APPLIED COMPUTING:
TECHNOLOGICAL CHALLENGES OF THE 1990'S

PROCEEDINGS OF THE 1992 ACM/SIGAPP
SYMPOSIUM ON APPLIED COMPUTING
VOLUME II

Kansas City Convention Center
March 1-3, 1992

editors:

Hal Berghel
Ed Deaton
George Hedrick
David Roach
Roger Wainwright

![Pie chart showing categories of computing challenges]

- Networks
- AI
- Info Sys
- Database
- Graphics
- OP Sys
- Distributed
- Parallel
- Languages
- OOP
- Hardware
- Scientific
- Other
An Efficient Algorithm for Sparse Matrix Computations

S. C. Park and J. P. Draayer
Department of Computer Science & Department of Physics and Astronomy
Louisiana State University, Baton Rouge, LA 70803

and

S.-Q. Zheng
Department of Computer Science
Louisiana State University, Baton Rouge, LA 70803

Abstract

A new representation of a sparse matrix is introduced that is very efficient for matrix multiplication when the non-zero elements 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 non-zero elements is less than two-thirds of the total number of non-zero elements. Time complexity is better or much better than that of existing algorithms depending on the number of groups of non-zero adjacent elements in the factor matrices.

*Funded in part by the U. S. National Science Foundation, Grant No. PHY89-22550.

Introduction

Matrices are abstract mathematical objects that arise in many numerical as well as non-numerical problems. Examples range from solving systems of equations to representations of graphs. In large-scale, high-performance scientific and engineering applications, it is not unusual to encounter matrices with tens of thousands of elements. Designing efficient algorithms for matrix operations is therefore a problem of fundamental importance in computational science.

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:

\[
\text{For } i = 1 \text{ to } m \\
\quad \text{For } j = 1 \text{ to } p \\
\quad \quad C_{ij} = 0 \\
\quad \quad \text{For } k = 1 \text{ to } n \\
\quad \quad \quad C_{ij} = C_{ij} + A_{ik} \times B_{kj} \\
\quad \text{endfor} \\
\text{endfor}
\]

Obviously this algorithm takes \( O(n^3) \) 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 \( \Omega(n^2) \). Several algorithms with \( O(n^2) \) running time, \( 2 < x < 3 \), have been proposed [1]. However, most of these algorithms are impractical because the constant factors associated with their time complexities are large.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

\* 1992 ACM 0-89791-502-X/92/0002/0919...$1.50

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.

It is important for computer scientists to understand the special requirements of large-scale matrix applications and design algorithms that meet these needs. These requirements can be identified by considering the common characteristics of abstract mathematical models of the physical world. In this paper the multiplication of large sparse matrices [2] [3], which enter for example in eigensystem analyses [4], least square problems [5], and solutions to systems of linear equations, will be considered. For sparse matrix multiplication, the lower bound mentioned above for the time complexity certainly does not hold. A trivial example of this is when both factor matrices are diagonal in which case the time complexity is simply \( O(n) \). It is therefore desirable to design special sparse matrix algorithms with time and space complexities that depend on the number of non-zero elements in the operand matrices.

A space-efficient representation of sparse matrices is given in [6] (denoted by \( \text{HS} \) in what follows). In this representation only non-zero elements are considered with each identified by a 3-tuple of the form \( (i, j, v) \), where \( i \) and \( j \) are the row and column numbers of the element, respectively, and \( v \) is the value of the element. A sparse matrix is stored in memory as a linear list of non-zero elements in row-major order. With this representation, multiplying two \( n \times n \) matrices can be carried out in \( O(n^2 (t_1 + t_2)) \) time, where \( t_1 \) and \( t_2 \) are the numbers of non-zero elements in the operand matrices.

