NAS Parallel Conjugate Gradient Benchmark on the Cray T3D

Tamara L. Gibson *
Department of Mathematics
University of Maryland
College Park, Maryland 20740

July 17, 1996

SRC Technical Report Number: SRC-TR-94-129

Abstract. The NAS Parallel Benchmark [1] kernels were developed to evaluate the performance of highly-parallel supercomputers. These benchmarks are unstructured in the sense that they give only an algorithm defining the benchmark; all implementation details are left to the programmer. We looked specifically at a kernel to solve an unstructured sparse linear system via the conjugate gradient method, the CG kernel. This kernel was implemented on one of the newest massively-parallel supercomputers, the Cray T3D. Currently, our implementation of the CG kernel on the T3D achieves 306 MFlops on 64 processors. In comparison, a Cray YMP single head gets 127 MFlops, a 128 processor iPSC/860 gets 181 MFlops, and a 32,768 processor CM-2 gets 105 MFlops [5].

*The bulk of this work was done at Supercomputing Research Center (SRC), 17100 Science Drive, Bowie, MD 20715-4300. I would like to thank Steve Kratzer and John Conroy of SRC for their help. In addition, I would like to acknowledge the National Physical Science Consortium (NPSC), the National Security Agency (NSA), and the University of Maryland for fellowship support. Lastly, I would like to thank my advisor, Dianne O'Leary, for her help and advice on this project.

1 Introduction

1.1 Problem Introduction

The Numerical Aerodynamical Simulation (NAS) Parallel Benchmark kernels were developed by a team of experts at the National Aeronautics and Space Administration (NASA) to evaluate the performance of highly-parallel supercomputers. The kernels exemplify the computation and data movements typical of large-scale computational fluid dynamics (CFD) applications. We will be concentrating on the conjugate gradient kernel. This kernel uses the inverse power method to find an estimate of the largest eigenvalue of a symmetric positive definite sparse matrix with a random pattern of nonzeroes. We will do timings on the 1400×1400 Class A matrix which has approximately 1.8 million nonzeroes. [1]

1.2 T3D Introduction

The Cray T3D, released in 1994, is a massively parallel multiple-instruction/multiple-data (MIMD) machine. It consists of a variable number of processing elements (PE's) connected in a three-dimensional torus with a Cray C90 or Cray C98 as a front end. Each PE consists of a 150 MHz Dec Alpha central processing unit (CPU) with 8K bytes of cache and 2 megawords (MW) of dynamic access random memory (DRAM). [6] All timings are based on code compiled with "AC" [1] which is a modification of the Gnu C compiler for the T3D.

2 T3D Description

2.1 Computations and Memory Organization on the T3D

Memory structure plays an important role in the performance of the individual processors. As mentioned before, we have 8K bytes of direct-mapped cache and 2 MW of DRAM. Memory is loaded into the cache in the form of four-word (32 bytes) cache lines. DRAM access takes 24 ticks/cache-line versus one tick/word for cache access. For example, if we need to use all four words in a cache line, the first access takes 24 ticks, and the additional three accesses take one tick each, giving an average access time of about 7 ticks/word. On the other hand, if we need only one word in a cache line from memory, the average access time is 24 ticks/word. In either case, using the date repeatedly before the cache line is written over lowers the average access time dramatically. As discussed in Section 5, the sparse matrix-dense vector multiply has poor memory access performance because it accesses the dense vector randomly and hence usually only reads one word per cache line. This poses serious performance penalties since the processor spends much of its time waiting for memory access.

[1] "AC" is written by Bill Carlson and Jesse Draper of SRC.

2.2 Communications on the T3D

The Cray T3D is arranged as a three-dimensional torus, although users are not always allocated PE's in that pattern. We will treat the PE's as a two-dimensional grid in our application. Communications are controlled via router chips; hence, the programmer has no control over the route that a message takes between any two PE's. We will not be concerned with communication patterns for the purposes of this research.

