Segmented Operations for Sparse Matrix Computation on Vector Multiprocessors

Guy E. Blelloch   Michael A. Heroux*   Marco Zagha

August 1993

CMU-CS-93-173

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*Cray Research, Inc.
655F Lone Oak Drive
Eagan, MN 55121

This research was sponsored in part by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597, and in part by the Pittsburgh Supercomputing Center (Grant ASC890018P) and Cray Research Inc. who provided Cray Y-MP and Cray Y-MP C90 time. Guy Blelloch was partially supported by a Finmeccanica chair and an NSF Young Investigator award (Grant CCR-9258525).

Keywords: Sparse matrix, matrix vector multiplication, parallel algorithms, vector algorithms, segmented scan, segmented sum, vector multiprocessor, Cray Y-MP C90

Abstract

In this paper we present a new technique for sparse matrix multiplication on vector multiprocessors based on the efficient implementation of a *segmented sum* operation. We describe how the segmented sum can be implemented on vector multiprocessors such that it both fully vectorizes within each processor and parallelizes across processors. Because of our method's insensitivity to relative row size, it is better suited than the Ellpack/Itpack or the Jagged Diagonal algorithms for matrices which have a varying number of non-zero elements in each row. Furthermore, our approach requires less preprocessing (no more time than a single sparse matrix-vector multiplication), less auxiliary storage, and uses a more convenient data representation (an augmented form of the standard compressed sparse row format).

We have implemented our algorithm (SEGMV) on the Cray Y-MP C90, and have compared its performance with other methods on a variety of sparse matrices from the Harwell-Boeing collection and industrial application codes. Our performance on the test matrices is up to 3 times faster than the Jagged Diagonal algorithm and up to 5 times faster than Ellpack/Itpack method. Our preprocessing time is an order of magnitude faster than for the Jagged Diagonal algorithm. Also, using an assembly language implementation of SEGMV on a 16 processor C90, the NAS Conjugate Gradient benchmark runs at 3.5 gigaflops.

1 Introduction

The solution of sparse linear systems is an essential computational component in many areas of scientific computing. In particular, the solution of unstructured sparse linear systems is important in computational fluid dynamics, circuit and device simulation, structural analysis, and many other application areas. As problems become larger and more complex, iterative solution methods for these sparse linear systems become extremely attractive and, in some cases, are the only possibility for obtaining a solution. One of the most important kernels in many iterative solvers is sparse-matrix multiplication. In the case of unstructured problems, this kernel is typically difficult to optimize and, since it must be performed at each iteration, often consumes the majority of the computing time.

The standard technique to represent sparse arrays is the compressed sparse row format (CSR)—the matrix is stored in row-major order, but only the non-zero elements and their column indices are kept (an analogous compressed sparse column format is also used). Although the serial algorithm for sparse-matrix vector multiply using the CSR format can vectorize on a per-row basis, the performance is often poor since the number of non-zeroes in each row is typically small.

Because of the importance of sparse-matrix vector multiplication as a kernel, researchers have developed several methods for improving its performance on parallel and vector machines. An early technique, which is used by Ellpack/Itpack [23], pads all rows so they have an equal number of elements and then vectorizes across the columns. This works well when the longest row is not much longer than the average row. A variation of this method collects the rows into groups based on sizes and processes each group separately [32]. A further enhancement, called the Jagged Diagonal (JAD) method, sorts the rows based on row size, and decreases the number of rows processed as the algorithm proceeds across the columns [2]. The problem with all these methods is that they can require significant preprocessing time and they usually require that the matrix be reordered and copied, therefore requiring extra memory. Also, even though JAD is significantly better than Ellpack/Itpack for matrices where the row sizes vary, it still suffers a performance degradation when the largest row is significantly longer than the average row.

In this paper we present a technique called SEGMV for sparse matrix multiplication based on the efficient implementation of *segmented sums*. A segmented sum views a vector as partitioned into contiguous segments, each potentially of different sizes, and sums the values within each segment [6]. Unlike the algorithms mentioned above, SEGMV is insensitive to relative row size and is therefore well suited for matrices that are very irregular. Such matrices are becoming more common with the increased interest in simulating more complicated geometries and the use of automatic mesh generators. SEGMV also leaves the matrix in standard CSR format therefore not requiring extra space for the reorganized matrix. Finally, although SEGMV does require preprocessing to generate a so-called *segment-descriptor*, this preprocessing is small—it takes no more time than a single sparse-matrix vector multiplication. The algorithm is therefore well suited for applications in which the matrix is only used a few times (as is often the case with adaptive meshes). Segmented operations have been used for sparse-matrix vector multiplication with good success on the Connection Machine [9, 36], but the application to vector multiprocessors is new.

We have implemented the SEGMV algorithm on the Cray Y-MP C90 and have compared its running time to various other algorithms on several sparse matrices from the Harwell-Boeing collection and industrial application codes. Figure 1 summarizes our results for 16 processors on our five largest test matrices. As expected, SEGMV is the least affected by the structure of the matrices. In addition to measuring times for matrix multiplication on the test matrices, we have used SEGMV as the core for the NAS Conjugate Gradient benchmark [3, 4]. On 16 processors of the C90, the benchmark using our algorithm achieves 3.5 Gigaflops[1].

[1]However, our implementation uses assembly language which is not permitted for official NAS results.

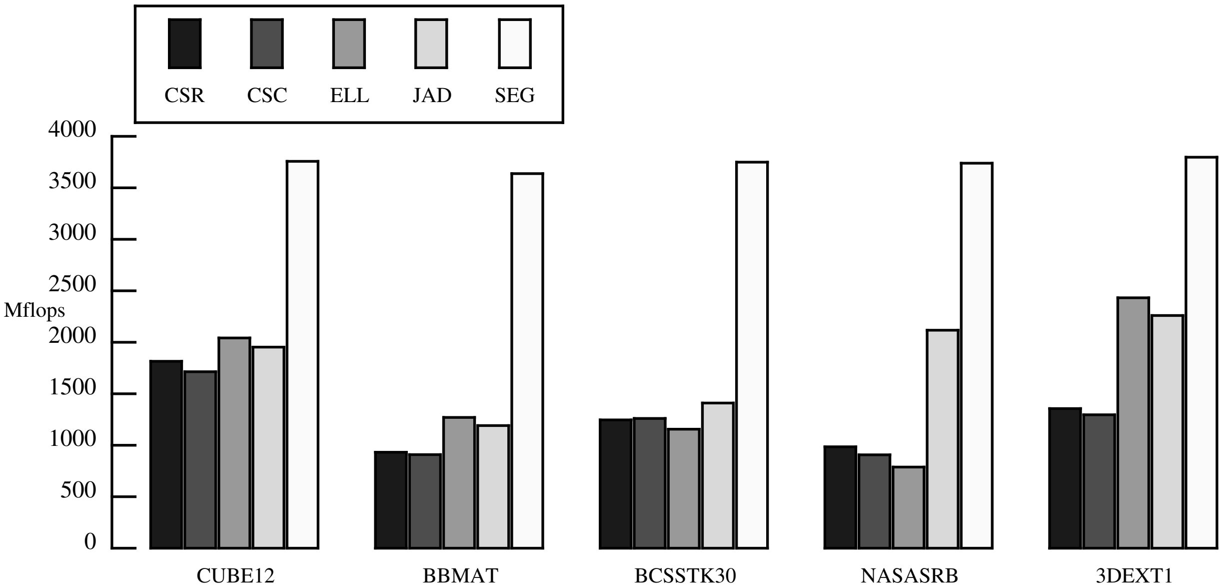


Figure 1: Performance results for sparse matrix multiplication comparing our implementation (SEG) to previous implementations: Compressed Sparse Row (CSR), Compressed Sparse Column (CSC), Ellpack/Itpack (ELL), and Jagged Diagonal (JAD). The graph shows the performance in Mflops on 16 processors of a Cray Y-MP C90 for the five largest matrices in our test suite. The previous implementations are described in Section 2. The full test suite and further performance results are presented in Section 4.

The remainder of this paper proceeds as follows. In Section 2 we give a brief presentation of commonly used sparse matrix data structures and algorithms. In Section 3 we introduce segmented computation and describe the SEGMV algorithm. Section 4 contains results comparing SEGMV to other techniques, using a collection of real-life test problems run on a Cray Y-MP C90 computer [29]. Section 5 discusses generalizations of the basic sparse matrix-vector multiplication algorithm. Finally, in Section 6, we present our conclusions.

2 Previous Work

This section discusses previous techniques for implementing sparse matrix-vector multiplication. We separate the known techniques into specialized techniques (techniques that work best for structured matrices), general techniques (techniques that work for sparse matrices with any structure) and hardware techniques (techniques that have been suggested for direct implementation in hardware). In general, the more regularity one can extract from the pattern of a sparse matrix, the better performance one can expect.

