SIAM J. COMPUT.
Vol. 28, No. 5, pp. 1576–1587

AUTOMATIC NONZERO STRUCTURE ANALYSIS*

AART J. C. BIK† AND HARRY A. G. WIJSHOFF‡

Abstract. The efficiency of sparse codes heavily depends on the size and structure of the input data. Peculiarities of the nonzero structure of each sparse matrix must be accounted for to avoid unsatisfying performance. Therefore, it is important to have an efficient analyzer that automatically determines characteristics of nonzero structures. In this paper, some efficient algorithms are presented that automatically detect particular nonzero structures.

Key words. nonzero structures, sparse computations, sparse matrices

AMS subject classifications. 68-04, 65F50

PII. S009753979529595X

1. Introduction.

Many methods have been developed that exploit the sparsity of matrices to reduce the storage requirements and computational time of particular applications (see, e.g., [9, 12, 15, 17, 20, 23]). The efficiency of each individual method, however, heavily depends on the specific characteristics of the nonzero structure of each sparse matrix. For example, although a particular band method may perform extremely well when actually applied to matrices with a small bandwidth, the explicit storage and manipulation of all elements within the band makes this method infeasible for sparse matrices in which the nonzero elements are scattered over the entire matrix. Because peculiarities of nonzero structures must be accounted for to obtain satisfactory performance, it is important to have an efficient analyzer that automatically determines certain characteristics of nonzero structures.

Such a nonzero structure analyzer can be used in a number of different fashions. First, if sparse applications are explicitly coded by hand, a nonzero structure analyzer can provide a programmer with useful insights about the characteristics of the matrices for which an application must be developed in case a representative set of sparse matrices is available beforehand. Although in this situation the efficiency of the analyzer is less important, excessively long running times would disable the analysis of large sets of matrices.

Second, in the past we have proposed a completely different approach to the development of sparse codes [1]. Rather than explicitly dealing with the sparsity of matrices at programming level, as done traditionally, this sparsity is dealt with at compilation level by a special kind of restructuring compiler, referred to as a "sparse compiler." We must refer to [1, 3, 4, 5] for a detailed presentation of this approach and some preliminary experiments with a prototype sparse compiler. It is obvious, however, that the automatically generated sparse code becomes more efficient if the sparse compiler can account for characteristics of the nonzero structures. For this,

*Received by the editors December 13, 1995; accepted for publication (in revised form) July 1, 1997; published electronically April 27, 1999. This research was supported by the Foundation for Computer Science (SION) of the Dutch Organization for Scientific Research (NWO) and the EC Esprit Agency DG XIII under grant APPARC 6634 BRA III. A preliminary version of this paper appears as "Nonzero structure analysis," in *ICS '94. Proc. 8th International Conference on Supercomputing*, July 11–15, 1994, Manchester, UK, ACM, New York, 1994, pp. 226–235.

http://www.siam.org/journals/sicomp/28-5/29595.html

†Intel Corp., 2200 Mission College Blvd., SC12-303, Santa Clara, CA 95052 (aart.bik@intel.com).

‡Computer Science Department, Leiden University, Niels Bohrweg 1, 2333 CA Leiden, The Netherlands (harryw@cs.leidenuniv.nl).

the compiler requires an automatic nonzero structure analyzer. Since in this approach analysis time contributes to compile time, it is again desirable to keep analysis time limited.

In most practical cases, however, sparse matrices are not all available beforehand. A possible approach to deal with this situation is to generate multiple versions of an application (either explicitly by hand or automatically by means of a sparse compiler), each of which has been optimized specifically for a particular class of nonzero structures. At run-time, the analyzer is invoked to determine which version is probably the most efficient. This version is subsequently executed. Using run-time analysis has the major advantage that nonzero structures do not have to be known in advance. In this case, however, the analyzer must be very efficient to avoid the situation in which savings in execution time using an optimized version are outweighed by analysis time. In general, it is desirable to keep analysis time proportional to the number of nonzero elements and order of a sparse matrix [11].

