# Blocked Data Distribution for the Conjugate Gradient Algorithm on the CRAY T3D*

Michael W. Berry† Charles Grassl‡ Vijay K. Krishna§

**Abstract**

In this paper, we present a sparse matrix-vector multiplication algorithm for massively-parallel computers such as the CRAY T3D. Performance results on a 256-processor CRAY T3D are presented along with a detailed analysis of the algorithm's computational complexity. The specific sparse matrix-vector multiplication algorithms discussed include the block-block algorithm (BBA) for implementation on the CRAY T3D, and the block row algorithm (BRA) for comparative results on the CRAY C90. The performance of these algorithms, when used with in the Conjugate Gradient kernel from the NAS Parallel Benchmarks, is presented for both the CRAY T3D and a 16-CPU CRAY C90. Results of this study demonstrate that the Conjugate Gradient benchmark for the class A problem size (matrix order $14,000$) can be executed in 1.3 seconds on a 256-processor CRAY T3D, which is quite competitive with published results for this benchmark on other massively-parallel machines.

## 1 Introduction

Sparse matrix-vector multiplication kernels are commonly used within iterative methods for solving large, sparse systems of equations and computing several of the largest singular values, or eigenvalues, of sparse matrices. The singular value decomposition of sparse matrices, in particular, is needed in applications such as information retrieval, seismic reflection tomography, and real-time signal processing [3]. Efficient parallel algorithms are especially needed to perform the matrix-vector multiplication operations associated with iterative methods on massively-parallel processors (MPP's). These MPP's may consist of tens to hundreds of processors each with its own local memory and a fast interconnect for

*Submitted to *Parallel Computing*.

†Department of Computer Science, 107 Ayres Hall, University of Tennessee, Knoxville, TN 37996-1301, berry@cs.utk.edu, fax number 615-974-4404

‡Cray Research Inc., Benchmarking, 655F Lone Oak Drive, Eagan, MN 55121, cmg@ferrari.cray.com

§Cray Research Inc., Network Media Section, 655F Lone Oak Drive, Eagan, MN 55121, krishna@ironwood.cray.com

movement of control information and data.

The main objective of this work is the design analysis of sparse matrix-vector multiplication algorithms suitable for a state-of-the-art massively-parallel computer system such as the CRAY T3D from Cray Research Inc. The first algorithm developed is similar to the one described in [6], but with extensions to handle a non-square number of processors. Other algorithms developed specifically address multiplication by over- or under-determined (rectangular) matrices. The application of these algorithms within real applications as well as a theoretical analysis of computational complexity are considered. The organization of the paper is provided below.

In Section 2 we discuss the hardware characteristics of the CRAY T3D including details of the processors used and the interconnection topology [4]. In Section 3 we state the various sparse matrix-vector multiplication operations for which algorithms are developed and then discuss each algorithm in detail. In Section 4 we discuss the performance of the sparse matrix-vector multiplication algorithms when they are used within the Conjugate Gradient kernel from the NAS Parallel Benchmarks, and we summarize and note future work in enhancing the performance of the sparse matrix-vector multiplication algorithms in Section 5.

## 2 Introduction to the CRAY T3D

The CRAY T3D is an MIMD (Multiple Instruction Multiple Data) machine comprised of up to 2048 DEC Alpha processors with a fast interconnect network for movement of control information and data. The memory is physically-distributed but is logically-shared [4].

### 2.1 Microprocessor Architecture and Network Topology

The peak speed of the DEC Alpha processor used in the CRAY T3D is 150 megaflop/s (millions of floating-point operations/second). The frequency of the clock in the processors is 150 megahertz. The Alpha is considered a *superscalar* processor capable of issuing two instructions (integer and/or floating-point) per clock cycle and a *superpipelined* processor with multiple segment functional units. The Alpha has 32 integer and 32 floating-point registers, an 8-kbyte on-chip data cache and an 8-kbyte instruction cache; it performs 64-

bit IEEE arithmetic. The particular machine used in this work has 256 processors, each processor having 2 MW (mega words) of memory and a total peak performance of about 38 gigaflop/s (billions of floating-point operations per second).

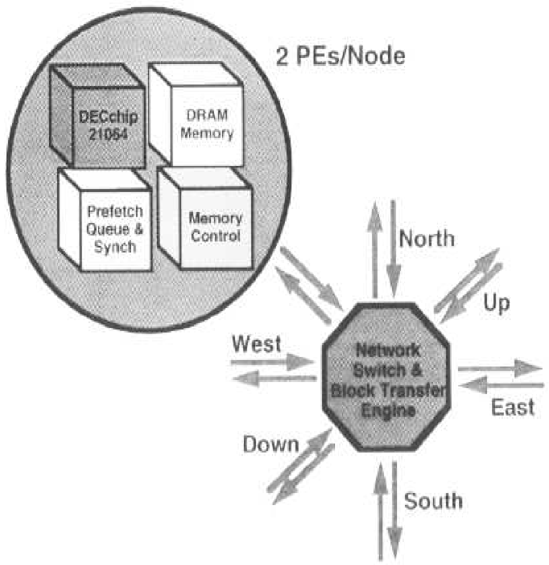


FIG. 1. *Node architecture of the CRAY T3D.*

The topology of the CRAY T3D is a 3-dimensional torus with 2 processing elements per node (see Figure 1). Each processing element (PE) can contain 2 or 8 MW of DRAM. It has redundant nodes to replace defective nodes. Each node has latency hiding hardware (Block Transfer Engine) which can transfer data without interrupting a PE. A peak rate of 300 MB/sec transfer rate is available in each of the 6 directions in which data can travel from a node (Figure 2). Data is sent in the form of packets, with each packet containing a header with destination information followed by data.

The CRAY T3D has a front-end UNICOS host system (CRAY Y-MP) through which all I/O activities (data and control) are performed. I/O communication between the T3D and the UNICOS host is done through CRAY T3D I/O Gateways (IOGs).