2.1 Specialized Techniques

Sparse Diagonal Systems: Standard discretizations of partial differential equations typically generate matrices which have only a few non-zero diagonals. An appropriate data structure for this type of structured sparse matrix would store only these non-zero diagonals along with offset values to indicate where each diagonal belongs in the matrix (see the DIA data structure in [33, 20]).

Feature Extraction: Another approach to optimizing sparse matrix operations is to decompose the matrix into additive submatrices and store each submatrix in a separate data structure. For example, the Feature

Extraction Based Algorithm (FEBA) presented by Agarwal, Gustavson, and Zubair [1] recursively extracts structural features from a given sparse matrix structure and uses the additive property of matrix multiplication to compute the matrix-vector product as a sequence of operations. The FEBA scheme first attempts to extract nearly dense blocks. From the remainder of entries, it then attempts to extract nearly dense diagonals and so on. This is a powerful technique, especially on machines where the performance of dense computations is substantially better than that of general sparse computations. However, one is required to have an extra copy of the original matrix with the potential for a non-trivial amount of fill-in. Also, pre-processing time can be significant.

2.2 General techniques

Although exploiting structural regularity is important, we consider in this paper data structures for unstructured sparse matrices. These data structures are appropriate in cases where the sparse matrix has no known regular sparsity pattern or where a general-purpose data structure is needed in order to handle a variety of sparse matrix patterns. There are many general-purpose sparse data structures, but we only discuss a few of them here. For more examples, see [1, 16, 18, 19, 20, 22, 27, 30, 31, 32, 33, 34].

Compressed Sparse Row: One of the most commonly used data structures is the compressed sparse row (CSR) format. The CSR format stores the entries of the matrix row-by-row in a scalar-valued array VAL. A corresponding integer array INDX holds the column index of each entry, and another integer array PNTR points to the first entry of each row in VAL and INDX. For example, let

$$A = \begin{pmatrix} 11 & 0 & 13 & 14 & 0 & 0 \\ 0 & 0 & 23 & 24 & 0 & 0 \\ 31 & 32 & 33 & 34 & 0 & 36 \\ 0 & 42 & 0 & 44 & 0 & 0 \\ 51 & 52 & 0 & 0 & 55 & 0 \\ 61 & 62 & 0 & 64 & 65 & 66 \end{pmatrix},$$

then A would be stored in CSR format as follows:

VAL	= (11	13	14	23	24	31	32	33	34	36	42	44	51	52	55	61	62	64	65	66),
INDX	= (1	3	4	3	4	1	2	3	4	6	2	4	1	2	5	1	2	4	5	6),
PNTR	= (1	4	6	11	13	16	21	).												

If the sparse matrix is stored in CSR format, then the sparse matrix-vector product operation, $y = Ax$, can be written as follows:

```
C  For each row of A, compute inner product of row I with X
      DO I = 1,M
         Y(I) = 0.0
         DO K = PNTR(I), PNTR(I+1)-1
            Y(I) = Y(I) + VAL(K)*X(INDX(K))
         END DO
      END DO
```

One should note that the vector lengths in inner loop are determined by the number entries in each row which is typically small and does not grow with the problem size. Thus, the vector performance of this kernel is usually poor.

The CSR format and its column analogue (CSC) are the most common general-purpose user data structures. They are flexible, easy to use and modify, and many matrix operations can be easily expressed with them. However, their performance is usually suboptimal for most matrix operations on vector multiprocessors. In order to achieve higher performance, loop interchange can be used in a variety of forms to operate on several rows at once. Because each row can be a different length, special data structures are often used to make the computation more regular. Two examples are the Ellpack/Itpack data structure and the Jagged Diagonal data structure.

Ellpack/Itpack: The Ellpack/Itpack (ELL) storage format [23] forces all rows to have the same length as the longest row by padding with zeros. Since the number of rows is typically much larger than the average row length, the sparse matrix-vector multiplication can be efficiently vectorized by expressing it as a series of operations on the columns of the modified matrix. The ELL format is efficient for matrices with a maximum row size close to the average row size, but the effective computational rate can be arbitrarily bad for matrices with general sparsity patterns because of wasted computation. In addition, memory space is wasted by storing extra zeros and possibly from having to maintain an unmodified version of the matrix.

Jagged Diagonal: The Jagged Diagonal (JAD) format [2] is also designed to exploit vectorizable column operations, but without performing unnecessary computation. In the JAD format, rows are permuted into order of decreasing length. The leftmost elements of each row (along with their column indices) are stored as a dense vector, followed by the elements second from the left in each row stored as a (possibly shorter) dense vector, and so on. Sparse matrix-vector multiplication proceeds by operating on each column in turn, decreasing the vector length as the length of the current column decreases. Sparse matrix-vector multiplication using the JAD format performs very well on vector multiprocessors for most matrices. However, performance can be suboptimal in some cases, e.g., when there are only a few long rows. The main disadvantage of the JAD representation is that constructing the data structure requires permuting the rows of the matrix. Thus, it is difficult to construct and modify, and makes expressing a standard forward/back solve or other operations difficult.

2.3 Hardware Techniques

Taylor, Ranade, and Messerschmitt [35] have proposed a modification to the compressed sparse column (CSC) data structure that stores the non-zero elements of each column with an additional zero value placed between each set of column values. Using this data structure, sparse matrix-vector multiplication can be expressed as a single loop over all non-zero elements, rather than a nested loop over columns and rows. Each iteration of the long vector loop tests the current matrix value for equality with zero and either continues accumulating or updates the current column index. The authors note that the conditional statement "cannot be executed efficiently on conventional vector machines because the vector length is not given explicitly" and propose special purpose zero-detection hardware to overcome this problem.

3 Segmented Scan Algorithms

Our approach combines several of the advantages of previous approaches. The data structure we use is an augmented form of the CSR format. The computational structure of our algorithm is similar to ELL and JAD in that we get parallelism from operating on many rows at once. But rather than using a heuristic approach to making the computation more regular, we reformulate sparse matrix-vector multiplication in terms of segmented vector operations. This approach is similar to the technique proposed by Taylor et al. since their data structure is one method for representing segmented vectors, and their algorithm for sparse matrix-vector multiplication using a single loop is essentially a segmented vector operation. However, our approach does not require special-purpose hardware because we are able to express the conditional computation in terms of standard vector merge instructions. The result of combining features of previous approaches is that we obtain an algorithm for vector multiprocessors that is insensitive to matrix structure, uses a convenient format, requires minimal preprocessing, exposes a large degree of parallelism (without performing unnecessary computation) and does not require special-purpose hardware. Section 3.1 introduces segmented scans, and Section 3.2 explains how segmented scans can be used for sparse matrix multiplication. Our algorithm is described in detail beginning in Section 3.3.

3.1 Segmented Scans: Background

The *scan operation*, also called the all-prefix-sums computation, takes a binary operator $\oplus$, and an array $[a_1, a_2, ..., a_n]$ of n elements, and calculates the values x_i such that

$$x_i \quad = \quad \begin{cases} a_1, & i = 1 \\ x_{i-1} \oplus a_i, & 1 < i \leq n \end{cases}$$

Although this calculation appears to be serial because of the the loop-carried dependence, if $\oplus$ is associative, it can be calculated efficiently in parallel in $\log_2 n$ time [24, 25]. This method also leads to an efficient algorithm for vector machines [8, 11]. The usefulness of scans in array computations has long been realized and the scan operations play a crucial role in the APL programming language. Associative operators that are often used include addition, maximum, minimum, logical-or, and logical-and.

The *segmented* scan operations take an array of values, and in addition take a second argument that specifies how this array is partitioned into segments [5]. A scan is executed independently within each segment. For example:

```
VAL                  = ( 5   1   3   4   3    9    2   6 )
FLAG                 = ( T   F   T   F   F    F    T   F )

ADD_SCAN(VAL,FLAG)   = ( 5   6   3   7   10   19   2   8 )
```

In this example, the `FLAG` array has a TRUE flag at the beginning of each segment. As with the the unsegmented version, the segmented scans can also be implemented in parallel in $\log_2 n$ time [5]. Chatterjee, Blelloch and Zagha showed how segmented scans can be implemented efficiently on vector computers [11] by using vector merge instructions to handle segment boundaries and a loop structuring technique called *loop raking* [8].

Segmented scans were originally introduced to solve problems with irregular data structures on the Connection Machine CM-2. Blelloch showed how segmented scans can be used to implement many data-parallel algorithms for problems with irregular structures, including sparse matrix routines [5, 6]. Other uses of segmented scans include computer graphics [15], object recognition [37], processing image contours [12], parallel quicksort [5], machine learning [9], and network optimization [28]. Because of their usefulness for such problems, hardware support was included in the Connection Machine CM-5 [26] for segmented scans, and the proposed High Performance Fortran (HPF) standard [21] contains scan intrinsics (called PREFIX and SUFFIX) with an optional argument for specifying segments.

