To the Graduate Council:

I am submitting herewith a thesis written by Vijay Kodangal Krishna entitled "Sparse Matrix-Vector Multiplication Kernels On The CRAY T3D." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Michael W. Berry

Michael W. Berry, Major Professor

We have read this thesis
and recommend its acceptance:

Accepted for the Council:

Associate Vice Chancellor
and Dean of the Graduate School

Sparse Matrix-Vector Multiplication Kernels On The CRAY T3D

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

Vijay K. Krishna

August 1994

Acknowledgments

I thank my thesis advisor, Dr. Michael Berry, for his guidance, support and encouragement throughout this project and my master's program. I also thank Dr. Micah Beck and Dr. David Straight who served on my thesis committee. I am grateful to Dr. Bill Harrod from Cray Research Inc. (CRI) for providing me with the opportunity to intern at CRI. I thank Dr. Charles Grassl from CRI for his help with the Conjugate Gradient benchmark on the CRAY T3D. I also thank Dr. Michael Heroux from CRI for his help during this project.

Abstract

In this thesis we examine sparse matrix-vector multiplication algorithms 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 each algorithm's computational complexity. The specific sparse matrix-vector multiplication algorithms discussed are the block-block algorithm (BBA) for square matrices, and the row and column block algorithms (RBA, CBA) for rectangular matrices. We also discuss the performance of these algorithms within applications, such as the Conjugate Gradient kernel from the NAS Parallel Benchmarks and a block-Lanczos method from SVD-PACK for finding the largest singular triplets of a sparse matrix. 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 which is quite competitive with published results for this benchmark on other massively-parallel machines.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Objectives 1

1.3 Thesis Overview 2

2 Introduction to the CRAY T3D **3**

2.1 CRAY T3D Hardware Characteristics 3

2.1.1 Alpha Microprocessor Architecture 3

2.1.2 CRAY T3D Topology 4

2.1.3 CRAY T3D Configuration 6

2.2 CRAY T3D Software Overview 6

2.2.1 Operating System 6

2.2.2 CRAY T3D Programming Environment 6

2.2.3 Libraries/Message-Passing 7

3 Parallel Matrix-Vector Multiplication Algorithms **8**

3.1 Statement of the Problem 8

3.2 Data Structures for Unstructured Sparse Matrices 9

3.3 Block-Block Algorithm (BBA) 10
3.3.1 Block-Block Domain Decomposition 11
3.3.2 Communication Primitives 12
3.3.3 Parallel Matrix-Vector Multiplication Algorithm 20
3.3.4 Complexity Model 20
3.4 Block Column and Row Algorithms (BCA, BRA) 22
3.4.1 Block Column Algorithm (BCA) 22
3.4.2 Block Row Algorithm (BRA) 23
3.4.3 Complexity Model (BCA, BRA) 23
3.5 Comparison of Algorithms 27

4 Performance of Sparse Kernels in Applications **29**
4.1 NAS Conjugate Gradient Benchmark 29
4.1.1 Implementation Details on the CRAY T3D 29
4.2 Heterogeneous Implementation of Block Lanczos Method 34

5 Summary and Future Work **43**
5.1 Summary 43
5.2 Future Work 44

Bibliography **45**

Vita **48**

List of Tables

3.1 An iterative framework for using matrix vector multiplication. 12

3.2 Fold Operation for processor $P_{\alpha\beta}$ in row α. 14

3.3 Transpose operation for processor $P_{\alpha\beta}$. 18

3.4 Expand operation for processor $P_{\alpha\beta}$ in column β. 25

3.5 Parallel matrix vector multiplication algorithm. 26

3.6 Block column parallel matrix-vector multiplication algorithm for computing $y = Ax$. 26

3.7 Block row matrix vector multiplication algorithm for computing $y = A^T x$. 26

3.8 Communication complexity for the sparse matrix-vector multiplication algorithms. 27

4.1 Conjugate Gradient algorithm. 31

4.2 Time in seconds for communication primitives for Class A problem in BBA. 34

4.3 Time in seconds for communication primitives for class B problem in BBA. 34

4.4 Matrix-vector multiplication kernels used on the CRAY T3D. 36

4.5 Sparse matrices used for heterogeneous experiments. 37

4.6 Time in seconds for running Block-Lanczos code when computing the 10-largest singular triplets on the CRAY Y-MP only and on the heterogeneous platform consisting of a CRAY Y-MP and a CRAY T3D. 38

4.7 Time in seconds for block-Lanczos code when computing the 100-largest singular triplets on the CRAY Y-MP only and on the heterogeneous platform consisting of a CRAY Y-MP and a CRAY T3D. . . 39

List of Figures

2.1 Node architecture of the CRAY T3D. 4

2.2 Interconnection topology of the CRAY T3D. 5

2.3 Machine configuration of the CRAY T3D. 6

3.1 Distribution of data for $npy = npx = 4$ with indices α=3 and β=2. . 11

3.2 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. 15

3.3 Transpose operation for a processor grid with $npx = npy = 4$. The movement of data is shown for the first row of processors. 17

3.4 Transpose operation for a processor grid with $npy = 8$ and $npx = 4$. Data movement is shown for the first and second row of processors. . 17

3.5 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 $n/4$. 19

3.6 Domain decomposition of data for computing $y = Ax$. 22

3.7 Domain decomposition of data for computing $y = A^T x$. 24

3.8 Communication time versus the number of processors used for BBA, CBA and RBA. The order of the sparse matrix (n) is fixed at 14,000. 28

4.1 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. 32

4.2 Performance in gigaflop/s as the number of processors used is increased for CG benchmark on Class A and Class B problems. 33

4.3 BLS2 runs on the CRAY Y-MP front-end and makes calls to Hett3d to execute matrix-vector multiplication operations on the CRAY T3D. 35

4.4 Time taken in seconds for block-Lanczos when computing the 10-largest singular triplets on the CRAY Y-MP only and on the heterogeneous platform consisting of a CRAY Y-MP and a CRAY T3D. . . 40

4.5 Time taken in seconds for block-Lanczos when computing the 100-largest singular triplets on the CRAY Y-MP only and on the heterogeneous platform consisting of a CRAY Y-MP and a CRAY T3D. . . 41

4.6 Time taken for sparse matrix-vector multiplication kernels on the CRAY Y-MP and time taken to run the kernels on the heterogeneous platform which includes time taken to exchange data using PVM calls and execution time on the CRAY T3D. 42

Chapter 1

Introduction

1.1 Motivation

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 [Berr92a]. Efficient parallel algorithms are especially needed to perform the matrix-vector multiplication operations associated with iterative methods on massively-parallel processors (MPP's). MPP's may consist of tens to hundreds of processors each with its own local memory and a fast interconnect for movement of control information and data.

1.2 Objectives

The main objective of this work is the design of sparse matrix-vector multiplication algorithms along with detailed analysis of their complexity. The first algorithm de-

veloped is similar to the one described in [HeLP93], but with extensions to handle a non-square number of processors [BeGV94]. Other algorithms developed specifically address multiplication by rectangular matrices. The next objective is to apply these algorithms within real applications and relate the performance results obtained to a theoretical analysis of computational complexity. All algorithms are developed on a state-of-the-art massively-parallel computer system, the CRAY T3D from Cray Research Inc.

1.3 Thesis Overview

In Chapter 2 we discuss the hardware characteristics of the CRAY T3D including details of the processors used and the interconnection topology. In Chapter 3 we state the various sparse matrix-vector multiplication operations for which algorithms are developed and then discuss each algorithm in detail. In Chapter 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 a block-Lanczos method from SVDPACK for finding several of the largest singular values (and corresponding singular vectors) of a sparse matrix. We summarize and note future work in enhancing the performance of the sparse matrix-vector multiplication algorithms in Chapter 5.

Chapter 2

Introduction to the CRAY T3D

2.1 CRAY T3D Hardware Characteristics

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 [MPPO93].

2.1.1 Alpha Microprocessor Architecture

The peak speed of the DEC Alpha processor used in the CRAY T3D is 150 mflop/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. 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 machine used for this thesis has 256 processors, each processor having 2 MW (mega words) of

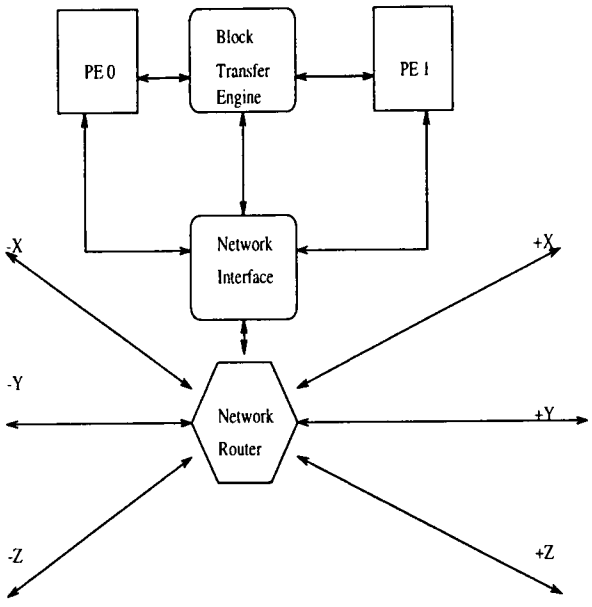


Figure 2.1: Node architecture of the CRAY T3D.

memory and a total peak performance of about 38 giga-flop/s (billions of floating-point operations per second).