In this paper, a modified data structure for 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 the existing methods if the sparse matrices contain segments consisting of adjacent non-zero elements in rows and/or columns. The improved performance of the algorithm 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 2 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 matrix multiplication algorithm introduced in this paper can result in a considerable performance improvement.
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 Figure 1. The data structure for sparse matrices introduced in [HS] for this example is shown in Figure 2.

\[
\begin{array}{ccc|ccc|ccc}
   & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
   1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\
   2 & 3 & 4 & 0 & 0 & 0 & 0 & 0 \\
   3 & 4 & 0 & 0 & 0 & 0 & 0 & 0 \\
   4 & 5 & 0 & 1 & 0 & 8 & 4 & 3 \\
   5 & 6 & 0 & 0 & 0 & 2 & 2 & 1 \\
   6 & 7 & 0 & 0 & 0 & 0 & 5 & 0 \\
\end{array}
\]

Figure 1 Standard two-dimensional array representation of a \( 7 \times 7 \) matrix. The standard two-dimensional array representation of a matrix uses fixed storage since zero and non-zero elements are handled the same.

Let \( A \) be represented by a \((t+1) \times 3\) array \( LA(0:t, 1:3) \), where \( t \) is the number of non-zero elements in the matrix. The 3-tuple \((LA(k, 1), LA(k, 2), LA(k, 3))\) specifies the \( k \)-th entry of the LA array. The 0-th 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 of non-zero elements. The \( k \)-th entry, \( k > 0 \), contains information on the \( k \)-th non-zero 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 non-zero elements be stored in this array in row-major order, that is, in increasing order of the row index, and for all non-zero elements with the same row index in increasing order of the column index. The LA array is a linear list of non-zero elements in the matrix \( A \).

\[
\begin{array}{cccccccccccc}
   LA: & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\
   1 & 7 & 1 & 2 & 3 & 4 & 5 & 5 & 5 & 5 & 6 & 6 & 6 & 7 \\
   2 & 7 & 1 & 1 & 2 & 4 & 4 & 2 & 4 & 5 & 6 & 5 & 7 & 5 \\
   3 & 13 & 2 & 3 & 4 & 5 & 6 & 1 & 8 & 4 & 3 & 2 & 2 & 1 & 5 \\
\end{array}
\]

Figure 2 Horowitz representation of a \( 7 \times 7 \) matrix. The elements \( LA(0, k) \) with \( k = 1, 2 \) and 3 specify, respectively, the number of rows, columns, and non-zero 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 non-zero element of \( A \) that are stored in \( LF(1:t, 3) \). Specifically, \( LA(i,1) \) and \( LA(i,2) \) are the row and column labels of the \( i \)-th non-zero element which has the value stored in \( LA(i,3) \), \( 1 \leq i \leq t \).

Using this data structure and the algorithms given in [HS], the transposition of an \( m \times n \) matrix can be done in \( O(n + t) \) time, where \( t \) is the total number of non-zero 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 \( O(p t_A + m t_B) \) time, where \( t_A \) and \( t_B \) are the number of non-zero 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 non-zero 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 [HS] 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 non-zero 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 \( A_{ij} \neq 0 \) for \( j \leq i \leq k \). 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 non-zero elements in \( A \) and \( s_A \) is the number of segments in \( A \) in row-major order. Each segment corresponds to a sub-array 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 \( SA(0, 3) = s_A \) gives the number of segments. That is, \( SA(0, 1) = m \), \( SA(0, 2) = n \) and \( SA(0, 3) = s_A \) for an \( m \times n \) matrix with \( s_A \) segments. The \( r \)-th entry, \( r > 0 \), contains information about the \( r \)-th segment. Specifically, if the \( r \)-th segment \((j, k)\) is in row \( i \), then \( SA(i, 1) = j \), \( SA(i, 2) = k \), and \( SA(i, 3) = k - j \). 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 Figure 1, this new representation is shown in Figure 3.

\[
\begin{array}{cccccccccccc}
   EA: & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\
   2 & 3 & 4 & 5 & 6 & 1 & 8 & 4 & 3 & 2 & 1 & 5 \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
   SA: & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\
   1 & 7 & 1 & 2 & 3 & 4 & 5 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
   2 & 7 & 1 & 1 & 4 & 4 & 2 & 4 & 5 & 5 & 6 & 7 & 8 & 9 & 10 \\
   3 & 8 & 1 & 2 & 4 & 4 & 2 & 6 & 7 & 5 & 6 & 7 & 8 & 9 & 10 \\
\end{array}
\]

Figure 3 New representation of a sparse matrix \( A \). This representation consists of two arrays, \( EA \) and \( SA \). The value of the non-zero 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 non-zero 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 \( i \)-th segment, \( 1 \leq i \leq t_A \).