### 2.2 CRAY T3D Software Overview

The operating system of the CRAY T3D is UNICOS MAX. It has a microkernel running on each PE which handles processor management, memory management and processor-to-processor communication. There is a UNIX agent running on the host system which

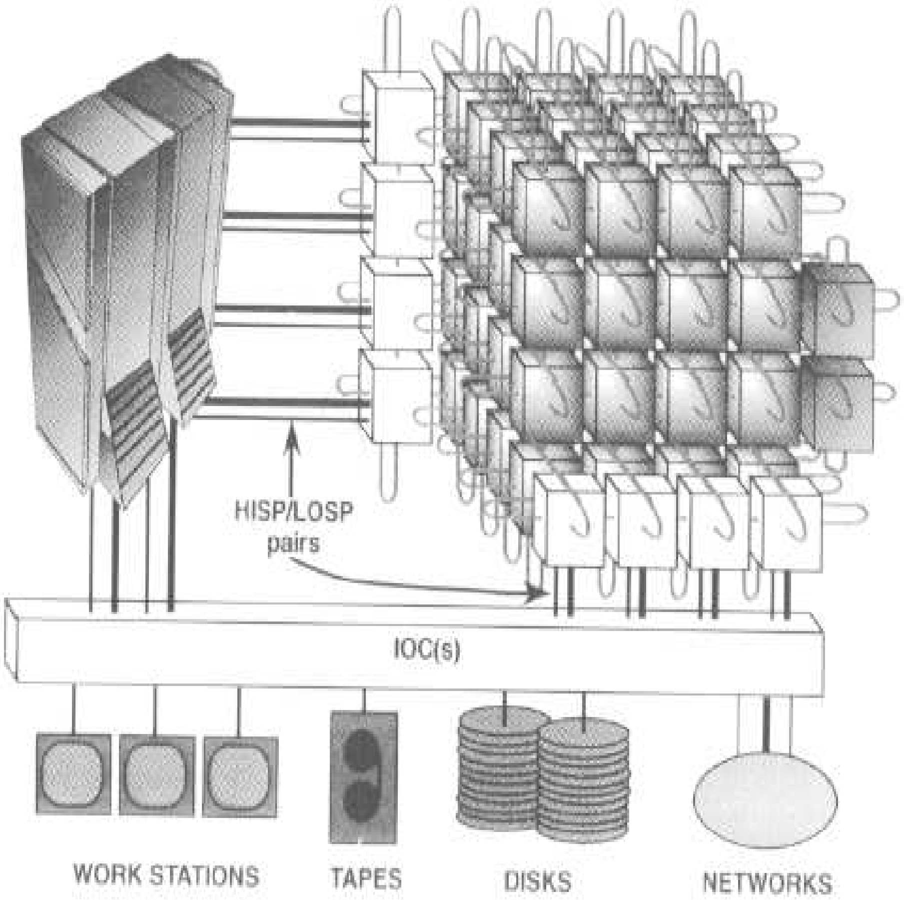


FIG. 2. *System configuration and interconnection topology of the CRAY T3D.*

handles T3D requests. There are enhanced accounting and new administrative commands. The front-end runs the UNICOS operating system.

The CRAY T3D programming environment provides a graphical performance analyzer, a graphical debugging tool, compilers, and libraries. The CRAY T3D has Fortran and C compilers along with a high-level parallel programming language called Cray Fortran Programming Model (CRAFT) and message passing libraries. The compilation of programs is done on the host system. The Cray Fortran Programming model provides a global shared address space. CRAFT is based on the sharing of data and work between processing elements. Data is shared among the processors and each processor works on the data present in its local memory. The CRAY T3D native debugger is *TotalView*. The performance monitoring and instrumenting tool is *Apprentice*.

The CRAY T3D also utilizes various standard libraries along with highly-optimized scientific library routines. Message-passing support is provided through the SMA Communication Library [7] and PVM (Parallel Virtual Machine) [10]. The PVM (Version 3.0) message-passing library is available on the CRAY T3D. PVM message passing can be used in either homogeneous or heterogeneous manner. A shared-memory access (SMA) library is available for performing direct memory access to remote PE's.

## 3 Parallel Matrix-Vector Multiplication Algorithms

Let $A$ be an $m \times n$ sparse matrix, $x$ be an $n \times 1$ dense vector and $y$ an $m \times 1$ dense vector. The target operation for algorithmic development is the sparse matrix-vector multiplication $y = Ax$. We assume that the nonzeros of the sparse matrix $A$ are stored in some compressed form in order to conserve memory and to avoid redundant computations on explicit zeros. Specifically, compressed sparse row and column formats (CSR and CSC from [2]) were used in our experiments.

### 3.1 Block-Block Algorithm (BBA)

The sparse matrix-vector multiplication algorithm considered uses a block-block decomposition quite suitable for iterative methods, in which the result vector $y$ has to be distributed in the same fashion as the input vector $x$. This is similar to the algorithm described in [6], but with extensions to handle a non-square number of processors. However, this algorithm for computing $y = Ax$ does assume that the number of processors is a power of 2.

**3.1.1 Block-Block Domain Decomposition.** For simplicity, we consider a sparse square matrix $A$ of order $n$, which is divided into a $npy \times npx$ processor grid ($npy \geq npx$) with each processor being assigned a block of the matrix $A$ with dimensions $(n/npy) \times (n/npx)$. Figure 3 illustrates the distribution of data for a $4 \times 4$ processor grid. Each block of the matrix assigned to a processor is stored in compressed-row (CSR) or compressed-column (CSC) format. BBA is independent of the data structure used to represent the blocks of the matrix $A$ as illustrated in Figure 3. This algorithm is well suited for iterative methods (see Table 1) in which the output vector $y_i$ is the input vector $x_{i+1}$ for the next iterate. Hence, at the end of the matrix-vector multiplication operation processor $P_{\alpha\beta}$ must

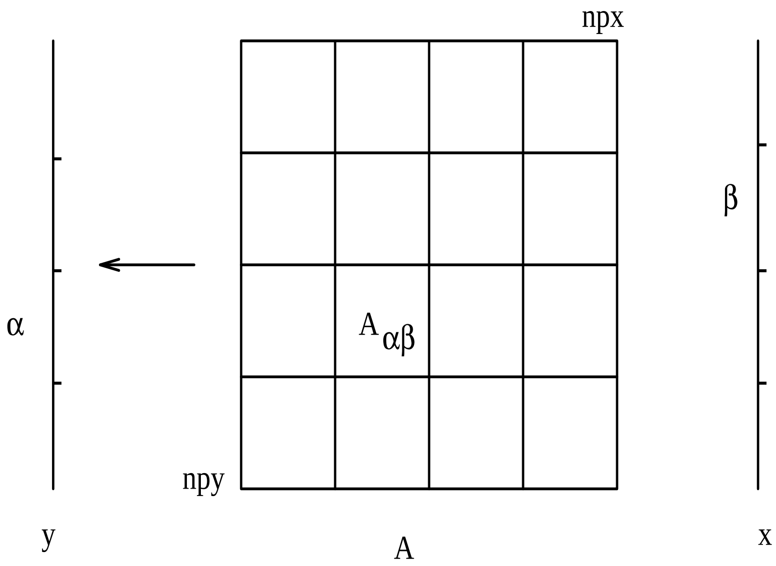


FIG. 3. *Distribution of data for npy = npx = 4 with indices $\alpha$=3 and $\beta$=2.*

contain $y_\beta$.

