and its effect on iterative solvers

Y. F. Hu, D. R. Emerson and R. J. Blake

Daresbury Laboratory, CLRC, Warrington WA4 4AD, United Kingdom

Email: y.f.hu@dl.ac.uk, d.r.emerson@dl.ac.uk

Abstract

On many distributed memory systems, such as workstation clusters or the Intel iPSC/860, the multigrid algorithm suffers from having extensive communication requirements and, in general, it is not very competitive in comparison to the conjugate gradient algorithm. This is in contrast to the sequential problem whereby the multigrid algorithm is very effective in reducing the global residual, particularly for very large linear systems of equations. These two algorithms are now compared on the Cray T3D for solving very large systems of linear equations (resulted from grids of the order 256^3 cells). The communication performance of the Cray T3D is first measured by the standard ping-pong tests and also by practical communication tasks that are found frequently in CFD calculations. It is found that the Cray T3D has a low latency (≈ 6 μs) and a high bandwidth interprocessor communication (120 MB/s) when the low-level intrinsic communication routines are used. As a result, the multigrid algorithm is found to be very competitive when compared with the conjugate gradient algorithm for solving the very large linear systems arising from the Direct Numerical Simulation of turbulent Combustion (DNSC). Results are contrasted by those on the Intel iPSC/860.

1. Introduction

Large scale linear systems arising from Computational Fluid Dynamics (CFD) are usually solved using iterative methods because they are efficient for many large sparse systems and have a low storage requirement. However, it is a common observation that

when using stationary-type iterative algorithms, such as the Jacobi or the Gauss-Seidel algorithms, the residual, which is an indication of the extent to which the algorithms have converged, normally drops very quickly in the first few iterations, and then it takes more and more iterations to reduce the residual by the same factor. This is even more so when solving problems on very fine grids and is due to the fact that most stationary iterative algorithms are good at reducing the high frequency errors, but not the low frequency errors. Multigrid methods (see, e.g., [2]) are therefore frequently employed to overcome this problem by solving the problem on a coarser grid, where the low frequency errors of the finer grid will appear as high frequency errors, whenever progress on the finer grid is slow. Nonstationary iterative algorithms, such as the conjugate gradient, are also popular for CFD calculations.

Multigrid algorithms tend to iterate the most on the coarsest grid. However, one coarse grid iteration only incurs a fraction of the computational cost of the finest grid iteration. Hence, on a sequential machine, even though multigrid algorithms may need a large number of iterations to converge over all levels, the equivalent number of iterations on the finest grid is usually quite low and insensitive to the grid size. Multigrid algorithms are therefore quite competitive with nonstationary schemes, such as the conjugate gradient algorithm, on sequential machines. The same is not generally true on a distributed memory machine. The large amount of iterations required on the coarse grids, where the computational work is relatively small in relation to the communication requirement (most of which involve short message lengths), means that multigrid algorithms tend to suffer significantly on many distributed memory parallel computers, such as the Intel iPSC/860. This is due to the high latency and low interprocessor bandwidth on many of these systems.

In this paper, the performance of a multigrid algorithm is considered and compared with the conjugate gradient algorithm on a Cray T3D, for solving very large (sparse) systems of equations from the Direct Numerical Simulation of Turbulent Combustion. The Cray T3D (at the Edinburgh Parallel Computing Center) was installed in April 1994 with 256 processors and became UK's successor to the Intel iPSC/860 (at Daresbury Laboratory). The Intel has 64 processors each running at 40 MHz and having 16 MB of memory. The Cray T3D is composed of 320 processor elements (PEs),

each consisting of a DEC Alpha 21064 processor running at 150 MHz and having 60 MB of memory. The nodes (each node comprises 2 PEs) are arranged in a 3D torus. All arithmetic operations, both integer and floating point, are performed using 64-bit arithmetic. Moreover, the machine supports both message passing as well as global address space. As there are no high level intrinsic communication routines (such as CSEND/CRECV on the Intel iPSC/860), PVM is generally used. To send a typical message using PVM involves 3 calls (i.e. PVMFINITSEND, PVMFPACK, PVMFSEND). Cray also provides a simplified version named PVMFPSEND, and the corresponding PVMFPRECV. On the CRAY T3D there are also a library of optimised lower level communication routines, for example SHMEM_PUT and SHMEM_GET, which use the CRAY T3D's shared memory feature and allows remote writing and reading of data. These routines are tested and compared in this paper. Practical communication tasks using these routines are also looked at. Finally, the multigrid algorithm and the conjugate gradient algorithm are compared.