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

Sparse Matrix Multiplication Algorithm

Consider the problem of multiplying an \(m \times n\) matrix \(A\) and an \(n \times p\) matrix \(B\) to obtain an \(m \times p\) matrix \(C\) using this new matrix representation. Logic similar to the matrix multiplication algorithm given in [HS] can be used provided an efficient matrix transposition algorithm is available. Let \((EB, SB)\) be the new representation matrix \(B\). Then \((EB, SB^T)\) can be constructed as follows: first, transform \((EB, SB)\) into \(LB\) in \(O(t_B)\) time; then transform \(LB\) into \(LB^T\) in \(O(n + t_B)\) time (see [HS]); and finally, transform \(LB^T\) into \((EB^T, SB^T)\), again in \(O(t_B)\) time. To reduce the overhead, a bucket sort algorithm can be applied.

Algorithm TRANSPOSE\((EB, SB, EB^T, SB^T)\)

Step 1: Initialize \(n\) empty queues;
Step 2: Scan \(EB[i]\) in increasing order of \(i\), use \(SB\) to calculate the row and column number of \(EB[i]\), and put \(EB[i]\) into the \(j\)-th queue if it is in the \(j\)-th column of \(B\);
Step 3: Scan queues to compute \(SB^T\) and concatenate the result to obtain \(EB^T\);
end TRANSPOSE

This algorithm also takes \(O(n + t_B)\) time. However, the constant factor of its complexity is smaller than that of the previous method. The details of TRANSPOSE are given in the appendix.

Before presenting the matrix multiplication algorithm, it is instructive to first consider the dot product of two vectors \(A = (A_1, A_2, \ldots, A_n)\) and \(B = (B_1, B_2, \ldots, B_n)\) of \(n\) elements each. Let \(A\) (resp. \(B\)) be represented as a sequence of segments of non-zero elements

\[L_A = ([j_1^A, k_1^A], [j_2^A, k_2^A], \ldots, [j_r^A, k_r^A])\]

(resp. \(L_B = ([j_1^B, k_1^B], [j_2^B, k_2^B], \ldots, [j_r^B, k_r^B])\))
such that

\[1 \leq j_1^A \leq k_1^A < j_2^A \leq k_2^A < \ldots < j_r^A \leq k_r^A \leq n\]

(resp. \(1 \leq j_1^B \leq k_1^B < j_2^B \leq k_2^B < \ldots < j_r^B \leq k_r^B \leq n\))

There are a total of nine possible different segment overlap conditions as shown in Figure 4. In the figure the left rectangle and right rectangle represent segment \([j_1^A, k_1^A]\) and \([j_1^B, k_1^B]\), respectively. The solid area corresponds to the overlap interval of the two segments. The nine cases can be subdivided into three groups which are labelled (1), (2), and (3) in the figure. Based on the overlap conditions, the following simple algorithm can be given:

![Figure 4. Overlap conditions for the multiplication of two matrices.](image)

**Algorithm VECTOR_MULTIPLY**

\[
a := 1, b := 1, c := 0,
while a \leq r \text{ and } b \leq s \text{ do}
\]

\[
[i, k] := [j_1^A, k_1^A] \cap [j_1^B, k_1^B]
\]

\[
if [j, k] = \emptyset \text{ then}
begin
compute the dot-product of sub-vectors of \(A\) and \(B\) defined by segment \([j, k]\), and let \(D\) be the result of this computation;
end
\]

\[
C := C + D;
\]

\[
if \text{ the overlap condition of } [j_1^A, k_1^A] \cap [j_1^B, k_1^B] \text{ is one of types 1.b and 1.c in Figure 4} \text{ then}
\]

\[
a := a + 1
\]

\[
\text{else if the overlap condition is one of types 2.b and 2.c then}
\]

\[
b := b + 1
\]
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
\( \{ k_a^A, k_b^A \} \cap \{ k_a^B, k_b^B \} \) is one of types
1.a and 1.b */
a := a + 1;
b := b + 1
endwhile
end VECTOR_MULTIPLY

Define

\[
\text{Overlap}_{ab} = \begin{cases} 
0, & \text{if segments } \{ k_a^A, k_b^A \} \text{ and } \\
\{ k_a^B, k_b^B \} \text{ do not overlap;} \\
\min(k_a^A, k_b^B) - \max(k_a^A, k_b^B), & \text{otherwise.}
\end{cases}
\]