2.1.2 CRAY T3D Topology

The topology of the CRAY T3D is a 3-dimensional torus with 2 processing elements per node (see Figure 2.1). Each processing element (PE) can contain 2-8 MW of DRAM. It has redundant nodes to replace defective nodes. Each node has latency hiding hardware (Block Transfer Engine) 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 (see Figure 2.1). Although not shown in Figure 2.2, each node lying on the edge of the torus has end-to-end connections to other edge nodes opposite to it. Data is sent in the form of packets, with each packet containing a header with destination information followed by data.

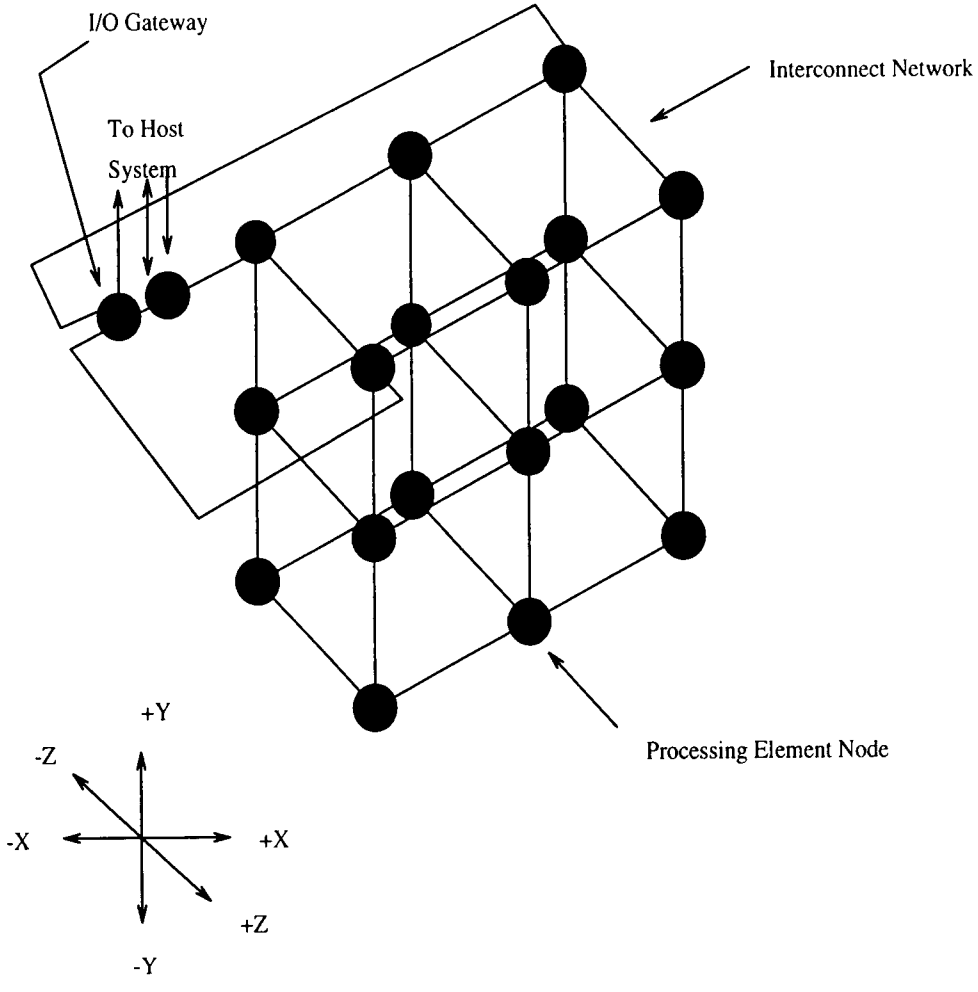


Figure 2.2: Interconnection topology of the CRAY T3D.

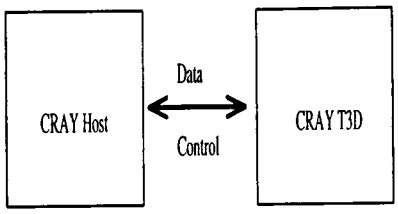


Figure 2.3: Machine configuration of the CRAY T3D.

2.1.3 CRAY T3D Configuration

The CRAY T3D has a front-end UNICOS host system (CRAY Y-MP) through which all I/O activities (data and control) are performed (see Figure 2.3). 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

2.2.1 Operating System

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

2.2.2 CRAY T3D Programming Environment

The CRAY T3D programming environment is very rich and provides a variety of compilers, assembler, loaders, graphical debugging tools 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 challenge is to keep data processing local to each node. The CRAY T3D provides a debugger called *TotalView* which has an X windows interface. *MPP Apprentice* is a performance utility which can be used to determine *where* performance problems are and provides data and observations to explain *what* the problem is.

2.2.3 Libraries/Message-Passing

The CRAY T3D also utilizes various standard libraries along with highly-optimized scientific library routines. Message-passing support is provided through the SHMEM Communication Library [KnBa93] and PVM (Parallel Virtual Machine) [Rigs93]. PVM is a defacto message-passing standard developed for heterogeneous systems tied to a network. PVM (Version 3.0) was used for this work for communication between the CRAY T3D and the front-end CRAY Y-MP. The PVM primitives used by the sending PE are `pvm_pack` and `pvm_send` and those used by the receiving PE are `pvm_receive` and `pvm_unpack`. The SHMEM Communication library routines (`shmem_get` and `shmem_put`) are highly optimized for message-passing within the CRAY T3D and have been used for intra PE communication.

Chapter 3

Parallel Matrix-Vector Multiplication Algorithms

3.1 Statement of the Problem

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. A^T denotes the transpose of the sparse matrix A, X is an $n \times k$ dense matrix and Y is an $m \times k$ dense matrix. The following sparse matrix-vector and sparse matrix-matrix multiplications are considered:

$$y \leftarrow Ax,$$
$$y \leftarrow A^T x,$$
$$Y \leftarrow A^T A X.$$

3.2 Data Structures for Unstructured Sparse Matrices

The nonzeros of very large sparse matrices are normally stored in compressed form in order to conserve memory and to avoid redundant computations on explicit zeros. Both compressed sparse row and column formats are discussed in this section.

CSR - Compressed Sparse Row

The sparse matrix A can be stored in the CSR format using three arrays. Let `NNZ` represent the number of nonzeros, `NROWS` be the number of rows, and `NCOLS` be the number of columns in the matrix.

- `VAL` - A scalar array of length `NNZ` consisting of the nonzero entries of A.
- `INDX` - An integer array of length `NNZ` consisting of the column indices of entries of A.
- `PNTR` - An integer array of length `NROWS + 1`, containing the pointers to the beginning of each row in the array `VAL`.

For example, let

$$A = \begin{pmatrix} 10 & 0 & 21 & 18 & 0 \\ 0 & 0 & 16 & 38 & 0 \\ 9 & 11 & 23 & 12 & 0 \\ 0 & 7 & 0 & 44 & 0 \\ 86 & 22 & 0 & 0 & 55 \end{pmatrix},$$

The CSR representation for matrix A would be:

$$\texttt{VAL} = \begin{pmatrix} 10 & 21 & 18 & 16 & 38 & 9 & 11 & 23 & 12 & 7 & 44 & 86 & 22 & 55 \end{pmatrix}$$

$$\texttt{INDX} = \begin{pmatrix} 1 & 3 & 4 & 3 & 4 & 1 & 2 & 3 & 4 & 2 & 4 & 1 & 2 & 5 \end{pmatrix}$$

$$\texttt{PNTR} = \begin{pmatrix} 1 & 4 & 6 & 10 & 12 & 15 \end{pmatrix}$$

CSC - Compressed Sparse Column

Similar to the CSR format, the matrix A can also be stored in the CSC format using three arrays. Let `NNZ` represent the number of nonzeros in the matrix.

- `VAL` - A scalar array of length `NNZ` consisting of the nonzero entries of A.
- `INDX` - An integer array of length `NNZ` consisting of the row indices of entries of A.
- `PNTR` - An integer array of length `NCOLS + 1`, containing the pointers to the beginning of each column in the array `VAL`.

