Computer Physics Communications 70 (1992) 557–568
North-Holland

Computer Physics
Communications

Fast sparse matrix multiplication

S.C. Park, J.P. Draayer [1]

Department of Computer Science and Department of Physics and Astronomy, Louisiana State University, Baton Rouge, LA 70803, USA

and

S.-Q. Zheng

Department of Computer Science, Louisiana State University, Baton Rouge, LA 70803, USA

Received 8 October 1991

A new space-efficient representation for sparse matrices is introduced and a fast sparse matrix multiplication algorithm based on the new representation is presented. The scheme is very efficient when the nonzero elements of a sparse matrix are partially or fully adjacent to one another as in band or triangular matrices. The space complexity of the new representation is better than that of existing algorithms when the number of sets of adjacent nonzero elements, called segments, is less than two thirds of the total number of nonzero elements. The time complexity of the associated sparse matrix multiplication algorithm is also better or even much better than that of existing schemes depending on the number of segments in the factor matrices.

PROGRAM SUMMARY

Title of program: SMM

Catalogue number: ACHR

Program obtainable from: CPC Program Library, Queen's University of Belfast, N. Ireland (see application form in this issue)

Licensing provisions: none

Computer: IBM 3090/600E

Operating system: MVS/XA (Version 3, Release 13)

Programming language used: FORTRAN

Correspondence to: J.P. Draayer, Dept. of Physics and Astronomy, Louisiana State University, Baton Rouge, LA 70803-4001, USA.

[1] Funded in part by the US National Science Foundation, Grant No. PHY89-22550.

Memory required to execute with typical data: The five routines that comprise the sparse matrix multiplication package (SMM) require a total of 11600 bytes of memory space on an IBM 3090 mainframe computer. The main program and array storage requirements are over and above this base amount.

Peripherals used: none

No. of lines in distributed program, including test data, etc.: 599 (161 in the sample DRIVER, 438 in the SMM package)

Keywords: sparse matrix, data structures, matrix multiplication, matrix transpose

Nature of physical problem
Sparse matrix multiplication [1–3] often arises in scientific computations. Since a sparse matrix includes many zero elements, the multiplication should not be handled in the same way as for dense matrices. The standard matrix multiplication algorithm for $n \times n$ factor matrices, represented in the usual two-dimensional array form, takes $\mathscr{O}(n^3)$ time [3]. This means that when the factor matrices are very large, e.g.

1000×1000, not only will the computation time be excessively long but the demands on storage can strain even the biggest of modern computers. Developing efficient data structures and algorithms for the multiplication of sparse matrices is therefore very important.

The new data structure and algorithm for sparse matrices that is presented in this paper is more time and space efficient than the existing methods if the sparse matrices contain nonzero elements which are partially or fully adjacent to one another as in band or triangular matrices. Space complexity is better than that of the existing algorithms when the number of the groups of adjacent nonzero elements is less than two thirds of the total number of nonzero elements. Time complexity is better or much better than that of existing algorithms depending on the number of groups of nonzero adjacent elements in the factor matrices.

Method of solution
The sparse matrix multiplication problem is addressed by introducing a space-efficient data structure for representing the matrices and a multiplication algorithm based on the new representation that can be easily vectorized. The new structure represents a sparse matrix with two arrays, one that contains only the nonzero elements and another integer array that holds information on the storage of matrix segments, which are the sets of adjacent nonzero elements in a row. For the multiplication of two matrices, $A \times B$, the transpose B^{τ} is determined first and then the segments of A are compared to those of B^{τ} to calculate segment overlaps. Details on how this is done are presented in the text.

Restrictions on the complexity of the problem
There are no restrictions on the factor matrices in the product. However, depending on the size of the matrices, memory overflow could be a problem.

Typical running time
For 300×300 band matrices with a bandwidth of 31, the matrix multiplication operation takes 0.74 s on an IBM 3090/600E.

Unusual features of the program
The routines in the package are generic and require no modification except for possible changes in the implicit statements that specify the character of the factor matrices.