Define

\[
\text{Overlap}_{AB} = \sum_{1 \leq a \leq r} \sum_{1 \leq b \leq s} \text{Overlap}_{ab}.
\]

Clearly, the time complexity of VECTOR_MULTIPLY is \( O(r + s + \text{Overlap}_{AB}) \). It is also easy to see that \( \Omega(\text{Overlap}_{AB}) \) is a lower bound for the time complexity of the dot product of two vectors A and B. If \( r + s \) and \( \text{Overlap}_{AB} \) are comparable, then VECTOR_MULTIPLY runs in optimal time. The worst case scenario is when \( A = (A_1, \ldots, A_n) \) and \( B = (B_1, \ldots, B_n) \) are of the forms \( A_i = 0 \), \( A_{i+1} = 0 \) & \( B_i = 0 \), \( B_{i+1} = 0 \) for either \( i \) even or \( i \) odd. In this case \( r + s = n \) and \( \text{Overlap}_{AB} = 0 \). To summarize, by introducing segment information into the data structure for sparse vectors, non-overlap portions of the two operands can be skipped, and if the two vectors contain many adjacent non-zero elements, the run time of comparison operations for determining overlapping portions of segments will be small.

Based on these observations, the following matrix multiplication algorithm is proposed. As input there is an \( m \times n \) matrix A and an \( n \times p \) matrix B represented by \((EA, SA)\), and \((EB, SB)\), respectively, while \((EC, SC)\) represent the product matrix C.

Algorithm MATRIX_MULTIPLY(EA, SA, EB, SB, EC, SC).

Use TRANSPOSE to obtain EB^T and SB^T;
Let \( a_1, a_2, \ldots, a_g \) be the row numbers of non-zero rows of A such that \( a_i \neq a_{i+1} \); Let \( b_1, b_2, \ldots, b_h \) be the row numbers of non-zero rows of B^T such that \( b_j \neq b_{j+1} \);
For \( i = 1 \) to \( g \) do
For \( j = 1 \) to \( h \) do
Perform vector dot-product operation to row \( A_{a_i} \) of A and row \( B_{b_j} \) of B^T as described in the VECTOR_MULTIPLY algorithm;
Store the result \( C_{a_i,b_j} \) in EC and update SC;
end for
end for
end MATRIX_MULTIPLY

Figure 5: Band matrix. A square sparse matrix of dimension D is shown. The shaded area represents the band of non-zero elements. The symbol L is used to denote the bandwidth which is simply the number of non-zero elements in a interior row or column of the matrix.

The row numbers \( a_i, 1 \leq i \leq g \), and \( b_j, 1 \leq j \leq h \), and segment information of row \( a_i \) of A and row \( b_j \) of B^T can be extracted from EA, SA, EB^T and SB^T. Since non-zero elements and segments of A and B^T are stored in EA, SA, EB^T, SB^T in row major order, \( a_i, 1 \leq i \leq g \), and \( b_j, 1 \leq j \leq h \), can be computed by linear scans of SA and SB^T, and rows \( A_{a_i} \) and \( B_{b_j} \), which are divided into segments, can be reconstructed from EA and EB^T while scanning SA and SB^T. A detailed implementation of the algorithm MATRIX_MULTIPLY is given in the appendix.

Let \( r_{a_i} \) and \( s_{b_j} \) be the number of segments in row \( A_{a_i} \) of A and row \( B_{b_j} \) of B^T. Then define

\[
r = \sum_{1 \leq i \leq g} r_{a_i},
\]

\[
s = \sum_{1 \leq j \leq h} s_{b_j}, \text{ and}
\]

\[
\text{Overlap}_{AB} = \sum_{1 \leq i \leq g} \sum_{1 \leq j \leq h} \text{Overlap}_{A_{a_i}B_{b_j}}
\]

