10-1-1994

A SIMD SPARSE MATRIX-VECTOR MULTIPLICATION ALGORITHM FOR COMPUTATIONAL ELECTROMAGNETICS AND SCATTERING MATRIX MODELS

Nirav Harish Kapadia
Purdue University School of Electrical Engineering

A SIMD Sparse Matrix-Vector Multiplication Algorithm for Computational Electromagnetics and Scattering Matrix Models

Nirav Harish Kapadia

TR-EE 94-31
October 1994

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907-1285

A SIMD SPARSE MATRIX-VECTOR MULTIPLICATION ALGORITHM FOR COMPUTATIONAL ELECTROMAGNETICS AND SCATTERING MATRIX MODELS

Nirav Harish Kapadia

Purdue University

ABSTRACT

Kapadia, Nirav Harish. M.S.E.E., Purdue University. May 1994. A SIMD Sparse Matrix-Vector Multiplication Algorithm for Computational Electromagnetics and Scattering Matrix Models. Major Professor: Jose Fortes.

A large number of problems in numerical analysis require the multiplication of a sparse matrix by a vector. In spite of the large amount of fine-grained parallelism available in the process of sparse matrix-vector multiplication, it is difficult to design an algorithm for distributed memory SIMD computers that can efficiently multiply an arbitrary sparse matrix by a vector. The difficulty lies in the irregular nature of the data structures required to efficiently store arbitrary sparse matrices, and the architectural constraints of a SIMD computer. We propose a new algorithm that allows the "regularity" of a data structure that uses a row-major mapping to be varied by a changing a parameter (the "blocksize"). The (block row) algorithm assumes that the number of non-zero elements in each row is a multiple of the blocksize; (additional) zero entries are stored to satisfy this condition. The blocksize can be varied from one to N, where N is the size of the matrix; a blocksize of one results in a rcw-major distribution of the non-zero elements of the matrix (no oveahead of storing zero elements), while a blocksize of N results in a row-major distribution corresponding to that of a dense matrix. The algorithm was implemented on a 16,384 processor MasPar MP-1, and for the matrices associated with the applications considered here (S-Matrix Approach to Device Simulation, and the Modeling of Diffractive and Scattering Objects), the algorithm was faster than any of the other algorithms considered (the "snake-like" method, the "segmented-scan" method, and a randomized packing algorithm). For matrices that have a wide variation in the number of non-zero elements in each row, a procedure for an "adaptive" block row algorithm is briefly mentioned. The block row algorithm is applicable to unstructured sparse matrices which have relatively sparse columns (dense rows arc not a problem), and it can be implemented on any distributed memory computer.

ACKNOWLEDGMENTS

This report includes the results of the author's Master of Science Thesis. This research was funded in part by the National Science Foundation under grant number CDA-9015696.

TABLE OF CONTENTS

Page

Page

CHAPTER 1
INTRODUCTION

1.1 Motivation

Sparse matrix-vector multiplication forms the computational core of a large number of problems in numerical analysis. Typical problems involving the solution of large sparse linear systems using iterative methods can take several hours of CPU time on a high performance workstation, making parallel computers very attractive for these applications. Additionally, the process of sparse matrix-vector multiplication intrinsically involves a large amount of fine-grained parallelism, which makes it an ideal application for massively parallel SIMD computers.

However, it is difficult to design an algorithm for distributed memory SIMD computers that can efficiently multiply an arbitrary sparse matrix by a vector. The difficulty lies in the design of a data structure that can efficiently store arbitrary sparse matrices, allow most interprocessor communication to be "regular" (with respect to the interprocessor connection network of the machine), and at the same: time distribute the non-zero elements of the matrix evenly among the processors in the processor an-ay. On most commercially available SIMD computers, regular interprocessor communication (with adjacent processors or processors along the same row, for example) is faster than communicating with an arbitrary processor in the processor array; data structures designed to efficiently store arbitrary sparse matrices, however, tend to be irregular in nature. An additional constraint is imposed by a SIMD architecture because all enabled processors must perform the same operation at any given time. Thus any data structure designed for unstructured sparse matrix-vector multiplication on a SIMD computer must compromise between a good load balance, a data distribution that allows most, if not all communication to be regular, and an efficient storage format.

In this thesis, we present an algorithm for matrix-vector multiplication that was primarily developed for unstructured sparse matrices arising from two different applications - a finite element approach for the numerical analysis and modeling of diffractive and scattering objects, and a scattering matrix approach to device simulation. The algorithm has been implemented and tested on a 16,384 processor MasPar MP-1, and, for our applications, it was found to be faster than the randomized packing algorithms described in [OgA93], the "segmented-scan" algorithm described in [Ham92], and the "snake-like" method explained in [RoZ93].

1.2 Prior Work

While there have been several algorithms for the multiplication of unstructured sparse matrices by vectors, most of them are designed for single program, multiple data (SPMD) and multiple program, multiple data (MIMD) type architectures; a relatively few algorithms exist for the more restrictive SIMD model. Most algorithms that are designed for SPMD or MIMD models would not work efficiently on SIMD architectures without extensive modifications because of the restriction that all enabled processors in a SIMD machine must do the same operation at a given time (implicit synchronization). A brief overview of sparse matrix computations along with additional references can be found in [KuG94].

A look at algorithms for dense matrix-vector or matrix-matrix multiplication on massively parallel computers shows that these procedures can be parallelized very efficiently, thus resulting in peak performance that is close to the peak speed of the machine [JoH89, Tic89, BjM92]. This efficiency is difficult to carry over to sparse-matrix algorithms because the data structures that are typically used to store sparse matrices are irregular in nature (regular data structures can be designed for sparse matrices if they have specific sparsity patterns). Keeping this in mind., when selecting a data structure for our algorithm, we attempt to maximize the regularity of the data structure, while minimizing the overhead (of storing zero elements) that goes with a regular data structure.

In his paper [Pet91], Alexander Peters discusses the implementation of several sparse matrix-vector multiplication algorithms on a vector machine. The discussed algorithms use scalar and vector ITPACK storage schemes, or some variants of them. A brief description of parallelizable sparse matrix data structures can also be found in [MPP93, KuG94]. For unstructured matrices on a massively parallel computer, we find that a different storage scheme, such as the one used in [DuR79], is more efficient (discussed in Chapter 3).

In [BiW92], Bik and Wijshoff present a method in which the selection of a data structure is postponed until the compile phase, thus allowing the compiler to combine code optimization with explicit data structure selection. This method is not considered here because of the unavailability of the necessary compiler technology on the MasPar MP-1.

Several VLSI implementations have also been proposed for the efficient parallel solution of sparse systems [LiS88, Mel88, MiK93]. In general, these methods involve the use of special architectural features and/or specialized interprocessor routing methods, thus making their implementation on general-purpose computers unfeasible.

In their paper on sparse matrix-vector multiplication on the DAP, M. Morjaria and G. Makinson [MoM82] have presented a block partitioning method for the storage of large sparse matrices on a two-dimensional mesh processor array. This method was improved upon by J. Anderson et. al. who used a less compact data structure and a heuristic scheduling procedure that enabled them to exploit more parallelism and reduce the amount of interprocessor communication [AnM92].

Romero and Zapata [RoZ93] have proposed two methods for sparse matrix-vector multiplication in multiprocessor computers with a two-dimensional mesh interconnection network and a distributed memory system: multiple recursive decomposition, and the block row scatter method. Multiple recursive decomposition involves dividing the matrix into submatrices such that each submatrix has approximately the same number of non-zero elements. In general, each submatrix will have a different size, which makes this method unsuitable for SIMD computers. A randomized packing algorithm proposed in [OgA93] achieves a similar load distribution, and at the same time divides the matrix into submatrices of equal size (except, possibly the ones on the edges). The block row scatter method is an improvement on the scatter methods presented in [AnM92], where the matrix is again divided into submatrices of size equal to the size of the processor array. This method is also conceptually similar to the one proposed in [OgA93]. A survey of the different data distributions for sparse matrix-vector multiplication on multiprocessor systems can be found in [RoZ93]. Some additional methods are also reviewed in [Ham92].

A. Ogielski and W. Aiello present two randomized packing algorithms that randomly permute the positions of the non-zero elements in the matrix before using a block partitioning method to store it in the processor array [OgA93]. The algorithms are implemented on a MasPar MP-1, and the distribution of the matrix elements is done so as to allow all interprocessor communications to be done using regular cornrnunication primitives. Scatter and gather techniques are used to perform the matrix-vector multiplication in parallel.

1.3 Results

We propose a new algorithm that allows the "regularity" of a data structure that uses a row-major mapping to be varied by a changing a parameter (the "blocksize"). The (block row) algorithm assumes that the number of non-zero elements in each row is a multiple of the blocksize; (additional) zero entries are stored to satisfy this condition. The blocksize can be varied from one to N, where N is the size of the matrix; a blocksize of one results in a row-major distribution of the non-zero elements of the matrix (no overhead of storing zero elements), while a blocksize of N results in a row-major distribution corresponding to that of a dense matrix. For matrices that have a wide variation in the number of non-zero elements in each row, a procedure for an "adaptive" block row algorithm is mentioned. The only assumption made about the matrix is that its columns are "sparse".

The block row algorithm was implemented on a 16,384 processor MasPar MP-1, and its performance was compared to that of three other algorithms. For the types of matrices under consideration, we found that our algorithm was up to an order of magnitude faster than the second randomized packing algorithm. described in [OgA93]. Of the two applications, for the finite element approach, our algorithm was about nine times faster than the randomized packing algorithm for the largest case, while for the scattering matrix approach, it was faster by a factor of two. In addition, the block row algorithm is much more memory-efficient - for the largest problem solved, involving 93,602 unknowns, and 1,427,614 non-zero elements, the block row algorithm used approximately 36 MBytes of memory, whereas the randomized packing algorithm of [OgA93] used approximately 237 MBytes (1,427,614 elements can be stored in approximately 11 MBytes of memory, using double precision).

1.4 Overview

The thesis is organized as follows. Chapter 2 deals with the analysis of parallel sparse matrix-vector multiplication on SIMD computers. Chapter 3 is divided into two main parts; in the first part, we present a brief description of the MasPar MP-1 computer, while the second part deals with the architecture-specific (to the MP-1, but generalizable to SIMD machines) issues of parallel sparse matrix-vector multiplication. The 'block row algorithm' is presented in Chapter 4, along with a theoretical analysis. Chapter 5 provides an experimental and a comparative analysis of the algorithm. Brief descriptions of the scattering matrix approach and the finite element approach along with experimental data for simulations are presented in Chapter 6 and Chapter 7, respectively. Finally, in Chapter 8, we conclude the thesis and present some ideas for future work. The code (in MPL) for the block row algorithm can be found in the appendix.

CHAPTER 2
SPARSE MATRIX-VECTOR MULTIPLICATION

2.1 Introduction

In this chapter, we analyze the basic procedure of sparse matrix-vector multiplication, and compare the differences in the sequential and parallel (SIMD) implementations of the procedure. We also provide a "generic" analysis for the procedure; this analysis forms the basis on which the algorithms in the next chapter are developed. Because the following (parallel) analysis is for SIMD machines, there is an implicit assumption that all enabled processors have to perform the same operation at any given time.

2.2 Sequential Sparse Matrix-Vector Multiplication

Consider the problem of matrix-vector multiplication with the notation b = Ax, where A is a sparse matrix of size N by N. Each element of the result vector can be computed as

$$b_i = \sum_j (a_{ij} \times x_j) , \tag{2.1}$$

where a_{ij} is a non-zero element of the matrix A.

On a sequential computer, the result vector is computed one element at a time, by computing the relevant products (Equation 2.1) and adding them. The actual order in which the computations are performed may vary depending on the architecture and the memory subsystems of the computer. Let R_i represent the number of non-zero elements in row i. Then, the sequential algorithm involves $\sum_{i=0}^{i<N} (R_i - 1)$ addition

operations and $\sum_{i=0}^{i<N} (R_i)$ multiplication operations. The time required to perform one matrix-vector multiplication using a sequential algorithm is

$$t_{sequential} = t_{add} + t_{multiply} , \qquad (2.2a)$$

where

$$t_{add} = c_1 \times \sum_{i=0}^{i<N} (R_i - 1) \text{ , and} \qquad (2.2b)$$

$$t_{multiply} = c_2 \times \sum_{i=0}^{i<N} (R_i) \qquad (2.2c)$$

Let $R_{\max}$ be the maximum number of non-zero elements in any one row of the sparse matrix A. Then, the complexity of the sparse matrix-vector multiplication operation is $O(R_{max}.N)$. The complexity can also be represented in terms of the total number of non-zero elements in the matrix (N_{elts}). The complexity of the algorithm in terms of N_{elts} is $O(N_{elts})$, where $N_{elts} = \sum_{i=o}^{i<N} (R_i)$.

2.3 Parallel (SIMD) Sparse Matrix-Vector Multiplication

2.3.1 Analysis of Parallel Sparse Matrix-Vector Multiplication

In contrast to the sequential implementation, an effective matrix-vector multiplication procedure for a massively parallel SIMD machine is quite different. The discussion is divided into two parts; the first part is based on the assumption that the number of processors in the processor array (N_{proc}) is greater than (or equal to) the size of the matrix (N) and the number of non-zero elements in the matrix (N_{elts}); that is, $N_{proc} \geq \max(N, N_{elts})$. The discussion in the second part deals with the cases where these assumptions are not true. This approach results in a clearer analysis of the problem.

The processors in the processor array can be visualized as a one dimensional array of processors. We assume that each processor can simultaneously support one incoming and one outgoing communication operation, and that interprocessor communication involving any permutation of processors can be done in one parallel operation. We also assume that the non-zero elements of the matrix and the elements of the vector are distributed in the processor array using some (unspecified) mapping method. Then, the process of parallel sparse matrix-vector multiplication can be divided into several basic steps, as considered below. The actual implementation of the algorithm may include additional steps to optimize the performance; they are ignored for now.

2.3.2 Case I ($N_{proc} \geq$ N, $N_{proc} \geq N_{elts}$)

2.3.2.1 Introduction

For the purpose of this discussion, and without loss of generality, we assume that each enabled processor of the machine has exactly one non-zero element of the matrix, and that the size of the matrix (N) is less than the number of processors in the processor array (N_{proc}). If the number of non-zero elements in the matrix (N_{elts}) is less than the number of processors in the processor array, some processors can be disabled. On the other hand, if some or all processors have more than one element, each element in a given processor needs to be processed sequentially. Similarly, if the size of the matrix is greater than the number of processors in the processor array, some or all processors will have multiple elements of the vector, and each element in a given processor will have to be processed sequentially. These cases require a virtual mapping of the data, and are considered later.

2.3.2.2 Procedure

Each processor that has a non-zero element of the matrix (a_{ij}) must first fetch the corresponding vector element (x_j) from the memory of the processor where it is stored. Because each processor can process only one communication request at one time, it is most efficient to store each vector element on a different processor. With this storage scheme, distinct vector elements can be fetched simultaneously; if more than one processor requires a particular vector element, each fetch for that vector element will have to be processed sequentially. Thus, the entire :€etch operation requires $C_{\max}$ parallel communication operations, where $C_{\max}$ is the maximum number of non-zero elements in any one column of the matrix.

Once all the vector elements have been fetched, each processor multiplies the local copy of the non-zero matrix element (a_{ij}) by the vector element that was just fetched (x_j). The resulting product ($a_{ij} \times$ x,) is called a partial product. All processors perform the multiplication in one parallel operation.

Once the partial products are available, the partial products from each row (say i) of the matrix must be added together to form the result-vector elements (b_i). The addition can be performed using a procedure known as recursive *doubling.* Using recursive doubling, n numbers can be added in $\lceil \log_2(n) \rceil$ steps, where each step consists of one communication (parallel) and one addition (parallel) operation.

Depending on the specific data layout and the architecture of the machine, it may not be feasible to use recursive doubling. In practice, for large problenis (as compared to the number of processors), the most efficient methods use a combiriation of (local) linear addition and recursive doubling to add the partial products. For the purposes of this analysis, we assume that it is feasible to use recursive doubling.

Finally, the elements of the result-vector must be sent to the appropriate processors to form a complete vector. If we assume that the layout of the result-vector is the same as that of the original vector (necessary for any iterative scheme), then this operation can be done in one parallel operation because each element will be stored on a distinct processor.

The sequence of operations just described is summarized below. Each of these steps is a parallel operation, and is executed by all processors that have a non-zero element of the matrix.

1. Fetch the required vector element (x_j).
2. Perform a local multiply ($c_{ij} = a_{ij} \times x_j$).
3. Add the partial products ($b_i = \sum_j c_{ij}$).
4. Put the result vector element in the appropriate processor.

2.3.2.3 Timing Analysis for Case I

Based on the analysis above, we now obtain an expression for the time taken to perform one sparse matrix-vector multiplication. The time taken to fetch the vector elements is proportional to the maximum number of non-zero elements in any one column; that is, fetching the vector elements takes $c_1 \times C_{\max}$ units of time, where c_1 is a constant. The multiplication operation can be completely parallelized, and so it can be executed in constant time (equal to c_2, say). The partial products can be added in $c_3 \times \lceil \log_2(R_{\max}) \rceil$ time units, where c_3 is a constant, and $R_{\max}$ is the maximum number of non-zero elements in any one row of the matrix. Finally, the results can be moved to the appropriate processors in constant (c_4) time. Thus, the time taken by the entire procedure is equal to:

$$t_{parallel} = t_{fetch} + t_{multiply} + t_{add} + t_{arrange} , \tag{2.3a}$$

where

$$t_{fetch} = c_1 \times C_{\max} , \tag{2.3b}$$

$$t_{multiply} = c_2 , \tag{2.3c}$$

$$t_{add} = c_3 \times \lceil \log_2(R_{\max}) \rceil , \text{ and} \tag{2.3d}$$

$$t_{arrange} = c_4 . \tag{2.3e}$$

It must be emphasized that these results are based on the assumptions made about the communication capabilities of the machine in Section 2.3.1. Then, under the assumptions that each processor can support exactly one incoming and one outgoing communication simultaneously, and that interprocessor communication involving any permutation of processors can be done in one parallel operation, if $N_{proc} \geq \max(N, N_{elts})$, the complexity of the parallel sparse matrix-vector multiplication operation is $O(C_{max} + \log_2 R_{max} + 1)$. In practice, because of the fact that it may not be possible to communicate data across any permutation of processors in one parallel operation, the above expression is actually a ***lower bound*** for the procedure of matrix-vector multiplication. In addition, depending on the architecture of the machine, and the method used to map the matrix into the processor array, the partial products may have to be reduced by using linear addition (as opposed to recursive doubling). On the other hand, a specific machine may be able to send/receive more than one simultaneous communication from each processor, which would modify the expressions obtained above.

However, assuming that the assumptions hold, the procedure involves $C_{max} + \lceil \log_2(R_{max}) \rceil + 1$ parallel communication operations, one parallel multiplication operation, and $\lceil \log_2(R_{max}) \rceil$ parallel addition operations.

2.3.3 Case II ($N > N_{proc}$, and *lor* $N_{elts} > N_{proc}$)

2.3.3.1 Introduction

The analysis resulting in Equation 2.3a was based on the assumptions that each enabled processor had exactly one non-zero element of the matrix, and that the length of the vector was less than or equal to the number of processors in the processor array. We now consider the cases where this is not true.

2.3.3.2 Case IIa ($N \leq N_{proc}$, and $N_{elts} > N_{proc}$)

If the number of non-zero elements in the matrix exceeds the number of processors in the processor array, some or all processors will have more than one element of the matrix; that is, the non-zero elements will be distributed in multiple *layers* (of memory) in the processor array.

A plural variable exists on all the processors of the processor array, and it can have a different value on each processor. Thus, on a processor array with N_{proc} processors, a plural variable can be interpreted as a one-dimensional array of size N_{proc} (Figure 2.1a). Similarly, a one dimensional plural array of size M is actually a two dimensional array of size N_{proc} by M, where each processor has one column (M locations) of the array (Figure 2.1b). The dimension along the processors (along N_{proc}) is called a layer, and so the above array would have a depth of *M* layers.

The first N_{proc} non-zero elements of the matrix will be mapped into the first layer (of data), the next N_{proc} non-zero elements will be mapped into the second layer, and so on. Let L_{elts} represent the number of layers that the non-zero elements of the matrix are mapped into. Then,

$$L_{elts} = \lceil N_{elts}/N_{proc} \rceil . \tag{2.4}$$

If $L_{elts} > 1$, each layer must be processed sequentially, and the quantities obtained in Equations 2.3b - 2.3e need to be multiplied by L_{elts}. The quantity C_{max} must be redefined as $C_{max,k}$, where $C_{max,k}$ represents the maximum non-zero elements from any one column of the matrix that are stored in layer 'k'. Similarly, $R_{max,k}$ represents the maximum number of non-zero elements from any one row of the matrix that are stored in layer 'k'. It should be noted that '$C_{max,k}$' and '$R_{max,k}$' are no longer constants based on the matrix; their values depend on the architecture of the machine and the method used to distribute the matrix into the processor array ($C_{max,k}$, $R_{max,k} \leq N_{proc}$). Then, the time taken for a matrix-vector multiplication is given by:

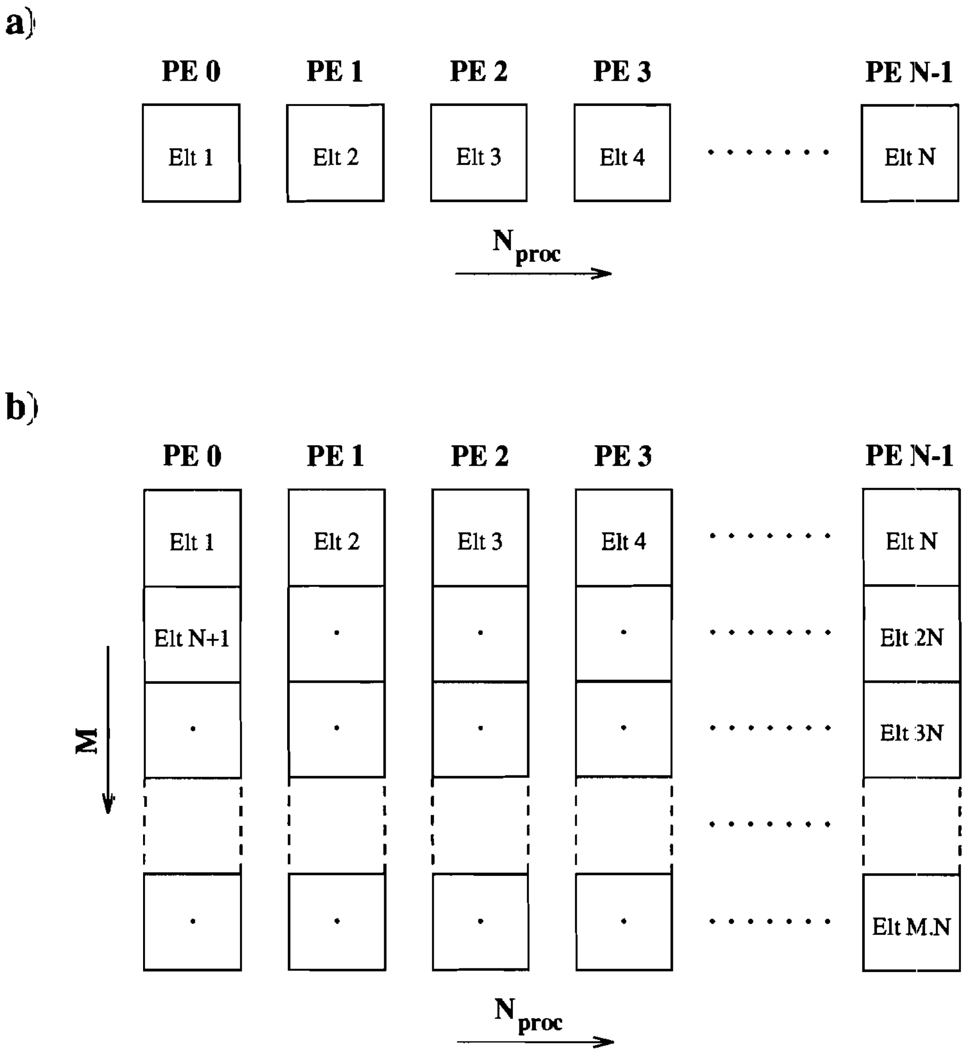


Figure 2.1: a) A plural variable as a one dimensional array of size N_{proc}, and b) A plural one dimensional array as a two dimensional array of size N_{proc} by M.

$$t_{parallel} = \sum_{k=0}^{k<L_{elts}} \left[t_{fetch,\ k} + t_{multiply} + t_{add,\ k} + t_{arrange} \right] , \tag{2.5a}$$

where

$$t_{fetch,\ k} = c_1 \times C_{max,k} , \tag{2.5b}$$

$$t_{multiply} = c_2 , \tag{2.5c}$$

$$t_{add,\ k} = c_3 \times \left\lceil \log_2(R_{max,k}) \right\rceil , \text{ and} \tag{2.5d}$$

$$t_{arrange} = c_4 . \tag{2.5e}$$

2.3.3.3 Case IIb (N > N_{proc}, and $N_{elts} \leq N_{proc}$)

On the other hand, if the size of the matrix exceeds the number of processors in the processor array, then one or more processors will have multiple elements of the vector. This affects the time required to fetch the vector elements and to arrange the result vector elements because each processor can support only one communication request at a time.