References
[1] J.K. Cullum and R.A. Willoughby, Lanczos Algorithms for Large Symmetric Eigenvalue Computations, Vol. 1 (Birkhauser, Boston, 1985).
[2] G.H. Golub and C.F. Van Loan, Matrix Computations, 2nd ed. (Johns Hopkins Univ. Press, Baltimore, 1989).
[3] E. Horowitz and S.S. Ahni, Fundamentals of Data Structures, (Computer Science Press, Rockville, MD, 1983).

LONG WRITE-UP

1. Introduction

Matrices are abstract mathematical objects that arise in many numerical as well as nonnumerical applications. Examples range from solutions of systems of simultaneous equations to representations of graphs. In large-scale, high-performance scientific and engineering computing applications, it is not unusual to encounter matrices with tens of thousands of elements. The underlying structure of these problems often dictates that the matrices encountered are sparse. Designing efficient algorithms for sparse matrix operations is therefore a problem of fundamental importance in computational science [1–4].

Consider the multiplication of an $m \times n$ matrix A and an $n \times p$ matrix B. If A and B are represented by two-dimensional arrays, the simplest method is as follows:

```
for $i = 1$ to $m$ do
    for $j = 1$ to $p$ do
        $C_{ij} = 0$
        for $k = 1$ to $n$ do
            $C_{ij} = C_{ij} + A_{ik} \times B_{kj}$
        endfor
    endfor
endfor
```

	col 1	col 2	col 3	col 4	col 5	col 6	col 7
row 1	2	0	0	0	0	0	0
row 2	3	4	0	0	0	0	0
row 3	0	0	0	5	0	0	0
row 4	0	0	0	6	0	0	0
row 5	0	1	0	8	4	3	0
row 6	0	0	0	0	2	2	1
row 7	0	0	0	0	5	0	0

Fig. 1. Two-dimensional array representation of a 7×7 matrix. The standard two-dimensional array representation of a matrix uses fixed storage since zero and nonzero elements are handled the same.

Obviously this algorithm takes $\mathcal{O}(n^3)$ time if $m = n = p$. This procedure will be called the *standard matrix multiplication algorithm*. A trivial lower bound on the time complexity for multiplying two general $n \times n$ matrices is $\mathcal{O}(n^2)$. Several algorithms with $\mathcal{O}(n^x)$ running time, where $2 < x < 3$, have been proposed [5]. However, most of these algorithms are impractical because the constant factors associated with their time complexities are large. The design and implementation of algorithms that yield an improved lower and upper bound for the general matrix multiplication problem therefore remains an outstanding open challenge.

In this article a new data structure for representing sparse matrices is introduced and a matrix multiplication algorithm based on this new structure is presented. The analysis shows that the new data structure and algorithm are more time and space efficient than existing methods if the sparse matrices consist of strings, called segments, of adjacent nonzero elements in rows and/or columns. The performance improvement is verified by numerical experiments. In this regard it is important to note that in large-scale, high-performance computing applications, even a factor of two improvement in the time efficiency of an operation may impact on whether a particular application can or cannot be run. Since the sparse matrices arising in scientific and engineering applications tend to be highly structured, such as band or triangular matrices, the implementation of the sparse matrix multiplication algorithm introduced in this paper can result in a considerable performance improvement.

2. Sparse matrix representations

It is obvious that using two-dimensional arrays to represent sparse matrices not only wastes space but also cannot lead to sub-quadratic-time matrix operations. Consider the sparse matrix A shown in fig. 1. The data structure for sparse matrices introduced in ref. [6] for this example is shown in fig. 2.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	7	1	2	2	3	4	5	5	5	5	6	6	6	7
2	7	1	1	2	4	4	2	4	5	6	5	6	7	5
3	13	2	3	4	5	6	1	8	4	3	2	2	1	5

Fig. 2. Horowitz representation of a 7×7 matrix. The elements LA(0, k) with $k = 1$, 2 and 3 specify, respectively, the number of rows, columns, and nonzero elements, t, in the matrix A. The elements LA(1: t, k) with $k = 1$ and 2 specify the row and column indices of the nonzero element of A that are stored in L(1: t, 3). Specifically, LA(i, 1) and LA(i, 2) are the row and column labels of the ith nonzero element which has the value stored in LA(i, 3), $1 \leq i \leq t$.