In this paper, some efficient algorithms are presented that can be used by an analyzer to automatically detect particular nonzero structures of square sparse matrices. The presented algorithms examine each matrix as it is; i.e., no attempts are made to permute the matrix into a particular form (as is frequently done in the context of LU-factorization, for example, to confine fill-in to certain regions in the matrix [6, 7, 8, 10, 12, 14, 15, 16, 17, 22]). Even if such a permutation is applied to the matrix, however, the analyzer can be used afterwards to determine whether an unforeseen nonzero structure arises (information about the specific form for which the permutation is intended is usually obtained as a side effect of computing the permutation).

2. Nonzero structures. We can distinguish between general sparse matrices and sparse matrices having a particular nonzero structure. In this section, some important nonzero structures of square matrices are identified [12, 18, 19, 20, 21].

2.1. Band forms. The *lower* and *upper semibandwidth* of an $N \times N$ matrix A are defined as the smallest integers $b_l \geq 0$ and $b_u \geq 0$, respectively, for which $(a_{ij} \neq 0) \Rightarrow (-b_u \leq i - j \leq b_l)$ still holds. Minimum values reveal the most information about the nonzero structure, because this constraint is trivially satisfied for semibandwidths $N - 1$. Allowing for negative semibandwidths would enable the specification of an arbitrary band in which the main diagonal is not necessarily included. However, in this paper we will assume that all matrices have a full transversal (i.e., all elements on the main diagonal are nonzero).

If the semibandwidths are relatively small, we say that the matrix is in *band form*, which means that all nonzero elements are confined to a small band. We define the shape count of a band form as the total number of elements that reside within the band:

$$N \cdot (b_l + b_u + 1) - (b_l^2 + b_l)/2 - (b_u^2 + b_u)/2.$$

Note that this number is likely to exceed the total number of nonzero elements, because the band is not necessarily full.

2.2. Block forms. Consider a block partition of a square matrix A into submatrices A_{ij}:

$$A = \begin{pmatrix} A_{11} & \dots & A_{1p} \\ \vdots & \ddots & \\ A_{p1} & & A_{pp} \end{pmatrix}.$$

Each submatrix A_{ii}, referred to as a *diagonal block*, is a square $n_i \times n_i$ submatrix. Hence, each submatrix A_{ij} with $i \neq j$, referred to as an *off-diagonal block*, is an $n_i \times n_j$ submatrix. The off-diagonal blocks A_{pi} and A_{ip} for $1 \leq i < p$ are referred to as the *lower border* and *upper border*, respectively. If a block A_{ij} contains at least one nonzero element, the block is called a *nonzero block*, denoted by $A_{ij} \neq 0$.

If $(A_{ij} \neq 0) \Rightarrow (i = j)$, then the matrix is in *block diagonal form*. Likewise, if $(A_{ij} \neq 0) \Rightarrow (i \geq j)$, or if $(A_{ij} \neq 0) \Rightarrow (i \leq j)$, then the matrix is in *block lower triangular form* or *block upper triangular form*, respectively. If a matrix is in block diagonal form except for some nonzero blocks in the borders, then the matrix is in *doubly bordered block diagonal form*. Matrices in *singly bordered block lower triangular form* or *singly bordered block upper triangular form* are defined likewise. Finally, if $p = 2$, $A_{21} \neq 0$, $A_{12} \neq 0$, and A_{11} is in band form, we say that the matrix is in *doubly bordered band form*.

For these forms, the shape count is defined as the total number of elements in the diagonal blocks (but only counting elements in the band of A_{11} for a bordered band form), *all* border blocks for the bordered forms, and, for the triangular forms, *all* remaining off-diagonal blocks in the lower or upper triangular part, even if not all these blocks are nonzero. Again, this number is likely to exceed the total number of nonzero elements, since the blocks are not necessarily full (and some of them may even be zero).