Let there be N_{proc} processors in the processor array, and let the length of the vector be N. Assume that the processors and the vector elements are numbered from zero to $N_{proc}-1$ and zero to $N-1$, respectively. Assume that the vector itself is stored completely; that is, all the elements of the vector are stored, even if they are zero. Also assume that the consecutive elements of the vector are stored on adjacent processors (assuming a one-dimensional model of the processor array), and that the vector is "wrapped around" in the processor array (Figure 2.1b); that :is, if x_i is stored in the last processor of the processor array, then x_{i+1} is stored in the first processor of the processor array. Then, the vector elements x_i, $x_{i+N_{proc}}$, $x_{i+2N_{proc}}$,...., and $x_{i+jN_{proc}}$, such that $j = 0,\ 1,\ 2,...$ and i + jN_{proc} < N, are stored in processor i.

Let L_{vect} represent the number of ***layers*** required to store the vector. The number of ***layers*** is equal to the maximum number of vector elements that are stored on any one processor. Then,

$$L_{vect} = \lceil N/N_{proc} \rceil \ . \tag{2.6}$$

The number of processors that would try to fetch one of the vector elements stored in processor i is given by the expression

$$C_{i(eff)} = \sum_{j=0}^{i+jN_{proc}<N} C_{i+jN_{proc}} \ , \tag{2.7}$$

where C_k (k = i + jN_{proc}) is the number of non-zero elements in the k^{th} column of the matrix ($0 \le k < N$).

Then, the maximum number of processors that would try to fetch one of the vector elements stored in any given processor is equal to

$$C_{max(eff)} = \max_{0 \le i < N} \left[\sum_{j=0}^{i+jN_{proc}<N} C_{i+jN_{proc}} \right] . \tag{2.8}$$

Thus, the time required to fetch the vector elements is given by

$$t_{fetch} = c_1 \times C_{max(eff)} \ .$$

The multiplication and addition steps are not affected by the size of the vector. Once the result-vector elements are computed, they have to be sent to the appropriate processors. Each vector element that is stored in a given processor must be sent sequentially because the destination processors can only handle one incoming communication at a time.

The time required for the matrix-vector multiplication when the size of the vector is greater than the number of processors in the processor array is

$$t_{parallel} = t_{fetch} + t_{multiply} + t_{add} + t_{arrange} \ , \tag{2.9a}$$

where

$$t_{fetch} = c_1 \times C_{max(eff)} , \tag{2.9b}$$

$$t_{multiply} = c_2 , \tag{2.9c}$$

$$t_{add} = c_3 \times \left\lceil \log_2(R_{\max}) \right\rceil , \text{ and} \tag{2.9d}$$

$$t_{arrange} = c_4 \times L_{vect} . \tag{2.9e}$$

2.3.3.4 Case IIc ($N > N_{proc}$, and $N_{elts} > N_{proc}$)

In the general case, both, the matrix size, and the number of non-zero elements in the matrix may be greater than the number of processors in the processor array. The result follows from a combination of the analyses in Case IIa and Case IIb.

$C_{max(eff)}$ must be redefined as $C_{max(eff),\,k}$; the term represents the maximum number of processors that need to fetch any one of the vector elements stored in any given processor, in layer 'k'.

The time required to send the result vector elements to the appropriate processors depends on which result vector elements are computed in a particular layer. If more than one of the result vector elements that are computed in a given layer need to be sent to the same processor, the send operation will have to be serialized. A worst case estimate is $c_4 \times L_{vect}$ time units.

The time taken to do the matrix-vector multiplication in the general case is given by:

$$t_{parallel} = \sum_{k=0}^{k < L_{elts}} \left[t_{fetch,\,k} + t_{multiply} + t_{add,\,k} + t_{arrange} \right] , \tag{2.10a}$$

where

$$t_{fetch,\,k} = c_1 \times C_{max(eff),\,k} , \tag{2.10b}$$

$$t_{multiply} = c_2 \,, \tag{2.10c}$$

$$t_{add,\,k} = c_3 \times \left\lceil \log_2(R_{\max,\,k}) \right\rceil \text{, and} \tag{2.10d}$$

$$t_{arrange} = c_4 \times L_{vect} \,. \tag{2.10e}$$

In Equations 2.10a - 2.10e, $t_{arrange}$ represents the worst-case time, whereas the other quantities represent execution times that would depend on the sparsity pattern of the matrix and on the mapping of the data in the processor array. A summary of the results for Case I and Case II are given in Table 2.1. To determine an upper bound for the procedure of sparse matrix-vector multiplication, a worst case analysis results in more compact results.

2.3.3.5 Worst-Case Analysis

The worst case value for $C_{max(eff),\,k}$ is equal to the smaller of: a) the maximum number of non-zero elements in any one column of the matrix multiplied by the maximum number of vector elements that are stored in any one processor, and b) the number of processors in the processor array. The upper bound for $R_{max,\,k}$ is the smaller of the maximum number of non-zero elements in any one row of the matrix, and the number of processors in the processor array. Using this in Equations 2.10b - 2.10e, we have

$$t_{parallel} = L_{elts} \times (t_{fetch} + t_{multiply} + t_{add} + t_{arrange}) \,, \tag{2.11a}$$

where

$$t_{fetch} = c_1 \times L_{vect} \times \min(C_{\max}, N_{proc}) \,, \tag{2.11b}$$

$$t_{multiply} = c_2 \,, \tag{2.11c}$$

$$t_{add} = c_3 \times \left\lceil \log_2(\min(R_{\max}, N_{proc})) \right\rceil \text{, and} \tag{2.11d}$$

$$t_{arrange} = c_4 \times L_{vect} \,. \tag{2.11e}$$

Table 2.1: A summary of the timing analysis for the parallel (SIMD) sparse matrix-vector multiplication procedure (assuming that recursive doubling is used to add the partial products).

Parallel Sparse Matrix Vector Multiplication		
	$N_{proc} \geq N_{elts}$ and $N_{proc} \geq N$	$N_{proc} < N_{elts}$ **and/or** $N_{proc} < N$
t_{fetch}	$c_1 \times C_{max}$	$c_1 \times \sum_{k=0}^{k < L_{elts}} C_{max(eff), k}$
$t_{multiply}$	c_2	$c_2 \times L_{elts}$
t_{add}	$c_3 \times \lceil \log_2(R_{max}) \rceil$	$c_3 \times \sum_{k=0}^{k < L_{elts}} \lceil \log_2(R_{max, k}) \rceil$
$t_{arrange}$	c_4	$c_4 \times L_{elts} \times L_{vect}$

The worst case fetch time is proportional to $L_{elts} \times L_{vect} \times \min(C_{\max}, N_{proc})$, and the worst case add time is proportional to $L_{elts} \times \log_2(\min(R_{\max}, N_{proc}))$. This translates to a complexity of $O(L_{eff}.\min(C_{\max}, N_{proc}) + L_{eff}.\log_2(min(\mathrm{R}_{\max}, N_{proc})))$, where

$$L_{eff} = L_{elts} \times L_{vect} , \tag{2.12}$$

and L_{elts} and L_{vect} are given by Equation **2.4** and Equation 2.6, respectively. Note that if both, the size of the matrix and the number of non-zero elements in the matrix are less than (or equal to) the number of processors in the processor array, $L_{elts} = L_{vect} = 1$, and $C_{\max}$ and $R_{\max}$ are less than (or equal to) the number of processors in the processor array, and the expression for the complexity reduces to $O(C_{\max} + \log_2 R_{\max})$. A summary of the results for the worst-case analysis is presented in Table 2.2.

2.4 Result Summary

If the size of the matrix and the total number of non-zero elements in the matrix are both less than (or equal to) the number of processors in the processor array, the data can be directly mapped on the processors - that is, there is no need for virtual mapping. For this case, a comparison of the complexity of the serial and the parallel algorithms and the number of steps involved in the sequential and parallel sparse miitrix-vector multiplication operations is given in Table **2.3.** In the more general case, where the above assumption is not true, the same data is shown in Table **2.4.**

2.5 Conclusions

In this chapter, we have provided a "generic" analysis for sparse matrix-vector multiplication on a SIMD machine and compared it with a sequential implementation. The analysis is "generic" in the sense that it includes only a limited amount of information about the distribution of the data across the processor array, and there are

Table 2.2: A summary of the worst-case timing analysis for the parallel (SIMD) sparse matrix-vector multiplication procedure (assuming that recursive doubling is used to add the partial products).

<table>
<tr><th colspan="3">Parallel Sparse Matrix Vector Multiplication
(Worst-Case Analysis)</th></tr>
<tr><th></th><th>$N_{proc} \geq N_{elts}$ and $N_{proc} \geq N$</th><th>$N_{proc} < N_{elts}$ and/or $N_{proc} < N$</th></tr>
<tr><td>t_{fetch}</td><td>$c_1 \times C_{max}$</td><td>$c_1 \times L_{elts} \times L_{vect} \times \min(C_{max}, N_{proc})$</td></tr>
<tr><td>$t_{multiply}$</td><td>c_2</td><td>$c_2 \times L_{elts}$</td></tr>
<tr><td>t_{add}</td><td>$c_3 \times \lceil \log_2(R_{max}) \rceil$</td><td>$c_3 \times L_{elts} \times \lceil \log_2(\min(R_{max}, N_{proc})) \rceil$</td></tr>
<tr><td>$t_{arrange}$</td><td>c_4</td><td>$c_4 \times L_{elts} \times L_{vect}$</td></tr>
</table>

Table **2.3:** Complexity and number of operations involved in the sequential and parallel algorithms for sparse matrix-vector **multiplication** when the size of the matrix and the total number of non-zero elements in the matrix are less than or equal to the number of processors in the processor array.

	Sequential Algorithm	**Parallel Algorithm**	**Ratio (Parallel/Sequential)**
Complexity	$O(R_{max}.N)$	$O(C_{max} + \log_2(R_{max}) + 1)$	-
Addition Operations	$\sum_{i=0}^{i<N}(R_i) - 1$	$\lceil \log_2(R_{max}) \rceil$	$\geq \frac{N_{elts} - 1}{\lceil \log_2(N) \rceil}$
Multiplication Operations	$\sum_{i=0}^{i<N}(R_i)$	1	N_{elts}
Communication Operations	-	$C_{max} + \lceil \log_2(R_{max}) \rceil + 1$	-
Total Operations	$2 \times N_{elts} - 1$	$C_{max} + 2\lceil \log_2(R_{max}) \rceil + 2$	$\geq \frac{2N_{elts} - 1}{N + 2\lceil \log_2(N) \rceil + 2}$

Table 2.4: Complexity and number of operations involved in the sequential and parallel algorithms for sparse matrix-vector multiplication when either the size of the matrix or the total number of non-zero elements in the matrix (or both) are greater than the number of processors in the processor array.

	Sequential Algorithm	**Parallel Algorithm**	**Ratio (Parallel/Sequential)**
Complexity	$O(R_{max}.N)$	$O(L_{eff}.\{\min(C_{max}, N_{proc}) + \log_2(\min(R_{max}, N_{proc}))\})$	
Addition Operations	$\sum_{i=0}^{i<N} (R_i) - 1$	$\sum_{k=0}^{k<L_{elts}} \lceil \log_2 R_{max, k} \rceil$	$\geq \frac{N_{elts} - 1}{L_{elts}.\min(R_{max}, N_{proc})}$
Multiplication Operations	$\sum^{i<N} (R_i)$	L_{elts}	$\frac{N_{elts}}{L_{elts}}$
Communication Operations		$\sum_{k=0}^{k<L_{elts}} (C_{max(eff), k} + \lceil \log_2(R_{max, k}) \rceil + L_{vect})$	-
Total Operations	$2 \times N_{elts} - 1$	$\sum_{k=0}^{k<L_{elts}} (C_{max(eff), k} + 2\lceil \log_2(R_{max, k}) \rceil + L_{vect} + 1)$	$\geq \frac{2N_{elts} - 1}{L_{elts}(2\lceil \log_2(N_{proc}) \rceil + 1) + L_{eff}(N_{proc} + 1)}$

no assumptions about the type of interconnection network present in the machine (except for the specific assumptions involving non-conflicting interprocessor communication).

As explained in this chapter, the procedure of multiplying a sparse matrix by a vector can be divided into four separate phases on a parallel computer. The first part of the analysis provides an insight into the effect of virtual mapping of data (Table 2.1); that is, what happens when either the size of the matrix or the number of non-zero elements in the matrix exceed the number of processors available: on the parallel computer. This is an important factor because, in the general case, the problem sizes of interest will require a virtual mapping (of data). Note that the 'virtually mapped data' that we talk about does not involve any swapping to secondary storage. We observe that increasing the number of elements in the vector (i.e., the size of the matrix) affects t_{fetch} and $t_{arrange}$, while increasing the number of non-zero elements in the matrix affects all the phases (Table 2.2). Based on the expressions in Table 2.1 (and Table 2.2), we can expect that t_{fetch} and t_{add} account for a large fraction of the time taken to perform a sparse matrix-vector multiplication. We can also expect that increasing the size of the matrix (while keeping the number of non-zero elements constant) will not increase the time (to perform the matrix-vector multiplication) as much as an increase in the number of non-zero elements will.

For the case where the size of the matrix and the number of non-zero elements are greater than the number of processors in the processor array, the results in Table 2.1 are dependent on the distribution of the non-zero elements in the memory of each processor, and across the processor array. To give a better "feel" for the results, a worst-case analysis is also performed, and the results are presented in Table 2.2. The results clearly show that even though the performance of the parallel implementation of sparse matrix-vector multiplication is dependent on the sparsity pattern of the matrix and the distribution (of the matrix in the processor array) selected, there is a definite lower bound on the performance, which is dependent on the number of processors in the processor array (Table 2.3 and Table 2.4). The actual algorithm used to perform the sparse matrix-vector multiplication will be based on the method used to distribute the data across the processor array, and so, as the number of processors in the processor array increases, the importance of using a "good" distribution (of the matrix in the processor array) also increases (scalability issue).

Finally, we compare the parallel procedure (of sparse matrix-vector multiplication) with the sequential procedure (Table 2.3 and Table 2.41). We find that as the number of processors in the parallel computer increases, the number of (sequential) addition and multiplication operations per processor decrease. In the best case (the number of processors are greater than or equal to the number of data items), apart from the communication overhead, the parallel procedure involves $\lceil \log_2(R_{max}) \rceil$ addition operations and one multiplication operation. This number can be significantly smaller than the corresponding numbers in the sequential procedure where there are $N_{elts}-1$ addition operations and N_{elts} multiplication operations.. The parallel procedure, however, involves interprocessor communication operations that are not present in the sequential procedure, which limits the amount of speedup that can be obtained over the sequential procedure.

The analyses in this chapter give us an idea of the constraints imposed on a parallel implementation of sparse matrix-vector multiplication on SIMD machines. In the next two chapters, we use this analysis in conjunction with the architectural features of the MasPar MP-1 to develop a new sparse matrix-vector multiplication algorithm.

CHAPTER 3
SPARSE MATRIX-VECTOR MULTIPLICATION ON THE MASPAR MP-1

3.1 Introduction

This chapter deals with the implementation details of sparse matrix-vector multiplication on a SIMD computer. Specific details regarding the design and iniplementation of data structures are provided; these form a basis for the design of the data structure used for the block row algorithm (Chapter 4). 'The architectural specifications of the MasPar MP-1, a 16,384 processor SIMD computer, are used for the discussion in this chapter, but the analysis can be easily extended to other distributed memory SIMD computers.

3.2 The MasPar MP-1 Computer

3.2.1 Introduction

The MasPar MP-1 is a massively parallel SIMD computer with up to 16,384 processing elements. This section provides a brief introduction to the architectural features of the computer.

3.2.2 The Processor Array

The MP-1 has a single instruction, multiple data (SIMD) architecture with 1,024 tc 16,384 processors. The processors are called processing elements *(PEs)* because they contain only the data path and no instruction logic. The instructions are fetched and decoded by the Array Control Unit (ACU), which is a scalar processor with a

a)

PEO	PE 1	PE2	PE3
PE4	PE 5	PE6	PE7
PE 8	PE 9	PE 10	PE 11
PE12	PE13	PE14	PE 15

b)

PE 0	PE 1	. . .	PE nxproc-1
PE nxproc	PE nxproc+1	. . .	PE 2nxproc-1
⋮	⋮	. . .	⋮
PE nyproc X nxproc	PE nyproc X (nxproc+1)	. . .	PE nproc-1

Figure 3.1: a) A cluster of processing elements, and b) The Processor Element Array.

RISC-style instruction set. The processing elements together form the Processor *Element* Array, and, the Array Control Unit and the Processor Element Array together form the Data Parallel Unit (DPU) [Nic90].

The processing elements are divided into clusters of sixteen processing elements each, and the processing elements in a cluster are logically arranged as a four-by-four array to form a two-dimensional mesh connection (Figure 3.1). Each printed circuit board contains 64 clusters, resulting in 1,024 processing elements. In the MP-I, each processing element has 48 32-bit registers, of which 40 are available to the programmer, and sixteen kilobytes of local memory. Thus, a 16,384 processing element system has 256 megabytes of memory [Nic90].

3.2.3 Interprocessor Communication

Interprocessor communication is handled by two different networks. One is the X-Net, which is functionally equivalent to an eight nearest-neighbor two-dimensional *mesh* network. The connections at the edge of the Processor Elernent Array are wrapped around to form a torus. The other network is the Global *Router* Network, which is used to handle arbitrary communication patterns between processing elements. Each cluster of sixteen processing elements shares one originating port and one target port. So, the router network can support as many simultaneous connections as there are clusters. Both, the X-Net and the Router Network are bit-serial and they are synchronously clocked with the processing elements [Nic90].

When using the X-Net for interprocessor communication, the communication time is proportional to either the product or the sum of the operand length and the distance (Table 3.1) [Nic90, MPA93]. Also, the X-Net operations are faster when the *xnet** construct is on the left hand side than when it is on the right: hand side of a statement [MPA93]. The approximate times required for interprocessor communication using the basic '*xnet*' construct, the '*xnetp*' construct (pipelined communication), and the '*xnetc*' construct (copy left on intermediate processors) are given in Table 3.1.

Table 3.1: Timings for interprocessor communication operations using the X-Net, where 'dist' is the distance between the communicating processors, and 'opsize' is the size of the operand in bits.

Operation		**Approximate Timing in Clock Cycles**	
		LHS	**RHS**
xnet[]	dist == 1	opsize + 7	opsize + 17
	dist > 1	(opsize + 2)*dist + 6	(opsize + 4)*dist + 17
xnetp[]	dist == 1	opsize + 10	opsize + 21
	dist > 1	opsize*5/4 + dist + 11	opsize*5/4 + 2*dist + 24
xnetc[]	dist == 1	opsize + 10	-
	dist > 1	opsize*2 + dist + 9	-

The Router Network provides a "distance insensitive" method of interprocessor communication because all communication paths are of equal length. However, because the router ports are multiplexed among the sixteen processing elements in each cluster, an arbitrary communication takes at least sixteen router cycles to complete [Nic90]. A random communication pattern using the Router Network, with all processing elements participating takes an average of 5,000 clock cycles for 32-bit operands.

On the whole, the X-Net is preferred if the communication patterns are regular; that is, all active processing elements need to communicate with processing elements that are in the same relative direction and distance. The '*xnetp*' and the '*xnetc*' constructs are faster than the 'xnet' construct (for distances greater than two, approximately), but they require intermediate processing elements to be disabled.

3.2.4 The Processing Elements

Each processing element has a four-bit load/store unit and a four-bit ALU. This is transparent to the programmer, who can directly operate on the any of the supported data types. Each cluster of processing elements has one sixteen-way rnultiplexed port to the local memory, and memory operations are overlapped with processing element computation wherever possible [Nic90]. An access to local memory is about ten times slower than an access to a local register[Chr90]. A processing element can access another processing element's memory by sending a message to the other processing element and requesting that it send the desired item; this procedure is approximately one hundred times slower than a local register access [MMP90, Chr90].

3.2.5 Software Options

The programs in this thesis were coded using the MasPar *Parallel* Application *Language* (MPL). MPL is C-derived, and provides a direct high-level control of the hardware. Two more languages, the MasPar C (MPC) and MasPar Fortran (MPF), are also supported, but they do not offer the flexibility of MPL [Chr90].

3.2.6 Architectural Configuration

Details about the architecture of the MasPar MP-1 can be found in [Bla90]. In the next few lines, some of the frequently used variables are described. These variables are pre-defined and represent the hardware configuration of the machine. The variables nproc, *nxproc*, and nyproc represent the actual configuration of the hardware of the MP-1. nproc represents the total number of processing elements in the system while *nxproc* (nyproc) represents the total number of processing elements per row (column), in the two-dimensional array (Figure 3.1). iproc is a unique number between 0 and nproc-1, given to a processing element, while ixproc, and iyproc tell a processing element its row and column positions in the Processor Element Array.

3.3 Sparse Matrix-Vector Multiplication on a SIMD Machine

3.3.1 Introduction

In Chapter 2, we saw that the procedure of sparse matrix-vector multiplication could be divided into four parts - namely, the fetch phase (where the vector elements are fetched), the multiplication phase, the reduction phase (where the partial products are summed), and the result phase (where the result-vector elements are sent to the appropriate processors). In this section, the implementation of each of these phases is discussed. Under the assumption that communication between adjacent processors, or between processors in one "row" is faster than communication between arbitrary processors, a specific data structure (for the matrix and/or the vector) is defined for each phase so that regular communication can be used as much as possible. The matrices are assumed to be unstructured, and no attempt is made to optimize the performance based on specific sparsity structures.