All the tests were carried out between December 1994 and January 1995. On the Cray T3D, the CF77 programming environment has since been updated but no significant effect was found to the results of this paper.

2. Measuring the Communication Performance of the Cray T3D

To measure the communication performance of the Cray T3D the standard ping-pong test ([10]) was carried out. In this test, processor A sends a message of length n to processor B whereby, upon receiving the message, processor B sends the message back to A. This process is repeated 1000 times and half of the averaged time is taken to be the time to do a send. Usually there is a linear relationship ([4]) between the time and the message size, namely:

$$t = t_0 + \beta n, \tag{2.1}$$

where t_0 is the startup time (in microseconds) and n is the message length (in Bytes). It is also useful to characterise the communication performance using the parameter pair $(r_\infty, n_{\frac{1}{2}})$ ([12]). Namely, the peak bandwidth is:

$$r_\infty = \frac{1}{\beta}, \text{ (MB/s)} \tag{2.2}$$

where MB/s stands for Million Bytes/second ([11]). The message size to reach half the peak bandwidth is then given by:

$$n_{\frac{1}{2}} = \frac{t_0}{\beta}. \quad \text{(Bytes)} \tag{2.3}$$

Table 2.1 lists the communication performance parameters obtained for the T3D using the standard test and the figures for Intel iPSC/860 are taken from Helin and Berrendorf ([8]). The test were done on a single node of the T3D comprising two processors, using standard FORTRAN 77 code with the $-O1$ $-Oscalar3$ compiler option, which was the highest optimisation available. The timings were obtained from the Cray intrinsic routine, *rtc*, which measures the time in clock cycles and is believed to be quite accurate. The overhead associated with a clock call was also taken into account. The time taken for the communication against the message length is plotted in Figure 2.1. The bandwidth as a function of the message length is given in Figure 2.2.

Table 2.1 Communication performance parameters for the Cray T3D and the Intel iPSC/860

	t_0	β	r_∞	$n_{\frac{1}{2}}$
Cray T3D, PVM	138(265)	$3.889 \times 10^{-2}(3.831 \times 10^{-2})$	25.7(26.1)	3357(6930)
Cray T3D, PVMFPSEND	32(222)	$2.969 \times 10^{-2}(3.807 \times 10^{-2})$	33.7(26.3)	1090(5834)
Cray T3D, SHMEM_PUT	6	8.321×10^{-3}	120.2	780
Cray T3D, SHMEM_GET	6	1.724×10^{-2}	58.0	362
Intel iPSC/860, CSEND	70(175)	$4.1 \times 10^{-1}(3.6 \times 10^{-1})$	2.4(2.8)	170(486)

As can be seen from Table 2.1, the PVMFPSEND has a measured startup time of around 32 μs, and a peak bandwidth of 26 MB/s for long messages. From Figure 2.1 it is seen that there is a jump in the communication time at 4096 Bytes. This corresponds to the value of the PVM parameter PVM_DATA_MAX (set in default to 4096 Bytes), which is the size of the PVM receiving buffer. If this parameter is increased to, say, 70 kB (70000 Bytes), then the jump does not happen until 70 kB, and the corresponding communication time before this size is reached is also lower, with a peak bandwidth of about 33 MB/s. However, this parameter affects the memory requirement of PVM

by the term PVM_SM_POOL * PVM_DATA_MAX, with PVM_SM_POOL set to the maximum of 10 and $2 \times$ *number of processors* by default, which allows each processor to receive 2 messages from every other processor. Thus a large increase of the parameter can significantly reduce the memory available for the applications. For that reason the default value (4096 Bytes) was used for the rest of this paper.

The Cray SHMEM_PUT routine allows data to be written to the memory of a remote processor by specifying the remote PE number and the address location. However, the remote processor's data cache is unchanged. Thus, when the remote PE references the same data, it may load the old value held in the data cache rather than the new value that was put in the main memory. There are a number of ways to avoid this problem. In our tests, the Cray routine SHMEM_UDCFLUSH was used to flush the cache after the put was done. It is also necessary to perform two synchronisations within a ping-pong cycle to ensure that the correct data are put between the two PEs. For the SHMEM_GET test, flushing the cache is not necessary, although two synchronisations still need to be done.