The CSC representation for matrix A would be:

$$\texttt{VAL} = \begin{pmatrix} 10 & 9 & 86 & 11 & 7 & 22 & 21 & 16 & 23 & 18 & 38 & 12 & 44 & 55 \end{pmatrix}$$

$$\texttt{INDX} = \begin{pmatrix} 1 & 3 & 5 & 3 & 4 & 5 & 1 & 2 & 3 & 1 & 2 & 3 & 4 & 5 \end{pmatrix}$$

$$\texttt{PNTR} = \begin{pmatrix} 1 & 4 & 7 & 10 & 14 & 15 \end{pmatrix}$$

3.3 Block-Block Algorithm (BBA)

The first sparse matrix-vector multiplication algorithm considered uses a block-block decomposition quite suitable for iterative methods, in which the result vector y has

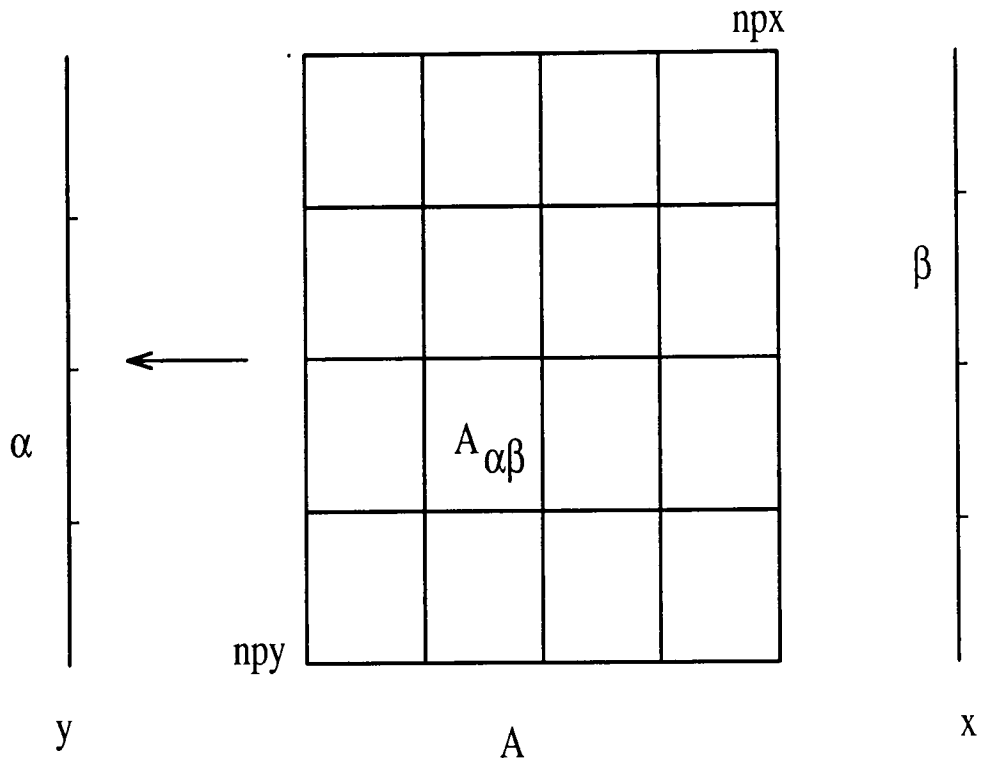


Figure 3.1: Distribution of data for $npy = npx = 4$ with indices α=3 and β=2.

to be distributed in the same fashion as the input vector x. This is similar to the algorithm described in [HeLP93], but with extensions to handle a non-square number of processors [BeGV94]. This algorithm which is used to compute $y = Ax$ assumes that the number of processors is a power of two.

3.3.1 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 3.1 illustrates the distribution of data for a 4×4 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 3.1. This algorithm is well suited for iterative methods (see Table 3.1) in which the output vector y_i is the input vector x_i for the next iterate. Hence, at the end of the matrix-vector multiplication operation processor $P_{\alpha\beta}$ must contain y_β.

Table 3.1: An iterative framework for using matrix vector multiplication.

For $i = 1, \ldots$

$y_i = Ax_i$

$x_{i+1} = y_i$

.

.

.

EndFor

3.3.2 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** [FJLO88]. As seen from Figure 3.1 processor, $P_{\alpha\beta}$ owns a block of the matrix $A_{\alpha\beta}$ and vector x_β. 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/npx present in npx processors having the same block row index α. At the end of this operation (see Table 3.2), 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_α. The algorithm, which requires $log_2 p$ steps, to accomplish this is *recursive halving*. As outlined in Table 3.2, 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 3.2, 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 3.2 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/npx 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 $(\frac{n}{npx} - \frac{n}{(npx*npx)})$. The total number of elements sent (received) by a processor can also be written as a geometric series with $log_2 p$ terms

$$TimeFold_{comm} = \sum_{k=1}^{log_2 p} \frac{n}{2^k * npx} = \frac{n}{npx}\left(1 - \frac{1}{npx}\right). \tag{3.1}$$

We use the **Put** primitive to copy blocks of data directly from one processor's memory to another processor's memory. 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 SHMEM Communication Library [KnBa93].

Table 3.2: Fold Operation for processor $P_{\alpha\beta}$ in row α.

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

1) **For** $i = 0, ..., (log_2 npx) - 1$

2) Partition $z_{\alpha\beta} = (z_1^T \mid z_2^T)^T$

3) $P_{\alpha\beta'} = P_{\alpha\beta}$ with i^{th} bit of β flipped

4) **If** bit i of β is 1 **then**

5) **Put** z_1 into w_1 of processor $P_{\alpha\beta'}$

6) Barrier()

7) $z_{\alpha\beta} = z_2 + w_2$

8) **Else**

9) **Put** z_2 into w_2 of processor $P_{\alpha\beta'}$

10) Barrier()

11) $z_{\alpha\beta} = z_1 + w_1$

12) **Endif**

13) **EndFor**

14) $y_{\alpha\beta} = z_{\alpha\beta}$

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

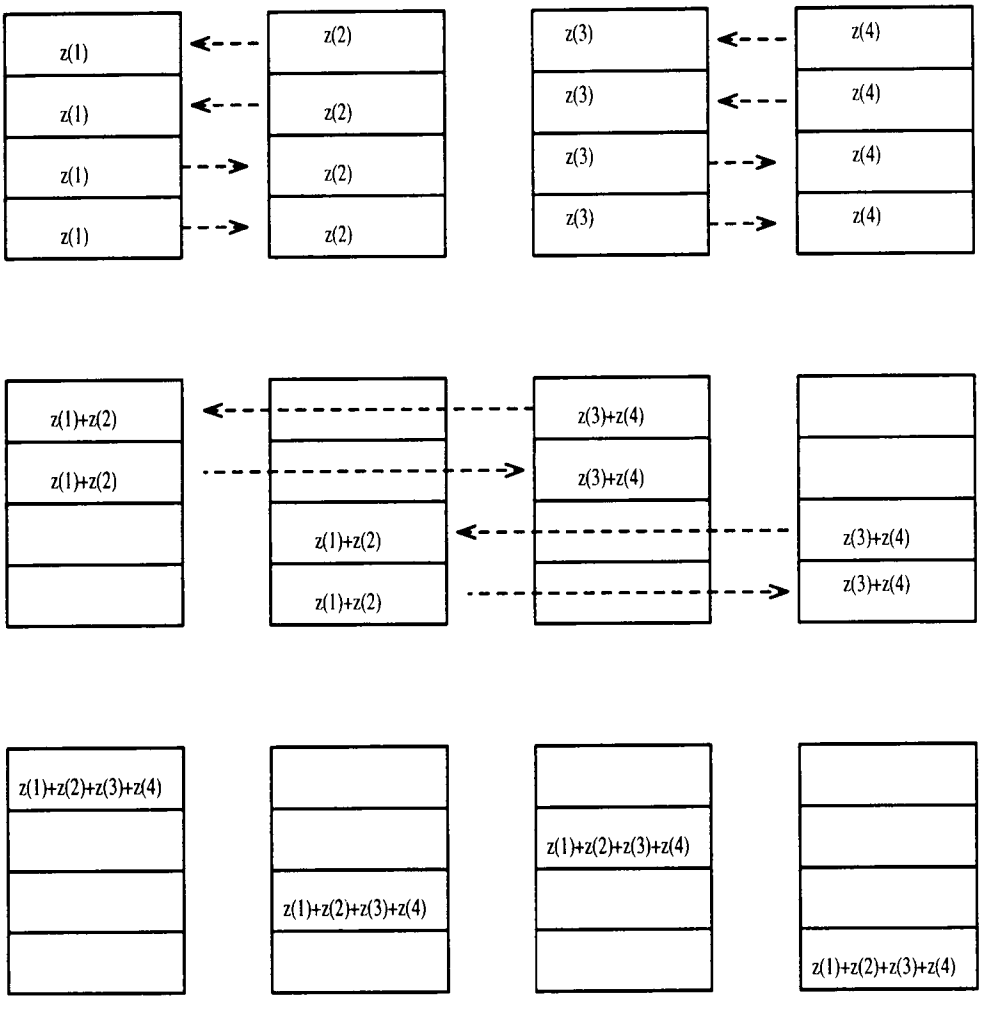


Figure 3.2: Fold operation with 4 processors in a row ($npx = 4$), each having a vector $z(i)$ of length $n/4$. At the end of the operation each processor has a subvector of length $n/16$ containing a unique piece of the result vector.

Transpose Operation

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 3.3). In this communication primitive, processor $P_{\alpha\beta}$ exchanges $y_{\alpha\beta}$ of length $n/(npx * npx)$ vector with processor $P_{new\alpha,new\beta}$, where

$$new\alpha = \beta + (\alpha.\mathbf{and}.(k-1)) * npx,\ k = npy/npx, \text{ and } new\beta = \alpha/2.$$

As seen in Figure 3.3, 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 3.4. The total number of data elements that are sent (received) by a processor in the transpose operation is given by $\frac{n}{(npx*npx)}$ so that

$$TimeTranspose_{comm} = \frac{n}{(npx * npx)} \ . \tag{3.2}$$

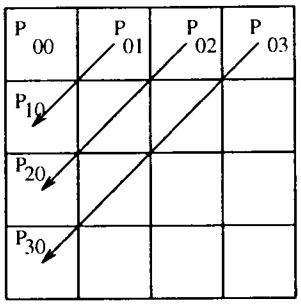


Figure 3.3: Transpose operation for a processor grid with $npx = npy = 4$. The movement of data is shown for the first row of processors.