3.2 Using Segmented Scans for Sparse Matrix Multiplication

A sparse matrix can be represented as a segmented vector by treating each row as a segment. Since the CSR (compressed sparse row) format is already organized in such segments, the only additional structure required is the flags array, which can be generated from the pointer array (`PNTR`). For example, for the matrix in Section 2.2 the flags would be organized as:

```
VAL  = ( 11 13 14  23 24  31 32 33 34 36  42 44  51 52 55  61 62 64 65 66 ),
FLAG = (  T  F  F   T  F   T  F  F  F  F   T  F   T  F  F   T  F  F  F  F ).
```

The following HPF code (which uses array syntax) can be used to generate the FLAGS array from the PNTR array:

```
FLAG = .FALSE.
FLAG(PNTR) = .TRUE.
```

The first line sets all the elements of the FLAG array to FALSE. The next line scatters TRUE's to the beginning of each row (i.e. each segment).

Using these flags, sparse matrix multiplication can be written as follows:

```
PRODUCTS = VAL * X(INDX)
SUMS = SUM_SUFFIX(PRODUCTS,SEGMENT=PARITY_PREFIX(FLAG))
Y = SUMS(PNTR)
```

This code first calculates all the products by indirectly accessing the values of X using INDX and multiplying them by VAL. The second line uses the SUM_SUFFIX function.[2] The SUFFIX functions in HPF execute the scan backwards, such that each element receives the sum of values that follow it, rather than precede it. The result of the SUM_SUFFIX on the PRODUCTS array leaves the sum of each segment in the first element of the segment. For example:

```
PRODUCTS                            = ( 5   1    3   4   3   9   2   6 )
FLAG                                = ( T   F    T   F   F   F   T   F )
SUM_SUFFIX(PRODUCTS, SEGMENT=
       PARITY_PREFIX(FLAG))         = ( 6   1   19  16  12   9   8   6 )
```

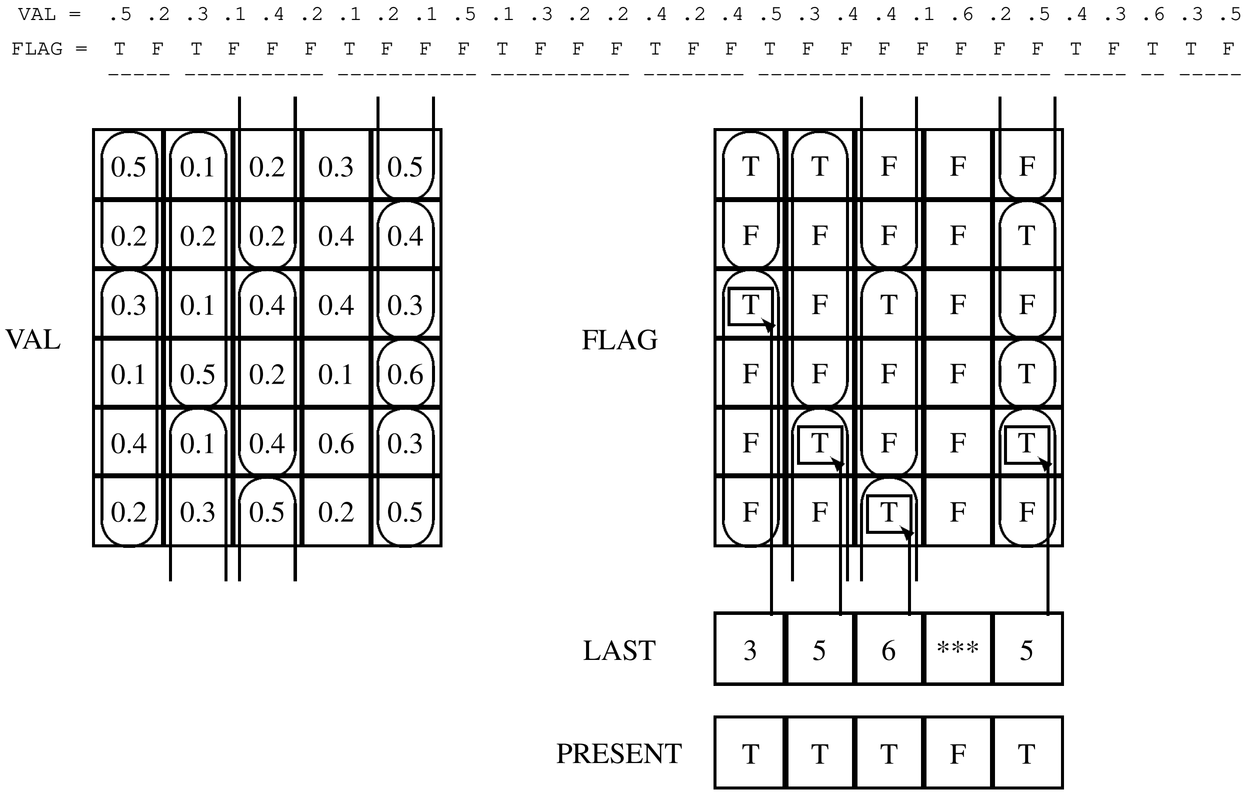
Once the sums are left in the first element of each segments, these are gathered using the PNTR array.

Using an implementation based on segmented scans gives reasonable running times on vector multiprocessors and massively parallel machines [36]. Our experiments show that on the Cray Y-MP C90, an implementation of this algorithm based on optimized segmented scans [11] is between 1.2 and 2.0 times slower than the Jagged Diagonal method (JAD). Because of the simplicity of the code, the fact that we can keep the data in CSR format rather than needing to permute it, and the much lower cost of setup, this might warrant using the scan-based method in many cases. However, as we describe below, we can modify the above algorithm such that it is almost always as fast or faster than JAD but still maintains these advantages.

The scan-based algorithm is slower than JAD mainly because it performs unnecessary work: the scan generates all the partial sums within each segment, but only the final sum at the beginning of the segment is used. It turns out that both the parallel and vector scan algorithms must pass over the data twice and execute a total of $2(n-1)$ additions instead of the $n-1$ required by a serial algorithm. However, if only the sum of each segment is needed, only one pass over the data is required, executing a total of only n additions. To do this we need to generate some additional information in a setup phase and pass this extra data to our segmented summing operation. Based on this method we introduce two new functions:

- SETUP_SEG(PNTR,M,N) generates a *segment descriptor* from the PNTR array of length M and the total number of non-zero elements N. It returns an array of integers; the format of this array will be described in the next section.

[2]HPF defines segments using an "edge-triggered" representation which marks segment boundaries by transitions from TRUE to FALSE or from FALSE to TRUE in the FLAG array. The conversion to HPF segments is done with the PARITY_PREFIX intrinsic, which executes a scan using exclusive-or as the binary operator.



(a) **(b)**

Figure 2: Figure **(a)** shows the values in the segmented vector VAL in two-dimensional form with $l = 5$ columns and $s = 6$ rows. Figure **(b)** shows how the elements of the FLAG array are TRUE at the beginning of a segment and FALSE elsewhere. Each column has a logical variable PRESENT which indicates whether any segments start within the column and an index LAST pointing to the last TRUE flag in the column.

- `SEG_SUM(VAL,PNTR,SEG)` sums each segment of the array `VAL` and returns an array of length `M` with one sum per segment stored in contiguous locations. The array `SEG` describes the segmentation of `VAL`.

Using these operations, the setup phase of a sparse matrix multiplication is simply implemented as

```
SEGDES = SETUP_SEG(PNTR,N)
```

and the sparse matrix multiplication can be coded as:

```
PRODUCTS = VAL * X(INDX)
Y = SEG_SUM(PRODUCTS,PNTR,SEGDES)
```

Although in this paper we will only use `SEG_SUM` for sparse matrix-vector multiplication, it has many other applications, and on most machines is likely to run twice as fast as a `PREFIX` or `SUFFIX` operation. We now discuss how the `SETUP_SEG` and `SEG_SUM` functions can be implemented.