TABLE 1
*An iterative framework for using sparse matrix-vector multiplication.*

**For** $i = 1, \ldots$

$y_i = Ax_i$

$\vdots$

$x_{i+1} = y_i$

$\vdots$

**EndFor**

**3.1.2 Communication Primitives.** BBA requires three kinds of communication primitives. The first primitive adds vectors present in different processors in each row and is called a **fold operation** [5, 6]. As seen from Figure 3 processor, $P_{\alpha\beta}$ owns a block of the matrix $A_{\alpha\beta}$ and vector $x_\beta$. If $z_{\alpha\beta} = A_{\alpha\beta}x_\beta$, then

$$y_\alpha = z_{\alpha 1} + z_{\alpha 2} + \cdots + z_{\alpha,npx}.$$

The fold operation is used for adding vectors $z_{\alpha\beta}$ of length $n/npy$ present in $npx$ processors having the same block row index $\alpha$. At the end of this operation (see Table 2), each processor $P_{\alpha\beta}$ has a vector $y_{\alpha\beta}$ of length $\frac{n}{(npx*npy)}$ which is a partition of the vector $y_{\alpha}$. The algorithm, which requires $log_2 npx$ steps, to accomplish this is *recursive halving.* As outlined in Table 2, at each step a pair of processors divide the vector $z$ into two halves and carry out the add operation on opposite halves. In Step 3) of Table 2, each processor exchanges data with a processor chosen from a set of processors in increasing powers of 2 by flipping bits from the left in the binary representation of the processor number. Figure 4 illustrates the fold operation in a row of 4 processors. The total number of elements of $z_{\alpha\beta}$ in each processor at the start of this operation is $n/npy$ and the total number of elements in the result $y_{\alpha\beta}$ in each processor at the end of this operation is $n/(npx * npy)$. Therefore, the total number of data elements that are sent (received) by a processor in the fold operation is given by ($\frac{n}{npy}$ - $\frac{n}{(npx*npy)}$). The total number of elements sent (received) by a processor can also be written as a geometric series with $log_2 npx$ terms

$$TimeFold_{comm} = \sum_{k=1}^{log_2 npx} \frac{n}{2^k * npy} = \frac{n}{npy}\left(1 - \frac{1}{npx}\right). \tag{1}$$

The **Put** primitive is used to copy blocks of data directly from one processor's memory to another processor's memory, while the **Barrier** call is used for synchronization, i.e, a processor is forced to wait until all other processors have executed the Barrier instruction. These communication primitives are provided by SMA Communication Library [7].

After the fold operation, processors in each row have data that must be shared by processors in a corresponding column to prepare for the next matrix-vector multiplication. This is achieved by a **transpose operation** in which processors in a row send data to corresponding processors in a column (see Table 3). In this communication primitive, processor $P_{\alpha\beta}$ exchanges $y_{\alpha\beta}$ of length $n/(npx * npy)$ vector with processor $P_{new\alpha,new\beta}$, where

$$new\alpha = \beta + (\alpha\textbf{.and.}(k-1)) * npx,\ k = npy/npx,\ \text{and}\ new\beta = \alpha/k.$$

As seen in Figure 5, on square processor grids processor $P_{\alpha\beta}$ communicates with processor $P_{\beta\alpha}$. An example of data movement in transpose operation for non-square processor grids is shown in Figure 6. The total number of data elements that are sent (received) by a

TABLE 2

*Fold Operation for processor $P_{\alpha\beta}$ in row $\alpha$.*

Processor $P_{\alpha\beta}$ contains $z_{\alpha\beta} \in \Re^{n/npy}$, and $w_1$, $w_2$, $z_1$ and $z_2 \in \Re^{n/npx}$ are temporary arrays.

**For** $i = 0, ..., (log_2\, npx) - 1$
    Partition $z_{\alpha\beta} = (z_1^T \mid z_2^T)^T$
    $P_{\alpha\beta'} = P_{\alpha\beta}$ with $i^{th}$ bit of $\beta$ flipped
    **If** bit $i$ of $\beta$ is 1 **then**
      **Put** $z_1$ into $w_1$ of processor $P_{\alpha\beta'}$
      Barrier()
      $z_{\alpha\beta} = z_2 + w_2$
    **Else**
      **Put** $z_2$ into $w_2$ of processor $P_{\alpha\beta'}$
      Barrier()
      $z_{\alpha\beta} = z_1 + w_1$
    **Endif**
**EndFor**
$y_{\alpha\beta} = z_{\alpha\beta}$

Processor $P_{\alpha\beta}$ now contains $y_{\alpha\beta} \in \Re^{n/(npx*npy)}$

processor in the transpose operation is given by $\frac{n}{(npx*npy)}$ so that

$$TimeTranspose_{comm} = \frac{n}{(npx*npy)} .$$

TABLE 3

*Transpose operation for processor $P_{\alpha\beta}$.*

Processor $P_{\alpha\beta}$ contains $y_{\alpha\beta} \in \Re^{n/(npx*npy)}$
$k = npy/npx$
$new\alpha = \beta + (\alpha.\mathbf{and}.(k-1)) * npx$
$new\beta = \alpha/k$
**Put** $y_{\alpha\beta}$ into $z_{\alpha\beta}$ of processor $P_{new\alpha,new\beta}$
**Barrier()**
$y_{\alpha\beta} = z_{\alpha\beta}$

The **expand operation** combines vectors from different processors to obtain a single vector which is duplicated on all the processors. After the transpose operation, each processor in a column has data which must be shared with other processors in the same column. A vector $y_{\alpha\beta}$ of length $n/(npx*npy)$ from each processor is combined to form a vector $y_\alpha$ of length $n/npx$ on all processors in the column.