The time for both SHMEM_GET and SHMEM_PUT is almost linear with the message size, with a startup time of around 6μs for both, and a peak bandwidth of 58 MB/s and 120 MB/s, respectively. The peak bandwidth is roughly the same as the figures quoted by Cray ([14]), although the startup time is higher than the Cray figures of 1-2 μs. It was found that the time to SHMEM_PUT an 8 Byte message varied between 4-7 μs and appeared to depend upon the size of the arrays defined and on where the code fragment was called from (i.e. from a complicated program or a simple program). The peak bandwidth was also subject to a deviation of around 10 MB/s. It should be noted that instead of using SHMEM_UDCFLUSH routine to flush the whole cache, it is possible to use SHMEM_UDCFLUSH_LINE to flush a specific cache line. This will bring the time to send an 8 Byte message to around 3-4 μs without affecting the peak bandwidth. However, inconsistent answers were occasional produced with practical applications when using SHMEM_UDCFLUSH_LINE and the most reliable approach was to flush the entire cache.

As can be seen from Table 2.1, in comparison with the Intel iPSC/860, the Cray T3D, with both PVM and the low level routines, has a much lower latency and

the peak bandwidth is also 10-50 times higher.

All of the foregoing communication parameters were derived using two neighbouring processors. Figures 2.3 and 2.4 show the time against the message length for performing the ping-pong test among one processor and any other processor from a 128 processor configuration. The percentage of fluctuation (scaled by 10) of the sending time over the 127 ping-pong tests is also shown in Figures 2.3 and 2.4. The fluctuation was calculated from the difference between the maximum time (t_{max}) and the minimum time (t_{min}) to send a message of size n divided by the average time ($\bar{t}$) to send the same message, which can be written as:

$$fluctuation = \frac{t_{max} - t_{min}}{\bar{t}}. \tag{2.4}$$

As can be seen, the distance between processors seems to have only a small influence on the communication time. For the size of the messages considered in Figures 2.3 and 2.4, the percentage of fluctuation for the SHMEM_PUT is under 7%, while that for the PVMFPSEND is under 11%. Tests were also done for larger message sizes up to 80kB on up to 256 processors. It was found that the percentage of fluctuation for SHMEM_PUT stays at around 6%, while that for the PVM gradually increases to about 20%.

Bidirectional message exchange is also tested on a two PE node for SHMEM_PUT and PVMFPSEND (with the default setting of the parameters). The SHMEM_PUT is agained used with SHMEM_UDCFLUSH and two synchronisations. Unlike in the ping-pong test, here each processor will send the message to the other at the same time. The results in Table 2.2 show that the startup time and the bandwidth for the PVMFPSEND have a marked increase over the uni-directional result of Table 2.1, while that for SHMEM_PUT improves only slightly.

Table 2.2 Communication performance parameters for bidirectional message exchange

	t_0	β	r_∞	$n_{\frac{1}{2}}$
Cray T3D, PVMFPSEND	19(148)	$1.925 \times 10^{-2}(2.074 \times 10^{-2})$	51.9(48.2)	964(7145)
Cray T3D, SHMEM_PUT	6	7.139×10^{-3}	140.1	854

3. Measuring Some Practical Communication Tasks

Global summation and transferring of halo data are two examples of communication tasks frequently encountered in CFD codes. For all the results shown, PVM refers to using PVMFPSEND and PVMFPRECV with the default setting of the parameters, and SHMEM refers to using SHMEM_PUT with SHMEM_UDCFLUSH.

Global Summation

At the time of writing this paper, there was no intrinsic global summation routine on the Cray T3D. An in-house hypercube global summation algorithm (see, e.g., [7]) is therefore tested using either PVMFPSEND and PVMFPRECV, or SHMEM_PUT. Tests have been done to look at the time taken to do a global summation on up to 64 processors, for vectors up to 16384 elements. It is found that the time taken to do a global summation using the binary tree algorithm is almost linear to the log_2 of the number of the processors involved. It is a bit surprising that for very long messages, the SHMEM version of the global summation routine is only around twice the speed of the PVM version. This is lower than the ratio between their bidirectional peak bandwidths. To analyse this the summation was repeated on a 2 processor cube, so that the summation involved only two stages; each processor first sends its vector to the other processor and the received vector is then added (summed) to the vector it owned originally. A detailed break down of the time is given in Table 3.1 for summing a very long vector.