3.3 Segmented Sum Algorithm

The intuition behind our segmented summing algorithm is that we can break a segmented vector into equally sized blocks, independent of where segment boundaries are located. Then, working on each block simultaneously, we can find the sum of each segment that is completely contained within a block, or the

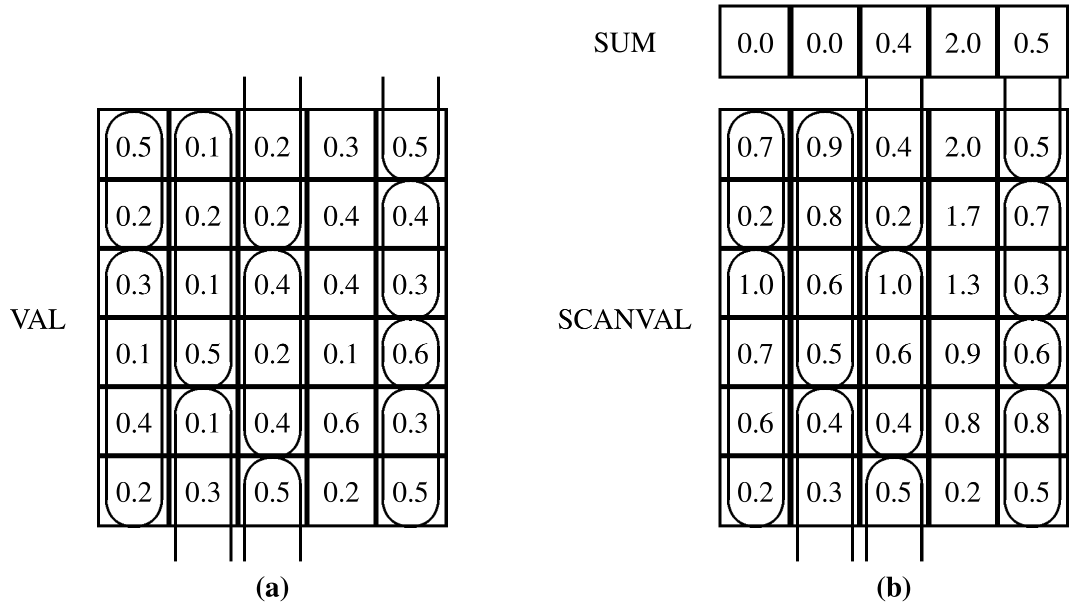


Figure 3: Figure **(a)** shows the representation of VAL with $l = 5$ columns and $s = 6$ rows. Figure **(b)** shows the result of performing a backwards segmented scan in each column. Notice that for segments contained entirely within a column the value at the beginning of each segment holds the sum of the segment.

partial sums of each segment that spans multiple blocks. Finally in a post-processing step, we can add the partial sums for each piece of the segment that span multiple blocks.

We will first present an algorithm for a single vector processor. Section 3.5 extends this algorithm to multiple vector processors. In order to express this algorithm in a vectorizable form, we will treat one-dimensional arrays as two-dimensional arrays, where the notion of "block" corresponds to a column in the two-dimensional representation. Specifically, we will treat the VAL and FLAG arrays of size n as two-dimensional arrays (in column-major order) with l columns and $s \approx n/l$ rows. (This conceptual layout is used to simplify the presentation and does not require any data motion.) The parameter l is chosen to be approximately equal to the hardware vector length so that we can express the algorithm in terms of vectorizable row operations. Figure 2 shows an example of a segmented vector in this form. Notice that some segments, which we will call "spanning segments", span multiple columns.

To implement `SETUP_SEG` we need to pre-compute some information about the structure of the segments. The preprocessing consists of the following (See Figure 2(**b**)):

1. As with the setup for segmented scans, set the FLAG elements to TRUE at the beginning of each segment, and FALSE elsewhere.

2. Determine for each column whether any segments start in that column and place the result in the logical array PRESENT(1:l). Place the row index of the last segment beginning in each column (if any) in LAST(1:l).

The result of `SEG_SUM` is a segment descriptor containing the LAST, PRESENT, and FLAG arrays. (In order to save memory, the flags can be represented using packed bit fields stored in an integer array.) Once we have this information, the `SEG_SUM` algorithm works in three stages:

1. Perform a backwards segmented scan within each column. Place the sum of the first partial segment in each column into SUM. (See Figure 3.) For all segments that are contained within a single column,

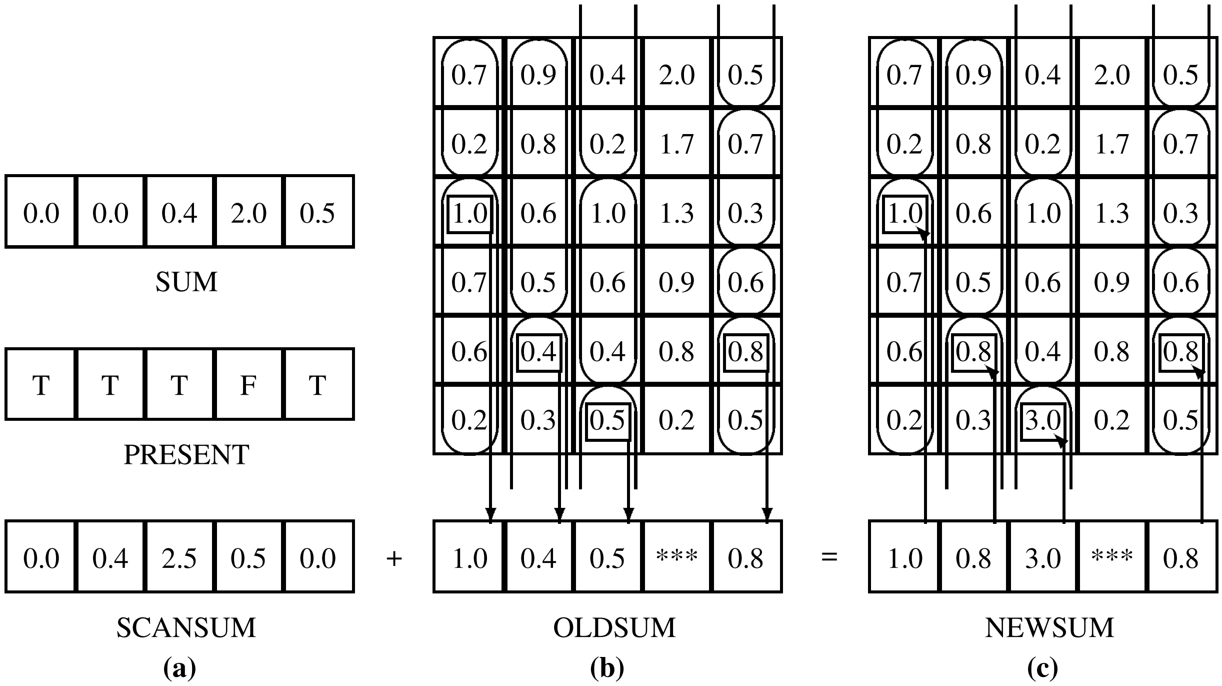


Figure 4: In Figure **(a)** the SCANSUM array is computed by performing a backwards segmented scan on SUM using the PRESENT flags as segment boundaries. Figures **(b)** and **(c)** show how in each column, the first value in the last segment is incremented by SCANSUM to produce the sum of the segment.

the sum of the segment will be in the first element of the segment.

2. Find the sum of spanning segments by adding the sum of each piece (See Figure 4):

   (a) Perform a backwards segmented scan on SUM into SCANSUM in order to compute the adjustment that must be made to each spanning segment.

   (b) For each spanning segment, increment the first element in the segment by the value of SCANSUM in its column.

3. Gather the sum of each segment from its first element.

3.4 Segmented Sum Implementation

We now consider the segmented summing algorithm in more detail. The description is made concrete by showing Fortran code fragments. In the following code, VAL is the source array, SCANVAL is a temporary array of the same size as the VAL array[3], N is the number of elements in VAL, and M is the number of segments. L and S are the number of columns and rows respectively in the two-dimensional representation of the arrays of length N.

[3]Section 4.4 describes how the temporary memory requirements can be greatly reduced by using a form of stripmiming.

Setup: The flag array is initialized such that flags are TRUE only at the beginning of each segment (as described in Section 3.1). The LAST pointers and the PRESENT flags can be computed from the FLAG array as follows:

```
PRESENT(1:L) = .FALSE.
DO I = 1,S
  DO J = 1,L
     IF (FLAG(I,J)) THEN
       LAST(J) = I
       PRESENT(J) = .TRUE.
     ENDIF
  END DO
END DO
```

The above code resets the value of LAST and PRESENT whenever a TRUE flag is encountered. The innermost loop is vectorized by converting the conditional statement into vector merge instructions that select one of two operands based on a vector of flags.

Column Scan: The code for segmented-sum performs a backwards segmented scan of each column of VAL into a temporary array of length N called SCANVAL, as shown in Figure 3. The first loop initializes the last row of SCANVAL by copying it from VAL.

```
DO J = 1,L
  SCANVAL(S,J) = VAL(S,J)
END DO
```

We now want to accumulate the sum in each column, resetting the sum at segment boundaries. This operation is the most important part of the SEGMV algorithm since it will consume almost all of the execution time for large matrices. The code can be written as:

```
DO I = S-1,1,-1
  DO J = 1,L
    IF (FLAG(I,J)) THEN
      SCANVAL(I,J) = VAL(I,J)
    ELSE
      SCANVAL(I,J) = VAL(I,J) + SCANVAL(I+1,J)
    ENDIF
  END DO
END DO
```

The structure of the column scan is very similar to the Ellpack/Itpack and Jagged Diagonal algorithms except that in our implementation the accumulation is conditional. This conditional statement vectorizes using vector merge instructions, and on many vector machines, the merge instructions chain with the multiplication and addition instructions.