Our communications are done via the shared memory library, otherwise known as the "shmem" library. We use what is known as a "shmem_put", hereafter referred to as a **PUT**. A **PUT** writes directly to the memory of another processor. It returns as soon as the message is launched onto the interprocessor network; it does not wait for receipt on the other PE. To wait for receipt on the other PE, a "net_quiet" command must be issued. The "net_quiet" does not return until all outstanding **PUT**'s have completed on the remote processors. A "barrier" is a synchronization command; all PE's must reach the barrier before the application continues to execute. If all PE's execute a **PUT**, a "net_quiet", and a "barrier", we are guaranteed that all messages have been sent and received by all PE's. Another issue is cache coherence; that is, we want the values in cache to match the values stored in memory. A **PUT** writes directly to a remote PE's memory but does not update the remote PE's cache. We handle this problem by doing a "cache_flush" which tells the PE that the current cache contents are invalid. A **BARRIER** command refers to doing a "net_quiet", "barrier", and "cache_flush" command. See the Shmem user's guide [2] for more information on this subject.

3 Description of Problem

3.1 Inverse Power Method

We are using the inverse power method to find the largest eigenvalue of a real, symmetric, positive definite matrix. This algorithm is outlined specifically as part of the benchmark. A denotes the real square matrix of order n, lower case Roman letters are vectors, and lower case Greek letters are scalars. We denote transpose by a "T" superscript, and $\|\cdot\|$ denotes the Euclidean norm. The inverse power method is implemented as follows with the significant subroutine names given in boxes:

(1) $x = [1, 1, \cdots, 1]^T$
(2) **Start Timing**
(3) For $it = 1, niter$
(4) Approximate the solution to the system $Az = x$ and return $\|r\| = \|x - Az\|$ $\boxed{\textbf{CG Solve}}$
(5) $\zeta = \lambda + 1/(x^T z)$
(6) Output: it, $\|r\|$, and ζ
(7) $x = z/\|z\|$ $\boxed{\textbf{Normalize}}$
(8) End it-Loop

(9) **Stop Timing**

The Class A problem has $n = 14,000$, $niter = 15$ and $\lambda = 20$. [1]

3.2 CG Solve

The conjugate gradient routine requires 25 iterations of the conjugate gradient method and is outlined specifically as part of the benchmark. We outline the conjugate gradient method below with the needed subroutines for each step given in boxes [1]:

(1) $z = 0$
(2) $r = x$
(3) $\rho = r^T r$ $\boxed{\textbf{Dot}}$
(4) $d = r$
(5) For $i = 1, 25$
(6) $\quad q = Ad$ $\boxed{\textbf{Mat-Vec Mult}}$
(7) $\quad \alpha = \rho/(d^T q)$ $\boxed{\textbf{Dot}}$
(8) $\quad z = z + \alpha d$ $\boxed{\textbf{Saxpy}}$
(9) $\quad \rho_0 = \rho$
(10) $\quad r = r - \alpha q$ $\boxed{\textbf{Saxpy}}$
(11) $\quad \rho = r^T r$ $\boxed{\textbf{Dot}}$
(12) $\quad \beta = \rho/\rho_0$
(13) $\quad d = r + \beta d$ $\boxed{\textbf{Saxpy}}$
(14) End i-Loop
(15) Compute residual: $\|r\| = \|x - Az\|$ $\boxed{\textbf{Residual}}$

See Golub and Van Loan [4] for further discussion of the conjugate gradient method. In the next section, we outline the needed subroutines and their implementations on the T3D.

4 Analysis of Subroutines

4.1 Notation

We will let p denote the number of PE's in use. The scalars $(\zeta, \lambda, \rho, \rho_0, \alpha, \beta)$ used in the **Inverse Power Method** and the **CG Solve** are stored redundantly on every PE. The vectors (d, q, r, x, z) are evenly divided among the PE's; i.e., the kth PE, $k = 1, \cdots, p$, holds elements $(k-1) \cdot \frac{n}{p} + 1$ through $k \cdot \frac{n}{p}$ of each vector. We will make the assumption that p divides n; if not, the problem can be "padded" to make it so. For convenience, we will let $\hat{n}$ denote the number of vector elements each PE holds; i.e., $\hat{n} = \frac{n}{p}$. The notation x_i denotes the ith element of the vector x. We will discuss the storage of the matrix A when we explain the **Mat-Vec Mult** subroutine in Section 4.4.