Even though this discussion is specifically aimed at the MasPar MP-1 computer, it is directly applicable to any SIMD computer with a two-dimensional mesh inlerconnection network, and primitives for regular and irregular communication (with the regular communication being cheaper). The analysis, with minor modifications, can also be applied to distributed memory SIMD computers with other types of interconnection networks.

3.3.2 The Vector-Fetch Phase

In the general case, each processor that has a non-zero element of' the matrix (a_{ij}) will need to fetch the appropriate element of the vector (x_j) from the processor in which it is stored. Because data stored in different layers in memory is processed sequentially, the following discussion is specific to one layer of data (say k), without any loss in generality. Under our assumption that each processor can support only one fetch request at a time, the fetch time for layer 'k' is proportional to $C_{max(eff),k}$ (Table 2.1). $C_{max(eff),k}$ represents the amount of communication conflicts that occur as a result of the data distribution in the processor array (versus the conflicts that occur as a result of the limitations of the interconnection network). The time taken by the fetch phase can be optimized by minimizing both, $C_{max(eff),k}$, and the cost of each fetch operation (which includes the cost of communication conflicts 'because of the limitations of the interconnection network).

The value of $C_{max(eff),k}$ can be reduced in two ways. As seen in Equation 2.8, the value of $C_{max(eff)}$ is proportional to the number of elements of the vector that are stored on any one processor. So, an obvious way to reduce its value is to distribute the elements of the vector evenly among the processors in the processor array. This has an added advantage that the memory requirement for vectors is distributed across the processor array; this is important because massively parallel machines with a distributed memory tend to have a relatively small amount of memory per processor (a maximum of 64kB for the MasPar MP-1).

One possible way of distributing the elements of the vector among the processors is as follows. Each element of the vector, starting from the first one (x_0), is stored on consecutive processors, starting from the first one (processor #0). If the size of the vector is greater than the number of processors in the processor array (N_{proc}), then the next element of the vector ($x_{N_{proc}}$) is again stored on the first processor. In general, the sparsity pattern in the vector is not taken into account; that is, all the elements of the vector, including the zero elements, are stored. This is done to avoid any "look-up" overhead when fetching the vector elements; if all the elements of the vector are stored, calculating the location of a particular element of the vector (in terms of the processor) is trivial (element x_j is in processor $j\%N_{proc}$, where the % sign represents the modulus operator). For specific cases, where entire blocks of the vector are zero, it may be advantageous to take the sparsity into account.

The value of $C_{max(eff),k}$ also depends on the actual distribution of the non-zero elements of the matrix in the given layer (of data) in the memory (Equation 2.7), and is proportional to the largest number of non-zero elements from any one column of the matrix present in the layer (strictly speaking, it is proportional to the largest sum of the non-zero elements from all the columns that need vector elements stored in a given processor). Thus, the value of $C_{max(eff),k}$ can also be reduced by a mapping of the non-zero elements of the matrix that distributes the (non-zero) elements from a column evenly among the layers (of data). This optimization is not considered in this thesis.

An arbitrary distribution of the non-zero elements of the matrix among the processors in the processor array would require irregular communication patterns (in the general case) to fetch the vector elements. On most SIMD computers, regular communication (with respect to the architecture of the interconnection network) is faster than communications between arbitrary processors. Consequently, a data distribution that utilizes only regular communication would result in a faster algorithm.

For the fetch phase, all non-zero elements of the matrix belonging to one column require the same element of the vector. Consequently, if non-zero elements belonging to a given column were stored on adjacent processors, then it would be possible to send the vector-element to the relevant processors by using a "broadcast" mechanism. In particular cases, it may also be possible to store the noin-zero elements of a column on a single processor; for unstructured matrices, however, this would usually lead to an unacceptable imbalance in the load distribution among the processors.

The storage format where the non-zero elements of a column of the matrix are stored in adjacent processors is called the column-major format. Obviously, if this format is used, one would store all (or as many as possible) non-zero elements of a column in one layer (of data). This is because, when using a "broadcast" mechanism, the cost of communication per processor involved is usually smaller than the cost of setting up the communication (establishing the channel, masking, etc.). Depending on the number of non-zero elements in a given column, the most efficient procedure may be a "hybrid" method where several processors first obtain the relevant vector-element using (irregular) communication primitives, and then broadcast it to the other elements. Even if an actual "broadcast" primitive is not available (as on the MP-1), it is often possible to send data down "rows" of processors with very little cost (relative to an irregular communication primitive).

For applications where a column-major format is inefficient (if the rows are relatively dense, for example; explained in Section 3.3.4), it is possible: to optimize the actual implementation of the fetch phase to reduce the communication conflicts. One possibility is to make multiple copies of the vector elements (this is only feasible if the size of the vector is less than the number of processors in the processor array); depending on the size of the vector, one or more copies can be made of the entire vector, or a part of it.

To do this, the processors are grouped into ***sets,*** with the number of processors in each set being equal to the size of the vector (the last set may be "incomplete"). Then, processors within each set would perform fetch operations "locally" (within the set). The improvement obtained from this scheme depends on the actual data distribution among the sets. For example, on one extreme, if, for a given column of the matrix (say *j*), all the (non-zero) elements of that column happen to be stored in one set, then there is no reduction in the time required to fetch the elements of the vector for that column. On the other hand, if the elements of the column are distributed evenly among all the sets, then the fetch time will be reduced by a factor equal to the total number of sets because the fetch requests (for that element of the vector) will be distributed evenly among the processors that contain copies of x_j.

Thus, we can conclude that to optimize the fetch phase, the vector elements must be evenly distributed among the processors, and the non-zero elements of the matrix must be distributed in a column-major format. For matrices that have relatively sparse columns and (relatively) dense rows, the column-major format is inefficient (for the reduction phase, as discussed below), and so, depending on the application, it may not be feasible to use the column-major format. In this case, the time taken for the fetch phase can be reduced by making multiple copies of the vector, and by distributing the non-zero elements of each column evenly across the layers of data in the memory.

3.3.3 The Multiplication Phase

This phase involves a local multiplication operation with no interprocessor communication (Section 2.3.2.2), and all (enabled) processors multiply the non-zero elements of the matrix in the current layer by the vector-elements fetched (in the fetch phase) in parallel. Thus, the multiplication phase takes constant time: for each layer. the resulting products are called partial products, and partial products from each row must be added to form the result-vector elements.

Note that even though it is possible to combine the fetch phase and the multiplication phase by performing a "remote" multiplication operation, this is effectively the same as fetching a vector element and then performing a "local" multiplication. In fact, depending on which processing element the result is computed, combining the two phases may result in an extra communication step (if a_{ij} is sent over to the processing element containing x_j, and then the result is sent back to the original processing element).

3.3.4 The Reduction Phase

This phase involves the addition of partial products corresponding to each row to form the elements of the result vector. In general, the partial product:; will be spread across the processors in the processor array, and consequently, this phase is communication intensive. Using arguments similar to those for the vector-fetch phase, it can be said that the execution time for this phase can be minimized by a row-major distribution of the elements of the matrix. A row-major distribution allows the use of regular communication to add the partial products, as discussed later in this section. For a row-major distribution, (non-zero) elements from any single row of the matrix are stored on the same processor, or on adjacent processors.

If the partial products from a row are arbitrarily distributed among the processors, then there are two options: the partial products could be sorted according to the row that they belong to (i.e., convert the distribution to a row-major mapping), or partial products belonging to each row could be sent to a unique processor, where they would be added. The first option involves (partially) sorting the non-zero elements for every matrix-vector multiplication, in addition to the actual reduction of the partial products - which would not be feasible for large problems. The second option involves sending multiple data items to each processor, which would result in serialization (our assumption regarding one communication per processor, at a time).

Assuming that the non-zero elements of the matrix are in a row-major format, the reduction can proceed in several ways; each of these methods involves the use of only regular communication primitives (no communication conflicts). If only one layer of data is considered at a time (assuming that the elements of a row are stored in adjacent processors, rather than the same processor), the reduction can be done in a logarithmic (base 2) number of steps using recursive doubling. On the other hand, if all the (non-zero) elements of a row are stored on one processor, then the number of steps required to add the partial products is equal to the number of partial products (minus one).

Using recursive doubling, a given set of numbers can be added in a fewer number of steps compared to linear addition, but at each step in the algorithm, the number of processors utilized is reduced by a factor of two (all processors in the first step, 1/2 in the second step, 1/4 in the next step, etc.). In addition, the i^{th} step in the algorithm involves interprocessor communication over distances of 2^{i-1}. Even though regular communication primitives may be used, unless the cornrnunication is distance-insensitive, the cost of adding 'n' numbers can increase faster than $O(\log_2 n)$.

Based on this analysis, if the cost of a regular communication is comparable to that of adding two numbers, recursive doubling is faster for adding a small set of numbers together, while a combination of linear (local) addition and recursive doubling is faster for adding large sets of data. The actual threshold is dependent on the cost of communication as compared to the cost of a floating point addition.

3.3.5 The Result Phase

Once the partial products have been summed, the results need to be sent to the appropriate processors so as to conform to the selected distribution for vectors. Note that the vector distribution that minimizes the communication conflicts for the fetch phase (elements distributed evenly among the processors) also minimizes the communication conflicts for this phase (because each processor gets sent approximately the same number of elements, and the worst case serialization is equal to the maximum number of vector elements that are stored on any one processor).

In practice, this step is done once for each layer of data, assembling a part of the vector each time. Consequently, the number of communication conflicts that occur in this step is determined by the number of elements of the result-vector that are generated in each layer (of data). For example, if the non-zero elements of the matrix are stored in a row-major format, for large problems (that is, $N_{elts} \gg N_{proc}$), elements from a relatively small number of rows will be present in each layer, and so a smaller number of results are likely to be generated in each layer (comparecl to the column-major format, say). This, in turn reduces the probability of communication conflicts. On the other hand, if the non-zero elements of the matrix are mapped in a column-major format, partial results for a larger number of rows are likely to be generated in each layer, thus increasing the probability communication conflicts.

3.3.6 Summary

In this section, specific requirements of each phase of the matrix-vector multiplication in terms of the data distribution were reviewed. To minimize communication conflicts, the vector elements should be distributed evenly among the processors. This has the added benefit of distributing the memory :requirement for vectors among the processors. A column-major distribution of the (non-zero elements of) the matrix is best for the fetch phase, while a row-major distribution is best for the reduction phase.

For unstructured sparse matrices, a data structure that is designed to satisfy the requirements of both the phases (the fetch phase and the reduction phase) is unlikely to have an acceptable load balance among the processors; a data structure that reduces the load imbalance has been implemented in [OgA93] with the help of randomization techniques.

3.4 Data Structures for Sparse Matrices on SIMD Computers

3.4.1 Introduction

As described in Section 3.3, the fetch phase can be carried out by using regular communication primitives if the elements of the matrix are distributed in a column-major format, whereas the reduction phase can be carried out by using (only) regular communication if the elements of the matrix are distributed in a row major format. A data-structure that simultaneously allows a row-major and a column-major mapping is said to preserve the ***integrity*** of the matrix [OgA93]. For unstructured sparse matrices, it is difficult (NP-complete, [OgA93]) to design a data structure thai simultaneously preserves the integrity of the matrix, and also distributes the elements evenly among the processors. A data structure that preserves the ***integrity*** of the matrix has been implemented on the MP-1 [OgA93]; randomization techniques are used to reduce the load imbalance among the processors.

The data structures considered in this thesis enforce an even distribution of the load among the processors, and do not attempt to preserve the integrity of the matrix. In this section, three data structures are considered: the row-major format, the column-major format, and the diagonal format. The row-major format forms the basis of the data structure used for the algorithm developed in the next chapter, while the diagonal format is used to show how data structures can be designed to exploit specific sparsity structures; the column-major format is considered here for the sake of completeness.

3.4.2 The "Row-Major" Format

A row-major mapping of the (non-zero) elements of the matrix allows the use of regular communication primitives in the reduction phase; partial products can be summed using either local additions or recursive doubling across processors, or both. However, in the general case, this mapping will result in an arbitrary distribution (among the processors and layers of data) of the non-zero elements of a column. Consequently, using a row-major format results in an inefficient implementation of the fetch phase.

Depending on how many non-zero elements are present in the :rows of a given matrix, a row-major mapping will result in elements from a relatively few rows being present in each layer of data (especially if the elements of a row are distributed in the same layer along adjacent (rather than one or two) processors). Thus, on an average (for a large problem), there will be a small number of (non-zero) elements from each column of the matrix in a given layer of data. As a result, the implementation of the fetch phase will not involve a large amount of serialization.

3.4.3 The "Column-Major" Format

Mapping the matrix in a "column-major" format allows the use of regular communications (using the X-Net) to fetch the vector elements during the fetch phase. On the other hand, the partial products can no longer be efficiently summed in the reduction phase without modifying their distribution (by sorting them, for example). For large problems with relatively dense columns, however, if the elements from a column are distributed in adjacent processors (and in the same layer of data), there will be a relatively few partial products from any one row in a given layer of data, Consequently, there is not much work involved in the reduction phase; but the work required to arrange the elements of the result-vector increases (because results from a larger number of rows are present in a given data layer).

Under the assumption that it is easier to resolve "one-to-many" (multiple reads) conflicts than it is to resolve "many-to-one" (multiple writes), if the rows and the columns of a given matrix contain approximately the same number of non-zero elements, the row-major format will result in better performance than the column-major format.

For example, on the MP-1, the router automatically resolves communication conflicts; if multiple processors attempt to communicate with a single processor, the communication requests are serialized in some (unspecified) order. However, if multiple processors attempt to send data to one (memory) location in a single processor, the last value that is communicated overwrites the others. Because of this, if multiple values are to be sent to a single processor (the partial products, in this case), it is necessary to "reduce" the values to one single value before the communication operation.

3.4.4 The "Diagonal" Format

The algorithm based on the row-major format exploits the fact that the non-zero elements from each row of the matrix are mapped in a regular fashion in the processor array, rather than being randomly spread across it. On the other hand, mapping the non-zero elements in a column major format allows one to use the adjacency of the non-zero elements in each column to advantage. A third possibility is to store the non-zero elements of the matrix in terms of diagonals; that is, elements from one diagonal are stored in adjacent processors.

If, instead of storing only the non-zero elements, one were to store (entirely) any diagonal that had at least one non-zero element in it, one would be able to exploit the advantages of both - the row-major format, and the column-major fonnat. Obviously, this approach would only work for matrices with relatively "dense" diagonals; that is, any diagonal that is not empty has a relatively small percentage of zero elements. For miitrices where this not true, a significant amount of computation time and memory resources will be spent on "zero" elements.

One possible way of implementing this method is as follows. Consider a matrix 'A' of size 'N'. Assign a number to each diagonal based on its distance from the main diagonal; positive numbers refer to diagonals above the main diagonal, and negative numbers refer to the ones below. For example, the number assigned to the main diagonal is '0', and the numbers assigned to the diagonals just above, and just below the main diagonal are '1' and '-1', respectively. Assume that the rows and the columns of the matrix 'A', and the elements of the vector (x) are numbered from zero to N-1.

Now, consider diagonal number '+*i*'. The first element of the: diagonal is in column number 'i', and the last element is in column number 'N-1' (the last column). The length of the diagonal (that is, the number of elements in the diagonal) is equal to 'N - i'. Then, for the vector-fetch phase, a "chunk" of the vector, from element number 'i' to element number 'N-1' is needed. The entire "chunk" can be fetched using the X-Net because all the elements of the vector need to be: communicated across the same distance and in the same direction (the elements of the diagonal are stored in adjacent processing elements). Thus, the vector elements can be efficiently fetched by using regular communication patterns.

The reduction phase can also be executed efficiently, though the actual computation proceeds in a different order. Consider a relatively large matrix that maps into several 'layers' in the Processor Array. If the row-major mapping is used, then a relatively small number of result-vector elements are computed (completely) in each layer (each layer is processed sequentially). On the other hand, if the diagonal format is used, a relatively large number of result-vector elements are partially computed in each layer. Because of this, and because the mapping allows the use of regular communication patterns, the partial products can be reduced quickly without the use of Recursive Doubling. For example, if the size of the matrix is approximately half the number of processing elements in the processor array, approximately two diagonals are stored in each layer (assuming that the diagonals are close to the main diagonal). This means that the reduction phase involves only one addition per row of the matrix; for larger matrices, the addition phase may be "eliminated" completely.

A similar analysis can be done for negatively numbered diagonals, the only difference being that the first element of the diagonal is always in the first column of the matrix. On the whole, an algorithm using the diagonal format for the matrix (for appropriate matrices, of course) can be expected to work faster than either the algorithms using the row-major format or the column-major format.

A preliminary version of the algorithm was coded and implernented, and the above analysis was verified for relatively small test cases. However, this algorithm is not considered further in this thesis.

3.5 Conclusions

The MasPar MP-1 is a SIMD computer with between 1,024 and 16,384 processors, and a two-dimensional toroidal mesh interconnection network. A 16,384 processor system can achieve 650 MFLOPS (average 64-bit of add and multiply operations). Interprocessor communication is handled by two networks; the X-Net is an eight nearest-neighbor two-dimensional mesh network, while the router network is a distance-insensitive multistage network that can support arbitrary communication patterns.

The analysis in Chapter 2 was based on the assumptions that processors can support one incoming and one outgoing communication simultaneously, and that interprocessor communication involving any permutation of processors can be done in one parallel operation. On the MP-1, even though the router can handle arbitrary communication patterns, the communications are not conflict-free. That is, each communication operation may involve several sequential steps. Consequently, sparse matrix-vector multiplication on the MP-1 will take more time than estimated by the analysis in Chapter 2 if the router is used for interprocessor communication; each communication operation in the analysis in Chapter 2 will become a number (indeterminate, in general) of steps. The X-Net can be much faster than the router network (81 cycles for an adjacent processor vs. an average of 5000 cycles for a router operation with all processors enabled), but the communication is limited to rows, columns, or diagonals of the processor array - and all enabled processors have to communicate in the same direction, at a given time.

Thus, given the limitations of the X-Net, it is desirable to design a data structure that can utilize it (the X-Net) as much as possible. As stated earlier, there is an implicit assumption in the analysis that the data is distributed evenly across the processor array. In Section 3.3, a data structure that utilizes regular communication is described for each (individual) phase; the cost of using a different data structure is also discussed.

As seen in Chapter 2 (Table 2.1), most of the work involved in parallel matrix-vector multiplication is concentrated in the fetch phase and the reduction phase. If regular communication is to be used for both these phases, two different data structures will be required (for unstructured matrices). Then, unless the (non-zero) elements of the matrix are dynamically redistributed (in the processor array) for one of the two phases, it is necessary to use the router network for at least one of the fetch and the reduction phases (not true if the matrix has a diagonal sparsity structures, for example). If the matrices under consideration have the same number of non-zero elements in the rows and columns (or, obviously, a lower number of non-zero elements in the columns), it is more efficient to use the router network for the fetch phase (Section 3.4.3).

In the next chapter, we propose a new algorithm that is based on a modified row-major distribution of the elements of the matrix. Specifically, in addition to the non-zero elements, some (specified by a parameter) zero elements are stored to obtain a more "regular" data structure.

CHAPTER 4
THE BLOCK ROW ALGORITHM

4.1 Introduction

In the block row algorithm, elements from each row of a sparse matrix are grouped into blocks, with blocksize elements in each block; zero elernents are stored only if the number of non-zero elements in a particular row is not a multiple of the blocksize. Then, each block (rather than each element) is processed as a basic unit, which facilitates the design of a data structure whose "regularity" can be varied by changing a parameter (the blocksize). The reduction phase and the result phase have to be executed only once for every block because of the regular nature of the associated data structure, which results in a faster algorithm.

In Section 4.2, the working of the block row algorithm is explained with the help of' an example. The algorithm is formally described in Section 4.3, and its performance is analyzed in Section 4.4. Section 4.5 deals with some of the practical aspects of the algorithm. For matrices with wide variations in the number of non-zero elements between the rows, an "adaptive" version of the block row algorithm is described (Section 4.6); this algorithm allows the use of different blocksizes within one matrix by partitioning the matrix along rows. Finally, the chapter is concluded in Section 4.7.

4.2 The Block Row Algorithm: An Example

Consider the matrix and the vector shown in Figure 4.1, and a processor array with four processors. The matrix is a 6 × 6 "sparse" matrix with 17 non-zero elements, and the corresponding vector is assumed to be dense (note that the indexes

$$A = \begin{bmatrix} a_{00} & 0 & a_{02} & 0 & 0 & a_{05} \\ 0 & all & 0 & a_{13} & a_{14} & al5 \\ 0 & a21 & a22 & 0 & a_{24} & a25 \\ a30 & 0 & 0 & a33 & 0 & 0 \\ 0 & 0 & 0 & 0 & a_{44} & 0 \\ a50 & 0 & 0 & a53 & 0 & a55 \end{bmatrix} \qquad x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix}$$

Figure **4.1:** An example "sparse" matrix with N = 6 and **17** non-zero elements, and the corresponding vector.

Row 0	a_{00}	ao2	a_{05}	0	0	0
Row 1	a_{11}	a_{13}	a_{14}	al5	0	0
Row 2	a_{21}	a22	a_{24}	a25	0	0
Row 3	a_{30}	a33	0	0	0	0
Row 4	a_{44}	0	0	0	0	0
Row 5	a50	a53	a55	0	0	0

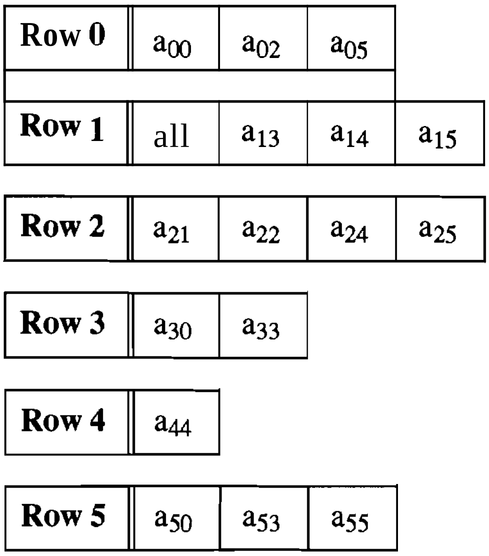
Figure 4.2: The intermediate-stage representation of the matrix shown in Figure 4.1.

of the elements of the matrix and the vector start from zero, rather than one). Assume the intermediate-stage representation of the matrix shown in Figure 4.2; this representation is obtained by "compressing" the non-zero elements in each row of the matrix - that is, by moving them to the left hand side of the zero elements.

From this intermediate-stage representation, data structures with different amounts of "regularity" can be obtained by changing the value of a parameter (the blocksize). The elements within each row of the intermediate-stage representation are divided into blocks such that the number of elements in each block is equal to the value of blocksize; zero elements can be added to the intermediate-stage representation if the number of elements in a row is not a multiple of the blocksize. Then, all blocks that contain at least one non-zero element are mapped on to the processor array, whereas blocks that have only zero elements are discarded. Let S_{blk} be the value of the blocksize. Then, N_{tot}/S_{blk} blocks are mapped on to the processor array, where N_{tot} is the total number of elements in the blocks.