Now since storing an element \( C_{a_i,b_j} \) in EC and updating SC can be done in constant time, as can be easily seen in the sub-algorithm STORE_SUM given in the appendix, the total run time for the algorithm MATRIX_MULTIPLY is \( O(h \times r + g \times s + \text{Overlap}_{AB}) \). Now compare the MATRIX_MULTIPLY algorithm with the one given in
The time complexity of the [HS] algorithm is \( O(h \times t_A + g \times t_B) \), where \( t_A \) and \( t_B \) are the number of non-zero elements in \( A \) and \( B \), respectively. Note that the time complexity of this algorithm was given as \( O(p \times t_A + m \times t_B) \). By slightly modifying the algorithm, \( O(h \times t_A + g \times t_B) \) time complexity is achievable. Since \( \text{Overlap}_{AB} \leq t_A \) and \( s \leq t_B \), theoretically the performance of the proposed sparse matrix multiplication algorithm is no worse than that of the algorithm given in [HS]. Considering that for most sparse matrix multiplication problems \( r < t_A, s < t_B \), and \( \text{Overlap}_{AB} < t_A + t_B \), the new algorithm outperforms the one given in [HS].

Experimental Results

The \text{MATRICES\_MULTIPLY} algorithm was tested using \text{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 Figure 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 and randomly breaking segments of length at least three in both \( A \) and \( B \) into two segments by replacing one of the non-zero elements with 0. The performance of the algorithm, which includes \text{TRANSPOSE} and \text{STORE\_SUM} as sub-algorithms, on this set of data is shown in Figure 6.

![Figure 6](image)

**Figure 6** Performance Test A. As an example, two 300 x 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 non-zero elements with 0, leaving segments of length at least three. The test was performed on an IBM 3090/600E using \text{FORTRAN}.

For comparison, results for the standard and sparse matrix multiplication algorithms (see Section 1 and [HS], respectively) 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 [HS] 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 non-zero element and 8 with two elements, the algorithm performs at about the same speed as the [HS] algorithm. The new algorithm is slower than the [HS] algorithm when the number of segments in each row is increased beyond 12. This is because when \( r = t_A \) and \( s = t_B \) the overhead in comparing segments to find overlapping intervals increases the number of comparisons required in the [HS] algorithm, a result that is made clear by a comparison of the time complexities, \( O(p + r + m \times s + \text{Overlap}_{AB}) \) and \( O(p + t_A + m \times t_B) \), respectively.

![Figure 7](image)

**Figure 7** Performance Test B. As an example, the product \( A \times A \) was calculated where \( A \) is a band matrix of dimension 300 multiplications with a single one segment per row, see Figure 5. The length \( L \) was changed so \( R \), the ratio of the number of non-zero elements to the total number of elements in \( A \) varied from 0 to 1. A above, this test was also performed on an IBM 3090/600E using \text{FORTRAN}.

One might expect the new algorithm to be slower than the standard matrix multiplication algorithm when the operand matrices are dense. However, experimental results can be used to demonstrate that this is not necessarily the case. For example, consider the performance of the three matrix multiplication algorithms shown in Figure 7 for a setup in which \( A \) and \( B \) are taken to be identical band matrices with one segment in each row. The quantity \( R \) measures the ratio of the number of non-zero elements to the total number of elements. 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 and B matrices contain no zero elements, the new algorithm executes about 20% faster than the standard one. An explanation for this can be found in the way most compilers allocate space in memory for the storage of two-dimensional arrays. The elements are usually arranged in either row-major or column-major order. This means, for example, that if the two-dimensional arrays defined by the FORTRAN language used in the calculation are arranged in row-major order, during each iteration the addresses of the \( A_{ik} \) and \( B_{kj} \) factors must be calculated before the innermost \( A_{ik} \times B_{kj} \) multiplication (see Section 1) can be performed. The overall overhead associated with the calculation of these addresses may involve as many as \( O(n^2) \) hidden integer multiplication problems and associated addition operations, a result that can easily account for why the standard method is slower than the new one. In contrasts, while the new algorithm requires the transformation of (EB, SB) into (EB_T, SB_T), during the multiplication process only the two linear arrays EA and EB_T are scanned with an occasional reference to SA and SB_T. The array addressing in the new data structure is therefore much simpler. These results show that the conventional two-dimensional array representation may not be optimal even for operations on dense matrices.

