Sparse matrix multiplication package (SMMP)*

Randolph E. Bank
Department of Mathematics C012, University of California at San Diego, P.O. Box 109, LaJolla, CA 92093, USA

and

Craig C. Douglas
Mathematical Sciences Department, IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598, USA and Department of Computer Science, Yale University, P.O. Box 2158 Yale Station, New Haven, CT 06520, USA

Received 26 May 1992

Abstract

Routines callable from Fortran and C are described which implement matrix–matrix multiplication and transposition for a variety of sparse matrix formats. Conversion routines between various formats are provided.

Keywords: Sparse matrices, matrix multiplication, transposition, numerical linear algebra.
Subject classification AMS(MOS): Numerical analysis: numerical linear algebra.

1. Introduction

The routines described here perform matrix–matrix multiplies, transposes and format conversions for sparse matrices. There are three formats supported. These include the old and new Yale sparse matrix package formats [4–6] and the more efficient one for square matrices advocated by Bank and Smith [1]. In each case, only the nonzeros are stored, not the zeros (or as few as possible).

The principal use of these routines will probably be in implementing parallel computation algorithms. For example, in one variant of the parallel multigrid [2,3], as the number of coarse grid problems per level grows, it becomes increasingly difficult to generate the coefficient matrices based on grids, as a serial multigrid solver does.

*The algorithms and routines described here were developed while both authors were visiting the Center for Applied Mathematics, Department of Mathematics, Purdue University. This research was supported in part by the Office of Naval Research, Grant No. N00014-895-1440.

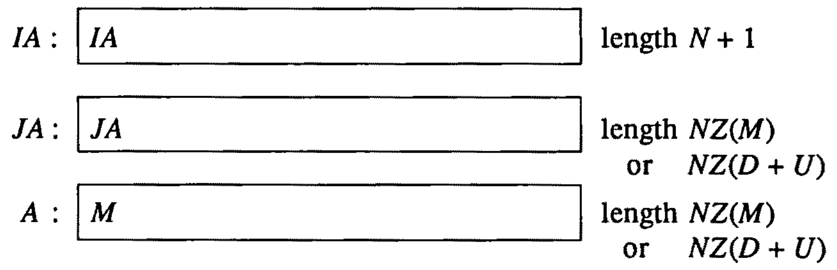
In section 2, we describe the sparse matrix formats we support. In section 3, we describe the structure of the package and the calling sequences of each routine. We describe the algorithms implemented by the package in section 4. Finally, instructions for obtaining a copy of the code are in section 5.

The routines described here differ from those in either SPARSKIT (see [9]) or the proposed sparse BLAS (see [7,8]). There is some overlap with SPARSKIT, but we are more interested in a data format not supported by SPARSKIT. The sparse BLAS have interfaces for multiplying a sparse matrix by a dense one, but not for multiplying two sparse matrices. Also, our routines are in the public domain with no restrictions, while the others are not necessarily so.

2. Sparse matrix formats

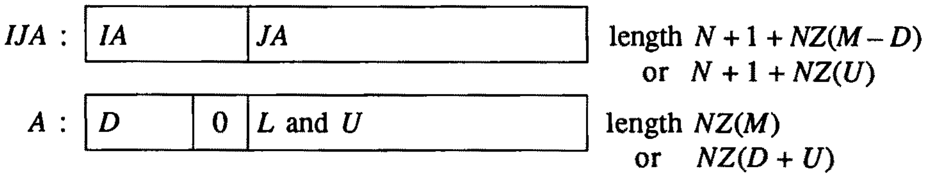
In this section, we define the three methods for storing a sparse matrix M. Let $M = D + L + U$, where D is the main diagonal part of M, L is the strictly lower triangular part of M, and U is the strictly upper triangular part of M. We define the number of nonzeros of M as $NZ(M)$.