Although, depending on which blocks are nonzero, a particular form of a matrix is defined once a block partition of that matrix is given, it is possible that *different* block partitions into one particular form differ in the accuracy of describing the nonzero structure. In Figure 2.1, for example, two different block partitions of a matrix into block diagonal form are shown with shape counts 15 and 25, respectively. Therefore, we say the most accurate description for a particular form is defined by a *minimum block partition into that form*, which means that there are no other block partitions of the matrix into the same form with a smaller shape count.

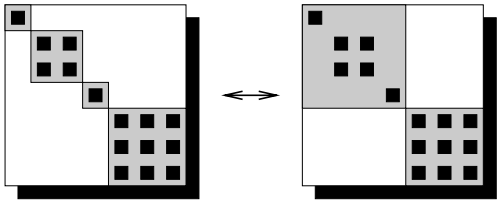

FIG. 2.1. *Two different block partitions into BDF.*

A square matrix has a unique minimum block partition into block diagonal form, which satisfies the following property (similar statements hold for block lower or upper triangular forms).

PROPOSITION 2.1. *A block partition of a square matrix into block diagonal form is minimum if and only if there is no diagonal block with a nontrivial block partition into block diagonal form.*

We will see that a matrix can have several minimum block partitions into a bordered band or block form.

3. Automatic nonzero structure analysis. In this section, efficient algorithms are presented that detect (bordered) band and (bordered) block forms. We assume that the nonzero structure of each $N \times N$ sparse matrix A to be analyzed is available on file in coordinate scheme. In this scheme, the file consists of the order N and an integer nnz that indicates the number of nonzero elements, followed by an unordered set of nnz triples (i, j, a_{ij}), indicating the row index, column index, and value of each individual nonzero element.

First, sky-line information is computed. Thereafter, this information is used to detect particular nonzero structures.

3.1. Preparatory analysis. Sky-line information, i.e., the lower and upper semibandwidth for each single row and column, respectively, can be obtained in a single pass over a file by executing the following procedure. In this fragment, the lower and upper sky-line are computed in the arrays `lsky` and `usky`, respectively:

```
procedure comp_skylines()
begin
  read(N, nnz);

  allocate lsky[1:N] and usky[1:N]
  for i := 1, N do
    lsky[i] := 0;
    usky[i] := 0;
  endfor

  for k := 1, nnz do
     read(i, j, aij);
     lsky[i] := max(lsky[i], (i-j));
     usky[j] := max(usky[j], (j-i));
  endfor
end
```

Sky-lines are computed under the assumption that the matrix has a full transversal, so that all elements of the arrays `lsky` and `usky` can be initialized to zero. All information requires $\Theta(N)$ storage and can be obtained in $\Theta(nnz + N)$ time.

Example. The following lower and upper sky-lines are obtained for the 15×15 sparse matrix that is depicted in Figure 3.1:

	1														15
lsky	0	0	1	0	4	0	1	0	1	0	0	0	0	0	3
usky	0	0	2	0	1	0	0	0	3	0	0	0	0	3	2

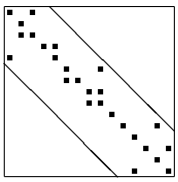

FIG. 3.1. *Band form.*

3.2. Band forms. Once the lower and upper sky-lines of a matrix have been computed, the lower and upper semibandwidths of this matrix are determined in $\Theta(N)$ time as follows:

```
procedure comp_bandform()
begin
  b_l := 0; b_u := 0;
  for i := 1, N do
    b_l := max(b_l, lksy[i]);
    b_u := max(b_u, uksy[i]);
  endfor
end
```

These semibandwidths directly determine the band form of a matrix.

Example. For the sparse matrix of Figure 3.1, the semibandwidths $b_l = 4$ and $b_u = 3$ result. This gives rise to the band form with shape count 104 that is shown in the same figure.