The exchange of data between processors in the expand operation is the inverse communication pattern of the fold operation as seen in Figure 7 and Table 4. The number

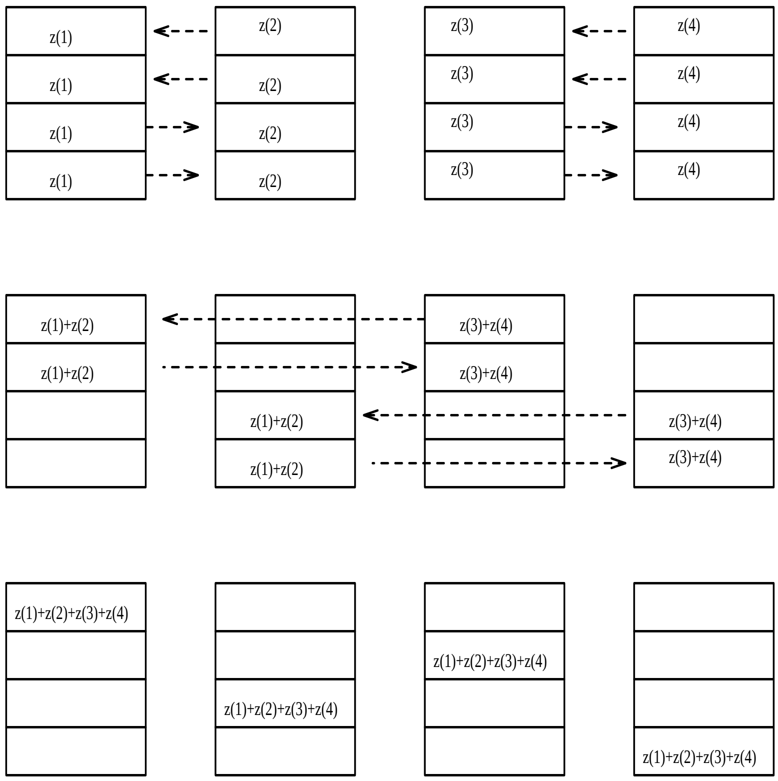


FIG. 4. *Fold operation with 4 processors in a row ($npx = 4$), each having a vector $z(i)$ of length $n/4$. At the end of the operation each processor has a subvector of length $n/16$ containing a unique piece of the result vector.*

of elements sent (received) is equal to the fold operation and from Equation (1) is given by $(\frac{n}{npx} - \frac{n}{(npx*npy)})$. For a non-square processor grid, an additional $log_2(\frac{npy}{npx})$ steps are required in which $\frac{n}{npx}$ number of values are exchanged per step. The total number of data elements that are sent (received) by a processor in the expand operation is given by

$$(2) \qquad TimeExpand_{comm} = \frac{n}{npx} - \frac{n}{(npx*npy)} + log_2\left(\frac{npy}{npx}\right) * \left(\frac{n}{npx}\right) .$$

**3.1.3 Parallel Matrix-Vector Multiplication Algorithm.** In the first step of the parallel matrix vector algorithm shown in Table 5, a local matrix vector product is computed on each processor with a block of the matrix $A_{\alpha\beta}$ and $x_\beta$ to obtain $z_{\alpha\beta}$. Using the fold operation illustrated in Table 2 and Figure 4, we add vectors $z_{\alpha\beta}$ from processors within rows to get $y_{\alpha\beta}$ which is a subvector of the vector $y_\alpha$. If the next matrix-vector multiplication on processor $P_{\alpha\beta}$ requires $y_\beta$ (see Table 1), the transpose operation can be used to copy subvectors of $y_\beta$ into vector $w$ of the processors in column $\beta$ (see Table 3, Figure 5, and

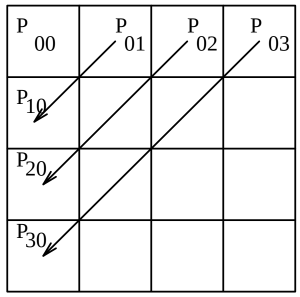


FIG. 5. *Transpose operation for a processor grid with $npx = npy = 4$. The movement of data is shown for the first row of processors.*

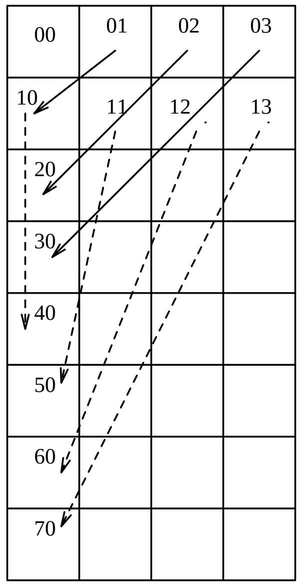


FIG. 6. *Transpose operation for a processor grid with $npy = 8$ and $npx = 4$. Data movement is shown for the first and second row of processors.*

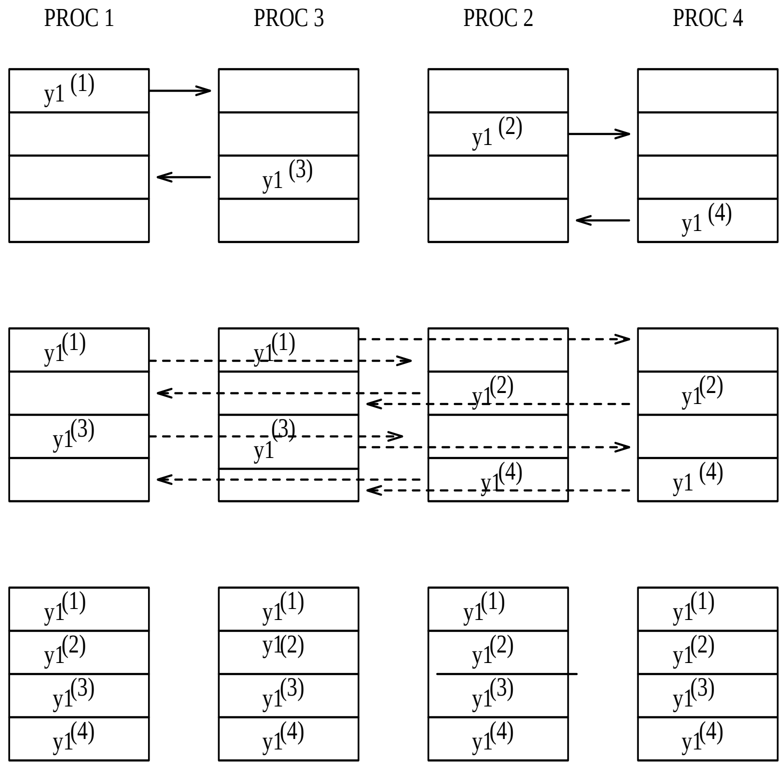


FIG. 7. *Expand operation for npy=4. Each processor in a column contains a vector* $y1^{(i)}$ *of length* $n/16$. *At the end of the operation each processor has the complete vector* $y1$ *of length* $n/4$.