We must be able to clearly differentiate between our global perspective and each PE's local perspective in our algorithm discussion. In general, a variable with a bar or a hat above it is a variable from the PE's perspective.

All other variables are from a global perspective. See Table 4.1 for a listing of all parameters.

Name	Relations	Description
n		Dimension of Problem
p		Number of processors
$\hat{n}$	$\hat{n} = n/p$	Dimension of vector size on each PE
n_h, n_w		The matrix A is divided into $n_h \times n_w$ blocks
p_h	$p_h = n/n_h$	Height of PE grid in **Mat-Vec Mult**
p_w	$p_w = n/n_w$	Width of PE grid in **Mat-Vec Mult**

Table 1: Summary of Parameters

4.2 Saxpy

This subroutine computes $u = \mu v + w$ where μ is a scalar and u, v, and w are real n-vectors. The kth PE, $k = 1, \cdots, p$, holds elements $(k-1)\hat{n}+1$ through $k\hat{n}$ of u, v, and w. We refer to a PE's local pieces of u, v, and w as $\hat{u}$, $\hat{v}$, and $\hat{w}$, respectively; furthermore, μ is stored redundantly on every PE. The algorithm goes as follows for each PE:

(1) For $i = 1, \hat{n}$
(2) $\quad \hat{u}_i = \mu \hat{v}_i + \hat{w}_i$
(3) End i-Loop

Hence, we see that **SAXPY** requires no communications and $\hat{n} = \frac{n}{p}$ multiplies and additions per PE. Hence, we will get linear speed-up in p, the number of PE's.

4.3 Dot

Here we compute $\delta = u^T v$ where u and v are real n-vectors. The final answer, δ, must be repeated on all PE's. Once again, we will let $\hat{u}$ and $\hat{v}$ denote each PE's local piece of u and v, and we let δ be repeated on every PE. We look at the inner product as follows:

$$\delta = \sum_{i=1}^{n} u_i v_i = \sum_{k=1}^{p} \sum_{j=(k-1)\hat{n}+1}^{k\hat{n}} u_i v_i.$$

PE k computes the sum $\delta = \sum_{j=(k-1)\hat{n}+1}^{k\hat{n}} u_i v_i$. All that remains is to add up the individual pieces and be sure that the result is distributed throughout all the PE's. We will do the second part using a butterfly-style accumulation [3]. The algorithm goes as follows on PE k:

(1) $\delta = 0$
(2) For $i = 1, \hat{n}$
(3) $\quad \delta = \delta + \hat{u}_i \hat{v}_i$

(4) End i-Loop
(5) For $i = 1, \log_2 p$
(6)   **PUT** δ into δ' on PE $k' = ((k-1) \oplus 2^i) + 1$
(7)   **BARRIER**
(8) $\delta = \delta + \delta'$
(9)   **BARRIER**
(10) End i-Loop

In the above algorithm, $\oplus$ symbolizes XOR. **PUT** and **BARRIER** are described in Section 2.2. The first **BARRIER** ensures that all **PUT**'s are completed before δ and δ' are summed. The second **BARRIER** is only necessary to ensure that δ' is not overwritten before it is used. This **BARRIER** can be eliminated by **PUT**-ing to a different spot for each value of the counter i.

The operation count on this routine is easy to compute. In steps (2) through (4), we require $\hat{n}$ additions and multiplies. In steps (5) through (10), we require $\log_2 p$ additions. Hence, the total number of computations required is

$$\frac{2n}{p} + \log_2 p.$$

The computational work decreases *almost* linearly with the number of PE's, depending on the relation between n and p.

The communications are a different matter. Each PE has to do $\log_2 p$ one-word **PUT**'s. Not only does the number of **PUT**'s per processor increase by one each time the number of PE's doubles, but the network contention also grows. This makes communication costs hard to scale. Sometimes increasing the number of PE's can actually lessen the network contention because it creates a different communication pattern.

4.4 Mat-Vec Mult

The matrix-vector multiply is by far the most expensive subroutine of the **CG Solve** routine. This is true for both computations and communications. To summarize, we wish to compute

$$u = Av,$$

where u and v are n-vectors and A is an $n \times n$ matrix. In our case, A is sparse, but we assume that dividing the matrix into evenly-sized blocks will yield blocks with approximately the same number of non-zeroes.