Let A be represented by a $(t+1) \times 3$ array LA$(0:t, 1:3)$, where t is the number of nonzero elements in the matrix. The 3-tuple (LA$(k, 1)$, LA$(k, 2)$, LA$(k, 3)$) specifies the kth entry of the LA array. The 0th entry contains global information on A, with LA(0, 1) and LA(0, 2) specifying the number of rows and columns, respectively, and LA(0, 3) the number t of nonzero elements. The kth entry, $k > 0$, contains information on the kth nonzero element. Specifically, LA$(k, 3)$ is its value and LA$(k, 1)$ and LA$(k, 2)$ are the row and column indices, respectively. Furthermore, let the nonzero elements be stored in this array in row-major order, that is, in increasing order of the row index, and for all nonzero elements with the same row index in increasing order of the column index. The LA array is a linear list of nonzero elements in the matrix A.

Using this data structure and the algorithms given in ref. [6], the transposition of an $m \times n$ matrix can be done in $\mathcal{O}(n+t)$ time, where t is the total number of nonzero elements in the matrix, and the multiplication of an $m \times n$ matrix A with an $n \times p$ matrix B can be accomplished in $\mathcal{O}(pt_A + mt_B)$ time, where t_A and t_B are the number of nonzero elements in A and B, respectively. It is easy to see that this data structure is not space efficient if the matrix contains segments of adjacent nonzero elements, since most of the row and column numbers in a segment are then redundant. More importantly, this redundant structure does not lead to more efficient matrix operations, which is contrary to the general algorithm design principle of introducing redundancy to improve performance.

The matrix multiplication algorithm given in ref. [6] performs repetitive linear scans of the entries of the arrays representing the factor matrices. In computing C_{ij}, the factors A_{ik} and B_{kj} are both checked to see if the multiplication operation $A_{ik} \times B_{kj}$ can be skipped because one or the other factor is zero. It would be nice to add additional information to the data structure that would allow redundant checks to be eliminated. This is the idea behind the new data structure. Specifically, a *maximal nonzero segment* (or simply *segment*) in row i of an $m \times n$ matrix A is defined as a closed interval $[j, k]$ such that all elements $A_{il} \neq 0$ for $j \le l \le k$ with $A_{i,j-1} = 0$ if $j > 1$ and $A_{i,k+1} = 0$ if $k < m$. Our representation for A consists of two arrays, EA$(1:t_A)$ and SA$(0:s_A, 1:3)$, where t_A is the number of nonzero elements in A and s_A is the number of segments in A in row-major order. Each segment corresponds to a subarray of EA and all segments of A are in row-major order in EA. SA contains $s_A + 1$ entries with each entry being a 3-tuple (SA$(r, 1)$, SA$(r, 2)$, SA$(r, 3)$). SA(0, 1) and SA(0, 2) specify, respectively, the number of rows and columns in A while SA(0, 3) gives the number of segments. That is, SA(0, 1) $= m$, SA(0, 2) $= n$

EA:

1	2	3	4	5	6	7	8	9	10	11	12	13
2	3	4	5	6	1	8	4	3	2	2	1	5

SA:

	0	1	2	3	4	5	6	7	8
1	7	1	2	3	4	5	5	6	7
2	7	1	1	4	4	2	4	5	5
3	8	1	2	4	4	2	6	7	5

Fig. 3. New representation of a sparse matrix. This representation consists of two arrays, EA and SA. The value of the nonzero elements are stored in a one-dimensional array EA while the two-dimensional array SA holds the information on the segments of the matrix A. Specifically, while as for the Horowitz case SA$(0, k)$ with $k = 1$ and 2 specify the number of rows and columns in A, SA(0, 3) gives the number of segments, t_A, rather than the number of nonzero elements. Furthermore, SA$(i, 1)$ specifies the row index of segments while SA$(i, 2)$ and SA$(i, 3)$ give the starting and ending columns of the ith segment, $1 \le i \le t_A$.

and SA(0, 3) = s_A for an $m \times n$ matrix with s_A segments. The rth entry, $r > 0$, contains information about the rth segment. Specifically, if the rth segment $[j, k]$ is in row i, then SA(r, 1) = i, SA(r, 2) = j, and SA(r, 3) = k. Since the segments of A are arranged in EA in a unique linear order, the indices of the segments can be used to calculate a linear scan of SA. We use the pair (EA, SA) to denote this data structure for the matrix A. For the example matrix A of fig. 1, this new representation is shown in fig. 3.