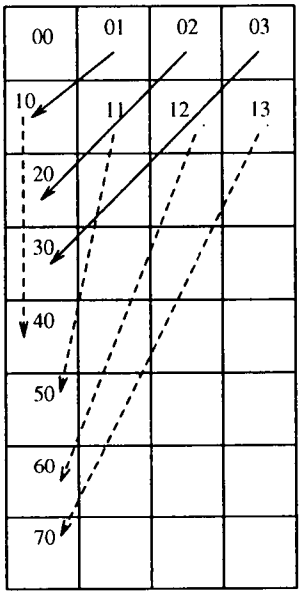


Figure 3.4: Transpose operation for a processor grid with $npy = 8$ and $npx = 4$. Data movement is shown for the first and second row of processors.

Table 3.3: Transpose operation for processor $P_{\alpha\beta}$.

Processor $P_{\alpha\beta}$ contains $y_{\alpha\beta} \in \Re^{n/(npx*npx)}$
$k = npy/npx$
$new\alpha = \beta + (\alpha.\mathbf{and}.(k-1)) * npx$
$new\beta = \alpha/2$
Put $y_{\alpha\beta}$ into $z_{\alpha\beta}$ of processor $P_{new\alpha,new\beta}$
Barrier()
$y_{\alpha\beta} = z_{\alpha\beta}$

Expand Operation

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 $n/(npx * npx)$ from each processor is combined to form a vector y_α 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 3.5 and Table 3.4. The number of elements sent (received) is equal to the fold operation and from Equation (3.1) is given by $(\frac{n}{npx} - \frac{n}{(npx*npx)})$. 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

$$TimeExpand_{comm} = \frac{n}{npx} - \frac{n}{(npx * npx)} + log_2\left(\frac{npy}{npx}\right) * \left(\frac{n}{npx}\right) . \qquad (3.3)$$

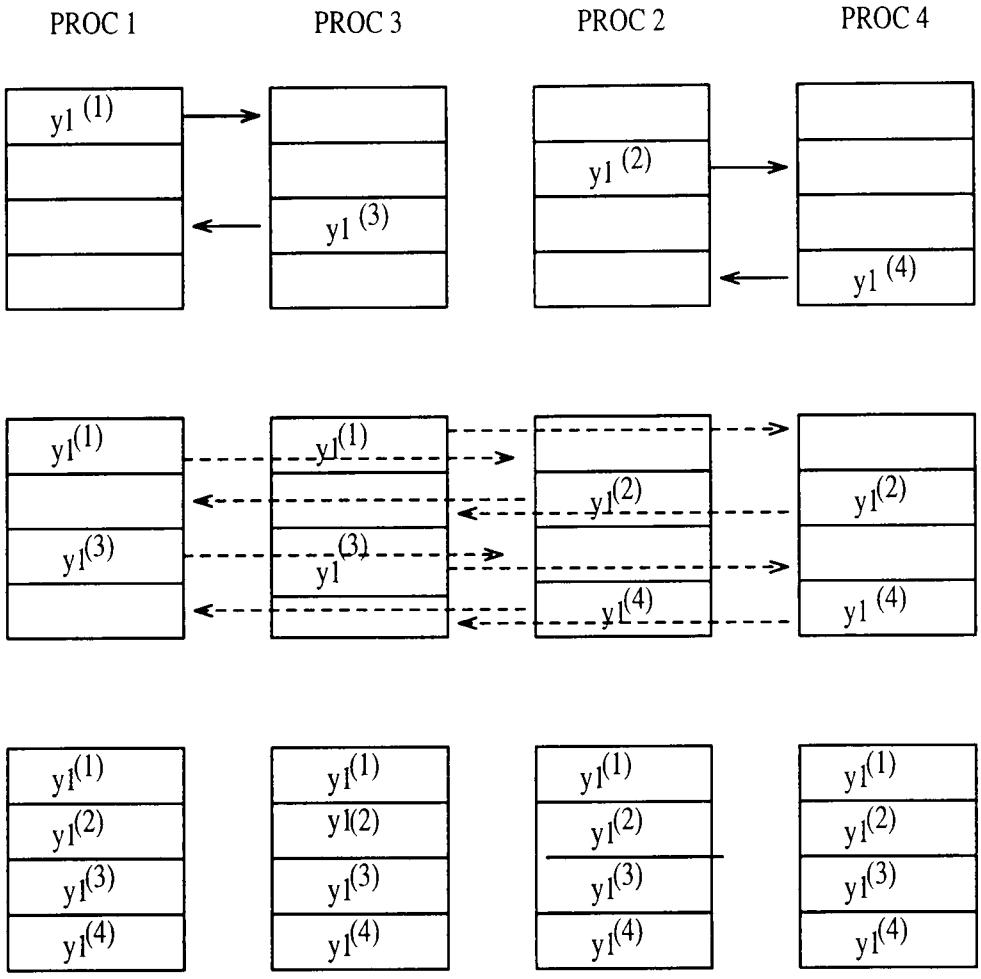


Figure 3.5: Expand operation for npy=4. Each processor in a column contains a vector $yl^{(i)}$ of length $n/16$. At the end of the operation each processor has the complete vector yl of length $n/4$.

Table 3.4: Expand operation for processor $P_{\alpha\beta}$ in column β.

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

For $i = (log_2 npx) - 1, ..., 0$

$P_{\alpha',\beta} = P_{\alpha\beta}$ with i^{th} bit of α flipped

Put $y_{\alpha\beta}$ into w of processor $P_{\alpha',\beta}$

If bit i of α is 1 **Then**

Prefix w so that $y^T_{\alpha\beta} = (w^T y^T_{\alpha\beta})$

Else

Append w so that $y^T_{\alpha\beta} = (y^T_{\alpha\beta} w^T)$

Endif

Endfor

If $(npx.ne.npy)$ **Then**

For $i = 1, ..., log_2(npy/npx)$

$new\alpha = \alpha.\mathbf{xor}.(npx * i)$

Put $y_{\alpha\beta}$ into w of processor $P_{new\alpha,\beta}$

$y_{\alpha\beta} = y_{\alpha\beta} + w$

Endfor

Endif

3.3.3 Parallel Matrix-Vector Multiplication Algorithm

In the first step of the parallel matrix vector algorithm shown in Table 3.5, a local matrix vector product is computed on each processor with a block of the matrix $A_{\alpha\beta}$ and x_β to obtain $z_{\alpha\beta}$. Using the fold operation illustrated in Table 3.2 and Figure 3.2, we add vectors $z_{\alpha\beta}$ from processors within rows to get $y_{\alpha\beta}$ which is a subvector of the vector y_α. If the next matrix-vector multiplication on processor $P_{\alpha\beta}$ requires y_β (see Table 3.1), the transpose operation can be used to copy subvectors of y_β into vector w of the processors in column β (see Table 3.3, Figure 3.3, and Figure 3.4). Finally, the expand operation shown in Figure 3.5 and Table 3.4 combines the subvectors w and places the complete vector y_β in each processor of column β.

Table 3.5: Parallel matrix vector multiplication algorithm.

$w, z \in \Re^{n/npx}$ 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 3.2.
4) Expand w to obtain y_β.

3.3.4 Complexity Model

The complexity of the algorithm shown in Table 3.5 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 size of the sparse matrix and p is the number of processors. From Equation (3.1) we know that the total number of steps in the fold operation is $log_2 npx$ and from Equation (3.3), 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

$$Time_{startup} = \left(2 * log_2 npx + log_2\left(\frac{npy}{npx}\right) + 1\right) * t_{startup}, \tag{3.4}$$

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

$$\begin{aligned} Time_{comm} &= TimeFold_{comm} + TimeTranspose_{comm} + TimeExpand_{comm} \\ &= \frac{n}{npx} - \frac{n}{npx * npx} + \frac{n}{(npx * npx)} + \frac{n}{npx} \\ &\quad - \frac{n}{(npx * npx)} + log_2\left(\frac{npy}{npx}\right) * \left(\frac{n}{npx}\right) \\ &= 2 * \left(\frac{n}{npx}\right) - \frac{n}{npx * npx} + log_2\left(\frac{npy}{npx}\right) * \left(\frac{n}{npx}\right) \\ &= \frac{n}{npx}\left(2 - \frac{1}{npx} + log_2\left(\frac{npy}{npx}\right)\right). \end{aligned} \tag{3.5}$$

The total run time is given by

$$TotalTime = \left(\frac{2 * nnz - n}{p}\right) * T_{flop} + Time_{startup} + Time_{comm}, \tag{3.6}$$

where T_{flop} is the time taken by a single floating point operation.

For simplicity, we consider the scalability of the algorithm in Table 3.5 for a square grid of p processors where $npx = npy = \sqrt{p}$. For this grid, the total communication time from Equation (3.5) is given by

$$Time_{comm} = \frac{n}{\sqrt{p}}\left(2 - \frac{1}{\sqrt{p}}\right) = n * \left(\frac{2 * \sqrt{p} - 1}{p}\right). \tag{3.7}$$

We can see from the above equation that the communication time is directly proportional to n and inversely proportional to the square root of the number of processors.