4.4.1 Discussion of two matrix decompositions

First we discuss two different approaches towards a parallel matrix-vector multiply, both of which are explained in Bertsekas and Tsitsiklis [3].

1. **Row Storage.** The row storage method splits the matrix by rows. PE $k, k = 1, ..., p$, holds rows $(k-1)\hat{n} + 1$ to $k\hat{n}$ of the matrix A and

all of the vector v. PE k then computes

$$u_i = \sum_{j=1}^{n} a_{ij} v_j, \qquad i = (k-1)\hat{n} + 1, \cdots, k\hat{n}.$$

At this point PE k knows elements $(k-1)\hat{n}+1$ through $k\hat{n}$ of u This method requires $n \cdot \hat{n}$ multiplies and $(n-1) \cdot \hat{n}$ additions. After this step, we need to do an *all-to-all* communication so that each PE can again know all the elements of u for the next step. Hence, PE k sends elements $(k-1)\hat{n}+1$ through $k\hat{n}$ of u to the other $p-1$ PE's. Depending on the network configuration and routing issues, this can take varying amounts of time, but we will generalize and say that the time is proportional to n because each PE needs to receive $n - \hat{n}$ elements.

2. **Column Storage.** The column storage method divides the matrix by columns. PE $k, k = 1, ..., p$, holds columns $(k-1)\hat{n}+1$ to $k\hat{n}$ of the matrix A and elements $(k-1)\hat{n}+1$ to $k\hat{n}$ of the vector v. PE k computes

$$\sum_{j=(k-1)\hat{n}+1}^{k\hat{n}} a_{ij} v_j, \qquad i = 1, ..., n.$$

Once these quantities are computed, we sum across all PE's to compute u.

$$u_i = \sum_{k=1}^{p} \sum_{j=(k-1)\hat{n}+1}^{k\hat{n}} a_{ij} v_j, \qquad i = 1, ..., n.$$

We divide up the work by having each PE sum up the p pieces necessary to compute elements $(k-1)\hat{n}+1$ to $k\hat{n}$ of u. Hence, each PE must do $n \cdot \hat{n}$ multiplies and $n \cdot 2\hat{n}$ additions. The summation across PE's requires that each PE send each of the other $p-1$ PE's a packet consisting of $\hat{n}$ elements and receive packets of size $\hat{n}$ from each of the other $p-1$ PE's. This is an *all-to-all* communication pattern. We will again generalize and say that the communication time is proportional to n because each PE receives $n - \hat{n}$ elements.

4.4.2 Block Storage Method

We use a combination of these two methods in our version, which is referred to as the block method [5]. Instead of dividing the matrix by rows or columns, we divide it into evenly-sized blocks of dimension $n_h \times n_w$. We aim to reduce communication cost without increasing computation cost.

Computations. Without loss of generality, we assume that the block dimensions are evenly divisible by p; if not, we can pad the matrix A so that they are. We let $p_h = \frac{n}{n_h}$ and $p_w = \frac{n}{n_w}$. We then assign a coordinate address to each PE based on its row and column coordinates as shown in Figure 1. For example, PE 2 is assigned coordinates (2,1). In general, PE k is assigned coordinates (k_h, k_w) where $k_h = (k-1) \bmod p_h + 1$ and $k_w = \lfloor (k-1)/p_h \rfloor + 1$.

	Column 1	Column 2	$\cdots$	Column p_w
Row 1	1	$p_h + 1$	$\cdots$	$p_h(p_w - 1) + 1$
Row 2	2	$p_h + 2$	$\cdots$	$p_h(p_w - 1) + 2$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
Row p_h	p_h	$2p_h$	$\cdots$	p

Figure 1: PE layout

The matrix A is divided into $n_h \times n_w$ blocks as shown in Equation 1:

$$A = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1p_w} \\ A_{21} & A_{22} & \cdots & A_{2p_w} \\ \vdots & \vdots & \ddots & \vdots \\ A_{p_h1} & A_{p_h2} & \cdots & A_{p_hp_w} \end{bmatrix}. \tag{1}$$

Each processor handles one block of A; specifically, PE (k_h, k_w) handles block $A_{k_hk_w}$.