3.3. Bordered band forms. By slightly extending the previous procedure, an algorithm is derived that constructs a minimum block partition into bordered band form in $\Theta(N)$ time:

```
procedure comp_bord_bandform()
begin
  b_l := 0; b_u := 0;
  tsz := N*N;

  for i := 1, N do
    b_l := max(b_l, lksy[i]);
    b_u := max(b_u, uksy[i]);
    if (bf_size(b_l, b_u, N-i) <= tsz) then
      t_l := b_l; t_u := b_u; b := N-i;
      tsz := bf_size(b_l, b_u, N-i);
    endif
  endfor
end
```

In this algorithm, the auxiliary function `bf_size` computes the shape count of a bordered band matrix with border size `b` and semibandwidths `bl` and `bu`:

```
integer function bf_size(bl, bu, b)
begin
  return ( (N-b) * (bl+bu+1) - (bl*bl+bl) / 2
           - (bu*bu+bu) / 2 + 2*N*b - b*b );
end
```

After the semibandwidths are updated in each step `i`, we test whether the shape count of a bordered band form with border size `N-i` is less than or equal to the best shape count seen so far. If this is true, we record this shape count and corresponding border size and semibandwidths. Consequently, after applying the algorithm, variables `b`, `t_l`, and `t_u` contain the border size and semibandwidths of a *minimum* block partition into block band form. If `b=0` holds, then effectively a band form results. For example, applying `comp_bord_bandform()` to the matrix of Figure 3.1 yields exactly the same band form as computed by `comp_bandform()`.

Example. In Figure 3.2, we present the resulting bordered band forms for some matrices, with shape counts 125, 119, 153, and 131, respectively. The last example shows that, although a minimum block partition into a *doubly* bordered band form is constructed, it is possible that only a single border block is actually nonzero.

Example. A matrix may have different minimum block partitions into bordered band form, as illustrated in Figure 3.3, where the shape count of all forms is 93. Our algorithm solves such ties in favor of the smallest border size.

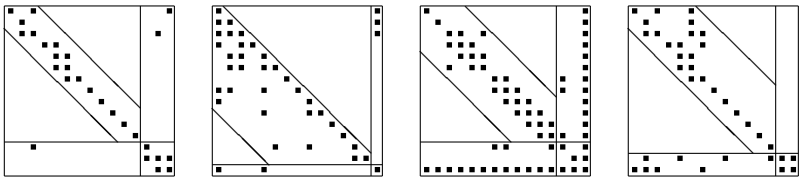

FIG. 3.2. *Bordered band forms.*

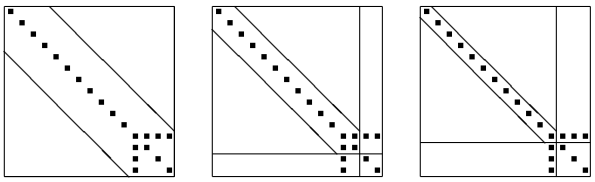

FIG. 3.3. *Different minimum block partitions into bordered band forms.*

3.4. Block forms. The following procedure constructs a block partition into diagonal block form in $\Theta(N)$ time by determining the size `k` of each next diagonal block with the lower right corner at row and column index `B` during a backward scan over the sky-lines:

```
procedure comp_blockdiag()
begin
  p := 0; k := 1; B := N;
  for i := N, 1, -1 do
S1: k := max(k, max(lsky[i],usky[i])+B-i+1);
    if (i = B-k+1) then
       p := p + 1;  part[p] := i;  /* Record Block */
       B := i - 1;        k := 1:  /* Next Block   */
    endif
  endfor
end
```

After application of this algorithm, `p` contains the number of diagonal blocks. The row (or column) indices of the upper left corners of all diagonal blocks of the block partition are recorded in reverse order in the first `p` locations of array `part`.

The following proposition states that the *minimum* block partition into block diagonal form is found. Likewise, if only the value `lsky[i]` or `usky[i]` is used in statement `S1`, then the minimum block partition into block lower, or block upper triangular form, respectively, is obtained in $\Theta(N)$ time.