**Discussion**

A new representation for sparse matrices has been introduced that can save a considerable amount of memory space, when compared with the standard 2-dimensional array representation and even the data structure introduced in [HS1], in many scientific and engineering applications. For example, for a \( 300 \times 300 \) band matrix with \( L = 31 \) (one segment in each row), there are 9060 non-zero elements. The new representation requires 9963 words of memory, which is less than the words required for the representation given in [HS1] and about 10.7% of the 90,000 words required for the standard 2-dimensional array representation. The associated matrix multiplication algorithm has been shown to outperform the one given in (HS) for matrices containing clusters of non-zero elements, such as band or triangular matrices, that are common in many applications.

Based on comparisons of the performance of these three approaches, a general scheme for matrix multiplication can be devised. Specifically, a profile can be associated with a matrix that gives the fraction of its elements that are non-zero and the number of segments it contains. The performance of the three matrix multiplication algorithms can then be correlated by experimentation with threshold values on the maximum and minimum, respectively, of these indicators for the matrices in the product. The preferred representation for a given problem can be selected based on average profile indicator measures for matrices that enter into the calculation and the matrices transformed accordingly into the representation that is predicted to achieve optimal performance. Since as indicated in Section 2, transformations between any two of the three representations can be carried out very efficiently, such an approach may lead to significant performance improvement. In the appendix, two procedures called PACKING and UNPACKING, are given for transforming between the standard 2-dimensional matrix form and the new representation. If the elements of a matrix are input in row-major order, PACKING can be used to construct the new representation in an on-the-fly fashion while UNPACKING can be used to convert a matrix in the representation into the standard 2-dimensional array form.

As the experimental results shown in Figures 6 and 7 indicate, a new data structure and an associated algorithm has been introduced that can yield improved matrix multiplication performance. In general, the improved performance can be expected when the factor matrices are relatively sparse matrices with the non-zero elements clustered into segment of length 1.5 or greater. However, even for relatively dense matrices the new scheme may result in performance algorithms based on the standard and (HS) representations for dense matrices. Beyond this, since a vector dot product is performed on the overlap portion of segments, one from each operand matrix, in each iteration of the algorithm, it may be possible to use the vector processing to realize additional gains. In contrast, the (HS) algorithm is less amenable to a vector processing adaptation. As a final remark, we note that the overlap segment concept introduced in this paper may be used to develop new (parallel) algorithms [7] for other matrix operations. Additionally, it may be possible to introduce hardware components based on the segment-algorithm concept into future generation computer to achieve additional performance improvements in large-scale scientific and engineering applications.

**References**


Appendix

a) Transpose

The procedure TRANSPOSE takes the transpose of matrix A and stores the result in matrix B.

procedure TRANSPOSE(EA, SA, EB, SB) // bucket sort algorithm simulation with 1-tuple using a two-dimensional array //
for i ← 1 to SA(0,2) do
    PNT(i) ← 1
end
i ← 1;
count ← 0
while i ≤ SA(0,3) do
    for j ← SA(i,1) to SA(i,3)
        count ← count + 1;
        QUEUE(j,PNT(j,1)) ← EB(count);
        QUEUE(j,PNT(j,2)) ← SA(i,1);
        PNT(j) ← PNT(j) + 1
    end
    i ← i + 1
end (while) // end bucket sort construction//
while p ≤ count do
    if QUEUE[j,k,2] ≠ 0 then
        EB(l) ← QUEUE(j,k,1)
        if last + 1 = QUEUE(j,k,2) then
            last ← last + 1;
            SB(seg,3) ← last
        else
            last ← QUEUE(j,k,2);
            seg ← seg + 1;
            SB(seg,1) ← j;
            SB(seg,2) ← last;
            SB(seg,3) ← last
        end
        k ← k + 1;
        p ← p + 1
    else
        k ← 1;
        j ← j + 1;
        last ← -1
    end (while)
SB(0,1) ← SA(0,2);
SB(0,2) ← SA(0,1);
SB(0,3) ← seg
end TRANSPOSE

b) Multiplication

Before giving the matrix multiplication procedure, it is useful to present a sub-procedure called STORE_SUM. The purpose of this sub-procedure is to store the results Cij in the proper position in the sequence of non-zero elements in EC and to update SC.