For an $m \times n$ matrix A with t_A nonzero elements and s_A segments, the conventional 2-dimensional array representation of A requires $m \times n$ words, LA requires $3(t_A + 1)$ words and our representation requires $t_A + 3(s_A + 1)$ words. If A is sparse, then it is possible that $3(t_A + 1) \ll m \times n$ and $[t_A + 3(s_A + 1)] \ll m \times n$. However, if $t_A > 1.5\ s_A$, then $[t_A + 3(s_A + 1)] < 3(t_A + 1)$. That is, if on the average each segment contains more than 1.5 elements, then our new representation requires less space than the representation LA introduced in ref. [6]. For large-scale scientific applications, this condition frequently holds. It is easy to see that $\mathcal{O}(t_A)$ time is sufficient to transform LA into (EA, SA), and vice versa. It is a simple fact that transforming the conventional 2-dimensional array representation of an $m \times n$ matrix into LA, or (EA, SA), and vice versa, takes $\mathcal{O}(m \times n)$ time.

3. Algorithms in the package

The SPARSE MATRIX MULTIPLICATION (SMM) package consists of five subprograms: MMULT, TRANSPOSE, PACKING, UNPACKING, PRINTOUT. The function of each of these programs is described in this section. It is important to note that the routines are generic and passive. They are generic because they can be used for one or more matrix multiplications in an application. And they are passive because they work on elements in the new representations of matrices and need no information other than that provided by the new representation on itself. Details about each of the five SMM routines are given below.

Before proceeding with a detailed description of the five SMM routines, however, first consider more carefully the new representation for sparse matrices used in the package. A matrix X consists of two arrays EX(1 : t) and NSX(0 : s + 1, 1 : 3), where t is the number of nonzero elements in X and s is the number of segments in X in row-major order. For bookkeeping purposes, the size of NSX must be one greater than the number of segments. The NSX array contains $s + 1$ integer entries with each entry a 3-tuple of type (NSX(r, 1), NSX(r, 2), NSX(r, 3)). NSX(0, 1) and NSX(0, 2) are the number of rows and columns in the matrix X, respectively, while NSX(0, 3) specifies the number of segments in X. Specifically, for an $m \times n$ matrix X with s segments NSX(0, 1) = m, NSX(0, 2) = n, NSX(0, 3) = s. The rth entry, $r > 0$, specifies information on the segments. For example if the rth segment $[j, k]$ is in row i then NSX(r, 1) = i, NSX(r, 2) = j, NSX(r, 3) = k. Since nonzero elements and segments of the matrix X are arranged in EX in a unique linear order, the indices of the nonzero elements can be calculated by a linear scan of NSX. Here the pair (EX, NSX) is used to denote matrix X in the new representation.

The subroutine MMULT multiplies two sparse matrices (EA, NSA) and (EB, NSB) which are given in the new sparse matrix representation. The result is (EC, NSC). The subroutine includes the subprogram VECTOR_MULTIPLY. To make the SMM package execute efficiently this subprogram has not been separated out as a subroutine in the SMM package, nonetheless, for convenience it is identified here as f it is a separate subroutine called VECTOR_MULTIPLY.

```
VECTOR_MULTIPLY
Step 1. initialize variables;
        $a := 1,\ b := 1,\ c := 0$;
Step 2. **while** $a \le r$ **and** $b \le s$ **do**
        $[j, k] := [j_a^A, k_a^A] \cap [j_b^B, k_b^B]$;
        /* $[j_a^A, k_a^A]$ is the $a$th segment in matrix $A$ and $[j_b^B, k_b^B]$ is the $b$th segment in matrix $B$ */
```