Table 3.1 Break down of the summation time (in milliseconds) using the binary tree summation routine

	msg. length (Bytes)	total time	comm. time	vector add. time
SHMEM	131072	5.19	1.86	3.30
PVM	131072	8.70	5.39	3.30

It can be calculated using the data from Table 3.1 that the vector summation part is doing 5.0 Mflop/s. For the bidirectional message exchange, PVM is achieving 49 MB/s and SHMEM is achieving 140 MB/s. It is seen that for very long vectors, the

time taken for the calculation (adding the two vectors) is of the same magnitude as the time spent in the communication. When SHMEM is used, more time is actually spent in the calculation. This explains why the ratio between the summation time is lower than that expected from the bidirectional peak bandwidth ratio. Incidentally, it is possible to replace the vector addition part of the routine with a Level 1 BLAS SAXPY operation, which would reduce the calculation time for very long vectors by a factor of about 3. Thus the ratio of total time will increase from 1.67 (Table 3.1) to 2.19. However, there is an overhead incurred using the BLAS routines ([6]) and it is found that it is not beneficial to use it in the summation routine for vector lengths below about 200 elements. Overall it seems that the CRAY T3D peak communication rate is so good compared to the computation, that improving the bandwidth further will not bring any significant reduction in the global summation time of very large vectors, for the particular summation routine considered. For an 8 Byte message, the SHMEM version of the global summation routine is 2 to 4 times faster than the PVM.

Halo Transfer

In many CFD calculations, the computational domain is meshed and for parallel computation, the mesh is partitioned into subdomains. Each subdomain, plus a halo region, is stored on a processor. The halo region includes the cells that are neighbours to the subdomain (or neighbours of neighbours, depending on the discretisation scheme used). Each processor will solve over its subdomain and assumes that the values in the halo region do not change during the iteration. After each iteration, each processor will update the values of the variables in the halo region by exchanging the halo data with the appropriate processors. For 3D calculations, the size of the data in the halo region of a processor is proportional to the surface area of the 3D subdomain resident on the processor. Three dimensional arrays are usually used for storing the grid data and in order to transfer the halo data, it is necessary to pack (or gather) 2D slices of the 3D arrays into a single array before sending it to the required destination. On receiving this array, that processor has to unpack (or scatter) the data back in to 3D arrays. This is illustrated in the following program fragment:

Packing:

for $k = ksta, ksto;\ j = jsta, jsto;\ i = ista, isto$

$vector(n) = array(i, j, k)$

$n = n + 1$

end do

Unpacking:

for $k = ksta, ksto;\ j = jsta, jsto;\ i = ista, isto$

$array(i, j, k) = vector(n)$

$n = n + 1$

end do

Here $ksta$, $ksto$ etc are the start and stop indices for the halo data. It is clear that, depending on which particular 2D slice of the 3D array is dealt with, the packing (unpacking) can involve large strides in the memory access.

Table 3.2 Break down (in milliseconds) of the halo data transfer time

	msg. length (Bytes)	total time	comm. time	pack and unpack time
SHMEM	9248	1.100	0.170	0.930
PVM	9248	1.663	0.733	0.930

Test cases were created by assigning each processor a cube of m^3 cells (including the halo region, which is assumed to have a thickness of one). The number of processors used ranged from 8 to 128 with m ranging from 2 to 34. The time taken to complete one halo data transfer, against the size of the cube m, was measured. It was found that for the shortest messages, using SHMEM is up to 3 times faster than PVM. For very large messages, however, using SHMEM was less than twice the speed of PVM, which is disappointing. The break down of time for the halo transfer on 2 processors for the 34^3 grid size is shown in Table 3.2. As can be seen from the table, the majority of the time is actually spent in packing (unpacking) rather than in the communication itself. In this case PVM and SHMEM are getting 25 MB/s and 109 MB/s for the bidirectional message exchange, respectively. However, the packing (unpacking) speed is only 5 Million operations per second (each pass of the packing or unpacking routine is assumed to incur 2 operations). So once again, because the CRAY T3D's communication bandwidth

for large messages is quite high, particularly when compared with the computation and main memory bandwidth, any significant improvement in the halo data transfer rate for large messages can only be achieved by increasing processor performance.

Balance between communication and computation

The previous discussion highlighted the importance for a balanced computation and communication performance. To formally measure the balance between communication and computation, we look at the *balance factor*, b ([8]; see also [7]) given by:

$$b = \frac{t_{comm}}{t_{calc}},$$

where t_{comm} is defined as the typical time to communicate one word between two PEs and t_{calc} is defined as the time to do a generic calculation, such as an addition, subtraction or multiplication (but not a division, which is performed in software). The parallel computer is defined as well-balanced if $b < 1$. As pointed out by Helin and Berrendorf ([8]), it is difficult to define the *typical time* for the communication and computation. For a sequential or shared memory computer, t_{calc} should be a measure of the speed of the processor while t_{comm} is a measure of the memory access speed. Thus b is a measure of how fast the memory is in feeding the processor. For distributed memory machines, remote data usually has to be fetched from (or put to) the remote processor explicitly before any calculation using this data can proceed. Therefore, for the purpose of comparing the speed of the remote data access (or message passing) and that of the computation, we use t_{comm} as a measure of the remote data access speed while t_{calc} a measure of computational speed, including the speed of the processor itself and that of the main memory access. Thus for example on a distributed memory system with $b >> 1$, rather than fetching some data from a remote processor, it may be better to derive the data if possible by some calculation on the local processor.