Figure 6). Finally, the expand operation shown in Figure 7 and Table 4 combines the subvectors $w$ and places the complete vector $y_\beta$ in each processor of column $\beta$.

TABLE 4
*Expand operation for processor $P_{\alpha\beta}$ in column $\beta$.*

```
Processor P_αβ contains y_αβ ∈ ℜ^{n/(npx*npy)}
For i = (log_2 npx) − 1, ..., 0
    P_α',β = P_αβ with i^th bit of α flipped
    Put y_αβ into w of processor P_α',β
    If bit i of α is 1 Then
      Prepend w so that y_αβ^T = (w^T y_αβ^T)
    Else
      Append w so that y_αβ^T = (y_αβ^T w^T)
    Endif
Endfor
If (npx.ne.npy) Then
      For i = 1, ..., log_2(npy/npx)
       newα = α.xor.(npx * i)
       Put y_αβ into w of processor P_newα,β
       y_αβ = y_αβ + w
      Endfor
Endif
```

TABLE 5
*Parallel matrix vector multiplication algorithm.*

| |
|---|
| $w, z \in \Re^{n/npy}$ are temporary arrays. |
| 1) Compute $z_{\alpha\beta} = A_{\alpha\beta} x_\beta$. |
| 2) Perform fold operation on $z_{\alpha\beta}$ within rows to form $y_{\alpha\beta}$. |
| 3) Transpose operation to put $y_{\alpha\beta}$ into $w$ of Processor $P_{new\alpha, new\beta}$ where $new\alpha$ and $new\beta$ are defined in Figure 4. |
| 4) Expand $w$ to obtain $y_\beta$. |

**3.1.4 Complexity Model.** The complexity of the algorithm shown in Table 5 can be represented by the total times for startup, communication, and arithmetic operations. The maximum number of floating-point operations per processor is given by $(2*nnz-n)/p$ where $nnz$ is the total number of nonzeros in the sparse matrix, $n$ is the order of the sparse matrix and $p$ is the number of processors. From Equation (1) we know that the total number of steps in the fold operation is $log_2\, npx$, and from Equation (2) the total number of steps in the expand operation is $log_2\, npx + log_2(\frac{npy}{npx})$. Only one message is initiated in the transpose operation. Therefore, the total startup cost for sending messages is

$$Time_{startup} = \left(2*log_2\, npx + log_2\left(\frac{npy}{npx}\right)+1\right)*t_{startup},$$

where $t_{startup}$ is the amount of time taken to initiate a **put** operation. The total communication time is given by

$$\begin{aligned}
Time_{comm} &= TimeFold_{comm} + TimeTranspose_{comm} + TimeExpand_{comm}\\
&= \frac{n}{npy} - \frac{n}{npx*npy} + \frac{n}{(npx*npy)} + \frac{n}{npx}\\
&\quad -\frac{n}{(npx*npy)} + log_2\left(\frac{npy}{npx}\right)*\left(\frac{n}{npx}\right)\\
&= \left(\frac{n}{npy}\right)+\left(\frac{n}{npx}\right) - \frac{n}{npx*npy} + log_2\left(\frac{npy}{npx}\right)*\left(\frac{n}{npx}\right)\\
&= \left(\frac{n}{npy}\right)+\frac{n}{npx}\left(1-\frac{1}{npy}+log_2\left(\frac{npy}{npx}\right)\right).
\end{aligned} \tag{3}$$

The total run time is given by

$$TotalTime = \left(\frac{2*nnz-n}{p}\right)*T_{flop} + Time_{startup} + Time_{comm},$$

where $T_{flop}$ is the time taken by a single floating point operation. For simplicity, we consider the scalability of the algorithm in Table 5 for a square grid of $p$ processors where $npx = npy = \sqrt{p}$. For this grid, the total communication time from Equation (3) is given by

$$Time_{comm} = \frac{n}{\sqrt{p}}\left(2-\frac{1}{\sqrt{p}}\right) = n*\left(\frac{2*\sqrt{p}-1}{p}\right).$$

From the above equation the communication time is directly proportional to $n$ and inversely proportional to the square root of the number of processors $p$ so that the communication complexity of BBA is given by

$$n * \left( \frac{2 * \sqrt{p} - 1}{p} \right) .$$

Hence, the block-block algorithm scales very well as the number of processors is increased (see Figure 8). Scaling refers to the decrease in the communication time as the number of processors increases, which results in overall better performance and a higher computation-to-communication ratio as $p$ approaches infinity.

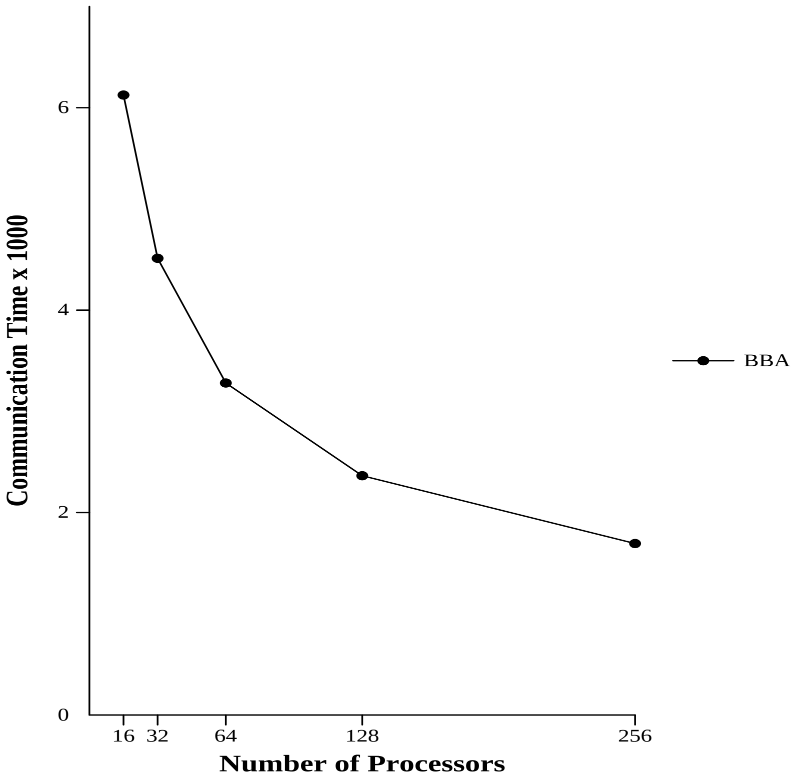


FIG. 8. *Communication time versus the number of processors used for BBA. The order of the sparse matrix (n) is fixed at* 14,000.