Spanning segments: The most involved part of the SEGMV algorithm is the adjustment for spanning segments. We can find the sum of the first partial segment in each column (SUM) by taking the first value of each column or zero if the column begins with a segment boundary, as shown in Figure 3. To find the sum of a spanning segment we can not simply add the partial sums from its neighbors, because a segment can span many columns. Instead, the partial sums for spanning segments are combined by performing an operation on SUM similar to a backwards segmented scan. The partial sums are accumulated, resetting the accumulation for each new spanning segment (indicated by the PRESENT flags):

```
SCANSUM(L) = 0.0
DO J = L-1,1,-1
  IF (PRESENT(J+1)) THEN
    SCANSUM(J) = SUM(J+1)
  ELSE
    SCANSUM(J) = SUM(J+1) + SCANSUM(J+1)
  ENDIF
END DO
```

Each element of the result, `SCANSUM(J)`, contains the contribution of values in higher columns to the last segment that begins in column j. Then, for each column containing a segment boundary, the first value of the last segment in that column is incremented by SCANSUM, as shown in Figure 4. This can be implemented as follows:

```
DO J = 1,L
  IF (PRESENT(J)) THEN
    SCANVAL(LAST(J),J) = SCANVAL(LAST(J),J) + SCANSUM(J)
  ENDIF
END DO
```

Result: Now that the first element of each segment contains the sum of the elements in that segment, the result is gathered from the first element of each segment using the PNTR array.

3.5 Multiprocessor Algorithm

The algorithm above has l-way parallelism in the columnwise scan because it operates on the rows of an $s = n/l$ by l matrix. In order to use p processors, the two-dimensional view of the data is changed to $s = n/(l \cdot p)$ rows by $l \cdot p$ columns. In parallel, each processor k can then process l columns starting at row $k \cdot l + 1$. In the pseudocode below showing the columnwise segmented scan, the iterations of the outer DOALL loop are run in parallel with one iteration assigned to each processor:

```
DOALL K = 0,P-1
  OFFSET = K * L
  DO I = S-1,1,-1
    DO J = OFFSET+1,OFFSET+L
      IF (FLAG(I,J)) THEN
        SCANVAL(I,J) = VAL(I,J)
      ELSE
        SCANVAL(I,J) = VAL(I,J) + SCANVAL(I+1,J)
      ENDIF
    END DO
  END DO
END DOALL
```

One way to handle the rowwise loops, such as the loops operating on the SCANSUM array, would be to execute $l \cdot p$ iterations on a single processor. However, the serial portions of the code can be minimized by having each processor adjust all spanning segments that are contained within its columns. Then because there are at most p spanning segments remaining to be processed, the time for rowwise loops is reduced from $l \cdot p$ steps to $l + p$ steps. It is possible to reduce this further to $l + \log_2 p$, but for small values of p, performing p steps on one processor is more practical. An alternative approach which eliminates serial loops entirely is to divide the matrix into p pieces such that no segment spans a processor, rather than using a optimally load-balanced algorithm. This technique is not as general but works well for most matrices used in practice. (Our current implementation does not use this technique.)

3.6 Sparse Matrix-Vector Multiplication

As discussed in Section 3.2, sparse matrix multiplication can be expressed in terms of a segmented sum primitive. However, we can build a direct implementation of sparse matrix multiplication by changing all uses of `VAL(I,J)` in the columnwise scan to `VAL(I,J) * X(INDX(I,J))`, which first multiplies the matrix element by the corresponding vector element. On many vector machines, a direct implementation of sparse matrix multiplication is more efficient because the additional load, gather, and multiply operations chain with other operations used in the segmented sum.

3.7 Implementation Details

We implemented SEGMV for the Cray Y-MP C90 in Cray Assembly Language [13] (CAL) using microtasking [14] from C to take advantage of multiple processors. In [11], we describe how to adjust l if the default value of s would cause bank conflicts on strided memory accesses, and how to accommodate vectors of arbitrary size, i.e., when n does not evenly divide l. To reduce the memory-bandwidth requirements, the FLAG array is represented using packed bit-masks (approximately 64 per word) stored in the form used by the vector mask register. Packed flags also reduce the additional storage requirements for the sparse matrix to approximately $n/64$ (plus a constant amount of storage for LAST and PRESENT). By using packed flags we are able to implement the conditional instruction in the columnwise scan at almost no additional cost, since loading the flags is virtually free, and the merge instruction chains with the multiplication and addition in the hardware pipeline.

We coded in assembly language to get maximal chaining and re-use of operands. However, to reduce the programming effort and enhance portability, we used a macro-assembler (written in LISP) with high-level support for register naming, loop unrolling, and handling packed masks. We have also implemented a version of SEGMV in Fortran (using packed flags) that performs about a factor of 2 slower than the CAL version for large matrices. The Fortran version was used for experimenting with extensions to the algorithm as described in Section 5.1.

4 Performance Analysis

In this section we present performance results for sparse matrix multiplication. The timings were run on a Cray Y-MP C90 in dedicated mode using sample problems from the Harwell-Boeing test set [17] and industrial application codes. We also present a performance model for the SEGMV implementation that accurately predicts its running time.

4.1 Test Suite

Table 1 describes the test problems we use, giving the dimension, the number of entries, average row size, maximum row size, and a brief description of each problem. For those matrices which are symmetric (indicated by *Sym* column), the matrix statistics describe the full nonsymmetric representation. Figure 5 shows the distribution of problem sizes and row sizes for the test suite. We have chosen a representative collection of matrices from several important application areas. Most of them are from the standard Harwell-Boeing collection and will facilitate comparison of our results with others. However, we have also included

Problem	*Entries* (n)	*Dim.* (m)	*Ave Row*	*Max Row*	*Sym.*	*Description*
Structures Problems (Finite Element)						
BCSSTK15	117816	3948	29.8	44	$\surd$	Stiffness matrix — Module of an Off-shore Platform
BCSSTK30	2043492	28924	70.7	219	$\surd$	Stiffness Matrix for Off-shore Generator Platform (MSC NASTRAN)
NASASRB	2677324	54870	48.8	276	$\surd$	Structure from NASA Langley, Shuttle Rocket Booster
Standard Finite Difference Operators						
GR-30x30	7744	900	8.6	9	$\surd$	Symmetric matrix from nine point start on a 30 by 30 grid
CUBE12	982600	8640	113.7	135	$\surd$	Symmetric pattern, 27 point operator on 12 by 12 by 12 grid
Oil Reservoir Problem						
SHERMAN2	23094	1080	21.4	34		Thermal Simulation, Steam injection, 6 by 6 by 5 grid, five unknowns
Computational Fluid Dynamics Problems						
BBMAT	1771722	38744	45.7	132		Beam + Bailey 2D Airfoil Exact Jacobian: mach = 0.08, angle = 2.0
3DEXT1	9045076	118564	76.3	108		3D Fully-coupled Incompressible NS Polymer Extrusion Problem
Chemical Process Simulation Problem						
FIFE1Q	392188	47640	8.2	44085		Chemical Process Simulation Test Matrix (Permuted)

Table 1: Statistics for the test matrices.

several large nonsymmetric matrices because the Harwell-Boeing collection does not have a very large set of nonsymmetric matrices.

4.2 Results for Matrix Multiplication

Table 2 and Figure 6 report timing information comparing our implementation to several other implementations on the set of test matrices. All results were obtained on a Cray Y-MP C90 (16 processors, 256Mw memory, 4.167 ns clock, S/N 4001) in dedicated mode. The JAD implementation is a highly-optimized Cray Research library routine implemented in Cray Assembly Language (CAL). The ELL, CSR, and CSC implementations are based on optimized Fortran kernels. Our SEGMV routine is coded in CAL. In all cases, even if the matrix is symmetric, all non-zero elements are stored. (Symmetric storage is not used in this study because efficient methods have not been developed for processing symmetric matrices with the Ellpack/Itpack, Jagged Diagonal, or SEGMV data structures.) The Flop rate is computed based on the running time t for one sparse matrix-vector multiplication (not including the setup computation), the number of equations m, and the total number of non-zero elements n. Matrix-vector multiplication requires n multiplications and $n - m$ additions, yielding the equation Flops $= (2n - m)/t$.