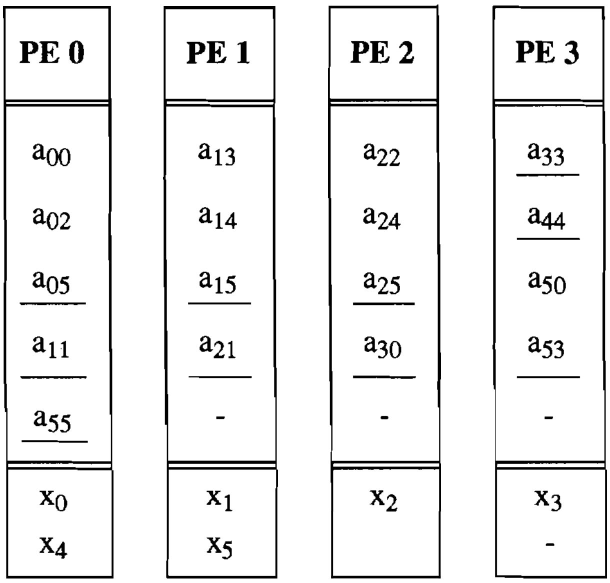
Consider a blocksize of one; each row is divided into blocks, with one element per block. The resulting data structure (Figure 4.3) does not have any zero elements because any number is an exact multiple of one; observe (Figure 4.3) that this data sbructure is simply a row-major mapping of the non-zero elements of the matrix. This data structure is then mapped into the processor array as shown in Figure 4.4. Each processor reads $\lfloor N_{tot}/N_{proc}/S_{blk} \rfloor$ (= 4, in this case) blocks from the data structure in a row-major format; thus, there are four ***complete layers*** of data in the memory of the processors. The remaining blocks are mapped into ***incomplete layers*** of data, with each processor (starting from the first one) being assigned one block. The vector is also distributed among the processors, as shown in the figure.

In general, to multiply a sparse matrix by a vector, the fetch phase, the multiplication phase, the reduction phase, and the result phase must be executed (in that sequence) for each layer of data (in the processor memory). In the block row algorithm, however, the reduction phase and the result phase are executed only once for each block; with a blocksize of one, though, each block has only one element, and all phases must be executed for each layer of data. In general, for a given processor,



Row 0	a_{00}	a_{02}	a_{05}	
Row 1	all	a_{13}	a_{14}	a_{15}
Row 2	a_{21}	a_{22}	a_{24}	a_{25}
Row 3	a_{30}	a_{33}		
Row 4	a_{44}			
Row 5	a_{50}	a_{53}	a_{55}	

Figure 4.3: The data structure for the matrix shown in Figure 4.1 witlh a blocksize of one; single vertical lines indicate block boundaries.



PE 0	**PE 1**	**PE 2**	**PE 3**
a_{00}	a_{13}	a_{22}	a_{33}
a_{02}	a_{14}	a_{24}	a_{44}
a_{05}	a_{15}	a_{25}	a_{50}
a_{11}	a_{21}	a_{30}	a_{53}
a_{55}	-	-	-
x_0	x_1	x_2	x_3
x_4	x_5		-

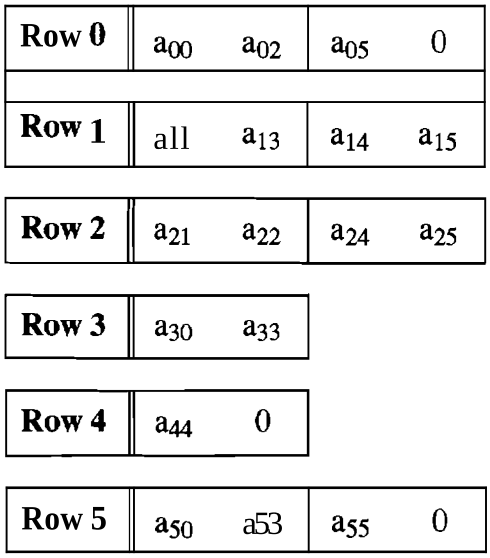
Figure 4.4: Distribution of the elements of the matrix shown in Figure 4.1 on a processor array with four processors for a blocksize of one..

the reduction phase and the result phase do not need to be executed for layer 'i' (of data) if the element in layer '*i+1*' belongs to the same row as the element in layer 'i'. In Figure 4.4, an underscore below a particular element indicates that the reduction phase and the result phase need to be executed for that layer (by the corresponding processor). Obviously, for the last layer of data, all phases must be executed.

For a blocksize of one, the reduction phase and the result phase need to be executed by at least one processor for every layer (Figure 4.4); in a SIMD computer, because of implicit synchronization, processors that do not need to execute the reduction/result phases must be disabled, and cannot do any useful work (simultaneously).

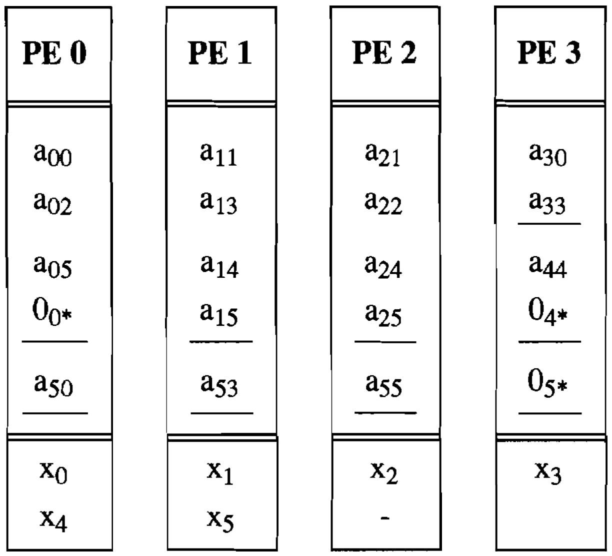
Now consider a blocksize value of two - the resulting data structure is shown in Figure 4.5. Notice that this data structure is more "regular" than the data structure for a blocksize of one; this regularity, however, is obtained at the cost of' having to store zero elements. For the reduction phase, the zero elements are assumed to belong to a specific block (specified by the row index), whereas for the fetch phase, the zero entries are ignored (indicated by a "*" for the column index in Figure 4.6). The entries in the data structure are mapped on to the processor array as shown in Figure 4.6. Again, each processor first reads $\lfloor N_{tot}/N_{proc}/S_{blk} \rfloor$ (= 2, in this case) blocks corresponding to the complete layers of data; the remaining (two) blocks corresponding to the incomplete layers of data are distributed evenly among the processors in the processor array (by "flattening" them; Figure 4.6). Note that this "flattening" of the blocks in the incomplete layers of data has an interesting side-effect; the reduction phase can become more expensive because the elements from a given block are distributed across a larger number of processors (because of the flattening).

In this case (because the blocksize is equal to two), the reduction phase and the result phase need to be executed once every two layers, as indicated by the underscores in Figure 4.6; also observe that not all processors execute the reduction/result phases at each block boundary. Thus, at the cost of storing three zero elements, the values of t_{add} and $t_{arrange}$ have been reduced by almost: a factor of two



Row 0	a_{00}	a_{02}	a_{05}	0
Row 1	all	a_{13}	a_{14}	a_{15}
Row 2	a_{21}	a_{22}	a_{24}	a_{25}
Row 3	a_{30}	a_{33}		
Row 4	a_{44}	0		
Row 5	a_{50}	a_{53}	a_{55}	0

Figure 4.5: The data structure for the matrix shown in Figure 4.1 witlh a blocksize of two; single vertical lines indicate block boundaries.



PE 0	**PE 1**	**PE 2**	**PE 3**
a_{00}	a_{11}	a_{21}	a_{30}
a_{02}	a_{13}	a_{22}	a_{33}
a_{05}	a_{14}	a_{24}	a_{44}
0_0*	a_{15}	a_{25}	0_4*
a_{50}	a_{53}	a_{55}	0_5*
x_0	x_1	x_2	x_3
x_4	x_5	-	

Figure 4.6: Distribution of the elements of the matrix shown in Figure 4.1 on a processor array with four processors for a blocksize of two.

(assuming that each execution of the reduction and the result phase takes the same amount of time).

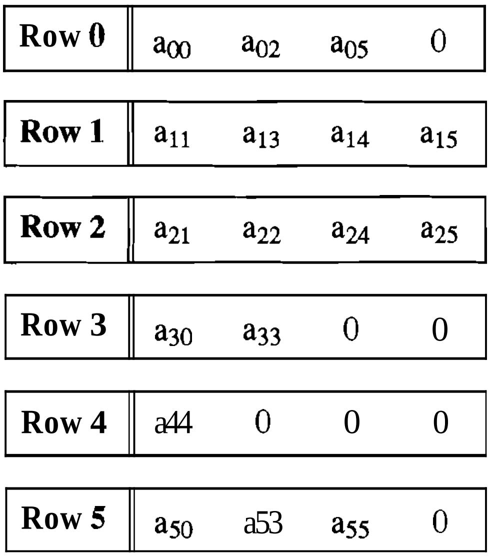
Finally, consider a blocksize of four: the corresponding data structure is shown in Figure 4.7, and the data distribution in the processors is shown in Figure 4.8. As before, the two blocks in the incomplete layers of data are flattened to maximize the utilization of the processors. With this blocksize, the reduction/result phases are executed only twice (compared to five times for a blocksize of one) for each matrix-vector multiplication. Observe that a further increase in the blocksize will add only zero elements; a meaningful increase in the blocksize is limited by the maximum number of non-zero elements in any one row of the given matrix.

4.3 Description of the Block Row Algorithm

As explained in Chapter 2, sparse matrix-vector multiplication on a SIMD computer can be divided into four phases; namely, the fetch phase, the multiplication phase, the reduction phase, and the result phase. Of these, t_{fetch} and t_{add} account for the largest fraction of the total time required for the matrix-vector multiplication (Chapter 2 and Chapter 3). For the matrices associated with our applications, the columns tend to be more (or about equally) sparse than the rows. Consequently, the algorithm is based a row-major mapping of data (Section 3.4.3).

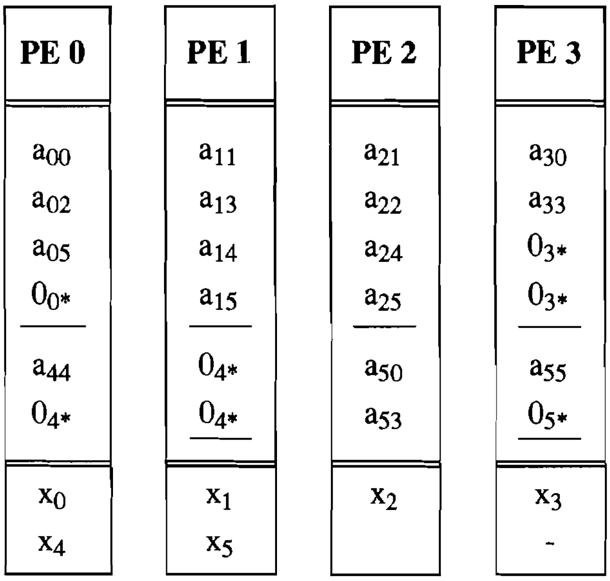
Consider a sparse matrix of size $N \times N$ with N_{elts} non-zero elements, and a processor array with N_{proc} processors. Assume that the non-zero elements in each row of' the matrix are "compressed" (as in the example in Section 4.2) to obtain the intermediate-stage representation of the matrix. Also assume that the elements from each row of the intermediate-stage representation are grouped into blocks, with S_{blk} elements in each block. Let N_{tot} be the total number of elements; in the blocks, including the zero elements.

Then, the blocks are distributed in the processor array as follows: each processor, starting with the first one, initially reads $\lfloor N_{tot}/N_{proc}/S_{blk} \rfloor$ blocks



Row 0	a_{00}	a_{02}	a_{05}	0
Row 1	a_{11}	a_{13}	a_{14}	a_{15}
Row 2	a_{21}	a_{22}	a_{24}	a_{25}
Row 3	a_{30}	a_{33}	0	0
Row 4	a44	0	0	0
Row 5	a_{50}	a53	a_{55}	0

Figure 4.7: The data structure for the matrix shown in Figure 4.1 with a blocksize of four; single vertical lines indicate block boundaries.



PE 0	**PE 1**	**PE 2**	**PE 3**
a_{00}	a_{11}	a_{21}	a_{30}
a_{02}	a_{13}	a_{22}	a_{33}
a_{05}	a_{14}	a_{24}	0_{3*}
0_{0*}	a_{15}	a_{25}	0_{3*}
a_{44}	0_{4*}	a_{50}	a_{55}
0_{4*}	0_{4*}	a_{53}	0_{5*}
x_0	x_1	x_2	x_3
x_4	x_5		-

Figure 4.8: Distribution of the elements of the matrix shown in Figure 4.1 on a processor array with four processors for a blocksize of four.

Matrix-vector Multiplication $y = Ax$

```
initialize
    in parallel in all processors
        partial−product = 0,
        for k = 0,..., $L_{vect} - 1$
            result_vect_elt[k] = 0.
        end for
for block = 0,..., $\frac{L_{tot}}{S_{blk}} - 1$
    for layer = 0,..., $S_{blk} - 1$
        in parallel in all processors
            if $a_{ij}$ in the current layer $\neq 0$
                temp = processor[j % $N_{proc}$].x[j / $N_{proc}$],
                partial−product += $a_{ij} \times$ temp.
            end if
    end for
    in parallel in all processors
        reduced−result = 0,
        if next block does not belong to the same row as this block
            reduced−result = reduce(partial_product),
            partial−product = 0.
        end if
    in parallel in the last processor in each reduction set
        processor[i % $N_{proc}$].result_vect_elt[i / $N_{proc}$] += reduced_result.
end for
```

Figure 4.9: Pseudocode for the block row algorithm; L_{tot} (L_{vect}) is the number of layers that the elements of the matrix (vector) map into, and N_{proc} is the number of processors in the processor array.

corresponding to the complete layers of data. Then, the remaining blocks (k, say) are distributed among the first 'k' processors, with one block per processor. The number of layers of data that the elements are mapped into is given by

$$L_{tot} = \left\lceil \frac{N_{tot}}{N_{proc} \times S_{blk}} \right\rceil \times S_{blk} \, . \tag{4.1}$$

Note that, for this analysis, the blocks in the incomplete layers of data are not flattened; this results in a clearer explanation, and "flattening" can be added to the algorithm with relatively minor modifications.

For the given setup, the pseudocode for sparse matrix-vector multiplication using the block row algorithm is given in Figure 4.9. With reference to the figure, observe that the fetch phase and the multiplication phase are executed for all layers of data, whereas the reduction phase and the result phase are executed only once for each block. In practice, as explained in Section 4.2, if a processor has more than one block from a given row of the matrix, the reduction/result phases only have to be executed after the last block (in that processor) belonging to that row has been processed.

The fetch phase is not executed for zero entries. Consequently, changing the value of the blocksize does not directly affect this phase; in practice, though, the fetch phase is dependent on the data distribution, which changes for different blocksizes. Also, a very high overhead (in terms of storing zero elements) can result in the under-utilization of the bandwidth of the interprocessor communication :network of the processor array - which, in turn, can cause an increase in the value of t_{fetch}.

4.4 Timing Analysis for the Block Row Algorithm

For this analysis, the setup in Section 4.3 is assumed. In general, for any value of the blocksize other than one, there will be some zero entries in the data structure corresponding to that blocksize. Thus, it can be expected that L_{tot} will be greater than L_{elts} for any blocksize other than one (in the general case).

Consider the fetch phase: as explained in Section 2.3.3.4, t_{fetch} depends on the number of non-zero elements in the columns of the matrix, and on the distribution of the non-zero elements from any one column among the layers of data. In the general case, different values of the blocksize will result in different data distributions (of the elements of the matrix) in the processor array. Consequently, even if no zero entries need to be stored, t_{fetch} will be different for different values of the blocksize. If this dependence of t_{fetch} on the data distribution is ignored, t_{fetch} remain:; approximately constant independent of the blocksize (because the fetch phase is not executed for zero entries), as long as there are a relatively few zero elements in each layer of data (otherwise, the communications would effectively be serialized due to the under-utilization of the communication network). Thus, up to a point, as the number of layers of data increase because of the storage of zero entries, the average fetch time per layer (proportionally) decreases because of the reduced amount of communication conflicts.

The time required for the multiplication phase is proportional to the number of layers that are processed (L_{tot}). As a result, the increase in $t_{multiply}$ is proportional to the number of zero entries that are stored. However, this does not have an adverse effect on the overall performance of the algorithm (except, possibly, for very small problems) because the multiplication phase is inexpensive as compared to the other phases (Table 2.1).

In the reduction phase, if the partial products belonging to a given row are on multiple processors, they are ''reduced'' - that is, they are summed. Thus, if the elements of a specific row are distributed across '*n*' processors, 'n-1' addition and 'n-1' communication operations are required for the reduction. Also, because the elements of a row are always in adjacent processors, the communication operations are conflict free.

Then, the reduction time for a given layer is proportional to the maximum number of processors across which the elements from a single row are distributed (in that layer). The total time taken for the reduction phase depends on the reduction time for each individual layer, and the number of layers for which the recluction phase is executed. As the blocksize increases, both these values decrease; for a higher blocksize, a given number of elements will be mapped across a fewer number of processors, and additionally, the reduction phase is executed less often. In practice, if a given processor has more than one block from a specific row, the reduction phase is executed only once - after the last block of that row is processed.

Finally, consider the result phase: $t_{arrange}$ is dependent on the data distribution among the layers (of data). If this effect is ignored, then, $t_{arrange}$ is (usually) lower for higher values of the blocksize. This can be explained as follows. Consider a row '*r*' that has six elements that are divided into two blocks which are on separate processors (Figure 4.10). Then, as indicated by the underscores, the result phase will be executed twice for this row (no reduction is necessary for this row). Generalizing this, it can be seen that, for one matrix-vector multiplication, there may be as many as 2 × N messages generated in the result phase (if all rows are similarly distributed in multiple processors). For higher values of the blocksize, more rows are likely to be stored on one processor, thus reducing the number of messages (down to N messages, if all rows are appropriately distributed). Consequently, $t_{arrange}$ is likely to be lower for higher blocksizes.

Given these results, the time taken to do one matrix-vector multiplication using the block row algorithm, in the general case ($N_{elts} > N_{proc}$ and $N_{vect} > N_{proc}$), is given by:

$$t_{block\_row} = \sum_{k=0}^{k < L_{tot}} \left[t_{fetch,\, k} + t_{multiply} + t_{add,\, k} + t_{arrange,\, k} \right] , \tag{4.2a}$$

where

$$t_{fetch,\, k} = c_1 \times C_{max(eff),\, k} , \tag{4.2b}$$

$$t_{multiply} = c_2 , \tag{4.2c}$$

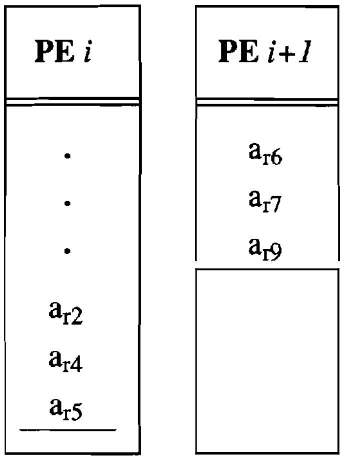


Figure 4.10: Execution of the reduction phase (indicated by an underscore) for row r.

$$t_{add,\,k} = \begin{cases} c_3 \times (W_{blk(max),\,k} - 1) & \text{if } k = i \times S_{blk},\ i=1, 2,\ldots \\ 0 & \text{otherwise} \end{cases} \text{, and} \tag{4.2d}$$

$$t_{arrange,\,k} = \begin{cases} c_4 \times L_{vect} & \text{if } k = i \times S_{blk},\ i=1, 2,\ldots \\ 0 & \text{otherwise} \end{cases} \tag{4.2e}$$

In Equation 4.2b, $C_{max(eff),\,k}$ represents the maximum number of processors that need to fetch any one of the vector elements stored in any given processor in layer 'k', as defined in Section 2.3.3.4. $W_{blk(max),\,k}$ (Equation 4.2d) represents the maximum number of processors across which the blocks from any one row are distributed, for the k^{th} layer of data.

4.5 Practical Considerations

4.5.1 Selection of the Optimal Blocksize

In general, a larger blocksize means lower values of t_{add} and $t_{arrange}$. *On* the other hand, a larger blocksize can result in more overhead in terms of storing zero elements, which, in turn, increases $t_{multiply}$, and can increase the value of t_{fetch} because of under-utilization of the communication bandwidth of the interprocessor communication network in the processor array. Thus, there is an optimal blocksize for which the sum of t_{fetch}, $t_{multiply}$, t_{add}, and $t_{arrange}$ is minimized.

In general, however, finding this optimal blocksize is not easy because the relative importance of a higher blocksize versus a lower overhead is not known, and because the execution times also depend on the specific data distribution achieved; obvious choices for the blocksize include a value that is equal to the number of non-zero elements in most rows (if such a value exists), or a common submultiple of the number of non-zero elements in each row. Given an upper limit for the acceptable overhead, an iterative procedure to find the best blocksize is described below.

Blocksize Selection

```
$S_{blk}$ = max(#non-zeros in a row).
while $S_{blk} > 1$
    overhead = 0.
    for each row i in the intermediate-stage representation
        blocks = $\left\lceil \frac{\text{#non–zeros in row i}}{S_{blk}} \right\rceil$,
        overhead += blocks × $S_{blk}$ – #non-zeros in row i.
    end for
    if overhead < MAX–OVERHEAD
        break while
    else
        $S_{blk}$ = next_max(#non-zeros in a row).
    end if
end while
if $S_{blk} < 1$
    $S_{blk} = 1$,
end if
```

Figure 4.11: Algorithm for the selection of the largest blocksize that results in an acceptable amount of overhead; a call to next_max() returns the next highest value of its argument (compared to the previous call to next_max() or max()), or a zero if there are no more enbies.

Observe that the blocksize is bounded on both sides - a blocksize of one is a "trivial" choice, requiring no preprocessing, while a blocksize value that is greater than the maximum number of non-zero elements in any row only adds zero elements to the data structure. Then, if a limit is imposed on the number of zero elements that are allowed, the best blocksize can be determined as shown in Figure 4.11. The function 'max()' returns the maximum of a set of numbers, while the function 'next_max()' returns the highest value (in the set) that is lower than the value returned by the most recent call to either max() or next–ma(). This procedure iterates through the values of the number of non-zero elements in the rows, starting from the highest value, and selects the first (largest) value of the blocksize that results :in an acceptable overhead. The procedure can also be modified to iterate through all integer values, from the maximum number of non-zero elements, down to a value of one.

4.5.2 "Flattening" Incomplete Layers of Data in the Processor Memory

With reference to Figure 4.12a, if for a given blocksize, the number of blocks is not a multiple of the number of processors in the processor array, then there will be some "incomplete layers" of data (as shown in the figure). If the number of blocks in the incomplete layers is such that a relatively small part of the processor array is utilized, the overall time for the matrix-vector multiplication will increase because of under-utilization of the resources of the machine (sequential processing). This problem can be avoided by using a different blocksize for the blocks in the incomplete layers of data (Figure 4.12b).