We particularly note that for BBA each processor only has that part of the vector corresponding to the rows in its local block ($x_\beta$). A larger portion of the multiplied vector could be *cached* in the local processor's data cache. This effect is measurable for smaller problems, but is not likely to be significant for larger problems where the size of the local vector is large compared to the size of the data cache.

### 3.2 Block Row Algorithm (BRA)

An alternate algorithm which can be used to compute $y = Ax$ divides the sparse matrix $A$ into row blocks[1] so that each processor gets one row block (see Figure 9). This particular method was used on a 16-CPU CRAY C90 for comparison purposes. Here, each row block of the sparse matrix is stored in compressed column format and processed by a unique CPU, i.e., the number of row blocks is equal to the number of CPU's. The parallel matrix-vector multiplication algorithm given in Table 6 ensures each CPU $P_i$ will compute $y_i$. Whereas on the CRAY T3D each processor $P_i$ can initially store $x_i$ and use a comparable expand operation on $p$ processors to acquire the entire $x$ vector [8], on the CRAY C90 $x$ is simply shared among the 16 CPU's. The number of floating-point operations per CPU is equal to $(\frac{(2*nnz-n)}{p}) * T_{flop}$, where $T_{flop}$ is the cost of a single floating-point operation.

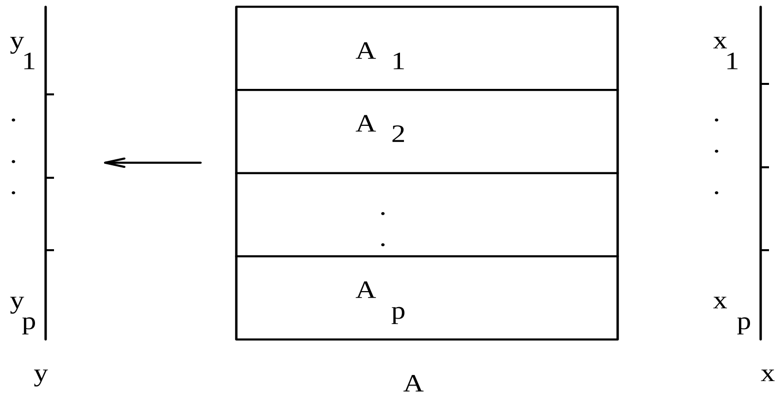


FIG. 9. *Domain decomposition of data for computing $y = Ax$.*

TABLE 6
*Block row matrix vector multiplication algorithm for computing $y = Ax$.*

| |
|---|
| 1) Assign $A_i$ and $x_i$ to processor $P_i$. |
| 2) Perform expand operation on $x_i$ within $p$ processors to obtain $x$ on each node. |
| 3) Compute $y_i = A_i x$ |

## 4 Performance of NAS Conjugate Gradient Benchmark

The BBA algorithm described in the previous Section is well-suited for massively-parallel systems such as the CRAY T3D. Although several key features of the CRAY T3D have

[1] A comparable Block Column Algorithm (BCA) which partitions the matrix $A$ into column blocks is another alternative, of course.

been exploited, the most important feature is the communication bandwidth for exchanging data between processors. The **put** communication primitive allows any processor to write data directly to the memory of any other processor. The processor-to-processor bandwidth of this primitive is approximately 130 Mbytes/second [9]. Another key feature of the CRAY T3D is that from any processor, it takes almost the same amount of time to write to farthest processor as it takes to write to a nearest neighbor.

Both the Conjugate Gradient kernel and BBA (discussed in Section 3) have been implemented in `Fortran 77` with communication between processors accomplished using message-passing. Message-passing was accomplished using routines `shmem_put` and `shmem_get` from the SMA Communication Library [7]. No assembly language optimization was applied.

TABLE 7
*Conjugate Gradient algorithm.*

| | |
|---|---|
| (1) | $x = 0$ |
| (2) | $r = b$ |
| (3) | $p = b$ |
| (4) | $\rho = r^T r$ |
| (5) | **For** $i = 1, \ldots$ |
| (6) | $\quad y = Ap$ |
| (7) | $\quad \gamma = p^T y$ |
| (8) | $\quad \alpha = \rho / \gamma$ |
| (9) | $\quad x = x + \alpha p$ |
| (10) | $\quad r = r - \alpha y$ |
| (11) | $\quad \rho' = r^T r$ |
| (12) | $\quad \beta = \rho' / \rho$ |
| (13) | $\quad \rho = \rho'$ |
| (14) | $\quad p = r + \beta p$ |
| (15) | **EndFor** |

The classical Conjugate Gradient method is outlined in Table 7. It mainly consists of sparse matrix vector multiplication, vector updates (saxpy's) and inner-product kernels. BBA has been implemented for the sparse matrix-vector multiplication computed in Step (6) of Table 7. The sparse symmetric positive definite matrix generated in the NAS Conjugate Gradient benchmark [1] has a random structure or pattern of non-zeros ($1,853,104$ in total) with full main-diagonal sparsity. However, for BBA the sparse input matrix was assumed to be unstructured.

For benchmarking purposes, two different problem sizes and known solutions are available. The two different size problems, called the class A and class B sizes, respectively compute the smallest eigenvector of matrices of order $14,000$ and $75,000$. For each problem, the given matrix is very sparse, with the class A problem being $0.1\%$ dense and the class B problem being $0.25\%$ dense.