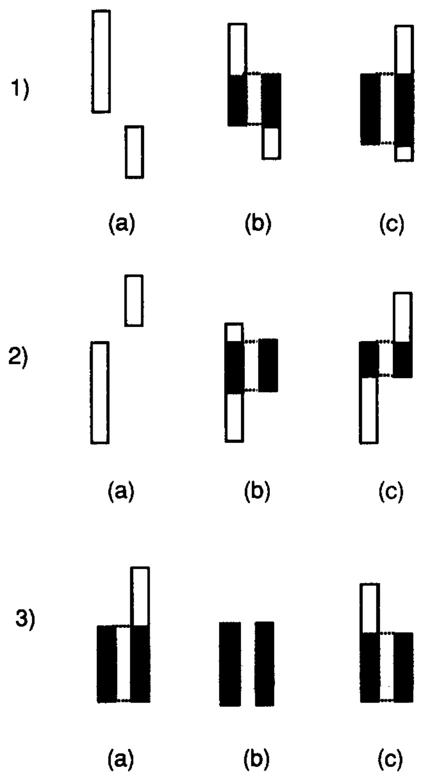

Fig. 4. Overlap conditions for the multiplication of two matrices. In the figure each vertical strip denotes a matrix segment. The segment on the left is from A and the one on the right is from B^T in the product $A \times B$. The solid block denotes the overlap interval. The patterns labelled types 1, 2 and 3 are complementary scenarios. Specifically, if the current overlap is of type 1 (2) then the next pair that must be considered is found by increasing the segment index of A (B^T). If the overlap is of type 3, that is, SA(i, 3) and SB(j, 3) are equal for the ith and jth segments of A and B, respectively, the segment indices of both A and B^T must be increased. The implementation of this sequential procedure yields a time complexity that is proportional to the total number of segments as compared to the number of nonzero elements.

```
if $[j, k] \neq \phi$ then
    begin
        compute the dot-product of subvectors of $A$ and $B$ defined by segment $[j, k]$, and let
            $D$ be the result of this computation;
        $C := C + D$;
        if the overlap condition of $[j_a^A, k_a^A] \cap [j_b^B, k_b^B]$ is one of types
            1.b and 1.c identified in fig. 4 then
            $a := a + 1$
        else if the overlap condition is one of types 2.b and 2.c then
            $b := b + 1$
        else /* the condition is one of types 3.a, 3.b and 3.c */
            $a := a + 1$;
            $b := b + 1$
    end
else
    /* the overlap condition of $[j_a^A, k_a^A] \cap [j_b^B, k_b^B]$ is one of types 1.a and 2.a */
```

```
            $a := a + 1$;
            $b := b + 1$
        endwhile
end of VECTOR_MULTIPLY
```

MMULT

EA	nonzero element array of matrix A.
NSA	segment array of matrix A.
EB	nonzero element array of matrix B.
NSB	segment array of matrix B.
EC	nonzero element array of matrix C.
NSC	segment array of matrix C.
ET	nonzero element array of matrix T.
NST	segment array of matrix T.
QUE	a queue array for the nonzero elements of a matrix.
KUE	a queue array for the low index of the nonzero elements.
NPNT	a pointer array.
IDIM	maximum row index.
JDIM	maximum column index.
NSPO	number of segments plus one.

```
Step 1. use TRANSPOSE to obtain ET and NST;
Step 2. let $a_1, a_2, \ldots, a_g$ be the row numbers of nonzero rows of matrix $A$ such that $a_i \le a_{i+1}$;
        let $b_1, b_2, \ldots, b_h$ be the row numbers of nonzero rows of matrix $T$ such that $b_j \le b_{j+1}$;
        for $i = 1$ to $g$ do
            for $j = 1$ to $h$ do
                perform vector dot-product operation to row $A_{a_i}$ of matrix $A$ and row $T_{b_j}$ of matrix $T$
                as described in the VECTOR_MULTIPLY algorithm;
                store the result $C_{a_i,b_j}$ in EC and update NSC
            endfor
        endfor
end of MATRIX_MULTIPLY
```

The subroutine TRANSPOSE generates the transpose of a matrix A given in the new representation, namely, (EA, NSA). The result is (ET, NST).