procedure STORE_SUM(EC, SC, last, flag, count, row, col, sum) if flag then last ← -1
if sum ≠ 0 then
    count ← count + 1;
    EC(count) ← sum
    if last + 1 = col then
        last ← col;
else
    noseg ← noseg + 1
    last ← col
    SC(noseg,1) ← row;
    SC(noseg,2) ← col;
    SC(noseg,3) ← sum
    flag ← false
end STORE_SUM

The procedure MMULT multiplies matrix B by A to obtain the product matrix C.

procedure MMULT(EA, SA, EB, SB, EC, SC) call TRANSPOSE(EB, SB, EB_T, SB_T)
    i ← p ← p.origin ← row.origin ← 1;
    sum ← 0;
    count ← 0;
    noseg ← 0;
    row ← SA(i,1);
i2 ← SA(i,2)
while i ≤ SA(0,3) do
    j ← k ← 1;
    flag ← true;
    col ← SB_T(j,1);
j2 ← SB_T(j,2)
while j ≤ SB(0,3) + 1 do
    case :
        SA(i,1) ≠ row :
        call STORE_SUM(EC, SC, last, flag, count, row, col, sum)
        i = row.origin;
p ← p.origin;
i2 ← SA(i,2)
        if SB_T(i,1) col and j2 > SB_T(j,2) then
            [k ← k + SB_T(j,3) - j2 + 1;
            j ← j + 1]
        while SB_T(j,1) = col do
            k ← k + SB_T(j,3) - SB_T(j,2) + 1;
            j ← j + 1
    end (while)
    j2 ← SB_T(j,2);
    col ← SB_T(j,1)
    if i2 > j2 then
        call STORE_SUM(EC, SC, last, flag, count, row, col, sum)
        i ← row.origin;
i2 ← SA(i,2);
p ← p.origin;
    col ← SB_T(j,1)
    if i2 > j2 then
        if i2 ≤ SB_T(j,3) then
            k ← k + i2 - j2
        if SA(i,3) > SB_T(j,3) then
            for m ← i2 to SB_T(j,3) do
                sum ← sum + EA(l) ×
                EB_T(k);
                p ← p + 1;
                k ← k + 1
            end
            j ← j + 1;
j2 ← SB_T(j,2)
else
    SA(i,3) = SB_T(j,3) then
        for m ← i2 to SA(i,3) do
            sum ← sum + EA(l) ×

925
\[\begin{align*}
    &EBT(k); \\
    &p \leftarrow p + 1; \\
    &k \leftarrow k + 1 \\
    &\text{end} \\
    &i \leftarrow i + 1; \\
    &j \leftarrow j + 1; \\
    &i2 \leftarrow SA(i,2); \\
    &j2 \leftarrow SBT(j,2); \\
    &\text{else [} \\
    &\text{for } m \leftarrow i2 \text{ to } SA(i,3) \text{ do} \\
    &\quad \text{sum} \leftarrow \text{sum} + EA(l) \times \\
    &\quad EBT(k); \\
    &\quad p \leftarrow p + 1; \\
    &\quad k \leftarrow k + 1 \\
    &\text{end} \\
    &j2 \leftarrow m; \\
    &i \leftarrow i + 1; \\
    &i2 \leftarrow SA(i,2)) \\
    &\text{else [} \\
    &\text{if } j2 > SBT(j,2) \text{ then } k \leftarrow k + \\
    &\quad SBT(j,3) - j2 + 1 \\
    &\text{else } k \leftarrow k + SBT(j,3) - \\
    &\quad SBT(j,2) + 1; \\
    &j \leftarrow j + 1; \\
    &j2 \leftarrow SBT(j,2); \\
    &\text{: } SA(i,3) \geq j2; \\
    &\text{if } SA(i,3) > SBT(j,3) \text{ then [} \\
    &\text{for } m \leftarrow j2 \text{ to } SBT(j,3) \text{ do} \\
    &\quad \text{sum} \leftarrow \text{sum} + EA(l) \times \\
    &\quad EBT(k); \\
    &\quad p \leftarrow p + 1; \\
    &\quad k \leftarrow k + 1 \\
    &\text{end} \\
    &i2 \leftarrow m; \\
    &j \leftarrow j + 1; \\
    &j2 \leftarrow SBT(j,2); \\
    &\text{else if } SA(i,3) = SBT(j,3) \text{ then [} \\
    &\text{for } m \leftarrow j2 \text{ to } SBT(j,3) \text{ do} \\
    &\quad \text{sum} \leftarrow \text{sum} + EA(l) \times \\
    &\quad EBT(k); \\
    &\quad p \leftarrow p + 1; \\
    &\quad k \leftarrow k + 1 \\
    &\text{end} \\
    &i \leftarrow i + 1; \\
    &j \leftarrow j + 1; \\
    &i2 \leftarrow SA(i,2); \\
    &j2 \leftarrow SBT(j,2); \\
    &\text{else [} \\
    &\text{for } m \leftarrow j2 \text{ to } SA(i,3) \text{ do} \\
    &\quad \text{sum} \leftarrow \text{sum} + EA(l) \times \\
    &\quad EBT(k); \\
    &\quad p \leftarrow p + 1; \\
    &\quad k \leftarrow k + 1 \\
    &\text{end} \\
    &i \leftarrow i + 1; \\
    &j2 \leftarrow m; \\
    &i2 \leftarrow SA(i,2)); \\
    &\text{else [} \\
    &\text{if } i2 > SA(i,2) \text{ then } p \leftarrow p + SA(i,3) - i2 + 1 \\
    &\text{else } p \leftarrow p + SA(i,3) - SA(i,2) + 1 \\
    &i \leftarrow i + 1; \\
    &i2 \leftarrow SA(i,2); \\
    &\text{end (case) [} \\
    &\text{if } SA(i,1) = \text{row and } i2 > SA(i,2) \text{ then [} \\
    &\quad l \leftarrow p + SA(i,3) - i2 + 1; \\
    &\text{end (case) [} \\
\end{align*}\]