If we assume the rate of computation (say multiplication) of vectors of length $n/8$ (where n is in Bytes) is $R(n/8)$ (Mflop/s), then:

$$t_{calc} = \frac{1}{R(n/8)}. \quad (\mu\text{s})$$

Using (2.1), the typical communication time for sending an n Byte message is $t = t_0 + \beta n$, thus:

$$t_{comm} = \frac{t_0 + \beta n}{(n/8)}. \quad (\mu\text{s})$$

Using (2.2) and (2.3) it is seen, after some algebraic manipulations, that:

$$b(n) = \frac{t_{comm}}{t_{calc}} = \frac{8R(n/8)}{r_\infty} \left(1 + \frac{n_{\frac{1}{2}}}{n}\right).$$

Thus for very large messages ($n \to \infty$) we have:

$$b(\infty) = \frac{8R(\infty)}{r_\infty}. \tag{3.1}$$

For a short message of 8 Bytes, as one usually has $n_{\frac{1}{2}} >> 1$, then

$$b(8) \approx \left(\frac{n_{\frac{1}{2}}}{8}\right) \left(\frac{8R(1)}{r_\infty}\right) = \frac{R(1)}{\pi_0}, \tag{3.2}$$

where $\pi_0 = r_\infty / n_{\frac{1}{2}} = t_0^{-1}$ is the specific performance ([9]) which characterises the short-message communication performance. It is therefore seen that for very large values of n, the balance factor is given by the ratio of the computational rate and that of the peak bandwidth whilst for the shortest message, the balance factor is also affected by the latency. To reach a crude approximation for the balance factor b for very short message sizes, we will make the assumption that the floating point performance is constant and independent of n (this is clearly not the case ([6]), but it is generally not possible to derive an analytical solution for the performance R). This approach is also consistent with the work of Helin and Berrendorf ([8]). With the foregoing assumption, the communication performance for a short message, as given by equation (3.2), can therefore be expressed as

$$b(8) \approx \left(\frac{n_{\frac{1}{2}}}{8}\right) b(\infty). \tag{3.3}$$

On the Cray T3D, it is found that the performance of a multiplication or addition ($z_i = x_i * y_i$ or $z_i = x_i + y_i$, see [8]) using pure FORTRAN is typically between 4 and 12 Mflop/s. Taking $R = 8$ Mflop/s as the average, then $t_{calc} = 1/8 = 0.125\ \mu s$. Using the data for t_0 and β in Table 2.1, the balance factor, as a function of message size, can be plotted in Figure 3.1. For comparison, the balance factor for the Intel iPSC/860 ([8]) is also plotted, where it was assumed that the Intel iPSC/860 (in 64-bit words and with vectorisation) could do 5.9 Mflop/s. Compared with the Intel iPSC/860, the Cray T3D is seen to be much better balanced. It is found that for the Cray T3D with SHMEM_PUT, the balance factor varies between 52 and 0.5 and it becomes less than one for messages above 888 Bytes. The machine is therefore balanced for medium to

large message sizes. In fact, it can be slightly over balanced for very large message sizes, as was found for the global summation and halo data transfer routines.

Assuming the same computational performance, in order for the balance factor of the Cray T3D to be less than one for all message sizes, it is necessary to have a start up time t_0 of less than 0.06 μs, or around 10 clock cycles of the DEC Alpha, which does not seem possible. Therefore, for small messages, the Cray T3D is unbalanced due to the start up cost, though much better balanced than the Intel iPSC/860.

4. Application to the Multigrid Algorithm

Multigrid algorithms have been applied successfully in solving many linear and nonlinear systems. A usual 2-grid algorithm on linear system $Ax = b$ on the fine grid can be written as follows, with the superscript ^{f} and ^{c} stands for quantities on fine and coarse grid respectively.

Iterate on the fine grid equation $A^f x = b^f$ to get x^f

Compute fine grid residual $r^f = b^f - A^f x^f$

Compute coarse grid residual $r^c = I_f^c r^f$

Form coarse grid equation $A^c x = r^c$

Iterate on $A^c x = r^c$ to get x^c

Correct the fine grid solution $x^f := x^f + I_c^f x^c$

Repeat the above process until converged

In the algorithm illustrated above I_f^c is the restriction operator to transfer from the fine to the coarse grid and I_c^f is the prolongation operator from the coarse to the fine grid.