The following procedure is used to flatten the blocks in the incomplete layers. If, for the current blocksize, the number of active processors in the incomplete layers is less than (or equal to) half the total number of processors in the processor array, then the blocks in the incomplete layers are split into two blocks; otherwise:, no flattening is done. If, after splitting the blocks, the new blocksize is greater than one, and if the number of active processors is still less than (or equal to) half the total number of processors, the above process is repeated; otherwise, the current (new) value of the blocksize is assigned to the blocks in the incomplete layers of data. If, at some point, the blocksize is not an even value, a zero element is added at the end of each block,

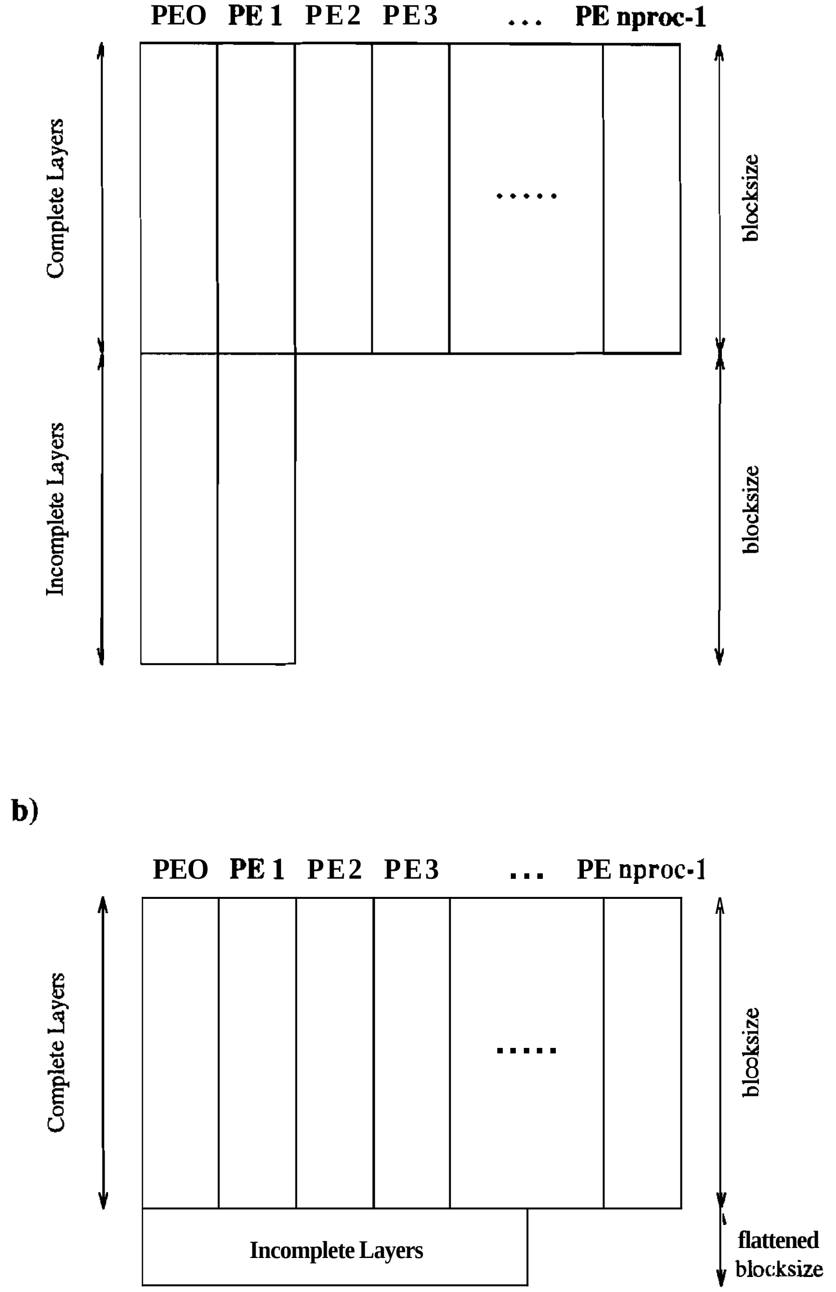


Figure 4.12: Distribution of the blocks in the memory before (a) and after (b) "flattening" the incomplete layers.

and the blocksize is increased by one before dividing it by two; this allows the processor utilization to be maximized regardless of the value of the blocksize, possibly at the cost of adding more zero entries.

The procedure described above is not optimal; as the blocks are flattened, the time required for the reduction phase increases because elements from each block (which belong to one row) are now distributed across a greater number of processors. On the other hand, the fetch time decreases (up to a point) because more fetch operations are being run in parallel, and the multiplication time decreases because it is proportional to the number of layers of data in which the elements are stored. The time required for the result phase is not affected directly because it is executed only once regardless of the new blocksize; it is, however, dependent on the data distribution, which is specific to each blocksize. Then, in the general case, there will be an optimal blocksize for which the sum of t_{fetch}, $t_{multiply}$, and t_{add} will be minimized - in effect, this is the exact problem of finding the optimal blocksize for a given matrix.

It should be noted that the analysis in this chapter does not include the effects of flattening the blocks in the incomplete layers of data; however, the modifications required are minor. The main side-effect of flattening the blocks is that t_{add} may increase as the blocksize is increased (instead of staying constant or decreasing); this increase is only significant for relatively small problems where the incomplete layers are a relatively large fraction of the total number of layers of data. This increase in t_{add} can be minimized by sorting the rows of the intermediate-stage representation in the decreasing order of the number of non-zero elements before generating the blocksize-specific data structure, and modifying the algorithm to ignore zero entries in the incomplete layers.

4.5.3 Coding the Block Row Algorithm

In the code for the algorithm, the implementation of each of the four phases is optimized as described in Chapter 3. The fetch phase is optimized for relatively small matrices ($N \leq N_{proc}$) by making additional copies of the vector, and interprocessor communication in the reduction phase is restricted to nearest-neighbor communication via the X-Net. Additionally, all frequently used variables are kept in registers (register operations are up to ten times faster than local memory operations on the MP-1). The code for the block row algorithm (written in MPL) can be found in the appendix; additional routines are required to load the matrix and the vector elements in the processor array.

4.5.4 Loading the Matrix in the Processor Array

To utilize the parallel read capability of the MP-1, the data for a given sparse matrix is stored in four files - the header file, the row-index file, the column-index file, and the data file. The header file contains the matrix size, the total number of elements to be stored (including any zero elements), and the blocksize. The row-index (column-index) file contains the row (column) indexes, while the data file contains the values of the matrix entries to be stored. The entries in each of the row-index, column-index and the data files are ordered so that the k^{th} entry in: the row-index (column-index) file represents the row (column) index of the k^{th} entry in the data file; the entries are stored in a row-major format. The vector is stored in a separate file (the vector is assumed to be dense). Using this storage format, all files except for the header file are read in parallel; on the MP-1, a matrix with approximately one million non-zero elements set up in this format can be read in about 1.5 seconds.

4.6 The "Adaptive" Block Row Algorithm

As explained earlier, it is difficult to find the optimal blocksize for an arbitrary sparse matrix. Additionally, for matrices with complex structures, the optimal blocksize may result in a large number of zero entries, for a relatively small improvement in the performance. On the other hand, it is relatively easy to find a good blocksize for simple matrices by using the iterative procedure described in Figure 4.11. Thus, it would be advantageous to be able to partition a complex matrix into simpler blocks.

Row 0	a_{00}	a_{01}	a_{02}	a_{03}	a_{04}	a_{05}	a_{06}	a_{07}	a_{08}
Row 1	a_{10}	all	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}	a_{17}	a_{18}
Row 2	a_{20}	a21	a22	a_{23}	a_{24}	a_{25}	a_{26}	a_{27}	a_{28}
Row 3	a30	a31	a32	a33	a34	a35	a36	a37	a_{38}
Row 4	a_{41}	a44	a46	a48	a49	0	0	0	0
Row 5	a_{50}	a_{53}	a_{55}	a_{56}	a_{58}	0	0	0	0
Row 6	a_{61}	a_{64}	a_{66}	a_{68}	a_{69}	0	0	0	0
Row 7	a_{70}	a_{72}	a_{74}	a_{76}	a_{77}	0	0	0	0
Row 8	a_{88}	0	0	0	0	0	0	0	0
Row 9	a_{99}	0	0	0	0	0	0	0	0
Row 10	$a_{10,10}$	0	0	0	0	0	0	0	0
Row 11	$a_{11,11}$	0	0	0	0	0	0	0	0

Figure 4.13: Intermediate-stage representation of an example matrix.

As an example, consider the intermediate-stage representation of a matrix (Figure 4.13), and a processor array with four processors. It can be seen that any choice of a blocksize, other than one, will result in some zero elements being stored. In general, this overhead is increases with increasing values of the blocksize.

Now, consider partitioning the intermediate-stage representation as indicated by the double horizontal lines in Figure 4.13. Then, the best blocksize value for each individual partition is obvious (the number of non-zero elements in one row of that partition). Consequently, if we allow a matrix to be partitioned on the basis of the number of non-zero elements in the rows, and assign a different blocksize to each partition, we should be able to achieve improved performance with a smaller amount of overhead.

This observation forms the basis for the "adaptive" block row algorithm. A sorted (in the decreasing order of the number of non-zero elements per row) intermediate-stage representation of an arbitrary matrix can be divided into partitions such that each partition contains rows with a similar number of non-zero elements. Then, each partition can be assigned an individual blocksize that is equal to the largest number of non-zero elements in a row in a given partition, if the partition contains enough elements ($\geq N_{proc} \times S_{blk}$). The largest feasible blocksize for a given partition is the smaller of: a) the largest number of non-zero elements in one row within the partition, and b) the total number of elements in the partition divided by the number of processors in the processor array.

The optimization problem associated with the partitioning of the matrix is similar to the problem involving the selection of the optimal blocksize. In general, a smaller partition will result in a lower overhead (in terms of storing zero elements), whereas a bigger partition allows a larger blocksize to be selected. Once the partitions have been made, however, the algorithm in Figure 4.9 can be directly applied to each individual partition.

4.7 Conclusions

The block row algorithm allows the "regularity" of a data structure that uses a row-major mapping to be varied by a changing a parameter (the "blocksize"). The (block row) algorithm assumes that the number of non-zero elements in each row is a multiple of the blocksize; (additional) zero entries are stored to satisfy this condition. The blocksize can be varied from one to N, where N is the size of the matrix; a blocksize of one results in a row-major distribution of the non-zero elements of the matrix (no overhead of storing zero elements), while a blocksize of N results in a row-major distribution corresponding to that of a dense matrix. However a meaningful increase in the blocksize is limited by the maximum number of non-zero elements in any row of the matrix.

Of the four phases in parallel sparse matrix-vector multiplication, the fetch phase and the multiplication phase are executed for each layer of data, while the reduction phase and the result phase are executed only once for each block. Consequently, as the blocksize increases, the values of t_{add} and $t_{arrange}$ decrease. On the other hand, for arbitrary unstructured sparse matrices, as the blocksize is increased, the number of zero entries that are stored also increases, leading to more overhead. The increase in the value of $t_{multiply}$ is proportional to the overhead; t_{fetch} is also affected if the overhead is more than a certain threshold. As a result, there is an optimal blocksize at which the sum of t_{fetch}, $t_{multiply}$, t_{add}, and $t_{arrange}$ is minimized, leading to the best performance.

An iterative method for determining a "good" blocksize is explained; the determination of the optimal blocksize is difficult because the execution times depend on the data distribution - which changes with a change in the value of the blocksize. Also, the improvement in t_{add} and $t_{arrange}$ obtained by a larger blocksize can be offset by a high overhead (in terms of storing zero elements); the impact of the overhead depends on the distribution of the zero elements among the layers of data. In practice, the reduction time is also dependent on the data distribution. If a given processor has multiple blocks from one row of the matrix, the reduction phase is executed only once for that row - after the last block (for that row) is processed. Flattening the blocks in the incomplete layers of data causes the reduction time to increase slightly - but the overall performance is improved because of better processor utilization..

For matrices that have a wide variation of non-zero elements between rows, it is advantageous to use different blocksizes for different parts of the matrix. This observation is the basis for the design of an adaptive block row algorithm. This algorithm allows a matrix to be partitioned along its rows; each partition can then be assigned an independent blocksize, and the basic algorithm described in this chapter is applied to each partition of the matrix; this algorithm is not evaluated further in this thesis.

In the next chapter, an experimental analysis of the block row is provided, using the matrices associated with our applications. It is shown that higher blocksizes result in improved performance, for a given amount of overhead. A comparative analysis of the algorithm is also given; the block row algorithm is faster for all the problems that were tested.

CHAPTER 5
EXPERIMENTAL EVALUATION

5.1 Introduction

In this chapter, we present an experimental analysis and a comparative evaluation of the block row algorithm. The chapter is divided into two parts. The first part describes the experimental results obtained for the block row algorithm for matrices associated with our applications (the finite element method, and the scattering matrix approach). In the second part, the performance of the block row algorithm is compared with the performance of the segmented-scan algorithm [Ham92], the snake-like method [RoZ93], and a randomized packing algorithm presented in [OgA93]. The block row algorithm is faster for all the matrices tested.

5.2 Experimental Analysis of the Block Row Algorithm

5.2.1 Introduction

The data in this section is obtained by evaluating the performance of the block row algorithm for four different matrices. The first three matrices represent systems discretized by using the finite element method, and the fourth matrix is a scattering matrix for a silicon device. A brief description of the finite element problem can be found in Chapter 7; a more detailed description is provided in [Lic93]. The scattering matrix approach is described in some detail in Chapter 6; a more thorough description can be found in [Ste91]. The execution times for the individual phases (t_{fetch}, $t_{multiply}$, t_{add}, and $t_{arrange}$, as defined in Chapter 2), as well as the (average) total time (t_{total}) taken to perform one matrix-vector multiplication are listed in the tables in this chapter. It should be pointed out that, in some cases the total time may be less than the sum of the times for the individual phases because of the (time) overhead involved in recording the execution times of the individual phases.

The matrices associated with the finite element approach have four to eight non-zero elements in each row. This makes them ideal for testing the performance of the block row algorithm because the blocksize only needs to be varied from one to eight for exhaustive testing. On the other hand, because any row has at most eight non-zero elements, the reduction phase can be expected to take a relatively small fraction of the time required for the matrix-vector multiplication (compared to a matrix with relatively dense rows). Consequently, optimizing this phase does not show up as a large change in the total time required for the matrix-vector multiplication. In the first matrix associated with the finite element problem (*M1*; 1633 unknowns), approximately 87% of the rows have exactly seven non-zero elements, and in the other two matrices (M2; 9385 unknowns, and M3; 36818 unknowns) more than 94% of the rows have seven non-zero elements. This distribution indicates that a blocksize of seven is likely to be the best choice. As an example, a matrix representing a (6 × 0.1)λ conducting scatterer in a circular domain with a radius of 5h and a node density of 20 nodes/λ is shown in Figure 5.1. Note that each '.' in the figure represents a square matrix (of size 184 ≡ the resolution of the map) with ***at least*** one non-zero element, and so the number of 'dots' do not directly indicate the number of non-zero elements in the matrix.

Matrices associated with the scattering matrix approach, unfortunately, are more complicated. For the matrix used here, the number of non-zero elements in a row ranges from 0 to 20,488. In addition, the variation in the number of non-zero elements in a row is relatively smooth; that is, there is no "common submultiple" that can be chosen as a "good" blocksize. Consequently, "sample" data for eight different blocksizes is presented in this section. The sparsity structure of a scattering matrix evaluated at an electric field of 300kV/cm is shown in Figure 5.2. As before, each '.' represents a non-empty submatrix (of size 486 this time).

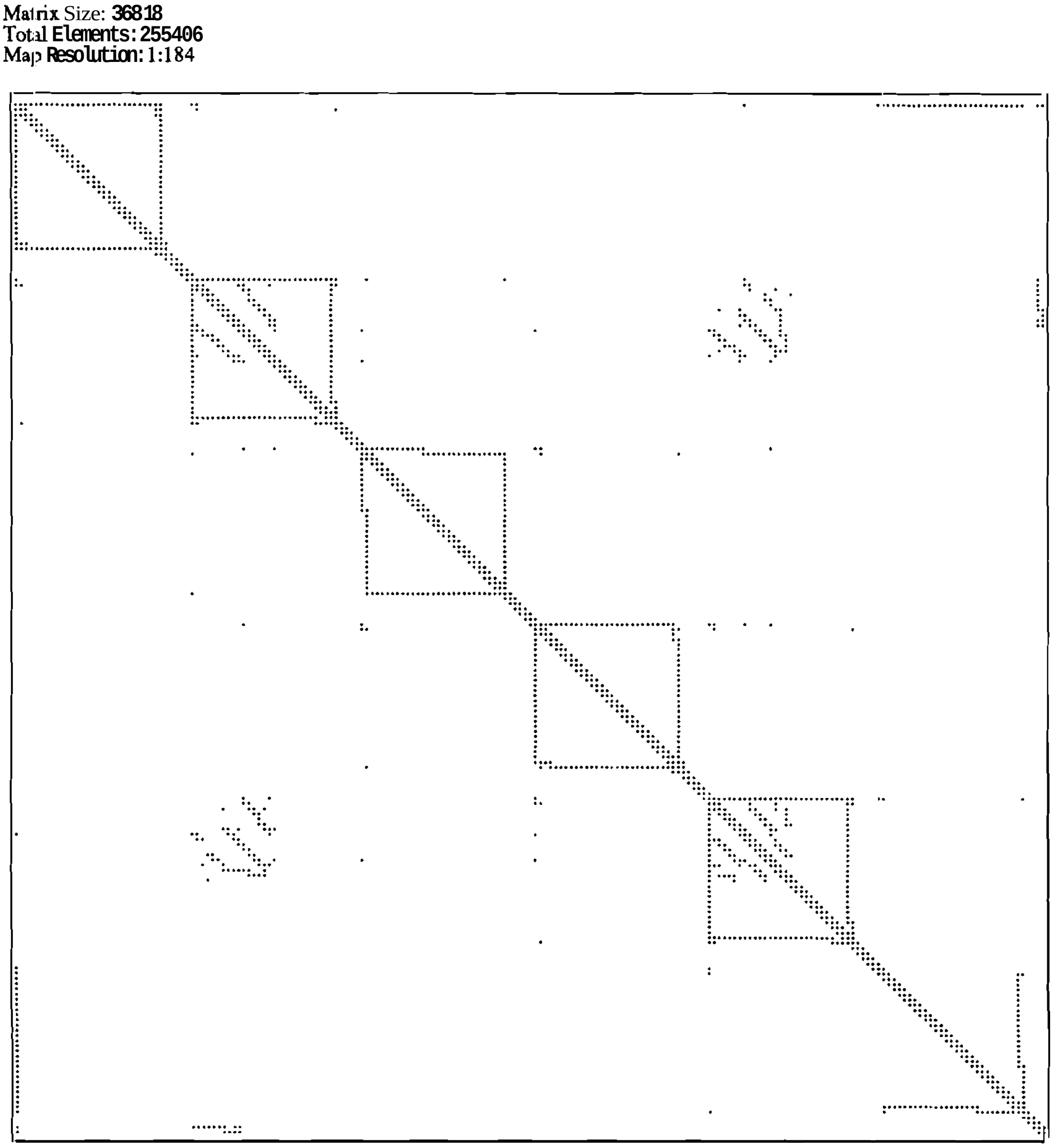


Figure 5.1: The sparsity structure of a matrix representing a (6 × 0.1)λ conducting scatterer in a circular domain of radius 5λ and node density 20 nodes/λ.

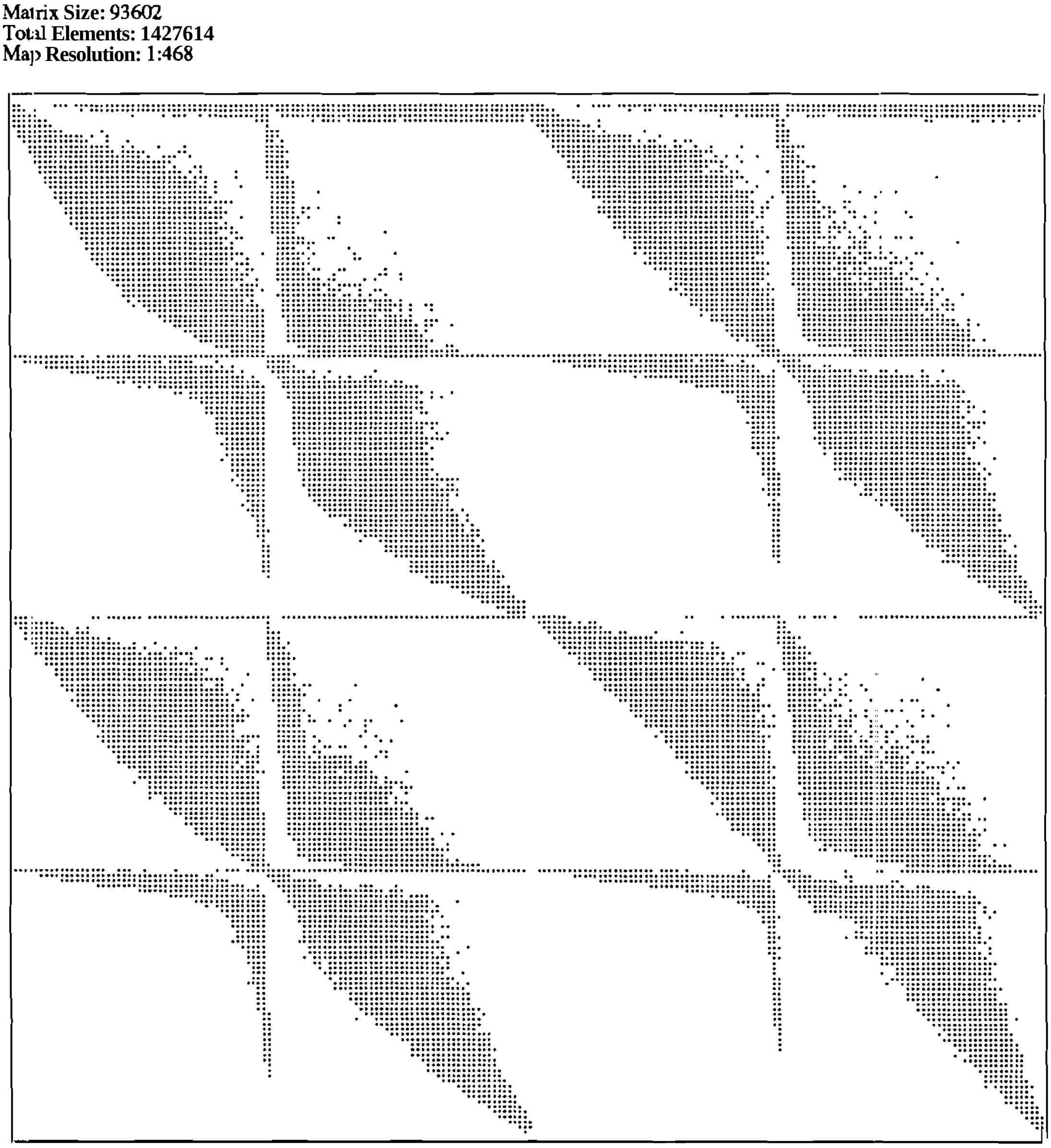


Figure 5.2: **The sparsity structure of a scattering matrix evaluated at an electric field of 300kV/cm.**

5.2.2 Part I - The Finite Element Approach

The matrices associated with this application have complex (as opposed to real) entries; however, for the experiments in this chapter, the matrices and vectors are assumed to be real (the imaginary parts are set to zero). If complex values are used, the execution time for each of the phases (and the total time) is almost exactly twice the time reported here.