Next, we must break the problem of computing $u = Av$ down. First, we look at it componentwise:

$$u_i = \sum_{j=1}^{n} a_{ij}v_j, \qquad i = 1, ..., n.$$

We can break this sum down into a sum of sums:

$$u_i = \sum_{k_w=1}^{p_w} \sum_{j=(k_w-1)n_w+1}^{k_wn_w} a_{ij}v_j, \qquad \begin{array}{l} i = (k_h - 1)n_h + 1, ..., k_hn_h, \\ k_h = 1, ..., p_h. \end{array} \tag{2}$$

The range that i is in determines what row of processors computes u_i, but the k_wth PE in the row computes the k_wth inner sum. Hence, PE (k_h, k_w) computes

$$\sum_{j=(k_w-1)n_w+1}^{k_wn_w} a_{ij}v_j, \tag{3}$$

for values of i between $(k_h - 1)n_h + 1$ and k_hn_h. This requires n_w additions and multiplications for n_h sums; hence, the total number of additions and multiplications per PE is $2n_hn_w$. Of course, we have not yet done all the sums we need to do. The next step is to do the outer summation of Equation 2. We need p_w inner sums from p_w different processors summed for each u_i, $i = 1, ..., n$. Hence, we need to do np_w additions. Because this work can be divided among the PE's, we actually only need to do $\hat{n}p_w$ additions per PE; so, the total number of operations that needs to be done per PE is

$$n(2\frac{n}{p} + \frac{1}{p_h}).$$

Communications. We assume that the vector v is initially distributed evenly over the PE's as is needed for the **Dot** (Section 4.3) and **Saxpy** (Section 4.2) routines, and we want to finish with u distributed in the same manner. To do the computations in Equation 3, we must have elements $(k_w - 1)n_w + 1$ through $k_w n_w$ of v in local storage for each PE in column k_w. These elements are initially distributed over the column, so we must do a column *all-to-all* communication. Each PE must broadcast $\frac{n}{p}$ elements to p_h processors. Hence, the communications cost here is n_w.

For the outer sum of Equation 2, we want to sum the pieces we have computed to form the elements of u. We will have the kth PE compute the kth portion of u; i.e., PE k will compute elements $(k-1)\hat{n} + 1$ through $k\hat{n}$ of u. For example, every PE in the first row will send the first $\hat{n}$ elements of its sum from Equation 3 to PE 1, the next $\hat{n}$ elements to PE 2, and so on. Specifically, each PE has a vector of length n_h that it must divide into p_w pieces, each of length $\hat{n}$. Each piece must then be broadcast to a different PE. Hence, we send p_w different packets of size $\hat{n}$. So for this part, we have communications proportional to n_h.

Hence, the total communications cost is proportional to

$$n_w + n_h.$$

If $n_w = n_h = \frac{n}{\sqrt{p}}$, then our communications are reduced from n in the row or column storage method to $2\frac{n}{\sqrt{p}}$ in the block method. Note that in all of these computations, if we set $p_w = 1$ then we have the row storage method, and if we set $p_h = 1$, we have the column storage method.

Algorithm. Now we lay out the algorithm from PE k's point of view. We will let $\bar{A}$ represent $A_{k_h k_w}$, $\hat{v}$ represent elements $(k-1)\hat{n}+1$ through kn of v, $\bar{v}$ represent elements $(k_w-1)n_w+1$ through $k_w n_w$ of v. $\bar{w}$ is a n_h-long vector representing the result of Equation 3. $\hat{u}$ represents elements $(k-1)\hat{n}+1$ through kn of u.