TRANSPOSE

EA	nonzero element array of matrix A.
NSA	segment array of matrix A.
ET	nonzero element array of matrix T.
NST	segment array of matrix T.
QUE	a queue array for the nonzero elements of a matrix.
KUE	a queue array for the low index of the nonzero elements.
NPNT	a pointer array.
IDIM	maximum row index.
JDIM	maximum column index.
NSPO	number of segments plus one.

```
Step 1. initialize $n$ empty queues;
Step 2. scan EA[$i$] in increasing order of $i$, use SA to calculate the row and column number of EA[$i$],
        and put EA[$i$] into the $j$th queue if it is in the $j$-th column of $A$;
Step 3. scan queues to compute NST and concatenate the result to obtain ET
end of TRANSPOSE
```

Two subroutines, one called PACKING for converting a conventional two-dimensional array into the new representation, and another called UNPACKING for the inverse process, are included for convenience. The subroutine PACKING converts a standard two-dimensional matrix A into its corresponding sparse representation, namely, (EA, NSA).

PACKING

A	two-dimensional array representation of the matrix A.
EA	nonzero element array of matrix A.
NSA	segment array of matrix A.
IDIM	maximum row index.
JDIM	maximum column index.
NSPO	number of segments plus one.

```
    count $\leftarrow$ 0; noseg $\leftarrow$ 0;
    for $i \leftarrow 1$ to ni do
        last $\leftarrow -1$;
        for $j \leftarrow 1$ to nj do
            if $A(i, j) \neq 0$ then [
                count $\leftarrow$ count + 1
                EA(count) $\leftarrow A(i, j)$
                if last + 1 = $j$ then [
                    last $\leftarrow j$; SA(noseg, 3) $\leftarrow j$]
                else [
                    last $\leftarrow j$; noseg $\leftarrow$ noseg + 1
                    SA(noseg, 1) $\leftarrow i$; SA(noseg, 2) $\leftarrow j$; SA(noseg, 3) $\leftarrow j$]]
        end
    end
    SA(0, 1) $\leftarrow$ ni; SA(0, 2) $\leftarrow$ nj; SA(0, 3) $\leftarrow$ noseg
end of PACKING
```

The subroutine UNPACKING converts a sparse matrix representation (EA, NSA) of A back into its two-dimensional array representation.

UNPACKING

EA	nonzero element array of matrix A.
NSA	segment array of matrix A.
A	two-dimensional array representation of the matrix A.
IDIM	maximum row index.
JDIM	maximum column index.
NSPO	number of segments plus one.

```
        count ← 1;
        for $i$ ← 1 to SA(0, 3) do
            for $j$ ← SA($i$, 2) to SA($i$, 3) do
                $A$(SA($i$, 1), $j$) ← EA(count);
                count ← count + 1
            end
        end
    end of UNPACKING
```

The subroutine PRINTOUT prints elements of a two-dimensional matrix. The user can decide the output ranges.

PRINTOUT...

A	two-dimensional array representation of the matrix A.
IDIM	maximum row index.
JDIM	maximum column index.
IDIM1	First position of the row that is to be printed out.
IDIM2	Second position of the row that is to be printed out.
JDIM1	First position of the column that is to be printed out.
JDIM2	Second position of the column that is to be printed out.

4. Performance characteristics

The MATRIX_MULTIPLY algorithm was tested using FORTRAN on an IBM 3090/600E. The matrices A and B were taken to be identical band square matrices of dimension D as shown schematically in fig. 5. The size D was fixed at 300 and its bandwidth L was set at 31 with each row of both A and B initially chosen to be a single segment. Test data sets were generated from these initial distributions by iteratively breaking segments of length of at least three in both A and B into two segments by replacing one of the nonzero elements with 0. The performance of the algorithm, which includes TRANSPOSE as a subalgorithm, on this set of data is shown in fig. 6. For comparison, results for the standard and sparse matrix multiplication algorithms using the same data are also shown. In addition to the fact that in scientific and engineering applications sparse band matrices are very common, it is also important to note that this random generation procedure yields test data with segment lengths that are normally distributed. When each row contains one segment, the results show that the new algorithm is between 7 and 8 times faster than the algorithm given in ref. [6] and about 40 times faster than the standard algorithm. When there are 12 segments in each row, except for the first and last 15, that is, 4 segments with one nonzero element and 8 with two elements, the algorithm performs at about the same speed as the HS algorithm of ref. [6]. The new algorithm is slower than the HS algorithm when the number of segments in each row is increased beyond 12. This is because the overhead in comparing segments to find overlapping intervals approaches the number of comparisons required in the HS algorithm.