The matrix (M1) associated with the first problem has 1,633 unknowns and 11,065 non-zero elements. The problem represents a homogeneous mesh (no scatterer in the domain) with a radius of 2h and a node density of 10 nodes/λ. The experimental results for this problem are shown in Table 5.1.

In this case, the best performance is obtained for a blocksize of one. This is explained by the "small" size of the problem - it involves only 11,065 non-zero elements, which means that even with a blocksize of one, all the processors in the processor array are not utilized. Nevertheless, it is instructive to see that the lowest values of t_{add} and $t_{arrange}$ are obtained for a blocksize of eight. As expected, t_{add} either decreases or stays constant as the blocksize is increased; t_{add} does not change when the blocksize is increased from four to five, and from five to six because rows with seven non-zero elements will be mapped into two blocks for each of these blocksizes (and more than 87% of the rows of the matrix have seven non-zero elements), thus keeping the work in the reduction phase constant. Because of the small size of the problem, the result phase is executed only once for any value of the blocksize. Consequently, in this case, $t_{arrange}$ is dependent only on the distribution of the non-zero elements among the layers (of data) - for different blocksizes, the source and destination processors in the result phase are different, thus resulting in a different number of router conflicts (or cycles).

The fetch time does not remain constant as the blocksize is varied; the variations in the distribution of the non-zero elements (of the matrix) in the processor array for different blocksizes result in a different number of router conflicts. Finally, as expected, $t_{multiply}$ increases with an increase in the number of layers into which the data is mapped; that is, the increase in $t_{multiply}$ is proportional to the overhead (in terms of storing zero elements).

The second matrix (M2) is of size 9,385 and has 64,837 unknowns. The associated problem is a homogeneous mesh with a radius of 2.51 and a node density of 20 nodes/λ. For this problem, the best performance is obtained with the largest blocksize - that is, a blocksize of eight (Table 5.2). This performance is obtained in spite of the fact that, for a blocksize of eight, the number of zero elements that are stored is approximately eight times the number of zero elements that are stored for a blocksize of seven. Observe that, in this case, t_{add} increases as the blocksize is increased (for some values of the blocksize). The reduction time is proportional to the largest number of processors across which elements of a row are spread. Consequently, when the last few layers of data are "flattened" in the memory to maximize processor utilization, t_{add} may increase slightly depending on the distribution of the matrix elements; this effect is seen clearly in this problem because of its relatively small size (no "flattening" is done for the first problem). $t_{arrange}$ depends on the distribution of the non-zero elements among the layers (of data), but, in general, it is lower for higher blocksizes (Section 4.4). As before, the fetch time depends on the distribution of the non-zero elements (of columns) in the memory, and the multiplication time is proportional to the overhead (in terms of the number of zero elements that are stored).

The third matrix (M3) for the finite element approach arises from a system consisting of 36,818 nodes, and 255,406 non-zero elements. It represents a $(6 \times 0.1)\lambda$ conducting scatterer in a mesh of radius 5 1 with a node density of 20 nodes/λ (actual application problem with no analytical solution). The timing information for this problem is shown in Table 5.3. For this problem, blocksizes of seven and eight result in approximately the same performance, which is better than the performance obtained with lower blocksizes. As expected, the lowest values of t_{add} and $t_{arrange}$ are obtained for the largest blocksize. However, the large increase in the overhead (approximately fifteen times) when going from a blocksize of seven to a blocksize of eight offsets this improvement (for the overall time). Note that t_{add} either decreases, or remains constant with an increase in the blocksize; "flattening" the last few layers (of data) has a smaller effect on t_{add} because of the relatively large size of this problem. Also, on the whole, $t_{arrange}$ decreases with increasing blocksizes. Finally, t_{fetch} depends on the distribution of the non-zero elements of the matrix, and $t_{multiply}$ depends on the number of layers into which the data is distributed.

5.2.3 Part II - The Scattering Matrix Approach

The results for a matrix-vector multiplication operation involving a scattering matrix (M4) evaluated at an electric field of 300kV/cm are presented in Table 5.4. As stated before, the scattering matrix (N = 93,602; 1,427,614 non-zero elements) is not as tractable as the matrices arising from the finite element approach It can be seen (Table 5.4) that better performance can be obtained at higher blocksizes, but it is not clear how to select a "good" blocksize. Note, however, that, in general, t_{add} and $t_{arrange}$ decrease as the blocksize increases. The data in Table 5.4 represents selected blocksizes that include the best and the worst performance obtained when varying the blocksize from one to twenty five.

5.3 A Comparative Analysis of the Block Row Algorithm

5.3.1 Introduction

In this section, the performance of the block row algorithm is compared with the performance of three other algorithms discussed in literature. A variation of the ‘‘snake-like'' method [RoZ93], the ‘‘segmented-scan'' method [Ham92], and a randomized packing algorithm [OgA93] were implemented on the MasPar MP-1, and compared with our algorithm. Each algorithm is described in brief before presenting the comparative analysis.

Table 5.1: Variations in the times (in seconds) for the individual phases of the sparse matrix-vector multiplication as the 'blocksize' is varied (M1; finite element approach).

N: 1633 N_{elts}: 11065

Blocksize (% Overhead)	t_{fetch}	$t_{multiply}$	t_{add}	$t_{arrange}$	t_{total}
1 (0%)	1.60E-03	1.06E-04	1.17E-03	3.06E-04	3.89E-03
2 (13%)	3.03E-03	2.12E-04	5.97E-04	3.82E-04	5.00E-03
3 (28%)	2.79E-03	3.20E-04	4.53E-04	3.28E-04	4.73E-03
4 (15%)	2.77E-03	4.24E-04	3.08E-04	4.38E-04	4.70E-03
5 (41%)	2.72E-03	5.31E-04	3.08E-04	3.60E-04	4.82E-03
6 (67%)	3.02E-03	6.37E-04	3.08E-04	3.51E-04	5.20E-03
7 (6%)	3.01E-03	7.43E-04	2.96E-04	3.83E-04	5.16E-03
8 (18%)	2.84E-03	8.52E-04	1.59E-04	2.30E-04	4.81E-03

Table 5.2: **Variations in the times (in seconds) for the individual phases of the sparse matrix-vector multiplication as the 'blocksize' is varied (M2; finite element approach).**

N: 9385 N_{elts}: 64837					
Blocksize (% Overhead)	t_{fetch}	$t_{multiply}$	t_{add}	$t_{arrange}$	t_{total}
1 (0%)	4.94E-03	4.25E-04	1.95E-03	1.08E-03	8.93E-03
2 (14%)	5.34E-03	5.32E-04	1.63E-03	1.12E-03	9.19E-03
3 (28%)	5.24E-03	6.37E-04	9.00E-04	7.93E-04	8.15E-03
4 (15%)	5.01E-03	5.32E-04	1.43E-03	7.42E-04	8.31E-03
5 (42%)	5.48E-03	6.37E-04	1.75E-03	7.36E-04	9.17E-03
6 (70%)	5.49E-03	7.45E-04	2.05E-03	7.14E-04	9.57E-02
7 (2%)	3.39E-03	7.44E-04	3.02E-04	4.39E-04	5.48E-03
8 (16%)	3.24E-03	8.51E-04	1.61E-04	2.29E-04	5.08E-03

Table 5.3: Variations in the times (in seconds) for the individual phases of the sparse matrix-vector multiplication as the 'blocksize' is varied (M3; finite element approach).

N: 36818 N_{elts}: 255406					
Blocksize (% Overhead)	t_{fetch}	$t_{multiply}$	t_{add}	$t_{arrange}$	t_{total}
1 (0%)	2.18E-02	1.70E-03	3.59E-03	8.14E-03	3.47E-02
2 (14%)	2.04E-02	1.91E-03	1.89E-03	6.43E-03	3.04E-02
3 (28%)	2.11E-02	2.23E-03	1.42E-03	6.01E-03	3.05E-02
4 (15%)	1.98E-02	1.91E-03	1.23E-03	4.41E-03	2.73E-02
5 (43%)	2.34E-02	2.44E-03	1.23E-03	4.96E-03	3.19E-02
6 (70%)	2.60E-02	2.86E-03	1.23E-03	4.95E-03	3.49E-02
7 (1%)	1.76E-02	1.81E-03	1.18E-03	2.87E-03	2.34E-02
8 (15%)	1.83E-02	1.91E-03	8.71E-04	2.31E-03	2.35E-02

Table 5.4: Variations in the times (in seconds) for the individual phases of the sparse matrix-vector multiplication as the 'blocksize' is varied (M4; scattering matrix approach).

N: 93602 N_{elts}: 1427614					
Blocksize (%overhead)	t_{fetch}	$t_{multiply}$	t_{add}	$t_{arrange}$	t_{total}
1 (0%)	1.23E-01	9.29E-03	5.07E-02	6.24E-02	2.41E-01
2 (3%)	1.23E-01	9.48E-03	4.80E-02	3.79E-02	2.16E-01
3 (6%)	1.25E-01	9.80E-03	4.25E-02	2.98E-02	2.05E-01
4 (9%)	1.26E-01	1.01E-02	3.92E-02	2.49E-02	1.98E-01
7 (21%)	1.30E-01	1.11E-02	3.09E-02	2.01E-02	1.90E-01
18 (64%)	1.47E-01	1.52E-02	2.59E-02	1.19E-02	1.99E-01
20 (71%)	1.52E-01	1.58E-02	2.35E-02	1.34E-02	2.03E-01
23 (83%)	1.56E-01	1.69E-02	2.23E-02	1.11E-02	2.05E-01

5.3.2 The "Snake-like" Method

This method requires the non-zero elements of a matrix to be stored in a column-major format; that is, the non-zero elements from one column of the matrix are stored in connected (adjacent) processors of the processor array. This distribution allows the fetch phase of the matrix-vector multiplication to be implemented using regular communication primitives (the X-Net on the MP-1), but results in an inefficient implementation of the reduction phase. This method performs well for matrices that have relatively sparse rows and (relatively) dense columns.

5.3.3 The "Segmented Scan" Method

A detailed discussion of the implementation of this method can be found in [Ham92]. A row-major storage format, along with a "scan" primitive is used to optimize the reduction phase. Each row is considered to be a "segment", and the reduction of partial products in all rows can be implemented in parallel. This method can be used for matrices that have relatively sparse columns.

5.3.4 The "Randomized Packing’' Algorithm

The randomized packing algorithm implemented here is the second (better) algorithm presented in [OgA93]. The data structure for this algorithm preserves the integrity of the matrix by requiring both, the non-zero elements from a row and from a column, to be stored in adjacent processors. As a result, both, the fetch phase and the reduction phase can be simultaneously optimized. However, this data structure no longer guarantees a good load balance among the processors; depending on the sparsity structure of the matrix, most of the non-zero elements may be distributed among a relatively few processors in the processor array. The algorithm presented in [OgA93] reduces this problem by randomly permuting the rows and columns of the matrix before mapping it on the processor array - as a result of the randomization, the non-zero elements are more uniformly distributed in the permuted matrix, and consequently a better load distribution is obtained.

As described in [OgA93], the randomized packing algorithm involves five "phases" - the vector distribution phase, the scatter phase, the multiplication phase, the gather phase, and the row-sum phase. For the purposes of this analysis, the vector distribution phase and the scatter phase are grouped together to form the fetch phase, and the gather phase and the row-sum phase are grouped together to form the "reduction + result" phase. The algorithm, as presented in the paper, has different storage formats for the input vector and the output (result) vector:; because most programs will require that the result vector be in the same format as the input vector, we have added a few lines of code to do that, and included the time in the "reduction + result'' phase.

The randomization changes the matrix 'A' to 'PAQ^T', and this effect must be reversed at the end of the computations. The time required to permute and subsequently unpermute the matrix is ignored in this analysis. It should be mentioned, however, that it took several minutes of CPU time to permute the rows and columns for the largest problem described above (as compared to tens of seconds for the preprocessing stage of the block row algorithm). Our implementation of the randomized packing algorithm achieved approximately 110 MFLOPS for the largest dense matrix-vector multiplication problem that could be solved on a MP-1 with 256 MBytes of memory - versus the approximately 116 MFLOPS achieved by the authors. Consequently, the times quoted for the randomized packing algorithm in this thesis are accurate to within a few percent (of the authors' implementation), for a given randomization.

5.3.5 Experimental Results

The algorithms described above are compared with the block row algorithm in this section. For each algorithm, the "best" performance is used for the evaluation; for the snake-like method and the segmented-scan method, the best time of several (ten) runs is used, for the randomized packing algorithm, the best randomization (of ten, using two different random number generators) is used, and the best blocksize is used for the block row algorithm. To give an idea of the structure of the matrices after the randomization, randomized versions of the two matrices shown in Figure 5.1 and

Figure 5.2 are shown in Figure 5.3 and Figure 5.4, respectively (as explained earlier, tht: 'dots' represent non-empty submatrices). The ***normalized time*** in the tables in this section is the total execution time, normalized with respect to the time taken by the block row algorithm.

The results for the matrix with 1,633 unknowns and 11,065 non-zero elements (M1) are shown in Table 5.5. Without randomization, the data structure for the randomized packing algorithm would have resulted in a maximum processor load of 63, and a minimum processor load of 0; the best randomization (of ten) improved the load distribution to a maximum load of 5, and a minimum load of 0 (as opposed to an ideal load of one).

Because of the small size of the problem, the performance of all the algorithms is approximately the same, though the block row algorithm is faster by about 10%. The randomized packing algorithm is slower than all the other algorithms because it is designed for relatively dense matrices [OgA93]; ours are less than 1% full.

The results in Table 5.6 represent the problem with 9,385 nodes, and 64,837 non-zero elements (M2). For the randomized packing algorithm, the best randomization resulted in a maximum processor load of 13, and a minimum processor load of 0 (without randomization: 450 and 0, respectively; ideal load: 4). Again, it should be emphasized that the main reason for the poor performance of the randomized packing algorithm is because the matrices associated with our applications are extremely sparse (the amount of sequential computation in the randomized packing algorithm is proportional to the size of the matrix). For this problem, the block row algorithm is more than twice as fast as the segmented-scan algorithm (which is the next fastest algorithm).

For the third problem, the matrix (M3) has 36,818 unknowns and 64,837 non-zero elements. Randomization achieved a maximum (minimum) load of 31 (4), and

Matrix Size: 36818
Total Elements: 255406
PE Load: Min 2, Max 31
Map Resolution: 1:184

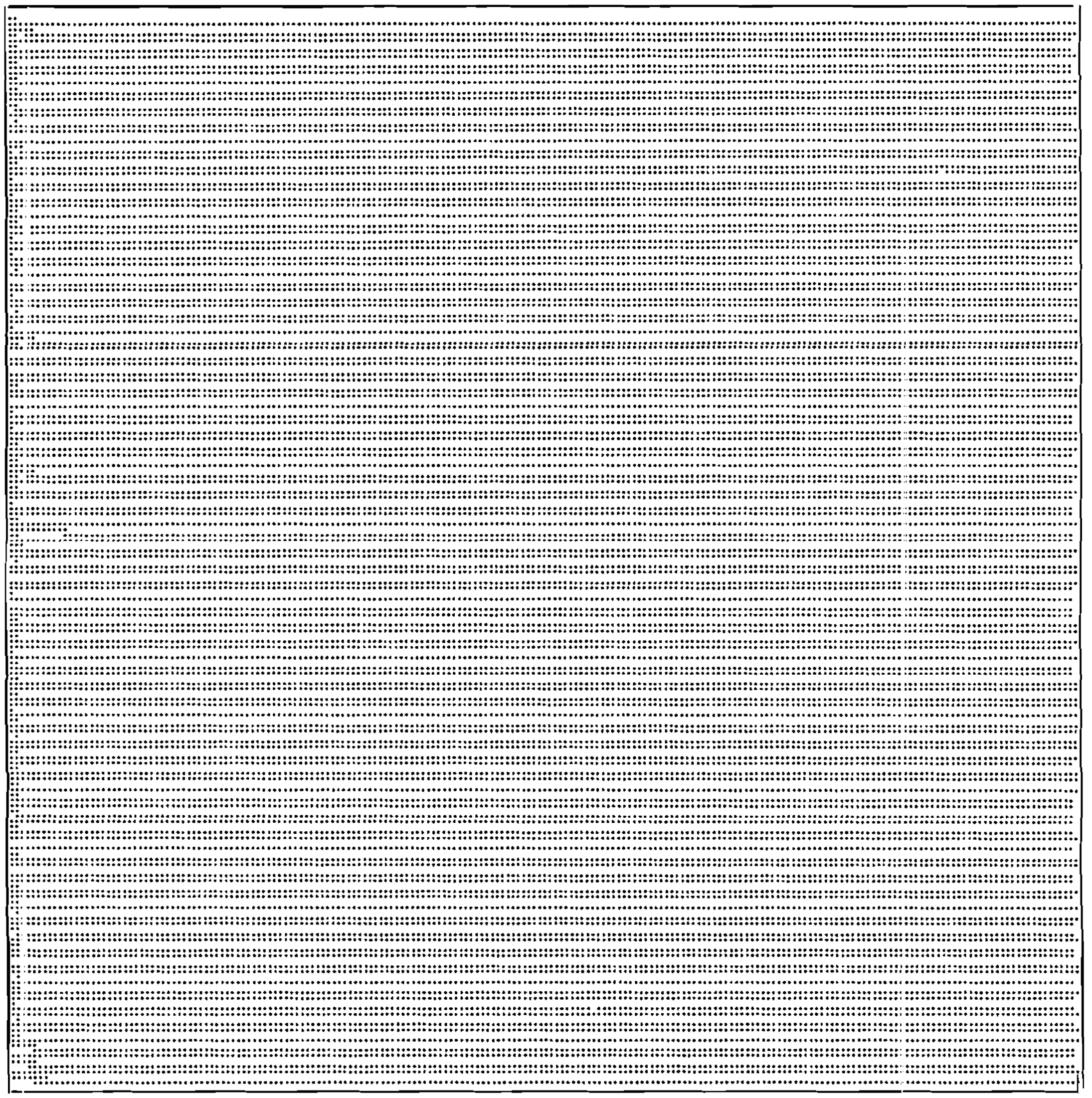


Figure 5.3: The sparsity structure of the randomized version of the matrix representing a $(6 \times 0.1)\lambda$ conducting scatterer in a circular domain of radius 5λ and node density 20 nodes/λ.

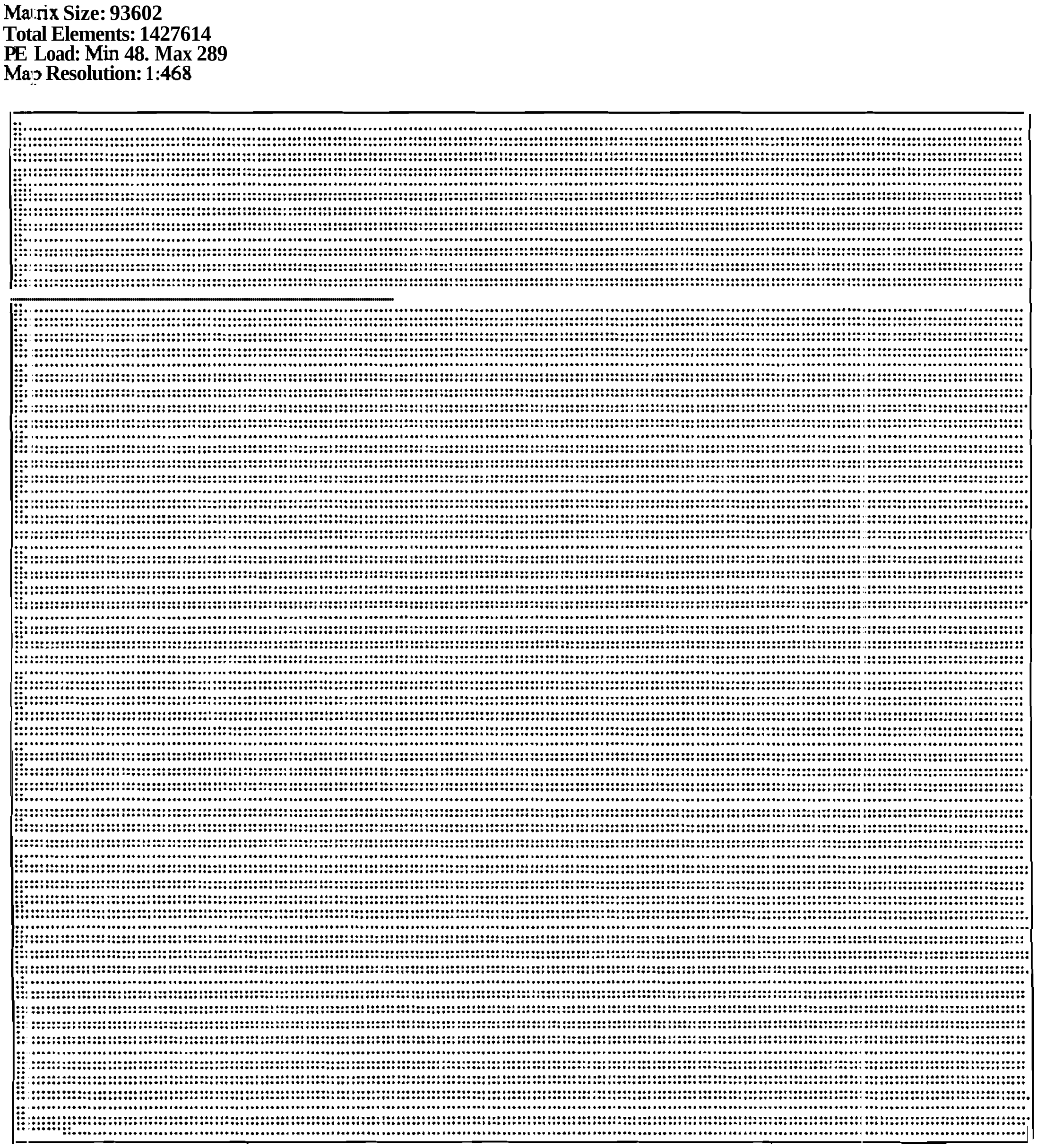


Figure 5.4: The sparsity structure of the randomized version of the scattering matrix evaluated at an electric field of 300kV/cm.

the maximum (minimum) load without randomization was 1880 (0); the ideal load for this problem was 16. The results are similar to those obtained in the earlier problems, with the block row algorithm being about twice as fast as the ‘‘segmented-scan’’ algorithm (Table 5.7).

The matrix for the fourth problem (M4) is a scattering matrix which is evaluated at 300kV/cm; the size of the matrix is 93,602 × 93,602, and it has 1,427,614 non-zero elements. The (best) random permutation of the rows and columns of the matrix resulted in a maximum processor load of 275, and a minimum processor load of 44. The ideal load was 88, and without randomization, the maximum processor load would have been 4654 (minimum 0). In this case, the block row algorithm is about 1.7 times faster than any of the other algorithms.