3.4 Block Column and Row Algorithms (BCA, BRA)

The next two sparse matrix-vector multiplication algorithms address the situation when the sparse matrix A is over- or under-determined (i.e., rectangular).

3.4.1 Block Column Algorithm (BCA)

An alternate algorithm which can be used to compute $y = Ax$ divides the sparse matrix A into column blocks so that each processor gets 1 column block (see Figure 3.6). Each block of the sparse matrix is stored in compressed column format within each processor. The number of column blocks is equal to the number of processors, which is assumed to be greater than or equal to the number of columns in the sparse matrix

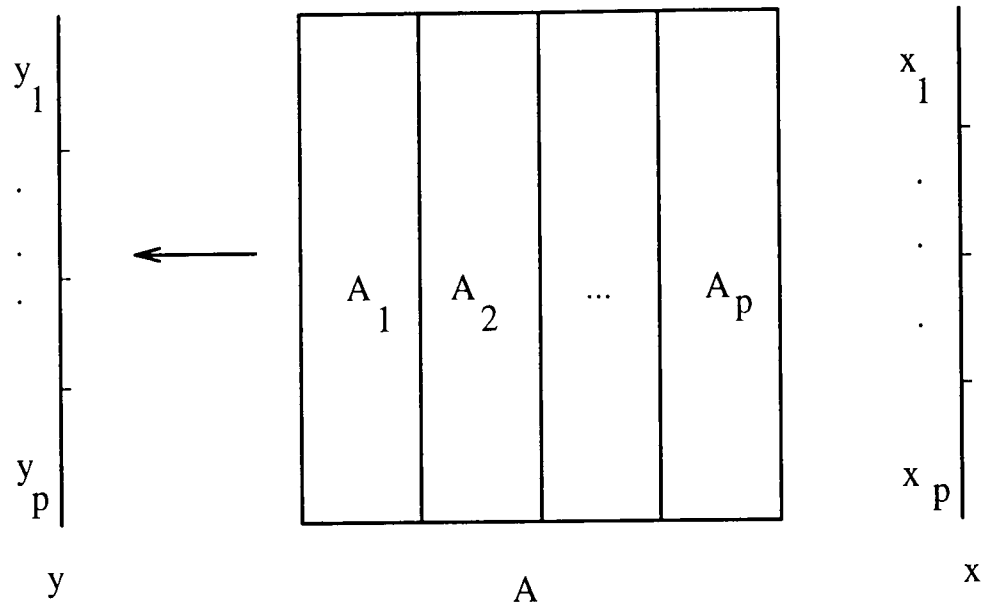


Figure 3.6: Domain decomposition of data for computing $y = Ax$.

A. If A_i, x_i and y_i are assigned to processor i, then

$$y = Ax = \left(A_1x_1 + A_2x_2 + ... + A_px_p. \right)$$

Table 3.6: Block column parallel matrix-vector multiplication algorithm for computing $y = Ax$.

1) Assign A_i and x_i to processor P_i.
2) Compute $z_i = A_ix_i$.
3) Perform fold operation on z_i with p processors to get y_i.

The parallel matrix-vector algorithm given in Figure 3.6 will compute vector y in a distributed fashion.

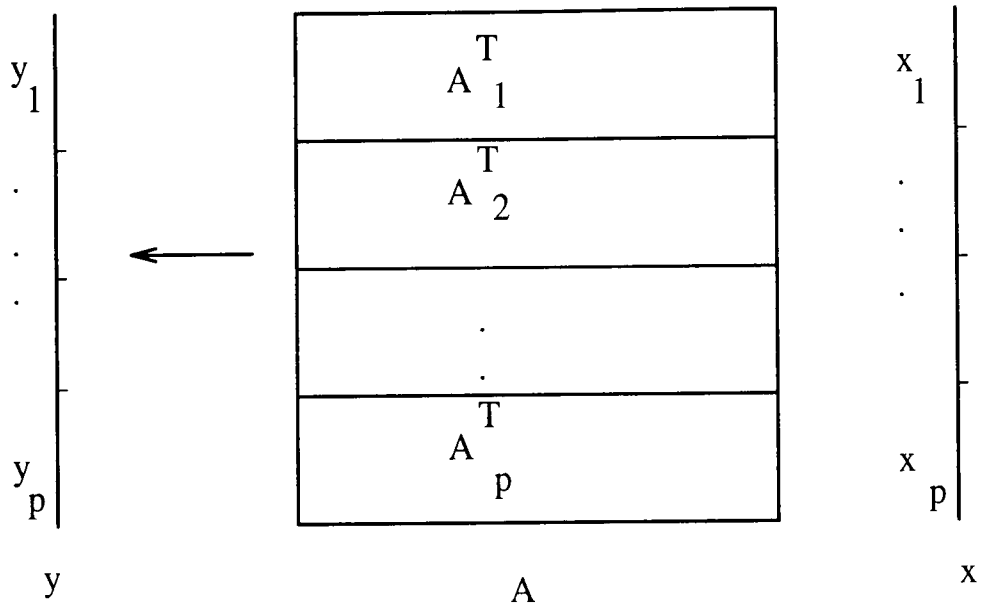


Figure 3.7: Domain decomposition of data for computing $y = A^T x$.

3.4.2 Block Row Algorithm (BRA)

This algorithm is used to compute $y = A^T x$, where the transpose of the sparse matrix A is divided into row blocks with each processor assigned 1 row block. The number of blocks is equal to the number of processors. The parallel matrix-vector multiplication algorithm given in Table 3.7 ensures each processor P_i will compute y_i. Processor P_i initially stores x_i. A subsequent expand operation on p processors will duplicate x on each processor.

Table 3.7: Block row matrix vector multiplication algorithm for computing $y = A^T x$.

1) Assign A_i^T 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 $y_i = A_i^T x$

Table 3.8: Communication complexity for the sparse matrix-vector multiplication algorithms.

BBA	$CBA(RBA)$
$n * (\frac{2*\sqrt{p}-1}{p})$	$n * (\frac{p-1}{p})$

3.4.3 Complexity Model (BCA, BRA)

The complexity model for both these algorithms is similar. The number of floating-point operations per processor is equal to $(\frac{(2*nnz-n)}{p}) * T_{flop}$, where T_{flop} is the cost of a single floating point operation. The number of steps needed in the fold operation for BCA or expand operation in BRA is $log_2 p$, and the total number of values that are sent (received) by a processor is given by $(m - \frac{m}{p})$. Therefore, the total run time is

$$TotalTime = \left(\frac{2 * nnz - m}{p}\right) * T_{flop} + log_2 p * T_{startup} + \left(m - \frac{m}{p}\right) * T_{put}, \quad (3.8)$$

where T_{put} is the amount of time taken for a put operation. The total communication time is equal to $m * (\frac{p-1}{p})$ and is directly proportional to the number of rows m and independent of the number of processors.

3.5 Comparison of Algorithms

For simplicity we compare the scalability of the three algorithms for p processors and sparse square matrices of order n. The total communication time for BBA, CBA and RBA is given in Table 3.8. The number of floating-point operations in all the methods are the same. The total communication time in the block-block algorithm is inversely proportional to the square root of the number of processors(p), whereas for large n the communication time of the block-column and block-row methods is independent upon the number of processors. Therefore, the block-block algorithm scales

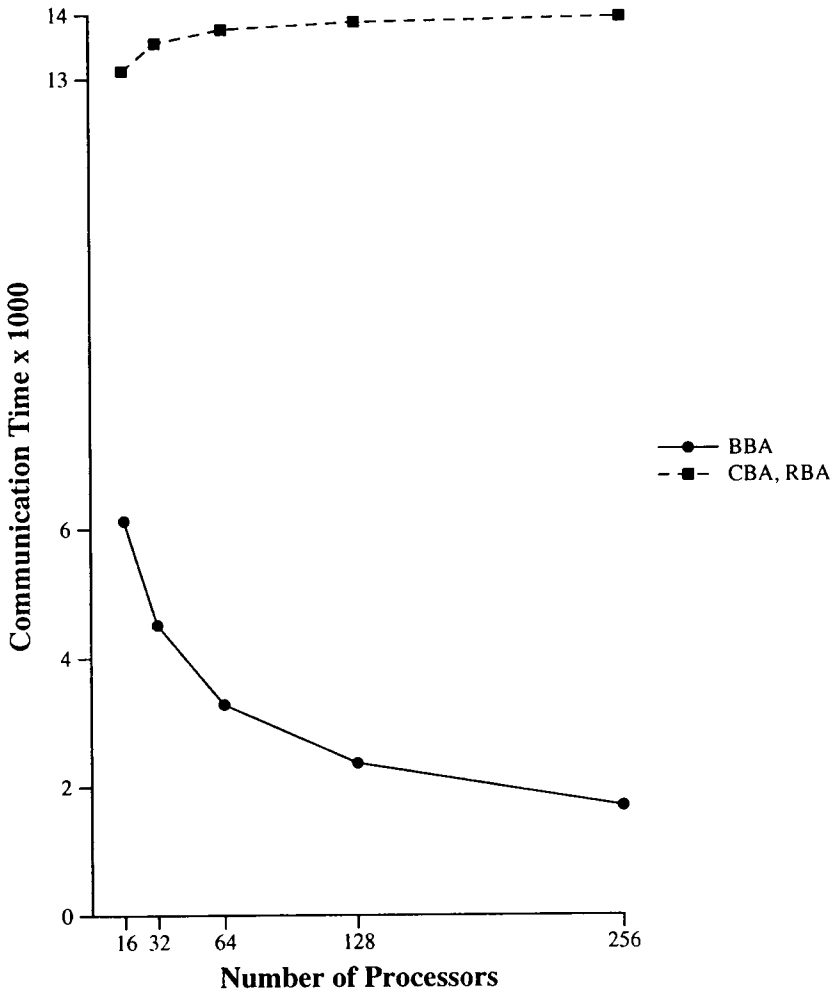


Figure 3.8: Communication time versus the number of processors used for BBA, CBA and RBA. The order of the sparse matrix (n) is fixed at 14,000.

very well as the number of processors is increased (see Figure 3.8). 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.