PROPOSITION 3.1. *Application of* `comp_blockdiag()` *to the lower and upper sky-line of a matrix yields the minimum block partition into block diagonal form.*

Proof. By construction, each nonzero element is incorporated in a diagonal block. Now assume that the resulting block partition is not a minimum block partition into diagonal form. Then Proposition 2.1 implies that there is a certain $k \times k$ diagonal block with the lower right corner at a row and column index B that has a nontrivial

block partition into block diagonal form; i.e., there is $1 \leq k' < k$ such that $\forall B - k' < i \leq B : \max(l_i, u_i) + (B - i) < k'$. Since no diagonal block is recorded during iterations $\texttt{i} = B \ldots B - k' + 1$, during at least one of these iterations a value is assigned to `k` that is greater than k'. However, this can only occur if $\max(l_i, u_i) + B - i + 1 > k'$ for some $B - k' < i \leq B$. This contradicts the assumption.

Example. Application of the different versions of this algorithm to the matrix of Figure 3.1 yields the block diagonal, block lower, and upper triangular forms shown in Figure 3.4, with shape counts 67, 140, and 138, respectively. Note that although in the block triangular forms many off-diagonal blocks are zero, the elements of these blocks are also included in the shape count.

The contents of array `part` for the first block partition are shown below:

part	11	10	6	1

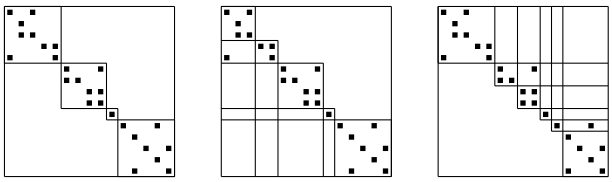

FIG. 3.4. *Block forms.*

3.5. Bordered block forms. Let $E(b)$ denote the shape count of a bordered block diagonal form with border size $b \in [0, N]$ arising from the minimum block partition of the remaining $(N - b) \times (N - b)$ matrix into block diagonal form. We define the improvement of using border size b' instead of b as $I(b', b) = E(b) - E(b')$, satisfying the following property.

PROPOSITION 3.2. *For $b, b', b'' \in [0, N]$, we have $I(b', b'') = I(b', b) + I(b, b'')$.*

Proof. $I(b', b'') = E(b) - E(b') + E(b'') - E(b) = I(b', b) + I(b, b'')$.

Suppose that for a given border size $b \in [0, N]$, we construct the minimum block partition of the remaining $(N - b) \times (N - b)$ submatrix into block diagonal form using the procedure `comp_diagblock()`. At any iteration $\texttt{i} = i$, the block partition found so far may be discarded and the algorithm may be restarted with $B = i - 1$ and $k = 1$ for a new border size $b' = N - i + 1$.

Obviously, selection of this border is only profitable if eventually we are able to determine that $I(b', b) > 0$. However, rather than constructing both block partitions completely, we are already able to compute the improvement during an iteration $\texttt{i} = i'$ in which the last diagonal block of the new block partition that overlaps with the diagonal block that was assumed during iteration $\texttt{i} = i$ has been found. This is *because the block partition of the remaining part of the matrix will be identical for both block partitions.* This new diagonal block may be contained in the old diagonal block (which occurs if the value of `k` would not have been incremented while constructing the old block partition), or these blocks may partially overlap.