5.4 Conclusions

In general, it can be expected that t_{add} will decrease (or stay constant) as the value of the blocksize is increased; the experimental results in the first part of this chapter (Tables 5.1 - 5.4) agree with this - with one exception. For the second problem (M2; Table 5.2), t_{add} increases when the blocksize is increased from three to four, from four to five, and from five to six; as explained earlier, this increase in the time is a result of "flattening" the last few layers of data in the memory (experimentally verified), and can be eliminated by sorting the rows in the decreasing order of the number of non-zero elements before the "row compression" stage:. This situation does not arise in larger problems (M3 and M4) because, for larger problems, the layers of data in the memory that are "flattened" represent a relatively small fraction of the total number of layers.

$t_{arrange}$ depends on the distribution of the non-zero elements across the layers of data and on the blocksize. The distribution of the elements determines the communication patterns involved in the result phase, which in turn. determines the number of router cycles necessary to complete the communication. On the other hand, the value of the blocksize determines the total number of (partial) results that are communicated in the result phase (Section 4.4). As expected, $t_{arrange}$ is lower for

Table 5.5: A comparison of the ***best*** times (in seconds) for the snake-like algorithm, the segmented-scan algorithm, the randomized algorithm, and the block row algorithm for test problem 1. Normalized times are with respect to the block row algorithm.

N: 1633 N_{elts}: 11065						
Algorithm	t_{fetch}	$t_{multiply}$	t_{add}	$t_{arrange}$	t_{total}	**Normalized Time**
Snake-like	9.90E-04	6.80E-05	5.55E-03		6.80E-03	1.75
Seg-Scan	1.60E-03	6.90E-05	1.52E-03	3.10E-04	4.26E-03	1.10
Randomized	1.31E-03	4.50E-04	8.87E-03		1.07E-02	2.76
Block-Row	1.60E-03	1.06E-04	1.17E-03	3.06E-04	3.88E-03	1.00

Table 5.6: **A comparison of the best times (in seconds) for the snake-like algorithm, the segmented-scan algorithm, the randomized algorithm, and the block row algorithm for test problem 2. Normalized times are with respect to the block row algorithm.**

N: 9385 N_{elts}: 64837						
Algorithm	t_{fetch}	$t_{multiply}$	t_{add}	$t_{arrange}$	t_{total}	**Normalized Time**
Snake-like	4.20E-03	2.73E-04	1.18E-02		2.27E-02	4.47
Seg-Scan	6.82E-03	2.74E-04	1.96E-03	1.22E-03	1.09E-02	2.15
Randomized	6.56E-03	1.15E-03	4.73E-02		5.48E-02	10.79
Block-Row	3.24E-03	8.51E-04	1.61E-04	2.29E-04	5.08E-03	1.00

Table 5.7: A comparison of the ***best*** times (in seconds) for the snake-like algorithm, the segmented-scan algorithm, the randomized algorithm, and the block row algorithm for test problem **3.** Normalized times are with respect to the block row algorithm.

N: 36818 N_{elts}: 255406						
Algorithm	t_{fetch}	$t_{multiply}$	t_{add}	$t_{arrange}$	t_{total}	**Normalized Time**
Snake-like	1.91E-02	1.09E-03	8.35E-02		1.04E-01	4.44
Seg-Scan	3.29E-02	1.09E-03	5.59E-03	7.91E-03	4.74E-02	2.03
Randomized	2.44E-02	3.24E-03	1.82E-01		2.06E-01	8.80
Block-Row	1.76E-02	1.81E-03	1.18E-03	2.87E-03	2.34E-02	1.00

Table 5.8: A comparison of the *best* times (in seconds) for the snake-like algorithm, the segmented-scan algorithm, the randomized algorithm, and the block row algorithm for the scattering matrix problem. Normalized times are with respect to the block row algorithm.

N: 93602 N_{elts}: 1427614						
Algorithm	t_{fetch}	$t_{multiply}$	t_{add}	$t_{arrange}$	t_{total}	**Normalized Time**
Snake-like	4.01E-01	6.09E-03	9.08E-01		1.31E-00	6.89
Seg-Scan	1.66E-01	6.01E-03	1.03E-01	4.93E-02	3.23E-01	1.70
Randomized	7.40E-02	2.40E-02	4.83E-01		5.75E-01	3.03
Block-Row	1.30E-01	1.11E-02	3.09E-02	2.01E-02	1.90E-01	1.00

higher values of the blocksize (it does not strictly decrease with each increment in the blocksize, though).

The fetch time depends on the distribution of the non-zero elements of the matrix among the layers of data (because this affects the communication patterns), and to a smaller extent on the total number of elements stored (including the zero elements), whereas the multiplication time depends only on the total number of elements stored. Then, ignoring the variation of t_{fetch} with changes in the distribution of data, it can be concluded that, for a given overhead (of storing zero elements), the performance will improve as the blocksize is increased. However, it is not clear what the relative weights of the blocksize and the overhead should be; that is, it is not clear how to find the point at which the performance drops even with an increase in the blocksize, because of the increase in the overhead. For example, in the second problem (M2), a blocksize of eight results in better performance even though the number of zero elements stored increases by a factor of eight (compared to a blocltsize of seven), while in the third problem (M3), the performance of the algorithm drops slightly for a blocksize of eight as compared to the performance for a blocksize of seven, with the overhead being fifteen times higher.

The comparative analysis in this chapter shows that the block row algorithm is faster than the snake-like method, the segmented-scan algorithm, and the randomized packing algorithm. Though the improvement in the performance is limited to a factor of' two (over the segmented-scan algorithm, approximately), this result needs to be qualified. For the matrices associated with the finite element approach, there are at most eight non-zero elements in a row; this limits the speedup that can be obtained by optimizing the reduction phase because the time taken by that phase is already small. On the other hand, though the scattering matrix considered has a maximum of 20,488 non-zero elements in a row, the variation of the non-zero elements across the rows is such that it is not possible to select a (one) good blocksize. An "adaptive" version of the block row algorithm that allows multiple blocksizes for a single matrix would be more appropriate for this application.

CHAPTER 6
THE SCATTERING MATRIX APPROACH

6.1 The Scattering Matrix Approach to Device Analysis

The ***Scattering Matrix Approach*** [Das90, Ste91] is a method used to simulate carrier transport in modern semiconductor devices. In this approach, the device is viewed as a set of interconnected thin slabs, where each slab is thin enough so that the electric field and the doping density can be considered constant within the slab. Carrier transport across each slab is described by a matrix equation which relates the incident fluxes to the emerging fluxes through transmission and reflection coefficients.

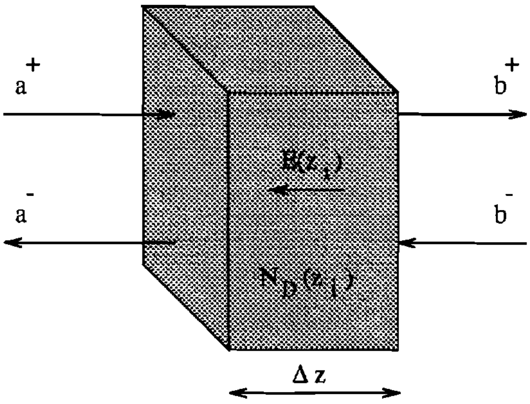


Figure 6.1: Electron transport across a thin slab of semiconductor in terms of incident and emerging fluxes.

The carrier fluxes emerging from the slab are related to those incident on the slab by

$$\begin{bmatrix} b^+ \\ a^- \end{bmatrix} = \begin{bmatrix} t & r' \\ r & t' \end{bmatrix} \begin{bmatrix} a^+ \\ b^- \end{bmatrix},$$

where t, t', and r, r' give the transmission and reflection probabilities for fluxes incident from the left (a') and right (b^-), respectively (Figure 6.1). The matrix,

$$S = \begin{bmatrix} t & r' \\ r & t' \end{bmatrix}$$

is called the scattering matrix, and its coefficients are dependent on the electric field, scattering mechanisms, and the recombination-generation processes that occur within the slab. As stated above, a semiconductor device is analyzed by dividing it into a number of small slabs. The scattering matrix S_i describes the transport across a thin slab centered at z_i with doping density N_{Di} and electric field E_i (Figure 6.1). To simulate the carrier transport in the entire device, the scattering matrices representing the slabs in the device are cascaded as shown in Figure 6.2.

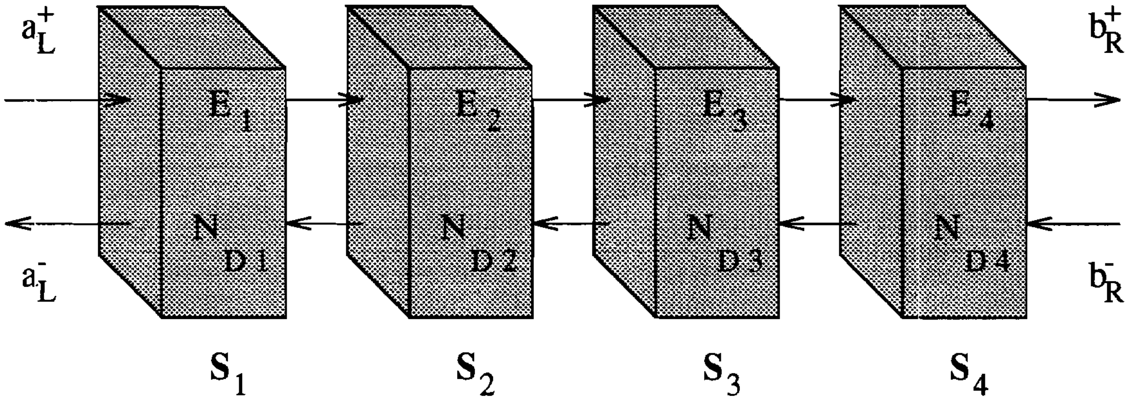


Figure 6.2: Cascading of individual scattering matrices (S_i).

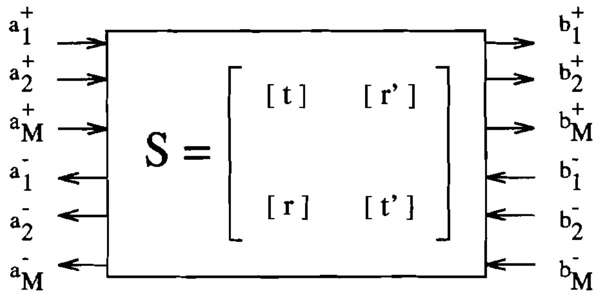
The carrier transport in a device is simulated by using an iterative technique. With reference to Figure 6.1, the emerging fluxes for a slab are computed by iteratively solving the equation $f_{i+1} = Sf_i$, where the initial value off is [a' $b^-]^t$, and S is the scattering matrix for the slab. First, the emerging fluxes are evaluated for each slab, starting from the left-most slab. An emerging flux for one slab acts as an incident flux for an adjacent slab. Also, when periodic boundary conditions are used, the emerging fluxes at the ends of the device are "wrapped around" to the other end ($a_L^+ = b_R^+$, and $b_R^- = a_L^-$ in Figure 6.2). Once the emerging fluxes for the last (right-most) slab have been evaluated, the process is repeated backwards - that is, from the right hand side of the device towards the left. The above steps are repeated until all the emerging fluxes converge.

6.2 The Multi-Flux Scattering Matrix Approach

To be applicable to modern devices with submicron dimensions, the flux method described above needs to be extended. This is done by resolving the incident and emerging fluxes in terms of energy and the angle to the normal axis of the slab.

The incident and emerging fluxes are discretized into a finite number (say *M*) of subfluxes, and each subflux is called a ***mode.*** As a result, the transmission and reflection coefficients t, t', r, and r' become M by M submatrices relating all the individual incoming and outgoing subfluxes, and the scattering matrix is 2M by 2M. The fluxes are treated as M by 1 vectors. The individual elements of the scattering matrix are real numbers between zero and one, and represent the transmission and reflection probabilities for the incident fluxes.

Several different discretizations of the energy space are possible; one of those is discussed here. Consider a carrier with crystal momentum $\hbar\vec{k}i$ incident on a thin isolated slab. The carrier is assigned a mode based on the Cartesian components of $\vec{k}$. Then, element t_{ji} (r_{ji}) in each of the transmission (reflection) submatrices represents the probability of a carrier that is incident in mode i being transmitted (reflected) in mode j, for a given electric field. The element accounts for both, scattering, and the acceleration by the electric field. Figure 6.3 shows an isolated slab for which each of the incident and emerging fluxes have been resolved into M modes, and the corresponding 2M by 2M scattering matrix.



$$[\,t\,] = \begin{bmatrix} t_{11} & t_{12} & \cdots & t_{1M} \\ t_{21} & t_{22} & \cdots & t_{2M} \\ \vdots & \vdots & & \vdots \\ t_{M1} & t_{M2} & \cdots & t_{MM} \end{bmatrix}$$

Figure 6.3: Discretization of incident and emerging fluxes.

The accuracy of the Scattering Matrix Approach strongly depends on the number of modes used to discretize the energy space. More modes mean a higher resolution, thus increasing the accuracy of the computation. A higher resolution also increases the computation time and the amount of memory required to solve the problem. The Multi-Flux Scattering Matrix Approach can result in very large prolblem sizes. For example, the simulation of carrier transport in a typical semiconductor device could involve scattering matrices with several hundred thousand non-zero elements, and several thousand matrix vector multiplications.

6.3 Simulations Involving Multi-band Transitions

Any change in the energy or the direction of travel of an incident carrier can cause it to change modes. In addition, depending on the material of the semiconductor device and the electric field strength, a carrier can also make transitions between different conduction bands. This effect can be ignored in devices made of specific semiconductors, if the electric field strength is below a specified threshold. In other cases, it becomes necessary to take the transitions between different bands into account.

Consider a simulation where a carrier may make transitions between B bands. Then, for this case, each of the transmission and reflection submatrices of the scattering matrix is made up of B^2 M by M submatrices, where M is the number of modes. The elements of submatrix ji of the transmission (reflection) submatrix represent the probabilities of carriers in band i making a transition to band j. Each individual element still represents the probability of a carrier in one mode being transmitted (reflected) in another mode, as described earlier.

For example, consider a two-band simulation. The transmission submatrix, t, is made up of four submatrices as follows.

$$t = \begin{bmatrix} T_{11} & T_{12} \\ T_{21} & T_{22} \end{bmatrix}$$

The element T_{21} represents the probability of a carrier incident in some mode in band *1* emerging in some mode in band 2. Similar comments can be made about the other three submatrices, t', r, and r'. Thus, for a two-band simulation, the size of the complete scattering matrix is 4M by 4M.

6.4 The Sparsity Pattern of the Scattering Matrix

The scattering matrix for a slab can be generated using Monte Carlo simulation. The Monte Carlo simulation is a statistical method which involves simulating the individual trajectories and scattering events of thousands of carriers as they pass through a device. Several thousand electrons distributed in mode i are injected into the semiconductor slab, and the elements t_{ji} and r_{ji} are determined by keeping track of the mode in which the electrons exit. This process is repeated for each mode to evaluate thr: entire scattering matrix. Because the scattering matrices are evaluated using a statistical solution method, the sparsity structure of the matrices has a certain amount of "fuzziness" to it.

A carrier injected into the slab in one mode can emerge in a different mode either by gaining or losing energy, or by being deflected, or both. Both, the change in energy, and the change in trajectory can occur because of either the scattering effects or the electric field. The amount of energy that an electron carrier could gain or lose is dependent upon the electric field strength and the scattering in the slab. The actual sparsity structure of a scattering matrix depends on how the energy space is discretized; the structures of two scattering matrices based on the discretization discussed earlier are shown in Figure 5.2 and Figure 6.4. Both scattering matrices are for the same semiconductor and slab thickness - the first one is evaluated at an electric field strength of 300kV/cm, and the second one is evaluated at a much lower electric field strength of 1kV/cm.

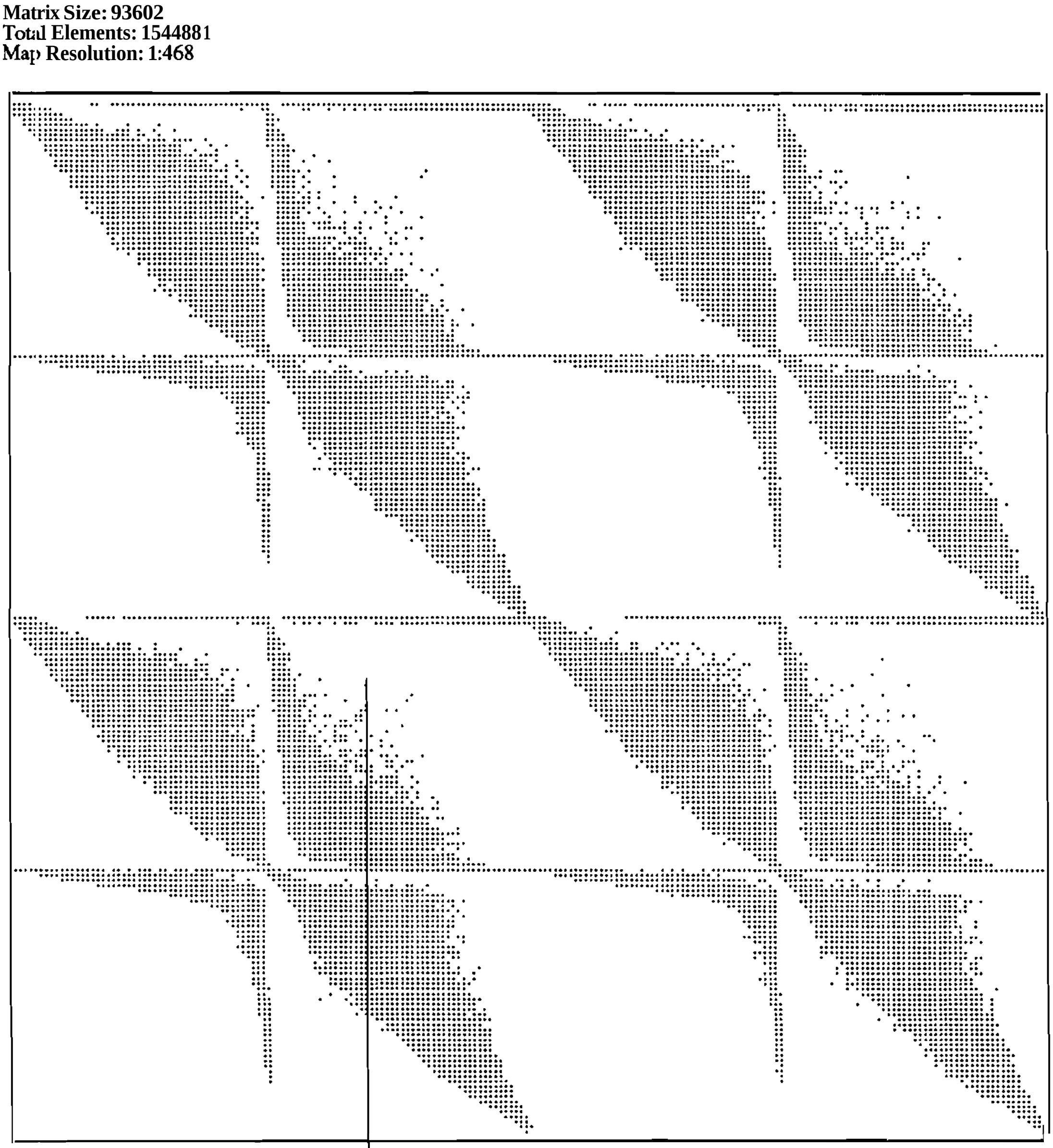


Figure 6.4: The sparsity pattern of a scattering matrix evaluated at an electric field strength of 1kV/cm.

6.5 Experimental Results

To benchmark the performance of the block row algorithm for the scattering matrix approach, parallel versions of two simulation programs existing on a IBM RS 6000/580 were written on a 16,384 processor MasPar MP-1. The sequential programs were written in FORTRAN, and were made available by [HuL93]. Sparse matrix-vector multiplications forms the computational core of the programs, and more than 98.5% of the total computation time is spent doing the matrix-vector multiplications.

The IBM RS 60001580 is a superscalar processor which is capable of issuing multiple instructions (up to four) in every cycle, and it has a floating point *multiply-add* instruction that can execute in one clock cycle (16ns) [War90]. In comparison, the MasPar MP-1 is a SIMD computer with up to 16,384, and can do a double precision addition (multiplication) on the processing elements in approximately 189 (557) 70ns clock cycles [MPA93]. The IBM RS 60001580 is rated at 62.5 MFLOPS for double precision (64-bit) floating point operations (38.1 MFLOPS for LINPACK), whereas a 16,384 processor MP-1 is rated at 630 MFLOPS peak (440 MFLOPS for LINPACK) for double precision floating point operations (average of add and multiply times). This makes the RS 60001580 significantly faster than an individual processing element on the MP-1; a 16k processor MP-1 is at the most about ten times faster (peak) than a RS 60001580.

The first program involves the simulation of carrier transport in a bulk *semiconductor*, which is a "thin" device with a uniform electric field; that is, the device consists of just one slab. This program was run on both machines with the 300kV/cm scattering matrix shown in Figure 5.2; the timing results for the simulation are: shown in Table 6.1. It can be seen that the MP-1 is approximately seven times faster than the RS 60001580.

The second program simulates carrier transport in a device with a low-high-low (1kV/cm - 300kV/cm - 1kV/cm) electric field distribution; the device was divided into sixty slabs. The scattering matrices shown in Figure 5.2 (300kV/cm) and Figure 6.4

(1kV/cm) were used; the results of the simulation on the two machines are shown in Table 6.2. This program also runs approximately seven times faster on the MP-1.

6.6 Conclusions

The simulations on the MP-1 ran approximately seven times faster than the corresponding simulations on the RS 6000/580 - as compared to a theoretically possible speedup of approximately ten (peak). However, the speedup obtained in terms of the number of floating point operations is higher than the figure indicated by the experimental results because of several reasons. The sparse matrix-vector routine on the RS 6000/580 has been optimized to take advantage of sparse vectors (with the current discretization for the generation of the scattering matrices, entire blocks of the vector, adding to approximately 20% of the total size, consist of zero elements), whereas the sparse matrix-vector routine on the MP-1 was not designed to take this sparsity into account. Also, for matrix-vector multiplication, the RS 6000 can take advantage of its multiply-add instruction; if this instruction is considered to be two FLOPS, the peak speed of the RS 6000/580 can be as much as 125 MFLOPS.

Additionally, as mentioned in Chapter 5, the scattering matrices have a rather complex structure, thus making it difficult to find the best blocksize for the basic version of the block row algorithm. The performance of the algorithm, was tested for blocksize values from one to twenty five for each matrix, and the best blocksize was used (for each matrix) to obtain the run times presented in this chapter.

Table 6.1: Computation times on the RS 6000/580 and the MP-1 for carrier simulation in a bulk semiconductor.

<table>
<tr><th colspan="4">Bulk Semiconductor Simulation</th></tr>
<tr><td colspan="4">S-Matrix: <b>300kV/cm,</b> N = 93602,1427614 non-zero elements.</td></tr>
<tr><td>Machine</td><td>Iterations</td><td>Run Time
(seconds)</td><td>Normalized
Run Time</td></tr>
<tr><td>IBM RS 60001580</td><td>42</td><td>57.00</td><td>7.05</td></tr>
<tr><td><b>MasPar</b> MP-1</td><td>42</td><td>8.09</td><td>1.00</td></tr>
</table>

Table 6.2: Computation times on the RS 6000/580 and the MP-1 for carrier simulation in a semiconductor device with a low-high-low electric field distribution.