Parallel multigrid solvers, and the associated set of iterative algorithms, can be used for solving the linear systems arising from CFD calculations. These equations are usually very large and sparse but may also be anisotropic due to high grid aspect ratios. However, they are usually diagonally dominant. These features decides our particular choice of the implementation. The parallel multigrid solver is based on the sequential algorithm proposed by Hutchinson and Raithby ([13], [15]), where the coarse grid equation is formed by the so called block correct approach. With this approach, the equations on the coarse grid cells are formed essentially by adding the equations for

the fine grid cells. This avoids the complicated interpolation during the restriction and prolongation process. Thus it is possible to lump an arbitrary number of cells along any direction into a coarse grid cell. So a coarse grid cell may contain, say, 2 fine grid cells, each in x and y directions, but more than two cells in the z direction. This is useful when the equation is anisotropic along the z direction. It also allows us to cope with subdomains whose number of cells along a certain coordinate direction is not even, which could easily happen when solving CFD problems in parallel where a mesh has been partitioned into many subdomains.

The multigrid algorithm is parallelised by the usual grid partitioning strategy. The computational domain is split into $p = p_x \times p_y \times p_z$ subdomains, where p is the number of processors. Coarsening factors along each of the coordinate directions are specified by the user. For example, the factors (2, 2, 4) would mean that the coarse grid cells are formed by lumping 2 cells each in the first or second coordinate direction, but 4 cells in the third coordinate direction. A negative factor means lumping all the cells available along the coordinate direction that correspond to the factor, thus amounting to a 1D block correction ([13]). A series of grids, each coarser than the previous one, is generated on each processor using the coarsening factors provided. When the number of cells for a fine grid subdomain on the current processor along, say, the x direction cannot be divided by the coarsening factor, say 4, for that direction, one of the coarse grid cells will contain less than 4 fine grid cells.

Starting from the finest level (level one), for each level of the grids, the linear systems are solved using an appropriate parallel stationary iterative algorithm. A number of iterative algorithms have been implemented, including the Jacobi, the Gauss-Seidel (GS) and the alternative direction line implicit (ADI) algorithms. A conjugate gradient algorithm (CG), using block diagonal modified incomplete LU factorisation (ILU) as the preconditioner, was also implemented. It was found that, in general, multigrid combined with GS takes less time to converge than with ADI, even though the latter may take less iterations. After each iteration, halo data are transferred and the residual is assessed to decide whether to continue solving on the current level, to solve for a correction on a coarser level (if the norm is not reducing faster than a given factor σ, we used $\sigma = 0.5$), or whether the current level has converged (if the residual is less

than α times the initial residual, we used 1.0^{-6} for the first level and $\alpha = 0.1$ for all the other levels). If the given tolerance has been satisfied, the correction can be added to the finer level. It was found that on the coarsest grid, as the residual has to go down by the factor α, it was faster to solve the problem using the conjugate gradient algorithm (CG), rather than using the GS algorithm.

The number of iterations (work units) for a multigrid is defined to be the sum of the equivalent finest iteration numbers on each level of the grids. Thus, with a coarsening factor of 2 in each coordinate direction, one iteration on level two is counted as only $1/2^3$ iterations on the finest grid (for a 3D calculation). This obviously ignores the cost of transferring between grids and, more importantly on a parallel computer, the cost of the communication.

The multigrid solver is used as a linear system solver for a Direct Numerical Simulation of Combustive Combustion (DNSC) project. The DNSC ([3]) involves the solution of the Navier-Stoke equations, augmented by two additional equations each describing the transport of a single scalar variable which together specify the thermochemical state of the system in the presence of differential diffusion effects. The usefulness of direct numerical simulations depends on the Reynolds number that can be attained. It can be shown that the attainable Reynolds number is given by:

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

where N is the number of cells along one direction of the computational domain and n is the number of cells required to resolve the flame front. The generally accepted figure for n is 10. Therefore, the Reynolds number is about 30 for a grid size of 128^3. On the CRAY T3D, grid sizes of 368^3 have been used during initial tests which give a Reynolds number of about 120. For problems of this size, computer storage requirements can be quite high even for the solution of the linear equations. The use of a multigrid algorithm will increase this storage requirement, although by less than a factor of 2.