The old Yale sparse matrix format [5,6] requires three vectors:



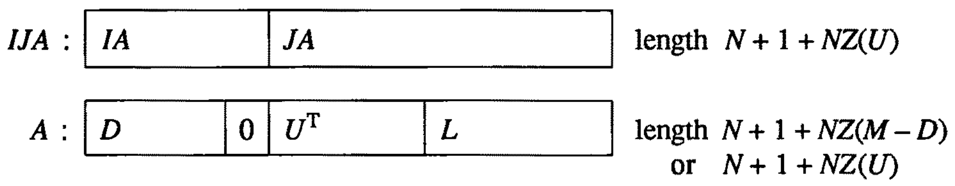
M is stored in row form. If M is symmetric, the elements of L do not have to be stored in A. The first element of row I is $A(IA(I))$. The length of row I is determined by $IA(I + 1) - IA(I)$, which is why IA requires $N + 1$ elements instead of the obvious N elements. The column indices of M are stored in the JA vector. For element $A(J)$, its column index is $JA(J)$. The elements in a row may be stored in any order.

The new Yale sparse matrix format [4] requires two vectors:



M is still stored in row form. The IA–JA vectors of the old format are combined into a single vector, sometimes referred to as an IJA vector. As before, the first element of row I is $A(IJA(I))$. In this case, the main diagonal of M is separated out from the nonzeros of L and U. The diagonal for row I is in $A(I)$. There is a zero after D so that the JA and L/U parts of the IJA and A vectors are aligned properly. Technically, the rows of A should be stored in ascending column order. However, this is not enforced.

The Bank–Smith sparse matrix format [1] requires M to be a square matrix with a symmetric (or nearly so) zero structure. It requires two vectors:



While M is stored strictly in row form, in a real sense it is stored in both column and row form. Since we assume that M has a symmetric zero structure (or L and U are padded by a small number of zeros), we need only store the row indices for U (when U is stored in column form). These are also the column indices for L (when L is stored in row form). However, we store the transpose of U in row form instead of U. If M is symmetric, the elements of L do not have to be stored in A. The first element of column I of U is $A(IA(I))$. The length of column I is determined by $IA(I+1)-IA(I)$, which is why IA requires $N+1$ elements instead of the obvious N elements. The row indices of U are stored in the JA vector. For element $A(J)$, its row index is $JA(J)$. The elements in a column must be stored in ascending row order. We define $LSHIFT$ to be 0 if M is symmetric and $IA(N+1)-IA(1)$ if M is nonsymmetric. The first element of L is $A(IA(1)+LSHIFT)$. L is stored in row format. The column index of an element $A(IA(I)+J+LSHIFT)$ is $JA(IA(I)+J)$.

For all three sparse matrix formats, we can assume there are three vectors IA, JA, and A which describe M. Except for the old Yale sparse matrix format, the vectors IA and JA are really the same vector IJA. We also need a variable $DIAGA$ which is one if the diagonal is separated from the rest of the nonzeros and zero otherwise. Last, we need a variable $SYMA$ which is one if M is stored in a symmetric manner and zero otherwise.

3. Calling sequences

In this section, we describe the five routines which comprise the package. These include two routines to multiply matrices, a routine for the transpose of a matrix, and two routines to convert between various sparse matrix formats.

For each routine in this section, the calling sequence assumes distinct IA and JA vectors for each matrix. Suppose a matrix is actually stored using an IJA vector. Then the routine should be called with IJA as an argument twice, once for each IA

and *JA*. The *IJA* vector should not be subscripted to point to either of the *IA* or *JA* parts; the routines will do this automatically.

We multiply two sparse matrices, resulting in a third:

$$C = AB.$$

Matrix–matrix multiplication is performed in two steps. These routines only support the Yale sparse matrix formats. First, the nonzero structure of the resulting matrix is determined symbolically in *SYMBMM*;

```
subroutine  SYMBMM   (N, M, L, IA, JA, DIAGA, IB, JB, DIAGB,
                      IC, JC, DIAGC, INDEX)

            integer   N, M, L, IA(*), JA(*), DIAGA,
                      IB(*), JB(*), DIAGB, IC(*), JC(*), DIAGC,
                      INDEX(*)
```

The number of rows and columns of the matrices are

matrix	rows	columns
A	N	M
B	M	L
C	N	L

INDEX is a scratch vector of length $\max\{L, M, N\}$. It is used to store linked lists. The output of *SYMBMM* is *IC* and *JC*. They are dependent on the value of *DIAGC*.