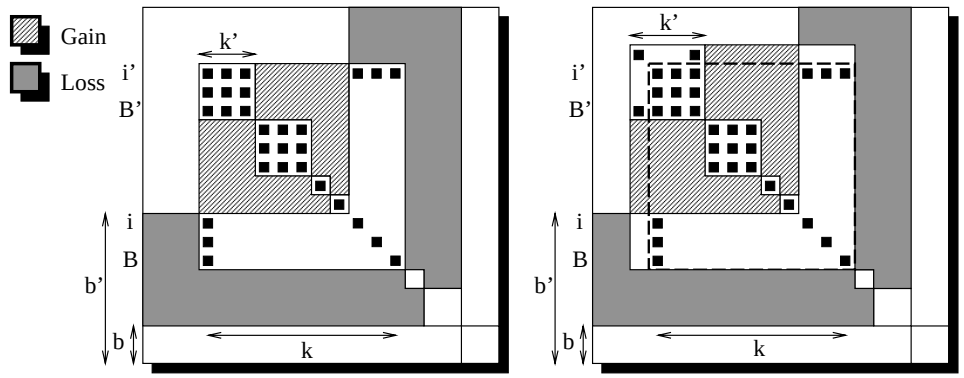

FIG. 3.5. *Gain and loss for border.*

Both cases are illustrated in Figure 3.5. In any case, the improvement is equal to the difference of the number of elements included in the border (*loss*) and the number of elements that do not have to be included in a diagonal block (*gain*). Let B, k and B', k' denote the value of `B` belonging to the iterations `i` $= i$ and `i` $= i'$, respectively. Furthermore, let Z and Z' denote the number of elements in the off-diagonal zero blocks of the old and new block partition below rows B and B', respectively. Then the improvement of using a new border size b' with respect to the old border size b is equal to the difference between the gain and the loss:

$$I(b', b) \quad = \quad Z' - Z - 2 \cdot (B' - k')(B - B').$$

If the gain exceeds the loss, i.e., $I(b', b) > 0$, then it is profitable to continue with the new block partition and border size b'. Moreover, border size b may be discarded from further consideration, since Proposition 3.2 implies that $I(b', b'') > I(b, b'')$ for all $b'' \in [0, N]$. If no improvement has been obtained, i.e., $I(b', b) \leq 0$, then the block partition corresponding to border size b must be restored and the algorithm can proceed with the search for the next diagonal block (which has at least size $\max(k, B - B' + k')$). In that case, we may discard border size b' from further consideration, since Proposition 3.2 implies that $I(b', b'') \leq I(b, b'')$ for all $b'' \in [0, N]$.

These observations enable us to construct a minimum block partition into block diagonal form in one pass over the sky-lines. At each step in which no diagonal block is recorded, the current status is saved on a stack, and a new border size is tried. If a diagonal block is recorded, no improvement can be obtained by trying a new border size. Instead, previously constructed block partitions belonging to smaller border sizes that can be verified are restored if an improvement is obtained (which is simply done by restoring the value of `p`), or discarded otherwise. The following slightly more complex version of procedure `comp_blockdiag()` results:

```
procedure comp_bord_blockdiag()
begin
  Z := 0; b := 0; s := 0;
  p := 0; k := 1; B := N;
  for i := N, 1,  -1 do
S2: k := max(k, max(lsky[i],usky[i])+B-i+1);
    if (i = B-k+1) then
       /* Last Overlapping Block? */
       while ( (s > 0) && (i == stackB[s]-new_k()+1) ) do /* Conditional AND */
          /* Improvement? */
          if (I() > 0) then
            s := s - 1;            /* Discard         */
          else
            pop_restore();         /* Restore         */
          endif
       endfor
       Z := Z + 2 * k * (B-k);
       p := p + 1;  part[p] := i;  /* Record Block    */
       B := i - 1;        k := 1:  /* Next Block      */
    else
       push();                     /* Save State      */
       Z := 0; B := i - 1;         /* New Search      */
       k := 1; b := N - i + 1;
    endif
  endfor
end
```

In this algorithm, the following auxiliary procedures are used to implement stack-like operations that save and restore states:

```
procedure push()
begin
  s := s + 1;
  stackk[s] := k;
  stackZ[s] := Z;
  stackB[s] := B;
  stackp[s] := p;
  stackb[s] := b;
end
```

```
procedure pop_restore()
begin
  k := new_k();
  Z := stackZ[s];
  B := stackB[s];
  p := stackp[s];
  b := stackb[s];
  s := s - 1;
end
```