On one processor, SEGMV outperforms all other implementations on large test matrices, as shown in Figure 6. On multiple processors, SEGMV outperforms the other implementations by a greater margin,

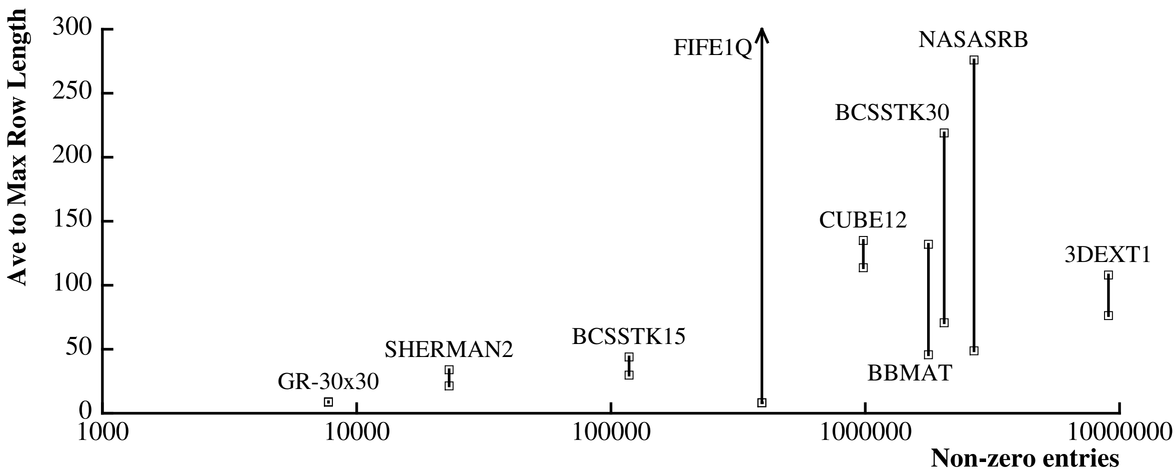


Figure 5: Statistics for the test matrices. Each test matrix is represented by a vertical line. The horizontal position of the line indicates the number of non-zero entries in the matrix. The bottom point indicates the average row length and the top point indicates the maximum row length. (The FIFE1Q matrix has a maximum row length of 44085 which is represented as an arrow.) The test suite includes a wide variety of row sizes and matrix sizes (ranging from under 10^4 to nearly 10^7).

	1 CPU					*4 CPU*					*16 CPU*				
Matrix	CSR	CSC	ELL	JAD	SEG	CSR	CSC	ELL	JAD	SEG	CSR	CSC	ELL	JAD	SEG
GR-30x30	15	26	333	336	278	48	86	784	784	471	163	141	927	900	205
SHERMAN2	34	65	117	248	288	121	216	386	501	795	419	364	1017	1314	488
BCSSTK15	45	84	205	312	304	164	293	746	893	1056	617	551	2303	2792	2479
FIFE1Q	14	25		165	249	50	88		284	577	199	164		436	611
CUBE12	126	182	153	163	322	476	645	579	540	1166	1816	1715	2042	1954	3758
BBMAT	63	118	103	255	311	235	422	396	389	1115	932	910	1270	1191	3639
BCSSTK30	86	157	90	168	312	323	561	335	446	1162	1246	1260	1157	1411	3749
NASASRB	67	123	65	224	311	249	444	241	739	1192	986	907	789	2117	3740
3DEXT1	92	165	187	244	314	345	590	709	616	1165	1357	1297	2433	2260	3797

Table 2: Performance results (in Mflops) comparing five implementations of sparse matrix multiplication: Compressed Sparse Row (CSR), Compressed Sparse Column (CSC), Ellpack/Itpack (ELL), Jagged Diagonal (JAD), and SEGMV (SEG). The test matrices are listed in order of increasing number of non-zero entries. Figure 6 shows this data using bar charts. (On the FIFE1Q matrix, no results are given for ELL because the data structure would require over 2 Gwords.)

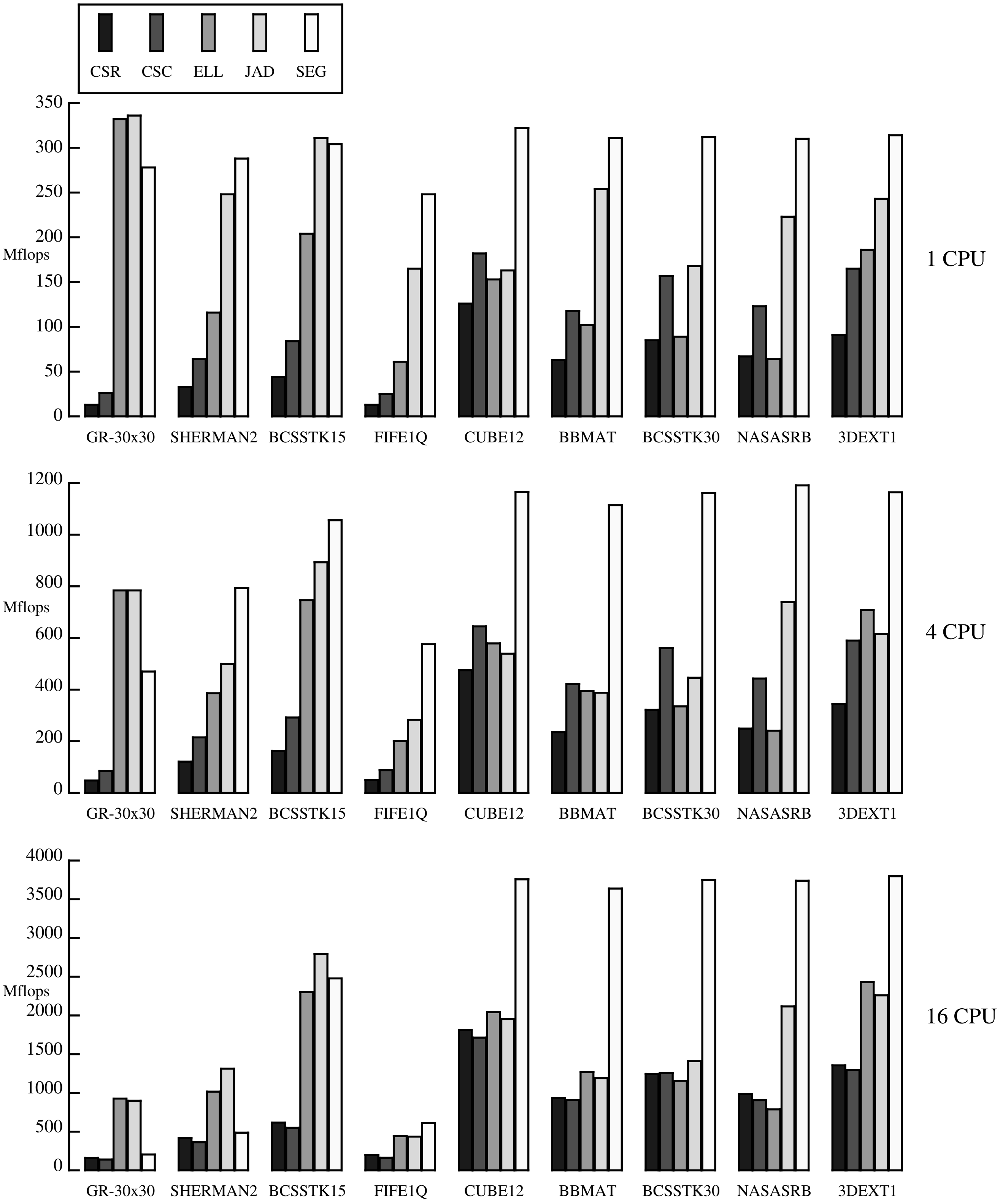


Figure 6: Performance results on a Cray Y-MP C90 comparing implementations of sparse matrix multiplication. The test matrices are shown in order of increasing number of non-zero entries. The SEGMV implementation outperforms all other implementations on the large test matrices.

CPU's	Mflops
1	326
4	1188
8	2140
16	3493

Table 3: Performance on the NAS Conjugate Gradient benchmark.

with an absolute performance on 16 processors of over 3.7 Gflops. Table 3 reports the performance of our implementation of the NAS Conjugate Gradient benchmark using our CAL implementation of SEGMV.

4.3 Performance Model

One advantage of the SEGMV algorithm is that its running time algorithm is very predictable. This section presents a performance model for the SEGMV algorithm with equations broken down into components for columnwise loops over the n matrix values, loops over each of the m segments, and loops over each of the p processors. All constants were measured on the C90 and are expressed in 4.167 nsec clock periods:

The setup time is given by the following equation:

$$t_{\text{setup}} = \frac{t_f \cdot n + t_s \cdot m}{p} + t_o$$

where the constants t_f, t_s, and t_o are defined as follows:

t_f = time per element for columnwise loops (to clear flags and find PRESENT and LAST) $\approx$ 1.7
t_s = time per element to mark segment starts $\approx$ 1.3
t_o = constant overhead $\approx$ 1000

The time to perform one matrix-vector multiplication is given by the following equation:

$$t_{\text{mvmult}} = \frac{t_c \cdot n + t_g \cdot m}{p} + t_p \cdot p + t_o$$

where the constants are defined as follows:

t_c = time per element for columnwise segmented scan loop $\approx$ 1.5
t_g = time per element to gather result from SCANVAL to Y$\approx$ 1.5
t_p = time per element for across-processor serial scan loop $\approx$ 110

These equations can accurately predict the running time as shown in Table 4.