Once the nonzero structure for *C* is known, the numerical matrix–matrix multiply is computed in *NUMBMM*:

```
subroutine  NUMBMM   (N, M, L, IA, JA, DIAGA, A, IB, JB, DIAGB, B,
                      IC, JC, DIAGC, C, TEMP)

            integer   N, M, L, IA(*), JA(*), DIAGA,
                      IB(*), JB(*), DIAGB, IC(*), JC(*), DIAGC,
            real      A(*), B(*), C(*)
```

TEMP is a scratch vector of length $\max\{L, M, N\}$. It is used to store partial sums.

We may also compute the transpose of a matrix, resulting in a second:

$$B = A^{\mathrm{T}}.$$

We do this operation in *TRANSP*:

```
subroutine  TRANSP   (N, M, IA, JA, DIAGA, A, IB, JB, MOVE)
                      IC, JC, DIAGC, C, TEMP)

            integer  N, M, IA(*), JA(*), DIAGA,
                     IB(*), JB(*), MOVE
            real     A(*), B(*)
```

The number of rows and columns of the matrices are

matrix	rows	columns
A	N	M
B	M	N

We assume that B will use the same diagonal storage method that is used for A. We do not actually move the elements of A and B unless *MOVE* is one.

Finally, we have two routines for converting between one of the Yale formats and the Bank–Smith format. This makes sense only when the matrices are square. The routine *YTOBS* will convert a Yale format sparse matrix into the Bank–Smith format:

```
subroutine  YTOBS   (N, IA, JA, DIAGA, SYMA, A, IB, JB, B, MOVE)

            integer  N, IA(*), JA(*), DIAGA, SYMA,
                     IB(*), JB(*), MOVE
            real     A(*), B(*)
```

By definition, *DIAGB* must be one. Hence, we do not need it as an argument. We determine whether or not B should be stored in a symmetric or nonsymmetric manner from *SYMA*. We do not actually move the elements of A into B unless *MOVE* is one.

The routine *BSTOY* will convert a Bank–Smith format sparse matrix into one of the Yale formats:

```
subroutine  BSTOY   (N, IA, JA, SYMA, A, IB, JB, B, DIAGB, B, MOVE)

            integer  N, IA(*), JA(*), SYMA,
                     IB(*), JB(*), DIAGB, MOVE
            real     A(*), B(*)
```

We determine which of the two formats by the value of *DIAGB*. We determine whether or not B should be stored in a symmetric or nonsymmetric manner from *SYMA*. We do not actually move the elements of A into B unless *MOVE* is one.

4. Algorithms

In this section, we describe the algorithm for *SYMBMM*, *NUMBMM*, and *TRANSP*. We use a metalanguage rather than real code. One of the facets of these algorithms is their ability to work well with matrices in a variety of formats.

4.1. SYMBMM

Initialization consists of setting up the first row pointer and clearing all of the links (contained in *INDEX*):

```
1.  do $i \in \{1, ..., max\{l, m, n\}\}$ {
2.          $index_i = 0$
3.          }
4.  if ($diagc == 0$) {
5.          $ic_1 = 1$
6.          }
7.  else {
8.          $ic_1 = n + 2$
9.          }
```

INDEX is used to store links. If an entry in *INDEX* is nonzero, it is a pointer to the next column with a nonzero. The links are determined as they are found, and are unordered.

The main loop consists of three components: initialization, a long loop that merges row lists, and code to copy the links into the *JC* vector. The initialization part is as follows:

```
10. do $i \in \{1, ..., n\}$ {
11.         $istart = -1$
12.         $length = 0$
```