The most expensive part of the DNSC calculation is the repeated solution of the linear equations from the pressure equation. As these linear systems are symmetric positive semi-definite, they can also be solved using the preconditioned conjugate gradient algorithm. The results of solving such a system associated with a mesh of size 64^3 are shown in Table 4.1. The results compare the conjugate gradient algorithm (CG) and

the multigrid algorithm (with Gauss-Siedel and 4 levels of grids, MGS4) on both the Intel iPSC/860 and the Cray T3D. The number of iterations, total CPU time (t_{cpu}, in seconds), time for halo data transfer (t_{trans}, in seconds) and time for global summation (t_{sum}, in seconds) are given. The computational domains are partioned into $4 \times 2 \times 2$ subdomains. The conjugate gradient algorithm is written in Level 1 BLAS uses the ILU factorisation as the preconditioner. On the Cray T3D, SHMEM_PUT routine is used for faster communication.

Table 4.1 Solving a 64^3 problem on 16 processors: comparing the Cray T3D and the Intel iPSC/860

machine	algorithm	iterations	t_{cpu}	t_{trans}	t_{sum}
Cray T3D	CG	144	11.3	0.4	0.01
Intel iPSC/860	CG	144	30.7	3.6	0.7
Cray T3D	MGS4	129.1	8.3	1.5	0.6
Intel iPSC/860	MGS4	129.1	42.0	15.7	9.2

As can be seen from Table 4.1, even with only 16 processors, a lot of time is spent in communication on the Intel iPSC/860, particularly for the multigrid algorithm. This is because there are a lot more iterations on the coarser grid which involve a lot of global summations and halo data transfers, and most with short message sizes. On the other hand, the computational work, relative to communication, is less for the coarse grid. Since the Intel iPSC/860 has a much higher start up time, this results in a very high communication overhead. Subsequently, the multigrid algorithm is slower than the conjugate gradient algorithm on the Intel iPSC/860 whilst on the Cray T3D, the multigrid is faster.

The same two algorithms were then used to solve a much larger problem associated with a grid size of 256^3 on the Cray T3D. The results are given in Tables 4.2 and 4.3 for 128 and 256 processors, respectively. The grids were partitioned into $8 \times 4 \times 4$ subdomains (for 128 processors) and $8 \times 8 \times 4$ (for 256 processors) subdomains. Results using both PVMFPSEND (PVMFPRECV) and SHMEM are presented. Results for the Intel are not available because the problem is too large for the machine.

As can be seen from the tables,the conjugate gradient algorithm takes a lot more

iterations for this large problem than when used for the 64^3 problem. It is also apparent that the communication takes only a small percentage of the total time for the conjugate gradient algorithm. This is because for the size of the problem being solved, the subdomain on each processor is still quite large (of size at least $34 \times 34 \times 66$), so the three global summations and one halo data transfer per iteration for the conjugate gradient algorithm do not take a significant amount of time compared with the computation. However, for the multigrid algorithm, because of the large amount of communication on the coarse grid, even though the Cray T3D has very fast communications, the amount of time spent in the global summation and halo data transfer are quite significant.

Table 4.2 Solving a 256^3 problem on 128 processors: comparing the conjugate gradient multigrid algorithms

	algorithm	iterations	t_{cpu}	t_{trans}	t_{sum}
SHMEM	CG	474	243.5	5.9	0.2
SHMEM	MGS4	193.5	98.5	12.1	6.0
SHMEM	MGS5	173.0	91.4	14.4	9.5
PVM	CG	474	246.3	8.3	0.6
PVM	MGS4	193.5	122.2	21.2	20.4
PVM	MGS5	173.0	126.8	26.8	32.4

It is also interesting to compare the results using PVMFPSEND (PVMFPRECV) and SHMEM. Compared with PVMFPSEND (PVMFPRECV), using SHMEM is seen to reduce the global summation time t_{sum} by a factor of over 3. This is consistent with the reduction factor of between 2 to 4 for 8 Byte messages mentioned in the last section, as all the global summations here are for the summation of scalars. Using SHMEM, the halo data transfer time, for the conjugate gradient method, is reduced by about 1.4-1.5 and for the multigrid algorithm by a factor of 1.8-2. This is again consistent with the reduction factor of up to 3 for the shortest message length and less than 2 for the largest message length, as seen in last section. This is because, for the conjugate gradient algorithm, the halo data size are quite large, while for the multigrid algorithm, the size of the halo data ranges from very small, on the coarsest grid, to quite large on the finest grid.