<table>
<tr><th colspan="4">Lo-Hi-Lo Simulation</th></tr>
<tr><td colspan="4">Lo-Field S-Matrix: <b>1kV/cm,</b> N = 93602,1544881 non-zero elements.
Hi-Field S-Matrix: <b>300kV/cm,</b> N = 93602,1427614 non-zero <b>elements.</b></td></tr>
<tr><td>Machine</td><td>Iterations</td><td>Run Time
(seconds)</td><td>Normalized
Ruin Time</td></tr>
<tr><td>IBM RS 60001580</td><td>80</td><td>14,198</td><td>7.39</td></tr>
<tr><td><b>MasPar</b> MP-1</td><td>80</td><td>1,920</td><td>1.00</td></tr>
</table>

CHAPTER 7
A FEM APPROACH FOR MODELING DIFFRACTIVE AND SCATTERING OBJECTS

7.1 Introduction

This chapter provides a short description of the finite element approach for the numerical analysis and modeling of diffractive and scattering objects [Lic93], and a brief summary of the results for a sample problem involving the solution of a complex matrix system associated with the finite element implementation. The complex conjugate gradient squared method [NaR92] is implemented on a 16,384 processor MasPar MP-1, and its performance is compared to the performance of the same method and the performance of a complex direct solver on serial machines.

7.2 Numerical Analysis of Diffractive and Scattering Objects [Lic93]

Several applications such as seismology, geophysics, weather prediction, and electromagnetics, require the solution of wave-like equations in an infinite domain. For such problems involving the computation of scattering and diffractive effects of objects in open regions, it is necessary to limit the computational domain to a finite size. This can be done either by mapping the infinite region onto a bounded one, or by constructing an artificial boundary and imposing conditions on the boundary to simulate the infinite region.

In order to compute an efficient numerical solution without too much reflection from the outer boundary, it is desirable to get as close as possible to the scattering object. Normally, the artificial boundaries are circular or spherical in shape. However, for elongated scatterers (objects), these special boundaries are inefficient because a large computational domain is required. The work done in [Lic93] focuses on generalizing the circular boundary condition to an arbitrary boundary shape for two-dimensional geometries.

A finite element implementation is used to discretize a variational form of the wave equation, which results in a unsymmetric, complex sparse system (typically, with a fill-in of less than 1%); the sparsity structure of the matrix depends on the size and the shape of the scatterer(s) in the domain (the structure is symmetric, however). A parallel iterative solver for complex systems using the conjugate gradient squared method was implemented on the MasPar MP-1; the results for one of the problems solved are presented in the next section.

7.3 Numerical Results

Consider a domain with a radius of 5λ that contains a conducting scatterer of size $(6 \times 0.1)\lambda$, where λ is the wavelength of the incident waveform. A node density of 20 nodes/λ is used to discretize the domain, resulting in a sparse system with 36,818 unknowns and 255,406 non-zero coefficients (matrix elements). It is required to find the transverse magnetic polarization in the domain when a plane wave with unit magnitude is incident at 45^o to the normal.

This problem was solved with the help of serial and massively parallel implementations of the conjugate gradient squared (CGS) method, and with the help of a direct (serial) sparse matrix solver (the Y12M, developed in Copenhagen, Denmark). The massively parallel implementation uses the block row algorithm (with a blocksize of seven). For the iterative solutions, a residual norm of 1E-03 is used as the stopping condition; this residual norm was verified to result in an accurate solution.

A comparison of the number of iterations and the solution times for the CGS method on three different machines is presented in Table 7.1. Recall that a 16,384 processor MP-1 is rated at 630 MFLOPS (440 MFLOPS for LINPACK); the RS

60001560 is rated at 50 MFLOPS (30.5 MFLOPS for LINPACK), and the Ardent Titan is a four processor vector computer with each processor rated at 16 MFLOPS (6 MFLOPS for LINPACK). Code on the RS 60001560 was compiled with the "-O" option, and the code on the Ardent Titan was compiled with the "-04" option. Iterative methods converge faster if the starting vector is a good guess - if the incident field is used as an initial guess, the number of iterations required to converge decreases somewhat (Table 7.2). Table 7.3 shows the relative speeds of the direct method on the Ardent Titan, and the CGS method on the MP-1 (with the incident field as the initial guess); the CGS method on the MP-1 is faster by a factor of seven. Finally, the verification of the accuracy of the result is shown in Table 7.4; the result was also verified graphically.

7.4 Conclusions

The CGS method on the MP-1 runs approximately 3.5 times faster than the same method on the RS 60001560, and approximately 23.5 times faster than the CGS method on a four processor Ardent Titan as compared to the theoretically possible (peak) speedups of about thirteen and ten over the RS 60001560 and the Ardent Titan, respectively. Note that the RS 60001560 can perform a double precision multiply-add every cycle [War90]; if the multiply-add instruction is considered to be two FLOPS, then a 16,384 processor MP-1 is only about six times faster than the RS 6000/560. The CGS method on the MP-1 also ran seven times faster than the direct method on the Ardent Titan; a higher speedup can be achieved with a better initial guess for the iterative method.

Table 7.1: A comparison of the iterations and the execution times for the solution of the problem using the conjugate gradient squared method; stopping condition: residual norm < le-03.

Conjugate Gradient Squared Method			
N: 36,818 N_{elts}: 255,406			
Machine	**Iterations**	**Computation Time (seconds)**	**Normalized Time**
Ardent Titan	5359	11,867	23.45
RS 6000/560	5362	1,747	3.45
MasPar MP-1	4616	506	1.00

Table 7.2: Effect of an initial guess vector on the number of iterations required to converge.

Conjugate Gradient Squared Method		
N: 36,818 N_{elts}: 255,406		
Starting Vector	**Iterations**	**Computation Time (seconds)**
$x_0 = 1.0$ $x_i = 0.0, i \neq 0$	4616	505.72
x = Incident Field	4497	492.53

Table 7.3: **A comparison of the performance of the parallel implementation of the conjugate gradient squared method with that of a serial sparse direct solver.**

N: 36,818 N_{elts}: 255,406		
Method/ Machine	**Solution Time (seconds)**	**Normalized Time**
Direct Solver on Ardent Titan	3,473	7.0
CGS Method on MasPar MP-1	493	1.0

Table 7.4: **The change in the error norm of the solution from the CGS method (relative to the solution from the direct method) with a change in the stopping condition.**

$\frac{\lVert x_{direct} - x_{cgs} \rVert_2}{\lVert x_{direct} \rVert_2}$	
N: 36,818 N_{elts}: 255,406	
Residual Norm: 1E-03	**Residual Norm: 1E-06**
4.371×10^{-5}	4.330×10^{-5}

CHAPTER 8
CONCLUSIONS

8.1 Summary

Sparse matrix-vector multiplication is an integral component of a large number of problems in numerical analysis. In spite of the inherent parallelism available in the procedure, it is difficult to design an algorithm for distributed memory machines that performs well for general unstructured sparse matrices. We have proposed a new algorithm that is designed for unstructured sparse matrices that have relatively sparse columns.

The procedure of sparse matrix-vector multiplication is analyzed for serial and parallel machines with a distributed memory system. The parallel procedure is divided into four phases - the fetch phase, the multiplication phase, the reduction phase, and the arrange phase. For distributed memory machines, the fetch phase and the reduction phase account for most of the interprocessor communication. It is difficult to simultaneously optimize the fetch phase and the reduction phase, and also achieve a good load balance between the processors, for unstructured sparse matrices.

A SIMD architecture represents an additional restriction on the design of the algorithm because all enabled processors must perform the same operation at any given time. The restrictions imposed by a SIMD computer with a two-dimensional mesh interconnection network on the design of an algorithm for sparse mamx-vector multiplication are studied, and the block row algorithm is developed based on the conclusions of the study.

A detailed description of the block row algorithm is presented., along with an example. The algorithm is then analyzed, and the analysis is supported with experimental evidence. For the types of matrices that are associated with the problems being considered, the experimental analysis presented in Chapter 5 shows that the block row algorithm is faster than the ''snake-like'' method, the ''segmented scan'' method, and the randomized packing algorithm.

8.2 Future Work

The work presented in this thesis only optimizes the ''reduction''' phase and the ‘‘result’’ phase. As seen from the experimental data in Chapter 5, the performance of the block row algorithm is limited by the fetch phase. This bottleneck occurs because a processor can only process one communication request at a time. While it is not possible to completely optimize the fetch phase simultaneously, it would be possible to permute the elements in the individual ''blocks'' in the block row algorithm so as to minimize the number of elements from any single column that are mapped to any one layer of memory. This could result in a significant improvement in the performance of the algorithm because the router conflicts will be minimized.

As seen in Chapter 5, the matrices arising from the scattering matrix approach are not very tractable for the block row algorithm because the number of non-zero elements in the different rows are very different. As a result, it is difficult to obtain one ''good'' blocksize. The block row algorithm can be extended to an ''adaptive'' block row algorithm, where different blocksizes can be used for different parts of the matrix. This algorithm needs to be implemented and tested.

LIST OF REFERENCES

LIST OF REFERENCES

[AnM92] J. Anderson, G. Mitra, and D. Parkinson, "The Scheduling of Sparse Matrix-Vector Multiplication on a Massively Parallel DAP Computer", Parallel Computing, Vol. 18, June 1992, pp 675-697.

[BiW92] Aart J. C. Bik and Harry A. G. Wijshoff, "Compilation Techniques for Sparse Matrix Computations", extended abstract of "Automatic Data Structure Selection and Transformation for Sparse Matrix Computations" by the same authors, TR No. 92-24, Dept. of Computer Science, Leiden University, 1992.

[BjM92] P. Bjorstad, F. Manne, T. Sorevik, and M. Vajtersic, ":Efficient Matrix Multiplication on SIMD Computers", SIAM Journal of Matrix Anal. Appl., Vol. 13, No. 1, January 1992, pp 386-401.

[Bla90] Tom Blank, "The MasPar MP-1 Architecture", Proceedings of IEEE Compcon Spring 1990, IEEE, February 1990, pp 20-24.

[Chr90] Peter Christy, "Software to Support Massively Parallel Computing on the MasPar MP-1", Proceedings of IEEE Compcon Spring 1990, IEEE, February 1990, pp 29-33.

[Das90] Amitava Das, "The Scattering Matrix Approach to Device Analysis", PhD Thesis, TR-EE 91-11, March 1991.

[DuR79] I. S. Duff and J. K. Reid, "Some Design Features of a Sparse Matrix Code", ACM Transactions on Mathematical Software, Vol. 5, No. 1, March 1979, pp 18-35.

[Ham92] Steven Warren Harnmond, "Mapping Unstructured Grid Computations to Massively Parallel Computers", PhD. Thesis, 1992, Chapter 5, pp 93-104.

[HuL93] Carl Huster, Prof. Mark Lundstrom, Personal Communications, April - July 1993.

[JoH89] S. Lennart Johnsson, Tim Harris, and Kapil K. Mathur, "Matrix Multiplication on the Connection Machine", Proceedings of Supercomputing, 1989, pp 326-332.

[KuG94] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis, "Introduction to Parallel Computing: Design and Analysis of Algorithms", The Benjamin/Cummings Publishing Company, Inc., 1994, Chapter 11.

[Lic93] Bernd Lichtenberg, "A Finite Element Approach for the Numerical Analysis and Modeling of Diffractive and Scattering Objects", A proposal submitted to the faculty of Purdue University for the Degree of Doctor of Philosophy, August 1993.

[LiS88] Hungwen Li and Ming-Cheng Sheng, "Sparse Matrix Vector Multiplication on a Polymorphic-Torus", Proceedings of the Second Symposium on the Frontiers of Massively Parallel Computing, 1988, pp 181-186.

[Mel88] Rami Melhem, "Parallel Solution of Linear Systems with Striped Sparse Matrices", Parallel Computing, Vol. 6, No. 2, February 1988, pp 165-184.

[MiK93] Manavendra Misra and V. K. Prasanna Kumar, "Efficient VLSI Implementation of Iterative Solutions to Sparse Linear Systems", Parallel Computing, Vol. 19, No. 5, May 1993, pp 525-544.

[MoM82] M. Morjaria and G. J. Makinson, "Unstructured Sparse Matrix Vector Multiplication on the DAP", Supercomputers and Parallel Computation, 1984, pp 157-166.

[MPA93] "MasPar Parallel Application Language (MPL) User Guide", MasPar Computer Corporation, Revision A5, July 1993, Chapter 4, pp 4.1-4.12.

[MMP90] "MasPar MP-1 Principles of Operation", MasPar Computer Corporation, Revision, July 1990, Chapter 3, pp 3.1-3.8.

[MPP93] "MP-PCGPAK3 User's Guide", MP-PCGPAK3 Version 1.O, Scientific Computing Associates, Inc., June 1993 pp 9-11.

[NaR92] Noel M. Nachtigal, Satish C. Reddy, and Lloyd N. Trefethen, "How Fast are Nonsyrnrnetric Matrix Iterations ?", SIAM J. Matrix Anal. Appl., Vol 13, No. 3, July 1992, pp 778-795.

[Nic90] John R. Nickolls, "The Design of the MasPar MP-1:: Cost Effective Massively Parallel Computer", Proceedings of IEEE Compcon Spring 1990, IEEE, February 1990, pp 25-28.

[OgA93] Andrew T. Ogielski and William Aiello, "Sparse Matrix Computations on Parallel Processor Arrays", SIAM J. Sci. Comput., Vol. 14, No. 3, May 1993, pp 519-530.

[Pet91] Alexander Peters, "Sparse Matrix Vector Multiplication Techniques on the IBM 3090 VF", Parallel Computing, Vol. 17, December 1991, pp 1409-1424.

[RoZ93] L. F. Romero and E. L. Zapata, "Data Distributions for Sparse Matrix Vector Multiplication", to be published, 1993.

[Ste91] Mark A. Stettler, "Simulation of Silicon Bipolar Transistors Using the Scattering Matrix Approach", A proposal submitted to the faculty of Purdue University for the Degree of Doctor of Philosophy, September 1991, pp 12-17.

[Tic89] Walter F. Tichy, "Parallel Matrix Multiplication on the Connection Machine", International Journal of High Speed Computing, Vol. 1, No. 2,1989, pp 247-262.

[War90] H. S. Warren, Jr., "Instruction Scheduling for the IBM RISC System/6000 Processor", IBM Journal of Research and Development, January 1990, Vol. 34, No. 1, pp 85-92.

APPENDIX

APPENDIX: CODE LISTING FOR THE BLOCK ROW ALGORITHM

```
#define     SEND              0
#define     EDGE              1
#define     NO–SEND           2
#define     INVALID–ROW      -1
#define     INVALID–COL      -1


void mat_vect_mul(plural double re_coef[], plural double im_coef[],
                  plural int row–index[], plural int col_index[],
                  plural double re–vector[], plural double im_vector[],
                  plural double re_result[], plural double im_result[],
                  int matrix–size, int total–elts, int blocksize[])
{
  register int i, j, k;
  register int vector–layers, complete_elt_layers, reg-blocksize;
  register int processors, remaining-elts;
  register int step, step2;
  register plural int base;
  register plural int remote–pe, layer;
  register plural int temp–col–index;
  register plural double re–reg–vector, im_reg_vector;
  plural double re–temp–result, im_temp_result;
  register plural double re–temp–vector, im_temp_vector;


  /* Compute necessary variables and copy frequently used vars to registers */
  processors = nproc;
  if(matrix_size <= nproc)
  {
    vector–layers = 1;
    step = matrix_size/nxproc + ((matrix_size%nxproc) != 0);
    step2 = nxproc * step;
    base = (iproc/step2) * step2;
    re–reg–vector = re_vector[0];
    im_reg_vector = im_vector[0];
    re_result[0] = 0.0;
    im_result[0] = 0.0;
  /* Make copies of the vector, if possible */
  if(iproc >= step2)
    for(i=step2; i < nproc && i < total_elts; i += step2)
    {
      re–reg–vector = xnetN[step].re_reg_vector;
      im_reg_vector = xnetN[step].im_reg_vector;
    }
}
else
{
  vector–layers = matrix_size/nproc + ((matrix_size%nproc) != 0);
  for(i=0; i < vector–layers; i++)
  {
    re_result[i] = 0.0;
    im_result[i] = 0.0;
  }
}

reg_blocksize = blocksize[0];
complete_elt_layers = ((total_elts/reg_blocksize)/nproc)*reg_blocksize;
i = 0;
re–temp–result = 0.0;
im_temp_result = 0.0;
while(i < complete_elt_layers)
{
  for(k=0; k < reg_blocksize; k++, i++)
  {
    /* Fetch Appropriate Vector Elements */
    temp–col–index = col_index[i];
    if(temp_col_index != INVALID–COL)
    {
      if(matrix_size > nproc)
      {
        remote–pe = temp_col_index%nproc;
        layer = temp_col_index/nproc;
        for(j=0; j < vector–layers; j++)
          if(layer == j)
          {
            re–regvector = re–vectorlj];
            im_reg_vector = im_vector[j];
            re–temp–vector = router[remote_pe].re–reg–vector;
            im_temp_vector = router[remote_pe].im_reg_vector;
          }
      }
      else
      {
        remote–pe = base + temp–col–index;
        if(processors > matrix–size)
          remote–pe -= (remote–pe >= processors)*step2;
        re–temp–vector = router[remote_pe].re_reg_vector;
    im–temp–vector = router[remote_pe].im_reg_vector;
   }
  }
  /* Multiply corresponding elements */
  re–temp–result += re_coef[i]*re–temp–vector - im_coef[i]*im_temp_vector;
  im_temp_result += re_coef[i]*im_temp_vector + im_coef[i]*re–temp–vector;
 }
 send_part_product(re_result, im_result, &re–temp–result, &im_temp_result,
           row–index, i, vector–layers, matrix–size,
           complete_elt_layers, nproc);
}

remaining_elts = total–elts - complete_elt_layers*nproc;
if(remaining_elts == 0)
 return;
reg–blocksize = blocksize[1];
processors = remaining_elts/reg_blocksize;

re–temp–result = 0.0;
im_temp_result = 0.0;
for(k=0; k < reg–blocksize; k++, i++)
{
 if(iproc < processors)
 {
  /* Fetch Appropriate Vector Elements */
  temp–col–index = col_index[i];
  if(temp_col_index != INVALID–COL)
  {
   if(matrix_size > nproc)
   {
    remote–pe = temp_col_index%nproc;
    layer = temp_col_index/nproc;
    for(j=0; j < vector–layers; j++)
     if(layer == j)
     {
      re_reg_vector = re_vector[j];
      im_reg_vector = im_vector[j];
      re–temp–vector = router[remote_pe].re–regvector;
      im–temp–vector = router[remote_pe].im_reg_vector;
     }
   }
   else
   {
    remote–pe = base + temp–col–index;
    if(processors > matrix–size)
     remote–pe -= (remote–pe >= processors)*step2;

    re–temp–vector = router[remote_pe].re_reg_vector;
    im–temp–vector = router[remote_pe].im_reg_vector;
   }
   /* Multiply corresponding elements */
   re–temp–result += re_coef[i]*re_temp_vector - im_coef[i]*im_temp_vector;
   im_temp_result += re_coef[i]*im_temp_vector + im_coef[i]*re_temp_vector;
  }
  send_part_product(re_result, im_result, &re–temp–result, &im_temp_result,
                    row–index, i, vector–layers, matrix–size,
                    complete–elt–layers+1, processors);
  return;
}



void send_part_product(plural double re–result[], plural double im_result[],
                    plural double *re–temp–result,
                    plural double *im_temp_result,
                    plural int row_index[], register int i,
                    int vector–layers, int matrix–size, int last–layer,
                    int processors)
{
  register int j;
  register plural char send–flag, temp–flag;
  register plural int itempl;
  register plural int temp–row–index;
  register plural int remote–pe, layer;
  register plural double re–dtempl, im_dtempl;
  register plural double re_dtemp2, im_dtemp2;
  register plural double re_dtemp3, im_dtemp3;
  register plural double re–dbuffer, im_dbuffer;


  re_dtemp2 = *re–temp–result;
  im_dtemp2 = *im_temp_result;
  temp–row–index = row_index[i-1];
  all send–flag = NO–SEND;

  if(i < last–layer)
    itemp1 = row_index[i];
  else
    itemp1 = INVALID–ROW,

  /* Collect Partial Results (of each row) from Adjacent Processors */
  if((iproc < processors) && (itempl != temp–row–index))
  {
    /* Reset Partial Products to zero (value held in temp register) */
    *re–temp–result = 0.0;
    *im_temp_result = 0.0;
```

```
/* Create the 'send mask' vector needed for reduction */
send–flag = SEND;
if((iproc+1) % nxproc == 0)
{
  if(xnetSE[1].temp–row–index != temp–row–index)
    send–flag = EDGE;
}
else if(xnetE[1].temp–row–index != temp–row–index)
  send–flag = EDGE;
if(processors < nproc)
  proc[processors-1].send–flag = EDGE;

if(send_flag == EDGE)
{
  re_dbuffer = 0.0;
  im–dbuffer = 0.0;
}
else
{
  re–dbuffer = re–dtemp2;
  im–dbuffer = im_dtemp2;
}

while(send_flag == SEND)
{
  all
  {
    re_dtemp3 = 0.0;
    im_dtemp3 = 0.0;
  }
  if((iproc+1) % nxproc == 0)
  {
    xnetSE[1].re_dtemp3 = re–dbuffer;
    xnetSE[1].im_dtemp3 = im–dbuffer;
  }
  else
  {
    xnetE[1].re_dtemp3 = re_dbuffer;
    xnetE[1].im_dtemp3 = im–dbuffer;
  }
  all
  {
    re_dbuffer = re_dtemp3;
    im–dbuffer = im_dtemp3;
    if(send_flag == EDGE)
    {
      re_dtemp2 += re–dbuffer;
      im_dtemp2 += im_dbuffer;

      re_dbuffer = 0.0;
    im_dbuffer = 0.0;
    }
   temp−flag = send−flag;
  }

  if(iproc % nxproc == 0)
  {
   if(xnetNW[1].temp−flag != SEND)
    send−flag = NO−SEND;
  }
  else if(xnetW[1].temp−flag != SEND)
   send−flag = NO−SEND;
 }
}

/* Send Collected Results to Respective Processors */
if(send_flag == EDGE)
{
 /* Send Result-Vector Elements to Appropriate Processors */
 if(matrix_size > nproc)
 {
  remote_pe = temp_row_index%nproc;
  layer = temp_row_index/nproc;
  for(j=0; j < vector−layers; j++)
  {
   all
   {
    re_dtemp1 = 0.0;
    im_dtemp1 = 0.0;
   }
   if(layer == j)
   {
    router[remote_pe].re_dtemp1 = re_dtemp2;
    router[remote_pe].im_dtemp1 = im_dtemp2;
   }
   all
   {
    re−resultti] += re−dtemp1;
    im_result[j] += im_dtemp1;
   }
  }
 }
 else
 {
  all
  {
   re_dtemp1 = 0.0;
   im_dtemp1 = 0.0;
  }
```

```
    router[temp_row_index].re_dtemp1 = re–dtemp2;
    router[temp_row_index].im_dtemp1 = im_dtemp2;
    all
    {
      re_result[0] += re_dtemp1;
      im_result[0] += im_dtemp1;
    }
  }
 }
 return;
}
```