The start column (*istart*) is reset and the number of column entries for the *i*th row is assumed empty. The loop to merge row lists is as follows:

```
13.         do $jj \in \{ia_i, ..., ia_{i+1}\}$ {
14.                 if ($jj == ia_{i+1}$) {
15.                         if ($diaga == 0$ or
                                $i > min\{m, n\}$) {
16.                                 next $jj$
17.                                 }
18.                          $j = i$
19.                          }
20.                  else {
21.                          $j = ja_{jj}$
22.                          }
23.                  if ($index_j == 0$ & $diagb == 1$ &
                         $j \le min\{l, m\}$) {
24.                          $index_j = istart$
25.                          $istart = j$
26.                          $length = length + 1$
27.                          }
28.                  do $k \in \{ib_j, \ldots, ib_{j+1} - 1\}$ {
29.                          if ($index_{jb_k} == 0$) {
30.                                  $index_{jb_k} = istart$
31.                                  $istart = jb_k$
32.                                  $length = length + 1$
33.                                  }
34.                          } (end of $k$ loop)
35.                  } (end of $jj$ loop)
```

Lines 14–22 determine if the *jj* loop has to execute an "extra" iteration when A is stored in the new Yale sparse matrix format. Lines 23–27 add column j to the linked list. Lines 28–34 determine the intersection of this row i with the nonzeros in column j of B. Finally, we copy the links into the JC vector as the column indices:

```
36.          if ($diagc == 1$ & $index_i \neq 0$) {
37.                  $length = length - 1$
38.                  }
39.          $ic_{i+1} = ic_i + length$
40.          do $j \in \{ic_i, \ldots, ic_{i+1} - 1\}$ {
41.                  if ($diagc == 1$ & $istart == i$) {
42.                          $istart = index_{istart}$
43.                          $index_i = 0$
44.                          }
45.                  $jc_j = istart$
46.                  $istart = index_{istart}$
47.                  $index_{jc_j} = 0$
48.                  } (end of $j$ loop)
49.          $index_i = 0$
50.          } (end of $i$ loop)
```

Lines 36–38 remove the diagonal element from the row if C is stored in the new Yale sparse matrix format. Note that in lines 43 and 47, the nonzero links are cleared. Due to the small number of links (with respect to N), it would be extremely inefficient to clear the entire vector. The resulting vectors IC and JC contain the nonzero structure of $C = AB$.

4.2. NUMBMM

Initialization consists of clearing all of the partial sums:

```
1.  do $i \in \{1, \ldots, max\{l, m, n\}\}$ {
2.          $temp_i = 0$
3.          }
```

The main loop forms the partial sums and then copies the completed sums into the correct locations in the sparse matrix structure:

```
4.  do $i \in \{1, \ldots, n\}$ {
5.          do $jj \in \{ia_i, \ldots, ia_{i+1}\}$ {
6.                  if $(jj == ia_{i+1})$ {
7.                          if $(diaga == 0$ or
                               $i > min\{m, n\})$ {
8.                                  next $jj$
9.                                  }
10.                         $j = i$
11.                         $ajj = a_i$
12.                         }
13.                 else {
14.                         $j = ja_{jj}$
15.                         $ajj = a_{jj}$
16.                         }
17.                 if $(diagb == 1 \& j \le min\{l, m\})$ {
18.                         $temp_j = temp_j + ajj * b_j$
19.                         }
20.                 do $k \in \{ib_j \ldots, ib_{j+1} - 1\}$ {
21.                         $temp_{jb_k} = temp_{jb_k} + ajj * b_k$
22.                         } (end of $k$ loop)
23.                 } (end of $jj$ loop)
24.         if $(diagc == 1 \& i \le min\{l, n\})$ {
25.                 $c_i = temp_i$
26.                 $temp_i = 0$
27.                 }
28.         do $j \in \{ic_i, \ldots, ic_{i+1} - 1\}$ {
29.                 $c_j = temp_{jc_j}$
30.                 $temp_{jc_j} = 0$
31.                 } (end of $j$ loop)
32.         } (end of $i$ loop)
```

Lines 6–16 determine if the *jj* loop has to execute an "extra" iteration when A is stored in the new Yale sparse matrix format. Lines 20–22 accumulate the product for row j, and store it in lines 28–31. Lines 17–19 and 24–27 deal with special cases when a matrix is stored in the new Yale sparse matrix format. The resulting vector C contains the numerical product AB.