(1) **PUT** $\hat{v}$ into positions $(k_w - 1)\hat{n} + 1$ through $k_w\hat{n}$ of $\bar{v}$ on every PE in the same column.
(2) **BARRIER**
(3) For $i = 1, n_h$
(4) $\quad\bar{w}_i = 0$
(5) $\quad$For $j = 1, n_w$
(6) $\quad\quad\bar{w}_i = \bar{w}_i + \bar{a}_{ij}\bar{v}_j$
(7) $\quad$End j-Loop
(8) End i-Loop

(9) $k' = (k_h - 1)p_w + 1$
(10) For $i = 1, p_w$
(11) **PUT** elements $(i-1)\hat{n}+1$ through $i\hat{n}$ of $\bar{w}$ into a holding spot on PE $k' + i$.
(12) End i-Loop
(13) **BARRIER**
(14) Sum pieces just received to form $\hat{u}$

Steps (1) through (2) concatenate the $\hat{v}$ vectors from each PE in each column to form the n_w-vector $\bar{v}$ which is repeated on each PE in the column. Steps (3) through (8) do the local matrix-vector multiplication as described in Equation 3 to compute $\hat{w}$. Step (9) computes the PE number which is the first PE to need $\hat{n}$ components of $\hat{w}$. Note that we actually do this using a sparse matrix format. Steps (10) through (12) distribute the components of $\hat{w}$ to the p_w PE's which are in charge of summing those components to form $\hat{u}$. The final step, (14), just adds up the pieces to form $\hat{u}$, and the pieces of u are now distributed evenly through the PE's as was specified for the other routines.

4.5 Normalize

We need to compute $v = u/\|u\|$. The algorithm uses procedures already outlined. Let $\hat{v}$ and $\hat{u}$ represent elements $(k-1)\hat{n}+1$ through kn of v and u on PE k. Then the algorithm goes as follows:

(1) $\delta = u^T u$ $\boxed{\textbf{Dot}}$
(2) $\delta = \sqrt{\delta}$
(3) For $i = 1, \hat{n}$
(4) $\quad \hat{v}_i = \hat{u}_i/\delta$
(5) End i-Loop

Step (1) computes $u^T u$, and step (2) computes $\|u\|$ using the results of step (1). Steps (3) through (5) divide each piece of $\hat{u}$ by $\|u\|$ and puts the result into $\hat{v}$. Hence, the global vector v is the normalized vector u. Hence, the **Normalize** subroutine is the cost of one **Dot** routine plus $\hat{n}$ divisions.

4.6 Residual

The residual routine requires that we explicitly compute Av and the norm of $w - Av$. Hence, we will let $\hat{u}$, $\hat{v}$, and $\hat{w}$ represent elements $(k-1)\hat{n}+1$ through kn of u, v and w, respectively, on PE k. The algorithm goes as follows:

(1) $\hat{u} = Av$ $\boxed{\textbf{Mat-Vec Mult}}$
(2) For $i = 1, \hat{n}$
(3) $\quad \hat{u}_i = \hat{w}_i - \hat{u}_i$
(4) End i-Loop
(5) Compute $\delta = \hat{u}^T \hat{u}$ $\boxed{\textbf{Dot}}$
(6) Return $\sqrt{\delta}$

Step (1) computes $\hat{u}$ on each PE. Steps (2) through (4) compute the difference between $\hat{w}$ and $\hat{u}$, storing the result in $\hat{u}$. Hence, $u = w - u = w - Av$. Steps (5) and 6) compute $\|u\|$ using **Dot**. As we can see, the **Residual** subroutine has $\hat{n}$ additions as well as the cost of the **Dot** and **Mat-Vec Mult** routines.

5 Summary of Timings

Table 2 presents timings on the T3D of the CG Parallel Benchmark on 2^4 through 2^7 PE's in various configurations. The code was implemented as described in Sections 3 and 4. Each row contains the following information: number of PE's (p), dimensions of processor grid (p_h and p_w), performance of code in megaflops per second (MFlops), total time to execute where the timing start and stop are specified in Section 3.1 (Secs), and the percent of time spent on communications (% Comm). The percent of time spent on communications is measured only on PE 1 where the timing starts before the initial **Put** and ends after the **Barrier** returns. It is meant to give only a general idea of the amount of time spent on communications.

p	p_h	p_w	MFlops	Secs	% Comm
16*	2	8	79.64	18.83	2
16	4	4	72.87	20.59	1
16	8	2	67.03	22.38	2
32	2	16	154.43	9.71	10
32*	4	8	155.10	9.67	2
32	8	4	142.90	10.50	3
32	16	2	130.09	11.53	6
32	32	1	121.01	12.39	11
64	2	32	259.14	5.79	22
64*	4	16	306.96	4.88	10
64	8	8	300.55	4.99	4
64	16	4	279.31	5.39	7
64	32	2	248.32	6.04	11
64	64	1	225.98	6.64	19
128	4	32	484.50	3.10	22
128*	8	16	563.50	2.66	13
128	16	8	516.40	2.90	13
128	32	4	498.95	3.00	15

* Best result for each value of p.