Chapter 4

Performance of Sparse Kernels in Applications

In this chapter, we discuss the performance of the sparse matrix-vector multiplication algorithms described in Chapter 3 within applications such as the Conjugate Gradient kernel from the NAS Parallel Benchmarks [BBLS91] and a block Lanczos method from SVDPACK [Berr92b] for finding several of the largest singular values of a sparse matrix. BBA has been applied to the Conjugate Gradient benchmark and RBA and CBA have been used in the block-Lanczos method.

4.1 NAS Conjugate Gradient Benchmark

4.1.1 Implementation Details on the CRAY T3D

The algorithms described in the previous chapter are well-suited for massively-parallel systems such as the CRAY T3D. Several key features of the CRAY T3D have been exploited. The most important feature being the communication bandwidth for ex-

changing data between processors. The **put** communication primitive allows any processor on the T3D to write data directly to the memory of any other processor. The processor-to-processor bandwidth of this primitive is approximately 130 Mbytes/second [NuSP94]. Another key feature of the 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, i.e, the number of *hops* taken does not matter. A *hop* can be defined as the distance from a processor to any of its nearest-neighboring processors.

Both the Conjugate Gradient kernel and BBA (discussed in Chapter 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 SHMEM communication library [KnBa93]. No assembly language optimization was used.

The classical Conjugate Gradient method is outlined in Table 4.1. 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 4.1. Though the sparse matrix generated in the NAS Conjugate Gradient benchmark has a particular structure or pattern of non-zeros with full main diagonal sparsity was not exploited. In other words, 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

Table 4.1: 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**

0.1% dense and the class B problem being 0.25% dense.

Figure 4.1 illustrates the performance of the local sparse matrix-vector multiplication operation in BBA for class A and class B problems. With 256 processors, this operations gives a performance of 1.5 gigaflop/s for 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 over the total number of floating-point operations done. Figure 4.2 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 4.2 and 4.3, respectively. The total 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 are increased.

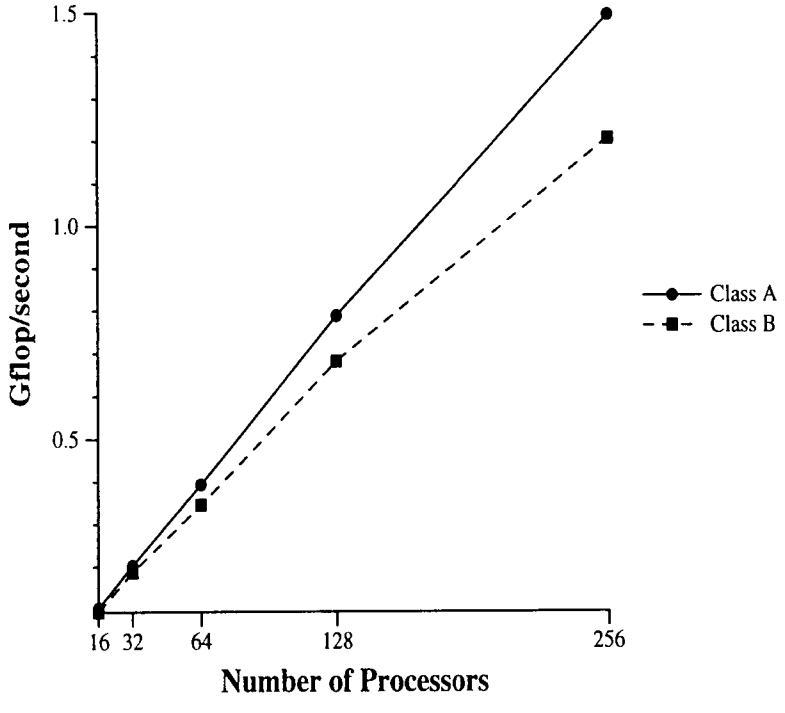


Figure 4.1: Performance in gigaflop/s (billions of floating-point operations per second) for the local sparse matrix-vector multiplication operation in BBA versus the number of processors used.

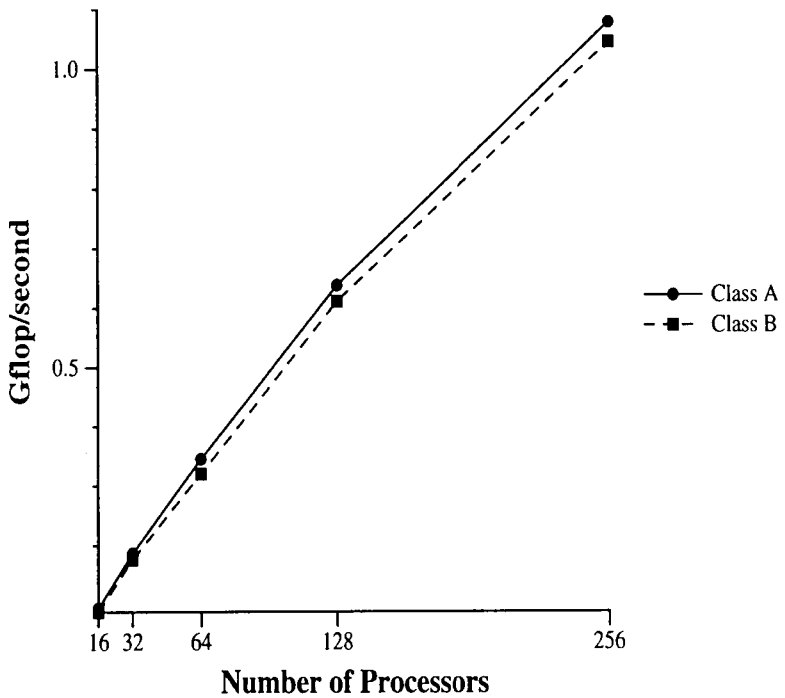


Figure 4.2: Performance in gigaflop/s as the number of processors used is increased for CG benchmark on Class A and Class B problems.

Table 4.2: 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

Table 4.3: 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

This agrees with our prediction in the complexity analysis of communication time for BBA in Chapter 3.

Our implementation solved the benchmark with class A problem size in 1.3 seconds which compares very well with other published results on massively parallel machines [HeLP93]. The slight degradation in gigaflop/s for the entire CG benchmark as compared to that of the BBA algorithm (Figure 4.1) can be attributed to the additional message-passing required to maintain the iterative framework in Table 3.1.

4.2 Heterogeneous Implementation of Block Lanczos Method

CBA and RBA discussed in Chapter 3 were used in a heterogeneous implementation of a block Lanczos method designed to find k-largest singular values of large *unstructured* sparse matrices arising from applications such as information retrieval, seismic

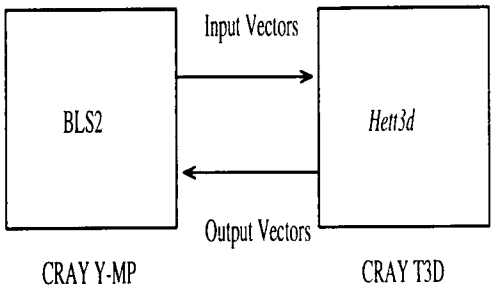


Figure 4.3: BLS2 runs on the CRAY Y-MP front-end and makes calls to Hett3d to execute matrix-vector multiplication operations on the CRAY T3D.

reflection tomography, and real-time signal processing. This method from the SVDPACK library [Berr92b] for determining singular values and corresponding singular vectors has been implemented on a variety of high-performance computer systems. SVDPACK uses Lanczos, block-Lanczos, subspace iteration and trace minimization methods for determining several of the largest singular values and corresponding singular vectors for large sparse matrices.

Without loss of generality, suppose A is sparse m by n ($m \gg n$) matrix with $\text{rank}(A) = r$. The singular value decomposition (SVD) of A can be defined as

$$A = U\Sigma V^T,$$

where $U^TU = V^TV = I_n$ and $\Sigma = diag(\sigma_1, ..., \sigma_n)$, $\sigma_i > 0$ for $1 \leq i \leq r$, $\sigma_i = 0$ for $i \geq r + 1$. The first r columns of the orthogonal matrices U and V define the orthonormalized eigenvectors associated with the r nonzero eigenvalues of AA^T and A^TA, respectively. The singular values of A are defined as the diagonal elements of Σ which are the nonnegative square roots of the n eigenvalues of AA^T. The set $\{u_i, \sigma_i, v_i\}$ is called the i-th singular triplet. The singular vectors (triplets) corresponding to large (small) singular values are called large (small) singular vectors (triplets) [Berr92a].