\text{while } \text{SA}(i,1) = \text{row do} \\
\text{i } \leftarrow p + \text{SA}(i,3) - \text{SA}(i,2) + 1; \\
\text{i } \leftarrow i + 1 \\
\text{end} \\
\text{row_origin } \leftarrow i; \\
p_origin \leftarrow p; \\
\text{row } \leftarrow \text{SA}(i,1); \\
\text{i2 } \leftarrow \text{SA}(i,2) \\
\text{end} \\
\text{SC}(0,1) \leftarrow \text{SA}(0,1); \\
\text{SC}(0,2) \leftarrow \text{SB}(0,2); \\
\text{SC}(0,3) \leftarrow \text{noseg}; \\
\text{end MMULT} \\

c) Miscellaneous Subroutines

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.

\text{procedure PACKING}(A, n0, n1, mA, EA, SA) \\
count \leftarrow 0; \\
noseg \leftarrow 0; \\
\text{for } i \leftarrow 1 \text{ to } n0 \text{ do} \\
\text{last } \leftarrow 1 \\
\text{for } j \leftarrow 1 \text{ to } n1 \text{ do} \\
\text{if } A(i,j) \neq 0 \text{ then [} \\
\text{count } \leftarrow \text{count } + 1 \\
\text{EA(count) } \leftarrow A(i,j) \\
\text{if last } + 1 \leftarrow j \text{ then [} \\
\text{last } \leftarrow j; \\
\text{SA(noseg,3) } \leftarrow j; \\
\text{else [} \\
\text{last } \leftarrow j; \\
\text{noseg } \leftarrow \text{noseg } + 1 \\
\text{SA(noseg,1) } \leftarrow i; \\
\text{SA(noseg,2) } \leftarrow j; \\
\text{SA(noseg,3) } \leftarrow j; \\
\text{end} \\
\text{end} \\
\text{SA}(0,1) \leftarrow n0; \\
\text{SA}(0,2) \leftarrow n1; \\
\text{SA}(0,3) \leftarrow \text{noseg}; \\
\text{end PACKING} \\

\text{procedure UNPACKING}(EA, SA, A) \\
count \leftarrow 1 \\
\text{for } i \leftarrow 1 \text{ to } \text{SA}(0,3) \text{ do} \\
\text{for } j \leftarrow \text{SA}(i,2) \text{ to } \text{SA}(i,3) \text{ do} \\
\text{A(SA(i,1), j) } \leftarrow \text{EA(count)} \\
\text{count } \leftarrow \text{count } + 1; \\
\text{end} \\
\text{end} \\
\text{end UNPACKING} \]