Table 2: Benchmark Timings on the Cray T3D

Examining Table 2 we note that the best performance does not always correspond to the least communications as is frequently the case on other machines. In fact, we see that for 64 processors, the best time is achieved by

p	$p_h \times p_w$	Subroutine Name	Total Ticks	Percent of Time	Calls	Avg. Ticks Per Call
16	2 x 8	Accumulate	4171218	0	810	5149
		Mat-Vec Mult	2766377730	98	390	7093276
		Dot	19909833	1	795	25043
		Saxpy	30498160	1	1125	27109
		Residual	106963668	4	15	7130911
32	4 x 8	Accumulate	5073147	0	810	6263
		Mat-Vec Mult	1413763494	97	390	3625034
		Dot	13003979	1	795	16357
		Saxpy	15352961	1	1125	13647
		Residual	54795423	4	15	3653028
64	4 x 16	Accumulate	5859420	1	810	7233
		Mat-Vec Mult	626033167	85	390	1605213
		Dot	9937444	1	795	12499
		Saxpy	7664343	1	1125	6812
		Residual	27739521	4	15	1849301
128	8 x 16	Accumulate	6861600	2	810	8471
		Mat-Vec Mult	375061396	94	390	961695
		Dot	9016210	2	795	11341
		Saxpy	3978993	1	1125	3537
		Residual	14687532	4	15	979168

Table 3: Subroutine Timings

the 4×16 layout which has 10% of its time spent in communications. As we shall later observe, computations can be a big bottleneck on the T3D - due largely to memory access delays explained in Section 2.2. Looking at the best results, we can compute that the MFlops per PE hovers between 4.4 and 5, getting worse as p grows. This is well below an individual processor's peak performance which is over 150 MFlops.

Table 3 gives a breakdown of the timings into the various subroutines: **Accumulate**, **Mat-Vec Mult**, **Dot**, **Saxpy,** and **Residual**. For each value of p, we have chosen the case with the best overall time shown in Table 2. Please note that these subroutines do call each other; for example, **Dot** calls **Accumulate**. "Ticks" is the total number of clock ticks spent in the subroutine on PE 1. We use PE 1 for all timings because using any other PE would yield similar results. We also give the percentage of the total time spent in the given subroutine (Percent of Time), the number of calls to the subroutine (Calls), and the average number of ticks per call (Avg. Ticks Per Call).

Table 3 shows the matrix-vector multiply routine is by far the most expensive, taking between 85 and 98 percent of the total time. As discussed before, we have poor caching performance in this routine because we randomly access the vector $\bar{v}$ when computing $\bar{A}\bar{v}$ on each processor. In Table 5 we examine the speed-ups as we increase the number of PE's.

Let us first examine the **Mat-Vec Mult** Routine in more detail. Table 4 gives the breakdown of **Mat-Vec Mult** for various values of p corresponding to the configurations in Table 3. The routine is broken down into four basic parts with the corresponding algorithm step numbers listed. (The algorithm is given in Section 4.4.) Each piece is called 390 times, and the "Avg. Ticks per Call" is based on this number. The "Percent of Time" gives the percent of total time spent in this section of code.

p	$p_h \times p_w$	Step(s) of **Mat-Vec Mult**	Total Ticks	Percent of Time	Avg. Ticks Per Call
16	2x8	Concatenate, 1-2	15004431	1	38472
		Local Multiply, 3-8	2652885855	94	6802271
		Scatter, 10-12	41451678	1	106268
		Add Pieces, 14	56325153	2	144423
32	4x8	Concatenate, 1-2	15186453	1	38939
		Local Multiply,3-8	1349871903	93	3461210
		Scatter,10-12	21592497	1	55365
		Add Pieces, 14	26394507	2	67678
64	4x16	Concatenate,1-2	8766255	1	22477
		Local Multiply,3-8	626033167	85	1605213
		Scatter,10-12	55081662	8	141235
		Add Pieces,14	22533699	3	57778
128	8x16	Concatenate,1-2	11369712	3	29153
		Local Multiply,3-8	319198700	80	818458
		Scatter,10-12	32487237	8	83301
		Add Pieces,14	11389872	3	29204