Table 4.4: Matrix-vector multiplication kernels used on the CRAY T3D.

code	*opb*()	*opm*()	*opa*()
BLS2	$y = A^T Ax$	$Y = A^T AX$	$y = Ax$

The block Lanczos implementation for a CRAY Y-MP was obtained taken from the SVDPACK library. This code (BLS2) was then reconfigured as a *heterogeneous* program with all the code except the sparse matrix-vector multiplication kernels given in Table 4.4 running on the CRAY Y-MP. The matrix-vector multiplication operations were done on CRAY T3D, a massively parallel processor. As shown in Figure 4.3 the block-Lanczos application for computing the singular values running on the CRAY Y-MP initially spawns a process *hett3d* for a 64-processor CRAY T3D[1]. Whenever the application running on the CRAY Y-MP needs to execute a matrix-vector multiplication operation it requests the process *hett3d* to perform this operation on the CRAY T3D. The sparse matrix A is initially loaded into the CRAY T3D from the CRAY Y-MP front-end.

The block-Lanczos code running on the CRAY Y-MP and matrix-vector multiplication kernels running on the CRAY T3D were written in Fortran 77. The code running on the CRAY Y-MP was taken and BCA and RBA were used for the matrix-vector multiplication operations. The input vectors for the matrix-vector multiplication operations were passed from the CRAY Y-MP front-end to the CRAY T3D using PVM (Parallel Virtual Machine) function calls [Rigs93].

[1] Only 64 processors of the 256-processor CRAY T3D were available for the heterogeneous program.

Table 4.5: Sparse matrices used for heterogeneous experiments.

Matrix	*ROWS*	*COLS*	*NNZ*
amoco1	1436	330	35210
bellmedt	5831	1033	52012
bellcist	5143	1460	66340
bellcrat	4997	1400	78942
belltect	16637	6535	327244

The aim here is to see if an application like the block-Lanczos method running on the CRAY Y-MP front-end would benefit by using the matrix-vector multiplication kernels running on the CRAY T3D. Timings have been taken to run the complete application on CRAY Y-MP only and to run the application using both the CRAY Y-MP and the CRAY T3D.

An 8-processor CRAY Y-MP (with autotasking) has been used with each processor running at clock rate of 6 nano-seconds per cycle, so that the CRAY Y-MP runs at a peak rate of 2.67 gigaflop/s. The 64-processor CRAY T3D attached to the Y-MP has each processor running at a clock rate of 6.67 nano-seconds and yields an an overall peak performance of 9.6 gigaflop/s. The various sparse matrices used (see [Berr92a]) for finding the k-largest singular values are listed in Table 4.5.

Table 4.6 compares the time required by the block-Lanczos method when computing the 10-largest singular triplets on the CRAY Y-MP only versus using a machine configuration of a CRAY Y-MP and a CRAY T3D. The total time, i.e. the time spent solely in the sparse matrix-vector multiplications (TimeinKernels) on the CRAY T3D, and the time taken for exchange of data between the Y-MP and the T3D using PVM function calls are both listed. As seen from Figure 4.4, the time required for the `belltect` sparse matrix on the CRAY Y-MP is only 19.6 seconds

Table 4.6: Time in seconds for running Block-Lanczos code when computing the 10-largest singular triplets on the CRAY Y-MP only and on the heterogeneous platform consisting of a CRAY Y-MP and a CRAY T3D.

Matrix	*NNZ*	*Configuration*	*TotalTime*	*TimeinKernels*	*PVMTime*
amoco1	35210	Y-MP	1.64	1.414	
		Y-MP-T3D	1.77	0.57	0.95
bellmedt	52012	Y-MP	3.5	3.01	
		Y-MP-T3D	5.88	1.44	3.97
bellcist	66340	Y-MP	3.98	3.42	
		Y-MP-T3D	8.01	1.23	6.19
bellcrat	78942	Y-MP	4.2	3.71	
		Y-MP-T3D	6.9	1.39	5.01
belltect	327244	Y-MP	19.6	17.84	
		Y-MP-T3D	34.6	5.3	27.5

whereas 34.6 seconds is required on the CRAY Y-MP and CRAY T3D. The reason why the application is taking more time to run on the heterogeneous platform can understood by looking at Figure 4.6 which compares the time taken to run the sparse matrix-vector multiplication kernels on the CRAY Y-MP, the time taken to pass data between the Y-MP and T3D(PVM), and the time taken to actually execute the kernels on the T3D. Figure 4.6 shows that even though the kernels perform better on the CRAY T3D, we do not see any decrease in the overall execution time due to the time taken for exchange of the input and output vectors using PVM calls between the CRAY Y-MP and CRAY T3D. For example, the time taken to run the sparse matrix-vector multiplication kernels for the `belltect` matrix on the Y-MP is 17.84 seconds and on the T3D is 5.3 seconds. This gain is offset by the time taken for the PVM calls which take about 27.5 seconds.

Table 4.7 and Figure 4.5 illustrate this scenario when the 100-largest singular triplets are computed by the block-Lanczos method for the same collection of matrices. From the results in Tables 4.6 and 4.7, it is clear that the application does benefit

Table 4.7: Time in seconds for block-Lanczos code when computing the 100-largest singular triplets on the CRAY Y-MP only and on the heterogeneous platform consisting of a CRAY Y-MP and a CRAY T3D.

Matrix	*NNZ*	*Triplets*	*Config.*	*Total Time*	*Time in Kernels*	*PVMTime*
amoco1	35210	100	Y-MP	26.5	8.77	
		100	Y-MP-T3D	34.15	5.3	8.97
bellmedt	52012	100	Y-MP	57.5	25.21	
		100	Y-MP-T3D	88.39	15.18	40.45
bellcist	66340	100	Y-MP	79.6	36.67	
		100	Y-MP-T3D	146.8	18.64	71.127
bellcrat	78942	100	Y-MP	77.7	39.48	
		100	Y-MP-T3D	122.0	18.1	63.93
belltect	327244	40	Y-MP	90.6	65.66	
		40	Y-MP-T3D	166.0	22.0	117.8

from running the kernels on the CRAY T3D, but this gain is more than offset by the time taken for exchange of the input and output vectors using PVM calls between the CRAY Y-MP and CRAY T3D. One way to improve communication bandwidth between the CRAY Y-MP and the CRAY T3D 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. From the results shown in Tables 4.6 and 4.7, we can predict that a three-fold increase in the communication rates between the CRAY Y-MP and CRAY T3D would yield performance gains for the the block-Lanczos method on the heterogeneous platform over using CRAY Y-MP only.

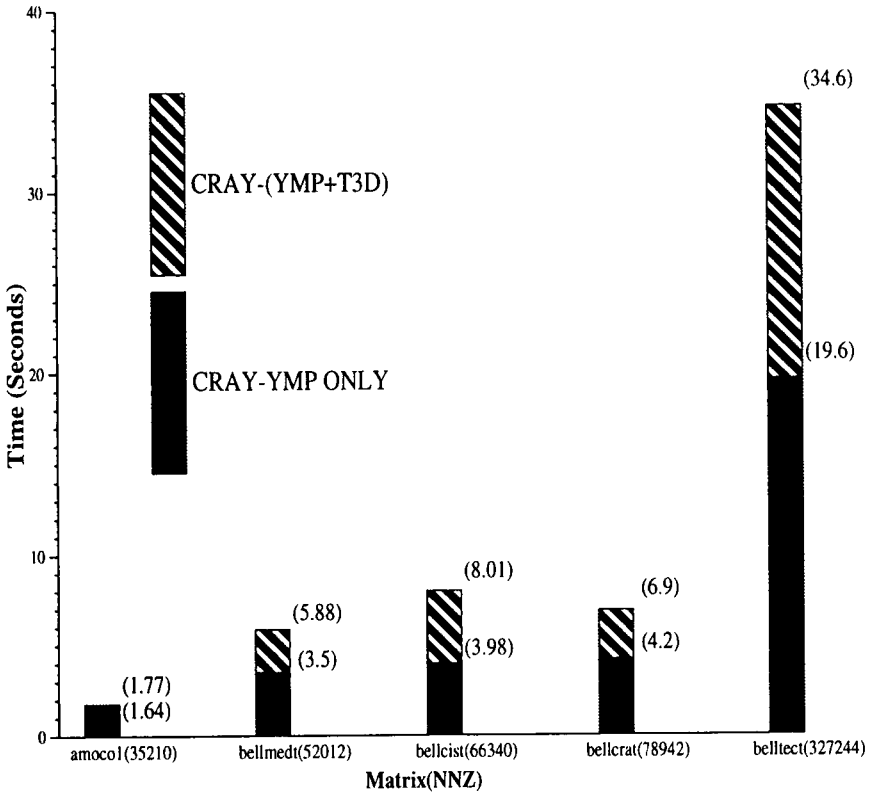


Figure 4.4: Time taken in seconds for block-Lanczos when computing the 10-largest singular triplets on the CRAY Y-MP only and on the heterogeneous platform consisting of a CRAY Y-MP and a CRAY T3D.