4.3. TRANSP

We begin by constructing IB. This requires setting up the first row pointer and counting indices for each column:

```
1.  do $i \in \{1, \ldots, m+1\}$ {
2.          $ib_i = 0$
3.          }
4.  if ($move == 1$) {
5.          do $i \in \{1, \ldots, m+1\}$ {
6.                  $b_i = 0$
7.                  }
8.          }
9.  if ($diaga == 1$) {
10.         $ib_i = m + 2$
11.         }
12. else {
13.         $ib_1 = 1$
14.         }
15. do $i \in \{1, \ldots, n\}$ {
16.         do $j \in \{ia_i, \ldots, ia_{i+1} - 1\}$
17.                 $ib_{ja_j+1} = ib_{ja_j+1} + 1$
18.                 }
19.         }
20. do $i \in \{1, \ldots, m\}$ {
21.         $ib_{i+1} = ib_i + ib_{i+1}$
22.         }
```

Lines 1–3 clear *IB*. If we are constructing *B* at the same time, then lines 4–8 clear the main diagonal of *B*. Lines 9–14 determine where the rows of *B* are stored. Lines 15–18 count the number of indices in each column and lines 20–22 convert this information into row pointers.

Next, we construct *JB*:

```
23.  do $i \in \{1, \ldots, n\}$ {
24.          do $j \in \{ia_i, \ldots, ia_{i+1} - 1\}$ {
25.                  $jj = ja_j$
26.                  $jb_{ib_{jj}} = i$
27.                  if $(move == 1)\}$
28.                          $b_{ib_{jj}} = a_j$
29.                          }
30.                  $ib_{jj} = ib_{jj} + 1$
31.                  }
32.          }
```

Lines 23–32 put *i* as a column index into row *JA*(*j*) in the first possible position (pointed to by *IB*(*jj*)) and increments the pointer. If we are constructing *B* at the same time, then lines 27–29 do the copy.

Finally, we have to restore *IB*:

```
33.  do $i \in \{m, m-1, \ldots, 2\}$ {
34.          $ib_i = ib_{i-1}$
35.          }
36.  if $(diaga == 1)$ {
37.          if $(move == 1)$ {
38.                  $j = min(n, m)$
39.                  do $i \in \{1, j\}$ {
40.                          $b_i = a_i$
41.                          }
42.                  }
43.          $ib_1 = m + 2$
44.          }
45.  else {
46.          $ib_1 = 1$
47.          }
```

Lines 34, 43, and 46 do the real work in restoring *IB*. Lines 36–42 finish copying the main diagonal of *A* when it is stored in the new Yale sparse matrix format.

5. FORTRAN source code

The FORTRAN source code for this package is freely available from Netlib as the file linalg/smmp.shar.

References

[1] R.E. Bank and R.K. Smith, General sparse elimination requires no permanent integer storage, SIAM J. Sci. Stat. Comp. 8(1987)574–584.

[2] C.C. Douglas and W.L. Miranker, Constructive interference in parallel algorithms, SIAM J. Numer. Anal. 25(1988)376–398.

[3] C.C. Douglas and B.F. Smith, Using symmetries and antisymmetries to analyze a parallel multigrid algorithm: The elliptic boundary value case, SIAM J. Numer. Anal. 26(1989)1439–1461.

[4] S.C. Eisenstat, H.C. Elman, M.H. Schultz and A.H. Sherman, The (new) Yale sparse matrix package, in: *Elliptic Problem Solvers II*, ed. G. Birkoff and A. Schienstadt (Academic Press, New York, 1984) pp. 45–52.

[5] S.C. Eisenstat, M.C. Gursky, M.H. Schultz and A.H. Sherman, Yale sparse matrix package I: The symmetric codes, Int. J. Numer. Meth. Eng. 18(1982)1145–1151.

[6] S.C. Eisenstat, M.C. Gursky, M.H. Schultz and A.H. Sherman, Yale sparse matrix package II: The nonsymmetric codes, Research Report 114, Department of Computer Science, Yale University, New Haven, CT (1977).

[7] I. Duff, M. Marrone and G. Radicati, A proposal for user level sparse BLAS, in preparation.

[8] M.A. Heroux, Proposal for a sparse BLAS toolkit, in preparation.

[9] Y. Saad, SPARSKIT: a basic tool kit for sparse matrix computations, preliminary version (1990). Available by anonymous ftp from riacs.edu.