Table 4: **Mat-Vec Mult** Routine Breakdown

We see that the local matrix-vector multiply is the most expensive operation that we do, taking between 80 and 94 percent of the total time. Recall that this part requires no communications. The performance penalty is primarily due to memory delays since we must randomly access the vector $\bar{v}$ to compute $\bar{A}\bar{v}$. Step 14 is also somewhat expensive (considering that we are only adding some numbers), but this phenomena may also be due to the fact that the pieces being added are not near each other in memory. Notice how the times for "Concatenate" and "Add Pieces" vary - this is because different layouts and sizes have an effect on the communications. We will not explore this issue in depth since the communication cost is such a small factor in the overall program and we expect that optimized routines will eventually be provided by Cray to do the types of communications we are doing.

Table 5 shows the speed-up achieved in the entire program and in some of the subroutines each time we double the number of processors. In this case, speed-up from $x \to y$ processors is defined as

$$\frac{\text{Time on } y \text{ processors}}{\text{Time on } x \text{ processors}}.$$

Subroutine Name	Change in p		
	$16 \to 32$	$32 \to 64$	$64 \to 128$
Entire Program	1.95	1.98	1.83
Accumulate	.82	.87	.85
Mat-Vec Mult	1.96	2.25	1.67
Dot	1.53	1.31	1.10
Saxpy	1.99	1.98	1.89
Residual	1.95	1.98	1.89

Table 5: **Speed-Up as the Number of Processors Increases**

Our goal is to double the speed of the code each time we double the number of processors. Because of Amdahl's Law (see [3]), we know that we can not achieve an exact doubling, but our numbers come close when changing between 16 and 32 PE's, and between 32 and 64 PE's. Since the **Accumulate** routine involves communications, we do not expect the time to get any better, but in fact to worsen. The **Mat-Vec Mult** routine demonstrates very strange behavior – this is directly influenced by the cost of the communications in that routine, not the computations. The **Dot** product also did not do very well, but this is because it involves communications. **Saxpy** had very good speed-up because it does not have any communications, and the amount of work it does directly scales with the number of processors. Despite the fact that **Residual** calls **Mat-Vec Mult**, it still scales well.

6 Conclusions

Our code achieves 306 MFlops on 64 processors. In comparison, a Cray YMP single head gets 127 MFlops, a 128 processor iPSC/860 gets 181 MFlops, and a 32,768 processor CM-2 gets 105 MFlops [5]. This performance, however, is not indicative of the peak performance of the Cray T3D. In fact, the local computations impede our performance a great deal due to memory access delays. An analysis of sparse vector operations on cache-based parallel systems is something which may be worth pursuing. We have also observed that different layouts of the processor grid are worth exploring because it's not always clear which layout will give the best performance. In our case, this is due primarily to communications cost and the **Mat-Vec Mult** routine (which has communications and computations that are directly affected by the layout of the grid). In other cases, however, we may want to break up the matrix into rectangular blocks because of the matrix structure as well.

References

[1] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, et al. The NAS parallel benchmarks. Technical report, NASA Ames Research Center, March 1994.

[2] Ray Barriuso and Allan Knies. Shmem user's guide. Technical report, Cray Research, Inc., Eagan, MN, 1994.

[3] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice Hall, Englewood Cliffs, NJ 07632, 1989.

[4] Gene H. Golub and Charles F. van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, 2nd edition, 1989.

[5] John G. Lewis and Robert A. van de Geijn. Distributed memory matrix-vector multiplication and conjugate gradient algorithms. In *Supercomputing '93*, Los Alamos, CA, 1993. IEEE Computer Society Press.

[6] Robert W. Numrich, Paul L. Springer, and John C. Peterson. Measurement of communication rates on the Cray T3D interprocessor network. In *Proceedings HPCN Europe '94*. Springer-Verlag, 1994.