The following auxiliary functions are used to compute the improvement and the new value of `k` for the block partition on top of the stack:

```
integer function I()
begin
 I := Z - stackZ[s] - 2 *
      (B-k) * (stackB[s]-B);
end
```

```
integer function new_k()
begin
  new_k := max(stackk[s],
               stackB[s]-B+k);
end
```

Although a while-loop occurs inside the `i`-loop, this algorithm still runs in $\Theta(N)$ time because each border size can only be pushed and popped from the stack once. Because the algorithm simply applies `comp_blockdiag()` to the submatrix that remains for the most profitable border size, it is clear that this extended algorithm constructs a *minimum* block partition into bordered block diagonal form.

After application of this algorithm, the scalar `b` contains the selected border size

(and hence the size of the last diagonal block). The first `p` locations of array `part` represent the block partition into block diagonal form of the remaining submatrix. If a zero border size is selected, the last diagonal block is empty and, effectively, a block partition into block diagonal form results.

If only the value `lsky[i]` or `usky[i]` is used in statement `S2`, then a minimum block partition into, respectively, singly bordered block lower or upper triangular form is obtained. In these cases, the constant 2 must be removed from the assignment to `Z` and the computation in function `I()` to compute the appropriate improvement.

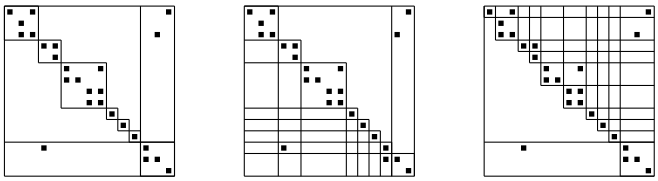

FIG. 3.6. *Bordered block forms.*

Example. In Figure 3.6, the bordered block forms that result for a matrix are shown, having shape counts $225 - 112 = 113$, $225 - 68 = 157$, and $176 - 14 = 162$, respectively. The contents of array `part` for the bordered block diagonal form having `b=3` are shown below:

part	12	11	10	6	4	1

Example. Applying the version operating only on `lsky[i]` to the matrix with 22 nonzero elements of Figure 3.7 yields a minimum partition into bordered block upper triangular form with border size 1 and shape count 166. However, the shape counts of similar forms with border sizes 2 and 3 are also 166. This example illustrates that a matrix may have different minimum block partitions into a particular bordered block form. Because a border is denied for a zero improvement (viz., $I(3,2) = 0$ and $I(2,1) = 0$ in the example), ties are solved in favor of the smallest border size.

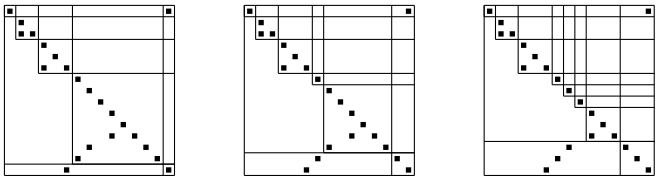

FIG. 3.7. *Different minimum block partitions into BBUTF.*

3.6. Classification. After the minimum block partitions into (doubly bordered) band form, (doubly bordered) block diagonal form, and (singly bordered) block lower and upper triangular forms have been constructed, shape counts can be used to determine which form most accurately describes the nonzero structure of a matrix. Moreover, the density of this form (i.e., *nnz* divided by the shape count) can be compared with a user-defined threshold to decide whether this nonzero structure is

used for the classification of the sparse matrix or whether the matrix is classified as a general sparse matrix.

Example. For the 59×59 matrix "impcol_b" of the Harwell–Boeing sparse matrix collection [13] with 312 nonzero elements, for example, the shape counts of the different forms are 2620, 3461, 3206, and 2930, respectively. In Figure 3.8, we show the band form and bordered block upper triangular form. Here, we classify this matrix as a band matrix if 312/2620 exceeds the threshold, or as a general sparse matrix otherwise.