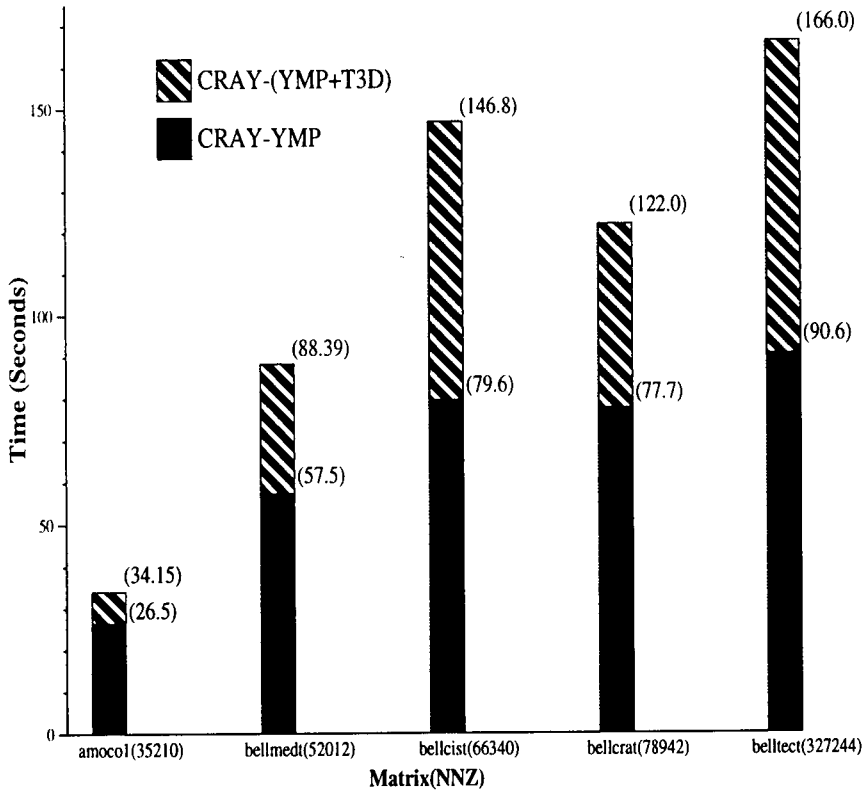


Figure 4.5: Time taken in seconds for block-Lanczos when computing the 100-largest singular triplets on the CRAY Y-MP only and on the heterogeneous platform consisting of a CRAY Y-MP and a CRAY T3D.

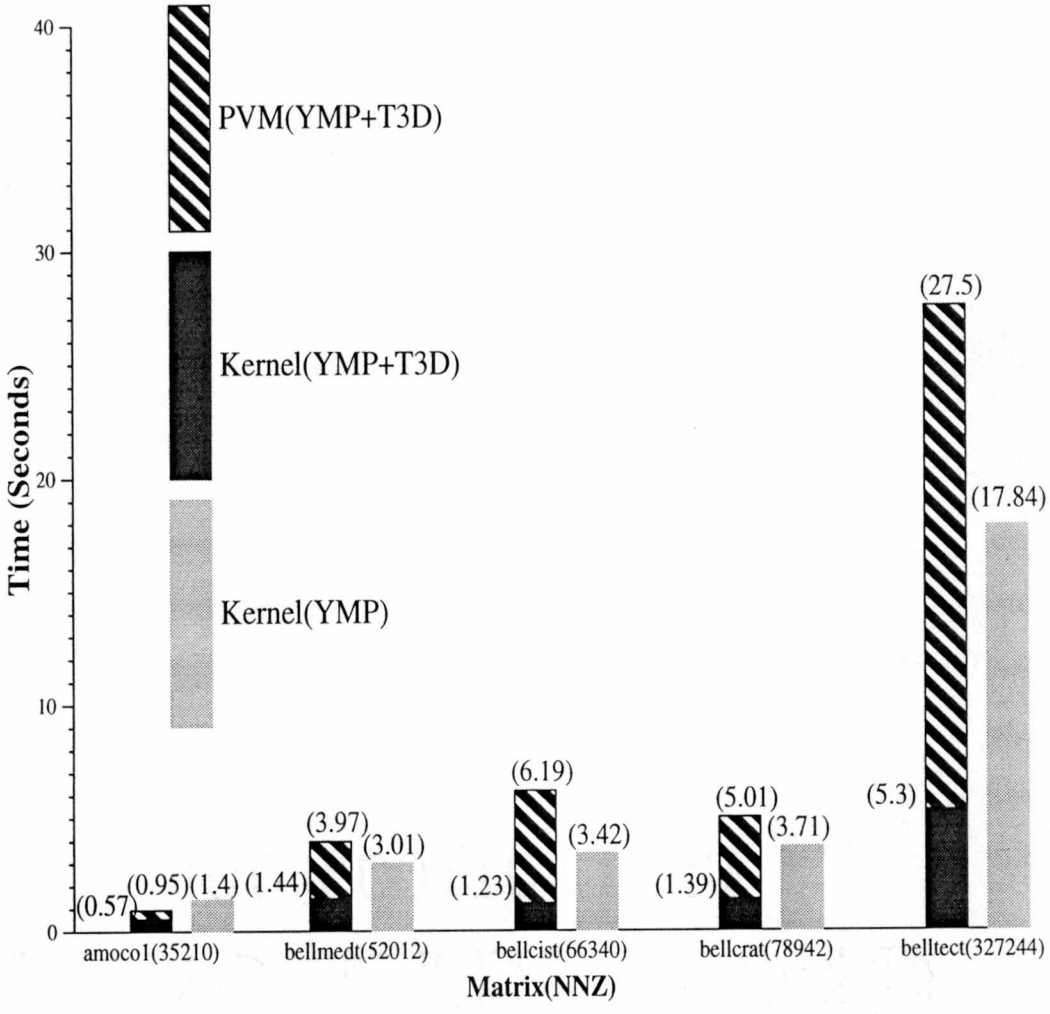


Figure 4.6: Time taken for sparse matrix-vector multiplication kernels on the CRAY Y-MP and time taken to run the kernels on the heterogeneous platform which includes time taken to exchange data using PVM calls and execution time on the CRAY T3D.

Chapter 5

Summary and Future Work

5.1 Summary

In this thesis, we have presented three algorithms for sparse matrix-vector multiplication along with detailed analysis of their complexity. The first algorithm discussed was block-block algorithm (BBA) which 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.

Row block (RBA) and column block (CBA) matrix-vector multiplication algorithms for rectangular sparse matrices were also presented. These algorithms were used in a heterogeneous implementation of a block-Lanczos method designed to find several of the largest singular triplets of large unstructured sparse matrices. In this implementation, all code except the sparse matrix-vector multiplication kernels ex-

ecuted on the CRAY Y-MP. The matrix-vector multiplication operations were done on the CRAY T3D. Results from this study show that the application significantly benefits from the execution of the kernels on the CRAY T3D, but that this gain is more than offset by the time taken for exchange of the input and output vectors using PVM calls between the CRAY Y-MP and the CRAY T3D.

5.2 Future Work

Future work may include modifying the block-block algorithm for structured sparse matrices such as block tridiagonal matrices arising from finite element-based computations. The fold operation used in the RBA and CBA could be modified to add only non-zero elements from vectors present in different processors. The communication bandwidth between the CRAY Y-MP and CRAY T3D must 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.

Bibliography

Bibliography

[BBLS91] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS Parallel Benchmarks. *NAS Systems Division Technical Report RNR-91-002*, NASA Ames Research Center, Moffet Field, CA, January 1991.

[Berr92a] M. Berry. Large scale singular value computations. *International Journal of Supercomputer Applications* 6:1 (1992), 13–49.

[Berr92b] M. Berry. SVDPACK: A Fortran-77 software library for the sparse singular value decomposition. *Computer Science Department Technical Report CS-92-159*, University of Tennessee, June 1992.

[BeGV94] M. Berry, C. Grassl, and V. Krishna. *Sparse Matrix-Vector Multiplication On The CRAY T3D*, In Preparation.

[FJLO88] 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.

[HeLP93] B. Hendrickson, R. Leland, and S. Plimpton. *A Parallel Algorithm for Matrix-Vector Multiplication*, Tech. Rep, SAND 92-2765, Sandia National Laboratories, Albuquerque, NM, March 1993.

[KnBa93] A. Knies, and R. Baruiso. *SHMEM User's Guide*, Cray Research Inc., Eagan, MN, August 1993.

[MPPO93] *MPP Overview*, Cray Research Inc., Eagan, MN, 1993.

[NuSP94] R. Numrich, P. Springer, and J. Peterson. Measurement of Communication Rates on the Cray T3D Interprocessor Network, Proceedings HPCN Europe, 1994.

[OgAi93] A. Ogielski, and W. Aiello. *Sparse matrix computations on parallel processor arrays*, SIAM J. Sci. Stat. Comput., 14:3 (1993), 519–530.

[Rigs93] P. Rigsbee. *PVM Reference Manual for CRAY Platforms*, Cray Research Inc., Eagan, MN, 1993.

[VaGe91] R. Van de Geijn. *On global combine operations*, LAPACK Working Note 29, Technical Report CS-91-129, University of Tennessee, 1991.

VITA

Vijay Kodangal Krishna was born in Hyderabad, India on September 26, 1969. He received his Bachelor of Computer Science Engineering from Chaitanya Bharathi Institute of Technology in August 1991. In January 1992, he entered the Computer Science program at the University of Tennessee and was awarded the Master of Science degree in Computer Science in August 1994.