In fig. 7, the quantity R measures the ratio of the number of nonzero elements to the total number of elements in the factor band matrices. The results show that the HS algorithm is more than 3 times slower than the new one for the dense $R = 1$ case. In fact, for matrices of this type the new algorithm is *always* faster than both the HS and standard methods. In particular, even for the dense $R = 1$ case, when the A

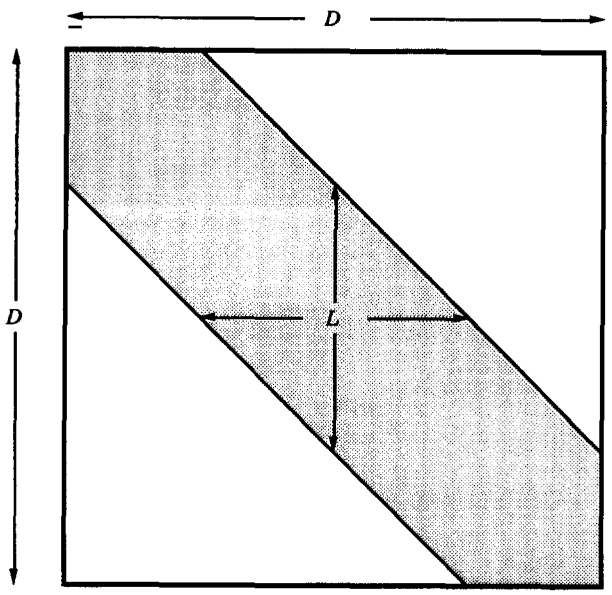

Fig. 5. Band matrix. A square sparse matrix of dimension D is shown. The shadowed area represents the band of nonzero elements. The symbol L is used to denote the bandwidth which is simply the number of nonzero elements in an interior row or column of the matrix.

and B matrices contain no zero elements, the new algorithm executes about 20% faster than the standard one.

Also, the representation for sparse matrices that has been introduced here can save a considerable amount of memory space, when compared with the standard 2-dimensional array representation and

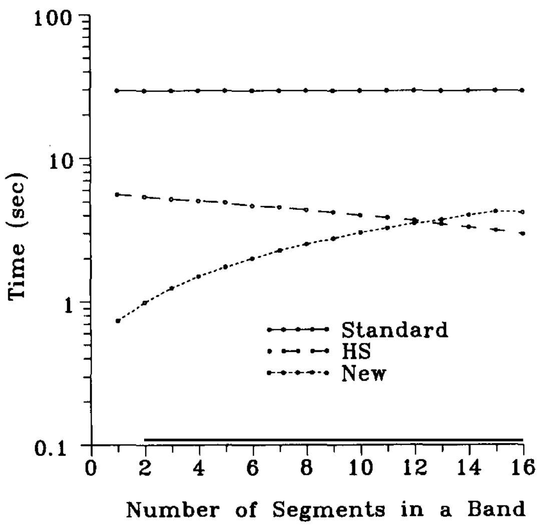

Fig. 6. Performance test A. As an example, two 300×300 band matrices were multiplied using the standard (Standard), Horowitz (HS), and new (New) matrix representations and the corresponding matrix multiplication algorithms. The value of L was fixed at 31 with one segment per row, and then these segments were broken iteratively and randomly by replacing one of the nonzero elements with 0, leaving segments of length at least three. The test was performed on an IBM 3090/700E using FORTRAN.

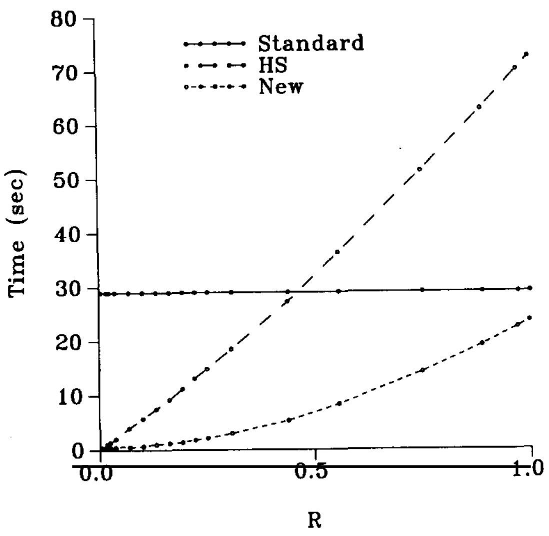

Fig. 7. Performance test B. As an example, the product $A \times A$ was calculated where A is a band matrix of dimension 300 with a single segment per row, see fig. 5. The length L was changed so R, the ratio of the number of nonzero elements to the total number of elements in A varied from 0 to 1. As above, this test was also performed on an IBM 3090/600E using FORTRAN.