4.4 Analysis of Results

Running Time: The performance of previous methods for sparse matrix multiplication can vary greatly with the structure of the matrix as shown by results on the test suite. To explain some of the underlying reasons for this variation, we have generated two sets of test matrices: one set for measuring the dependence on average row size and one set for measuring the dependence on maximum row size. Figure 7(a) measures the performance on matrices that contain the same number of elements in each row. As expected, CSR is very sensitive to row size because the row size determines the hardware vector length. On the other hand,

Matrix	Measured	Predicted	Error
GR-30x30	278	264	5.0%
SHERMAN2	288	295	-2.4%
BCSSTK15	304	305	-0.3%
FIFE1Q	249	283	-13.7%
CUBE12	322	314	2.5%
BBMAT	311	311	0.0%
BCSSTK30	312	313	-0.3%
NASASRB	311	311	0.0%
3DEXT1	314	313	0.3%

Table 4: Predicted and measured performance for SEGMV (MFlops on 1 CPU), using the equation: $t = 1.5n + 1.5m + 1000$. The Error column indicates that the SEGMV performance can be accurately predicted from n and m without additional information about the matrix structure.

ELL, JAD, and SEG all perform equally well independent of the row size, because their computational structure is nearly identical for matrices with a fixed row size. Figure 7(b) measures the performance on matrices where all rows are the same length except for one long row. The performance of CSR is insensitive to maximum row length. The performance with the Ellpack/Itpack data structure degrades severely as the maximum row size is increased, because all rows must be padded with zeros to the maximum row size. The JAD data structure shows a slight dependence on the maximum row size. Notice that the SEGMV performance is independent of the maximum row size and nearly independent of the average row size.

The effects of matrix structure can be seen from the performance results on the test suite. The CSR and CSC implementations do fairly well on matrices with long rows and columns (such as CUBE12) and poorly on matrices with short rows and columns (such as GR-30x30). The ELL and JAD implementations do quite well on matrices that have a maximum row length near the average row length (such as GR-30x30 and BCSSTK15). One of the most interesting performance results is that ELL, JAD, and SEG all do poorly on the FIFE1Q matrix. This matrix contains one very dense row and column which is common in some application areas, such as chemical process simulation. Because the dense column accounts for over 10% of the non-zero matrix elements, a substantial fraction of all memory fetches are made to the same element of the input vector. These references all contend for access to a single bank of the C90's 1024-bank interleaved memory system [29]. Bank conflicts on the gather operations cause a noticeable slowdown on one processor, and as the number of processors is increased, the memory system becomes a serial bottleneck. Memory conflicts can be avoided by replicating frequently used vector elements and modifying the INDX vector to distribute the references among the copies. Replication can be done by the user if the matrix structure is known in advance or possibly by automatic feature extraction techniques.

Setup Time: Execution time for matrix-vector multiplication is not the only important consideration. Large setup times can be prohibitive for certain applications, e.g. adaptive mesh algorithms. As shown in Table 5, the setup time for Jagged Diagonal is significantly larger than the setup time for SEGMV. The setup time for SEGMV is approximately the same as a single matrix-vector multiplication.

Memory Usage: The CSR and CSC implementations do not require scratch space. In practice, the JAD and ELL data structures consume additional memory when the user must maintain the original matrix in addition to the permuted or padded version. The SEGMV data structure requires only approximately $n/64$

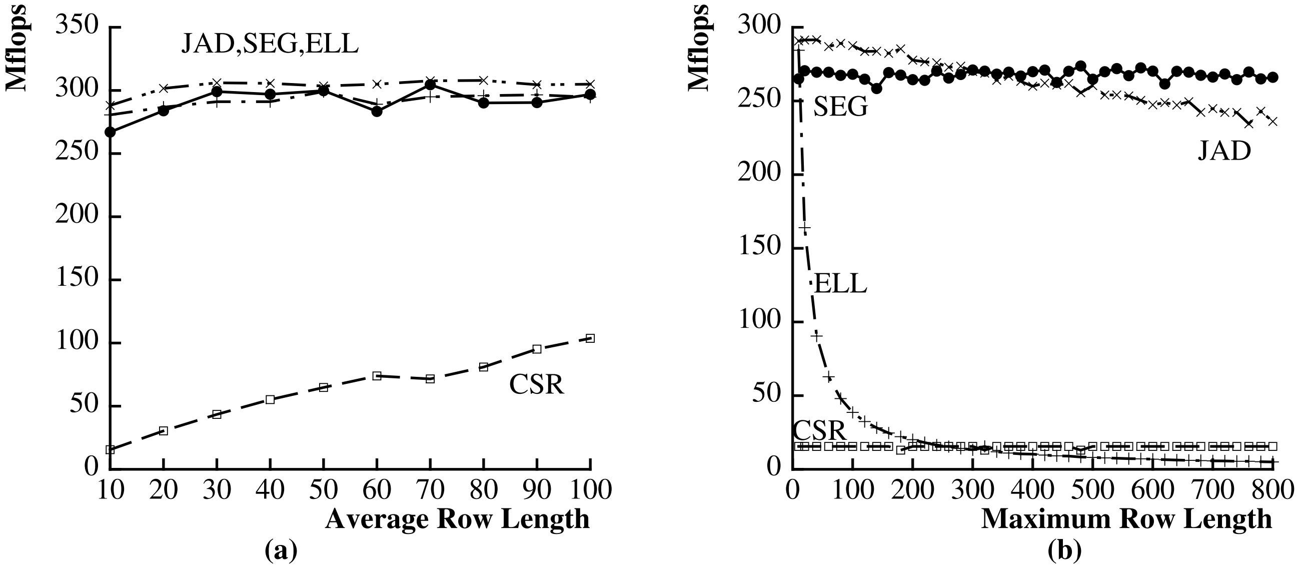


Figure 7: **(a)** Performance as the average row size is varied. Each row contains the same number of non-zero entries, and the total number of non-zero entries is fixed at 10^5. Column indices are generated randomly. The CSR performance varies with row length, while the other formats are independent of row length when all rows are the same length. **(b)** Performance as the maximum row size is varied. The number of non-zero entries is fixed at 10^5 and the average row size is fixed at 10. Column indices are generated randomly. As the maximum row size is increased, the ELL performance degrades quickly and the JAD performance degrades slightly.

words for the data structure (in addition to the space for the CSR data structure). The SEGMV algorithm requires n scratch space for the SCANVAL temporary array, but we can greatly reduce this requirement by breaking the matrix into a number of strips and processing one strip at a time. This stripmining, which can be hidden from the user, reduces the scratch space required from the size of the matrix to the size of a single strip. The strip size should be chosen to balance the memory consumption with the startup time incurred for processing each strip. For the large matrices in which scratch space is an issue, the stripmining would not significantly affect performance.

5 Generalizations

In this section, we consider generalizations of the SEGMV algorithm. Two generalizations we consider, block entry matrices and multiple right-hand sides, can be implemented in terms of sparse matrix-vector multiplication, but offer the potential for improved performance if implemented directly. We also consider the application of other segmented operations to sparse matrix computation. Finally we outline a column-oriented algorithm which is useful for symmetric matrices.

5.1 Block Entry Matrices

Simulating systems with several unknowns per gridpoint can lead to a matrix with block structure. These matrices can be stored in a "block entry" format in which each logical entry of the matrix is a k by k block [20]. We have implemented a version of sparse matrix-vector multiplication for block entry matrices, and our preliminary results are shown in Table 6. Our preliminary version, which is implemented in

Matrix	1 CPU JAD	1 CPU SEG	1 CPU ratio	4 CPU JAD	4 CPU SEG	4 CPU ratio
GR-30x30	3	0.1	23	3	0.2	12
SHERMAN2	4	0.2	16	4	0.3	13
BCSSTK15	14	0.9	15	14	0.4	31
FIFE1Q	124	3.1	41	124	1.0	127
CUBE12	50	7.0	7	50	2.0	25
BBMAT	161	13.0	12	158	3.4	46
BCSSTK30	141	14.5	10	141	3.9	36
NASASRB	219	19.1	11	219	5.0	44
3DEXT1	606	65.0	9	560	16.7	34

Table 5: Setup Times (in msec) for Jagged Diagonal and SEGMV. The setup times for SEGMV are an order of magnitude faster on one processor. On multiple processors this performance difference is amplified because the SEGMV implementation achieves near linear speedup for large matrices while the Jagged Diagonal setup does not take advantage of multiple processors. (The JAD setup is parallelizable although achieving linear speedup might be difficult.)

Fortran, is between 1.6 and 2.2 times faster than the Fortran point entry version. The block version performs significantly better because of reduced memory bandwidth requirements; the INDX vector is only accessed once per block. Currently, our CAL implementation does not take advantage of blocking, but we would expect a blocked CAL implementation to perform between 1.3 and 2.0 times faster than the point entry implementation.

Matrix	Blocksize	Mflops
CUBE12	1×1	160
CUBE12	2×2	263
CUBE12	4×4	288
GR-30x30	1×1	111
GR-30x30	2×2	199
GR-30x30	4×4	249

Table 6: Performance on 1 CPU for our block entry implementation in Fortran.

5.2 Multiple Right-hand-sides

As with block entry matrices, memory bandwidth requirements are reduced for multiplying a sparse matrix by a dense matrix, i.e. for handling multiple right-hand-sides. In addition to only accessing INDX once per block, the matrix values (VAL) only need to be loaded once per block, allowing further speedup.