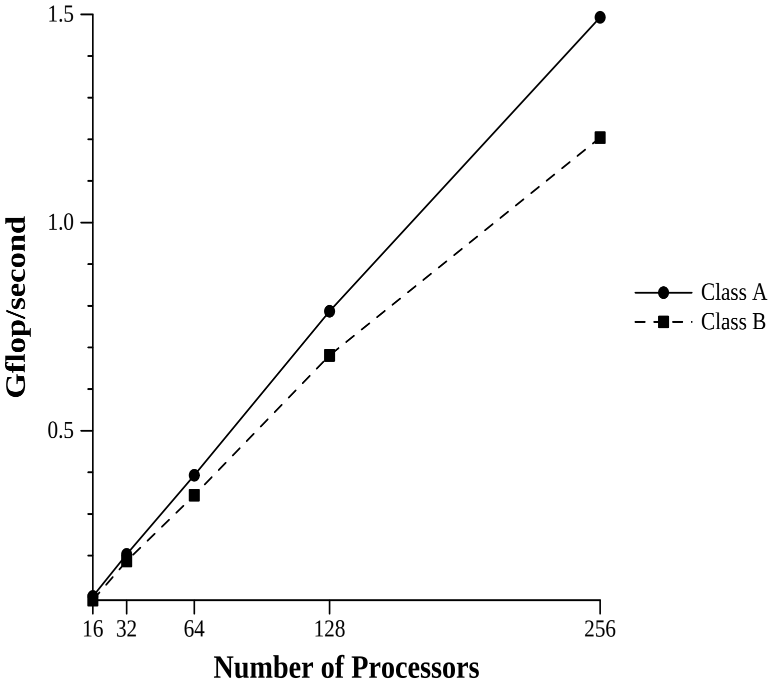


FIG. 10. *Performance in gigaflop/s (billions of floating-point operations per second) for the local sparse matrix-vector multiplication operation in BBA versus the number of processors used.*

Figure 10 illustrates the performance of the local sparse matrix-vector multiplication operation in BBA for class A and class B problems. With 256 processors, this operation yields a performance of 1.5 gigaflop/s for the class A problem. Note that the routine runs faster on the class A size problem. This can be attributed to the sparsity of the class B problem. The performance was calculated by dividing the total CPU time in seconds into the total number of floating-point operations. Figure 11 shows the performance of the Conjugate Gradient kernel in gigaflop/s. Notice that with 256 processors we obtain about 1.1 gigaflop/s for the class A problem.

Time for the expand, transpose and fold communication operations used in BBA for class A and class B problems are given in Tables 8 and 9, respectively. The total

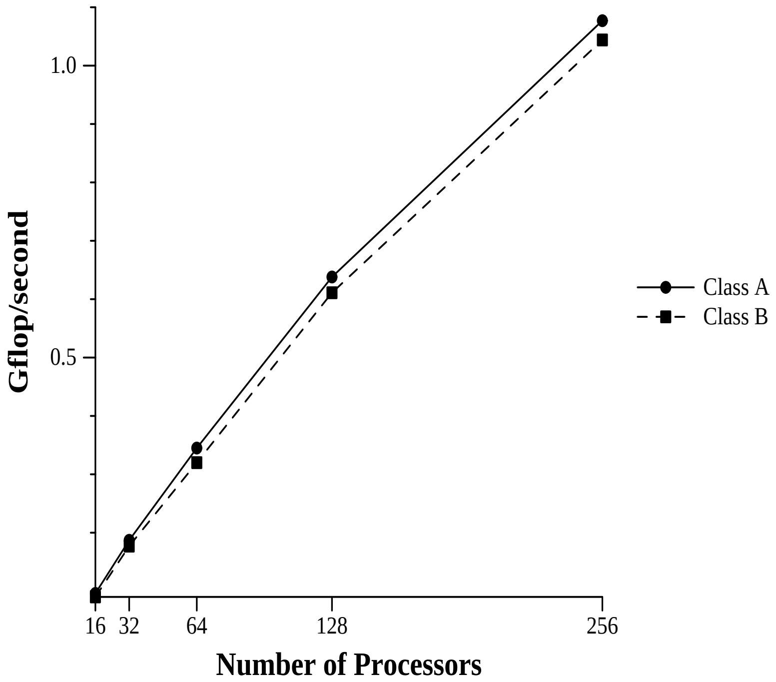


FIG. 11. *Performance in gigaflop/s as the number of processors used is increased for CG benchmark on Class A and Class B problems.*

communication time in BBA along with the total user CPU time for CG is also given. Clearly the communication time decreases as the number of processors is increased. This agrees with our prediction in the complexity analysis of communication time for BBA in Section 3.

TABLE 8
*Time in seconds for communication primitives for Class A problem in BBA.*

| $PE's$ | $TRANSP$ | $EXPAND$ | $FOLD$ | $TotalCommTime$ | $TotalUserTime$ |
|---|---|---|---|---|---|
| 16 | 0.071 | 0.182 | 0.313 | 0.566 | 15.6 |
| 32 | 0.054 | 0.225 | 0.173 | 0.452 | 8.0 |
| 64 | 0.041 | 0.116 | 0.219 | 0.376 | 4.3 |
| 128 | 0.029 | 0.150 | 0.134 | 0.313 | 2.3 |
| 256 | 0.025 | 0.093 | 0.144 | 0.262 | 1.3 |

Our implementation of the benchmark for the class A problem size in 1.3 seconds compares very well with other published results on massively parallel machines [6]. The slight degradation in gigaflop/s for the entire CG benchmark as compared to that of the BBA algorithm (Figure 10) can be attributed to the additional message-passing required to maintain the iterative framework in Table 1.

Table 9

*Time in seconds for communication primitives for class B problem in BBA.*

| $PE's$ | $TRANSP$ | $EXPAND$ | $FOLD$ | $TotalCommTime$ | $TotalUserTime$ |
|---|---|---|---|---|---|
| 16 | 1.766 | 3.93 | 7.203 | 12.899 | 609.6 |
| 32 | 1.411 | 5.506 | 5.732 | 12.649 | 309.9 |
| 64 | 1.187 | 2.282 | 6.378 | 9.847 | 171.2 |
| 128 | 0.701 | 3.243 | 3.774 | 7.718 | 89.8 |
| 256 | 0.569 | 1.609 | 3.626 | 5.804 | 52.5 |

Tables 10 and 11 compare the performance of the CG kernel on the CRAY T3D (using BBA) with that of the CRAY C90[2] For the C90, the BRA method (see Section 3.2) was implemented for sparse matrix-vector multiplication. Notice that the performance relative to the number of CPU's or PE's scales similarly on the CRAY T3D and the CRAY C90. For both systems, the measured per cent parallelism[3] is 96% or 97%.