even the data structure introduced in ref. [6], in many scientific and engineering applications. For example, for a 300×300 band matrix with L-31 (one segment in each row), there are 9060 nonzero elements. The new representation requires 9963 words of memory, which is less than 36% of the 27 183 words required by the representation given in ref. [6], and about 11% of the 90 000 words required for the standard 2-dimensional array representation. The associated matrix multiplication algorithm has been shown to out perform the one given in ref. [6] for matrices containing clusters of nonzero elements, such as band or triangular matrices, that are common in many applications.

A breakdown on the storage requirements for routines in the SMM package on an IBM 3090/600E is given in table 1.

5. Conclusion

The routines in the SMM package are simple to use and execute efficiently. Input and output matrices can be given either in the standard 2-dimensional array form or directly in the new representation since subroutines are provided for transforming from one to the other. Specifically, the subroutine called PACKING can be used to construct the new representation of a matrix in an on-the-fly fashion from its

Table 1
Storage requirements in bytes for routines in the SMM package on an IBM 3090/600E

MMULT	6116
TRANSPOSE	2146
PACKING	1232
UNPACKING	972
PRINTOUT	1134
Total (SMM)	11600

standard form while the routine called UNPACKING can be called to convert a matrix in the new representation into its standard form. Since these are elementary transformations that can be carried out very efficiently, the implementation of the new sparse matrix multiplication algorithm may led to significant performance improvement.

As the experimental results shown in figs. 6 and 7 illustrate, the sparse matrix multiplication algorithm implemented in the SMM package can lead to a very significant improvement in performance. In general, improved performance is expected when the factor matrices are relatively sparse with the nonzero elements clustered into segments of average length 1.5 or greater. However, even for relatively dense matrices the new scheme may out perform other algorithms like those based on the standard and HS representations. Beyond this, since in every iteration of the multiplication algorithm a vector dot-product is performed on the overlap portion of segments, one from each operand matrix, it may be possible to use vector and even parallel processing to realize additional gains [7].

The routines in the SMM package have been written in a manner that makes incorporating them into existing programs an easy task. In particular, the codes are written in FORTRAN as, despite attempts to adopt simpler and more powerful languages, this is still the language of choice for most CPU-intense scientific applications. One reason for this is the ready availability of many other FORTRAN codes that have been tuned for efficiency in numerically-intensive, high-performance computing applications. Another reason is that the efficiency of FORTRAN compilers is generally very high.

The purpose of the SMM package is to place in the public domain a set of software tools that allow scientists to incorporate the new structure and logic into FORTRAN programs with a minimum of inconvenience. We have found the SMM routines to be extremely efficient in dense as well as sparse matrix computations. It is our hope that others will also.

References

[1] J.K. Cullum and R.A. Willoughby, Lanczos Algorithms for Large Symmetric Eigenvalue Computations, Vol. 1 (Birkhauser, Boston 1985).
[2] G.H. Golub and C.F. Van Loan, Matrix Computations, 2nd ed. (Johns Hopkins Univ. Press, Baltimore, 1989).
[3] J.W. Negele and H. Orland, Quantum Many-Particle Systems (Addison–Wesley, Reading, MA, 1988).
[4] J. Neter, W. Wasserman and M.H. Kutner, Applied Linear Regression Models, 2nd ed. (Irwin, Homewood, IL, 1989).
[5] V. Strassen, The Asymptotic Spectrum of Tensors and the Exponent of Matrix Multiplication (IEEE Press, New York, 1986) p. 49.
[6] E. Horowitz and S. Sahni, Fundamentals of Data Structures, (Computer Science Press, Rockville, MD, 1983).
[7] M.J. Quinn, Designing Efficient Algorithms for Parallel Computers (McGraw-Hill, New York, 1987).