Table 4.3 Solving a 256^3 problem on 256 processors: comparing the conjugate gradient multigrid algorithms

	algorithm	iterations	t_{cpu}	t_{trans}	t_{sum}
SHMEM	CG	443	116.3	3.4	0.2
SHMEM	MGS4	213.8	57.9	9.8	7.2
SHMEM	MGS5	193.0	61.0	13.2	12.6
PVM	CG	443	118.6	5.2	0.7
PVM	MGS4	213.8	82.5	17.6	23.9
PVM	MGS5	193.0	103.8	25.9	54.2

It was also observed that it is sometimes beneficial to use fewer grid levels to reduce the communication overhead associated with the coarse grids. For example, on 256 processors, MGS4 is faster than MGS5 due to the smaller communication times, even though the former requires more iterations and more computing than the latter.

5. Conclusions

In this paper, the communication performance of the Cray T3D, relative to its computational performance, was measured. In general, the Cray T3D communication is seen to have a small start up time (≈ 6 μs) and a high peak bandwidth (120 MB/s), especially when compared with the Intel iPSC/860. The machine is also found to be well balanced for medium to large message sizes.

The good communication performance of the Cray T3D makes the multigrid algorithm, which performs poorly on the Intel iPSC/860, very competitive against the conjugate gradient algorithm, even though the multigrid algorithm still has a communication overhead of over 30% on 256 processors of the Cray T3D for the large scale problem considered. To further improve the communication time for the multigrid algorithm, it is necessary to reduce the startup time of the communication (a main limiting factor for the global scalar summation using the binary tree algorithm, and the halo transfer of short messages) as well as to improve the packing/unpacking speed (a limiting factor for the transfer of halo data of large sizes).

It is noted that to reduce the communication cost of the multigrid algorithm, it may be useful to solve the coarse grid problem on only one or part of the processors available (see, e.g., [1]). The preconditioning part of the conjugate gradient algorithm can also be improved by employing an approach suggested by Eisenstat [5]. One interesting thing that was observed during our investigation of the multigrid and conjugate gradient algorithms was that, even though multigrid was suggested to overcome the inability of the stationary type iterative algorithms in reducing the low frequency errors, in practice, the combination of multigrid with the conjugate gradient algorithm on very large problems frequently converges faster than the conjugate gradient algorithm itself, or the combination of the multigrid and the GS algorithm. It is planned to study all these further.

Acknowledgement The authors wish to thank Prof. R. W. Hockney, who made many useful comments on an early version of this paper.

References

[1] M. Alef, Concepts for efficient multigrid implementation on SUPRENUM-like architectures, Parallel Computing, 17 (1991) 1-16.

[2] W. L. Briggs, A Multigrid Tutorial (Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, 1987).

[3] R. S. Cant, C. J. Rutland and A. Trouve, in: Proceeding of 1990 CTRSummer Program (Center for Turbulence Research, Stanford University, 1990) 271-279.

[4] T. H. Dunigan, Performance of the Intel iPSC/860 and Ncube 6400 hypercubes, Parallel Computing, 17 (1991) 1285-1302.

[5] S. C. Eisenstat, Efficient implementation of a class of preconditioned conjugate gradient methods, SIAM Journal of Scientific and Statistical Computing, 2 (1981) 1-4.

[6] D. R. Emerson and R. S. Cant, Direct simulation of turbulent combustion on the CRAY T3D - initial thoughts and impressions from an engineering perspective, submitted to Parallel Computing, 1995.

[7] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, D. Walker, Solving Problems on Concurrent Processors, Vol. I (Prentice Hall, Englewood Cliffs, NJ, 1989).

[8] J. Helin and R. Berrendorf, Analyzing the performance of message passing hypercubes: a study with the Intel iPSC/860 (the 1991 ACM International Conference on Supercomputing, Cologne, Germany, 1991).

[9] R. W. Hockney, The communication challenge for MPP: Intel Paragon and Meiko CS-2, Parallel Computing, 20 (1994) 389-398.

[10] R. W. Hockney, Synchronization and communication overheads on the LCAP multiple FPS-164 computer system, Parallel Computing, 9 (1989) 279-290.

[11] R. W. Hockney and M. Berry, Public international benchmarks for parallel computers, PARKBENCH Committee report 1, Scientific Programming, 3 (1994) 101-146.

[12] R. W. Hockney and C. R. Jesshope, Parallel Computers 2: Architecture, Programming and Algorithms (IOP Publishing/ Adam Hilger, Bristol & Philadelphia, 1988).

[13] B. R. Hutchinson and G. D. Raithby, a multigrid method based on additive correction strategy, Numerical Heat Transfer, 9 (1986) 511-537.

[14] W. Oed, The Cray Research Massively Parallel Processor System CRAY T3D (Cray Research, November, 1993).

[15] A. Settari and K. Aziz, A generalization of the additive correction methods for the iterative solution of matrix equations, SIAM Journal of Numerical Analysis, 10 (1973) 506-521.