5.3 Other Segmented Operations

In addition to segmented summing, several other segmented operations are useful for sparse matrix computation. A segmented *permute* operation can be used to re-order the elements in each row. A segmented *pack* operation can be used to create a new sparse matrix consisting of a subset of elements from another sparse matrix. A segmented *copy* operation can be used to distribute a different value to the elements of each row. These operations have been efficiently implemented for a variety of machines [6, 7, 10].

5.4 CSC SEGMV and Symmetric Matrices

Segmented vector operations can also be used to implement a column-oriented version of sparse matrix multiplication. This could be used along with a row-oriented version to process symmetric matrices directly, rather than expanding them into a full non-symmetric representation.

The column-oriented version works by spreading the value of x_j along the jth column of the matrix and then adding the product $A_{ij} \cdot x_j$ to y_i. Starting from a Compressed Sparse Column (CSC) representation, an HPF implementation of CSC sparse matrix multiplication is similar to the CSR version, except that we use a segmented *copy* operation instead of a segmented summing operation, and a combining-scatter instead of a gather. A segmented copy spreads the first value of each segment to all other values in the segment. A combining scatter operation sends each element to a specified index and combines all elements scattered to the same destination using an associative operator such as addition. In HPF, the setup code is identical to the CSR version and the matrix vector multiplication can be coded as follows:

```
SPREADX(PNTR) = X
SPREADX = COPY_PREFIX(SPREADX,SEGMENT=PARITY_PREFIX(FLAG))
PRODUCTS = VAL * SPREADX
Y = 0
Y = SUM_SCATTER(PRODUCTS,Y,INDX)
```

As with the segmented summing operation, the segmented copy operation can be implemented with only one pass over the data and an adjustment for spanning segments that takes constant time. (However, for segmented copy, the adjustment is made before the columnwise scan.) Our implementations of segmented sum and segmented copy run at approximately the same speed. The main difficulty with using a column-oriented version is that designing an efficient algorithm for `SUM_SCATTER` is an open research question, particularly for using multiple processors.

6 Conclusions

This paper has presented the SEGMV algorithm for sparse matrix multiplication on vector multiprocessors. Because our algorithm is based on segmented operations rather than heuristic preprocessing techniques, our method has several advantages over previous methods. SEGMV requires minimal preprocessing, uses a convenient data structure (an augmented form of the standard CSR data structure), fully vectorizes and parallelizes for any matrix structure (without performing unnecessary computation), and has predictable performance. On the Cray Y-MP C90, our implementation outperforms all previous approaches on large sparse matrices.

References

[1] R. C. Agarwal, F. G. Gustavson, and M. Zubair. A high performance algorithm using pre-processing

for the sparse matrix-vector multiplication. In *Proceedings Supercomputing '92*, pages 32–41, Nov. 1992.

[2] Edward Anderson and Youcef Saad. Solving sparse triangular systems on parallel computers. *International Journal of High Speed Computing*, 1(1):73–95, 1989.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weernatunga. The NAS parallel benchmarks. *International Journal of Supercomputer Applications*, 5(3):63–73, 1991.

[4] D. H. Bailey, E. Barszcz, L. Dagum, H. D. Simon, NAS Parallel Benchmark Results. In *Proceedings Supercomputing '92*, pages 386–393, Nov. 1992.

[5] Guy E. Blelloch. Scans as Primitive Parallel Operations. *IEEE Transactions on Computers*, C-38(11):1526–1538, November 1989.

[6] G. E. Blelloch. *Vector Models for Data-Parallel Computing*. The MIT Press, 1990.

[7] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, M. Reid-Miller, J. Sipelstein, and M. Zagha. CVL: A C vector library. Technical Report CMU-CS-93-114, School of Computer Science, Carnegie Mellon University, Feb. 1993.

[8] G. E. Blelloch, S. Chatterjee, and M. Zagha. Solving Linear Recurrences with Loop Raking. In *Proceedings Sixth International Parallel Processing Symposium* pages 416–424, March, 1992.

[9] G. E. Blelloch and C. R. Rosenberg. Network learning on the Connection Machine. In *Proceedings of the AAAI Spring Symposium Series: Parallel Models of Intelligence*, pages 355–362, August 1987.

[10] S. Chatterjee. *Compiling Data-Parallel Programs for Efficient Execution on Shared-Memory Multiprocessors*. PhD thesis, School of Computer Science, Carnegie Mellon University, Oct. 1991.

[11] S. Chatterjee, G. E. Blelloch, and M. Zagha. Scan primitives for vector computers. In *Proceedings Supercomputing '90*, pages 666–675, Nov. 1990.

[12] L. T. Chen, L. S. Davis, and C. P. Kruskal. Efficient parallel processing of image contours. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(1):69–81, January 1993.

[13] Cray Research Inc., Mendota Heights, Minnesota. *Symbolic Machine Instructions Reference Manual, SR-0085B*, March 1988.

[14] Cray Research Inc., Mendota Heights, Minnesota. *CRAY Y-MP, CRAY X-MP EA, and CRAY X-MP Multitasking Programmer's Manual*, July 1989.

[15] F. Crow, G. Demos, J. Hardy, J. Mclaughlin, and K. Sims. 3D image synthesis on the Connection Machine. *International Journal of High Speed Computing*, 1(2):329–347, 1989.

[16] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Monographs on Numerical Analysis. Oxford University Press, New York, 1986.

[17] I. S. Duff, R. G. Grimes, and J. G. Lewis. Sparse matrix test problems. ACM Transactions on Mathematical Software, 15(1):1–14, March 1989.

[18] Jocelyne Erhel. Sparse matrix multiplication on vector computers. *International Journal of High Speed Computing*, 2(2):101–116, 1990.

[19] P. Fernandes and P. Girdinio. A new storage scheme for an efficient implementation of the sparse matrix-vector product. *Parallel Computing*, 12:327–333, 1989.

[20] Michael A. Heroux. A proposal for a sparse BLAS toolkit. Technical Report TR/PA/92/90, CERFACS, December 1992.

[21] High Performance Fortran Forum. *High Performance Fortran Language Specification*, Version 1.0 Draft, January 1993.

[22] David R. Kincaid and Thomas C. Oppe. Recent vectorization and parallelization of ITPACKV. Technical report, Center for Numerical Analysis, The University of Texas at Austin, November 1989.

[23] David R. Kincaid, Thomas C. Oppe, John R. Respess, and David M. Young. ITPACKV 2C user's guide. Technical Report CNA-191, Center for Numerical Analysis, The University of Texas at Austin, November 1984.

[24] Peter M. Kogge and Harold S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, C-22(8):786–793, August 1973.

[25] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the Association for Computing Machinery*, 27(4):831–838, October 1980.

[26] C. Leiserson, Z. S. Abuhamdeh, D. Douglas, C. R. Feynmann, M. Ganmukhi, J. Hill, W. D. Hillis, B. Kuszmaul, M. St. Pierre, D. Wells, M. Wong, S-W Yang, and R. Zak. The Network Architecture of the Connection Machine CM-5, In *Proceedings Symposium on Parallel Algorithms and Architectures*, July, 1992.

[27] Rami Melhem. Parallel solution of linear systems with striped sparse matrices. *Parallel Computing*, 6:165–184, 1988.

[28] S. S. Nielsen and S. A. Zenios. Data structures for network algorithms on massively parallel architectures. *Parallel Computing*, 18:1033–1052, 1992.

[29] Wilfried Oed. Cray Y-MP C90: System features and early benchmark results. *Parallel Computing*, 18:947–954, 1992.

[30] Thomas C. Oppe, Wayne D. Joubert, and David R. Kincaid. *NSPCG User's Guide*. Center for Numerical Analysis, The University of Texas at Austin, December 1988.

[31] Gaia Valeria Paolini and Giuseppe Radicati Di Brozolo. Data structures to vectorize CG algorithms for general sparsity patterns. *BIT*, 29:703–718, 1989.

[32] Alexander Peters. Sparse matrix vector multiplication techniques on the IBM 3090 VF. *Parallel Computing*, 17:1409–1424, 1991.

[33] Youcef Saad. SPARSKIT: a basic tool kit for sparse matrix computations. Preliminary Version.

[34] Thomas J. Sheffler. Implementing the multiprefix operation on parallel and vector computers. Technical Report CMU-CS-92-173, School of Computer Science, Carnegie Mellon University, August 1992.

[35] Valerie E. Taylor, Abhiram Ranade, and David G. Messerschmitt. Three-dimensional finite-element analyses: implications for computer architectures. In *Proceedings of Supercomputing '91*, pages 786–795, Nov. 1991.

[36] Thinking Machines Corporation, Connection Machine Scientific Software Library (CMSSL) for Fortran Version 3.1, 1993.

[37] L. W. Tucker, C. R. Feynman, and D. M. Fritzsche. Object recognition using the Connection Machine. In *Proceedings CVPR '88: the computer society conference on computer vision and pattern recognition*, pages 871–878, June 1988.