Each CPU of the CRAY C90 is approximately 70 times faster than each PE in the CRAY T3D. This speed difference is a reflection of the speed and number of memory banks in the C90. It is not indicative of the CPU speeds themselves. For MPP technology used in the CRAY T3D, the memory bandwidth and speed are attained from using many slower and smaller local memories. Specifically, 64 PEs of the T3D have 64 aggregate local memory banks, while the entire 16-CPU C90 system has 1024 memory banks, or 64 banks per CPU. The 1-CPU C90 and 64 PEs from a T3D have similar performance (see Tables 10 and 11), and it is not surprising based on the number of memory banks available.

Table 10

*Comparison of execution time in seconds for CG with Class A and B problems on the CRAY T3D and CRAY C90.*

| CRAY T3D | | | CRAY C90 | | |
|---|---|---|---|---|---|
| $PE's$ | Class A | Class B | $CPU's$ | Class A | Class B |
| 16 | 15.6 | 609.6 | 1 | 3.6 | 122.9 |
| 32 | 8.0 | 309.9 | 2 | 1.8 | — |
| 64 | 4.3 | 171.2 | 4 | 1.0 | 33.9 |
| 128 | 2.3 | 89.8 | 8 | 0.5 | 18.3 |
| 256 | 1.3 | 52.5 | 16 | 0.3 | 10.6 |

[2] Results on exactly 2 dedicated CPU's were not available.

[3] The measured percent parallelism is the parallelism parameter extracted from a fit of the data to an Amdahl's law curve.

TABLE 11

*Speed comparison in gigaflop/s of CG with Class A and B problems on the CRAY T3D and CRAY C90.*

| CRAY T3D | | | CRAY C90 | | |
|---|---|---|---|---|---|
| $PE's$ | Class A | Class B | $CPU's$ | Class A | Class B |
| 16 | 0.096 | 0.090 | 1 | 0.425 | 0.446 |
| 32 | 0.187 | 0.177 | 2 | 0.841 | — |
| 64 | 0.345 | 0.320 | 4 | 1.568 | 1.615 |
| 128 | 0.638 | 0.611 | 8 | 2.765 | 2.993 |
| 256 | 1.077 | 1.044 | 16 | 4.378 | 5.035 |

This *memory banking* problem can be partially alleviated on the CRAY T3D with schemes which will further enhance localization. If the density of the matrix $A$ is high enough, then the blocks can be further broken down into subblocks which have better localization. Such *subblocking* can only be carried out if there are a sufficient number of elements per subblock. However, the subblocking scheme has the liability of being data dependent in that unstructured sparse matrices do not usually benefit from localization.

## 5 Summary and Future Work

We have presented a competitive algorithm for sparse matrix-vector multiplication along with detailed analysis of its complexity. The block-block algorithm (BBA) is well suited for iterative methods in which the result vector has to be distributed in the same fashion as the input vector. This algorithm was used in the Conjugate Gradient kernel from the NAS Parallel Benchmarks on the CRAY T3D. For this particular benchmark, a class A problem size (matrix order $14,000$) is solved in 1.3 seconds which is competitive with other published results on massively-parallel machines.

Comparisons in the performance of the BBA method on the CRAY T3D with a a block row algorithm (BRA) for sparse matrix-vector multiplication on a 16-CPU CRAY C90 were also made. Differences in the number of memory banks associated with each processor (or CPU) on the different architectures account for the disparities in speed of the CG kernel from the NAS Parallel Benchmarks. It was shown that the performance of a 64-processor CRAY T3D roughly approximates that of a single CPU of the CRAY C90.

The BRA and BRA-like algorithms have recently been used in a heterogeneous implementation of a block-Lanczos method designed to find several of the largest singular triplets of large unstructured sparse matrices [8]. In that implementation, all code except the sparse matrix-vector multiplication kernels executed on the CRAY Y-MP host; the matrix-vector multiplication operations executed on the CRAY T3D. Future performance analysis of such implementations is warranted.

Future work may also include modifying the block-block algorithm for structured sparse matrices such as block tridiagonal matrices arising from finite element-based computations. From a future hardware/software perspective, the communication bandwidth between the CRAY Y-MP and CRAY T3D should also be improved. One way to achieve this would be to allow the processor(s) on the T3D to directly access the shared memory of the Y-MP. New communication primitives could be devised to `get` and `put` data directly from Y-MP memory without interrupting the Y-MP. This might require special hardware and software features.

## Acknowledgements

The authors would like to thank Marco Zagha (`marcoz@cs.cmu.edu`) at the School of Computer Science, Carnegie Mellon University for use of his implementation of the NAS Conjugate Gradient kernel which employs the BRA method for sparse matrix-vector multiplication.

## References

[1] D. BAILEY, J. BARTON, T. LASINSKI, AND H. SIMON, *The NAS Parallel Benchmarks*, Tech. Report RNR-91-002, NAS Ames Research Center, Moffet Field, CA, January 1991.

[2] R. BARRETT, M. BERRY, T. CHAN, J. DEMMEL, ET AL., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[3] M. W. BERRY, *Large scale singular value computations*, International Journal of Supercomputer Applications, 6 (1992), pp. 13–49.

[4] CRAY RESEARCH INC., *MPP Overview*, Cray Research Inc., Eagan, MN, 1993.

[5] G. FOX, M. JOHNSON, G. LYZENGA, S. OTTO, J. SALMON, AND D. WALKER, *Solving Problems on Concurrent Processors: Volume 1*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[6] B. HENDRICKSON, R. LELAND, AND S. PLIMPTON, *A parallel algorithm for matrix-vector multiplication*, Tech. Report SAND 92-2765, Sandia National Laboratories, Albuquerque, NM,

March 1993.

[7] A. KNIES AND R. BARUISO, *SHMEM User's Guide*, Cray Research Inc., Eagan, MN, August 1993.

[8] V. K. KRISHNA, *Sparse Matrix-Vector Multiplication Kernels on the CRAY T3D*, Master's thesis, University of Tennessee, Knoxville, August 1994.

[9] R. NUMRICH, P. SPRINGER, AND J. PETERSON, *Measurement of communication rates on the CRAY T3D interprocessor network*, in HPCN Europe, 1994.

[10] P. RIGSBEE, *PVM Reference Manual for CRAY Platforms*, Cray Research Inc., Eagan, MN, 1993.