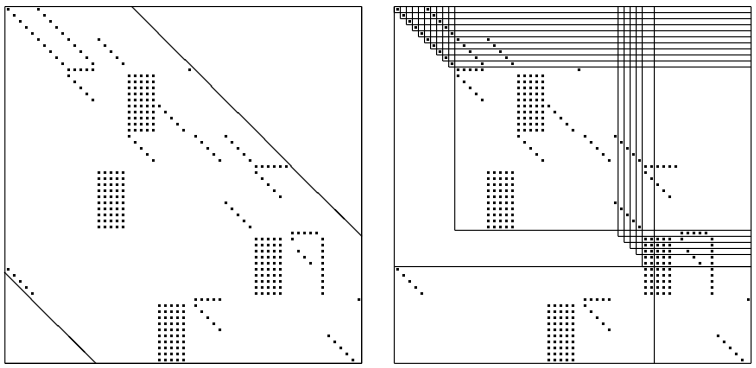

FIG. 3.8. *Classification of "impcol_b."*

4. Conclusions. In this paper, we have presented some efficient algorithms that automatically detect particular nonzero structures of sparse matrices. First, we have extended an algorithm that constructs a band form into an algorithm that constructs a minimum block partition into bordered band form (possibly yielding a band form as a special case). Likewise, an algorithm that constructs a minimum block partition into block diagonal or triangular form has been extended into an algorithm that constructs a minimum block partition into bordered block diagonal or triangular form (possibly yielding a block form with an empty border as a special case). All algorithms require only sky-line information, which can be obtained in $\Theta(N + nnz)$ time for an $N \times N$ sparse matrix with nnz nonzero elements, and have a running time of $\Theta(N)$.

REFERENCES

[1] A. J. C. BIK, *Compiler Support for Sparse Matrix Computations*, Ph.D. thesis, Department of Computer Science, Leiden University, Leiden, The Netherlands, 1996.

[2] A. J. C. BIK AND H. A. G. WIJSHOFF, *Nonzero structure analysis*, in ICS '94. Proc. 8th International Conference on Supercomputing, July 11–15, 1994, Manchester, UK, ACM, New York, 1994, pp. 226–235.

[3] A. J. C. BIK AND H. A. G. WIJSHOFF, *Advanced compiler optimizations for sparse computations*, J. Parallel Distrib. Comput., 31 (1995), pp. 14–24.

[4] A. J. C. BIK AND H. A. G. WIJSHOFF, *Automatic data structure selection and transformation for sparse matrix computations*, IEEE Trans. Parallel Distrib. Systems, 7 (1996), pp. 109–126.

[5] A. J. C. BIK AND H. A. G. WIJSHOFF, *The use of iteration space partitioning to construct representative simple sections*, J. Parallel Distrib. Comput., 34 (1996), pp. 95–110.

[6] T. F. COLEMAN, *Large Sparse Numerical Optimization*, Lecture Notes in Comput. Sci. 165, Springer-Verlag, Berlin, 1984.

[7] E. CUTHILL, *Several strategies for reducing the bandwidth of matrices*, in Sparse Matrices and Their Applications, D. J. Rose and R. A. Willoughby, eds., Plenum Press, New York, 1972, pp. 157–166.

[8] E. Cuthill and J. Mckee, *Reducing the bandwidth of sparse symmetric matrices*, in Proc. 24th National Conference of the ACM, 1969, pp. 157–172.
[9] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst, *Solving Linear Systems on Vector and Shared Memory Computers*, SIAM, Philadelphia, 1991.
[10] I. S. Duff, *A survey of sparse matrix research*, Proc. IEEE, (1977), pp. 500–535.
[11] I. S. Duff, *A sparse future*, in Sparse Matrices and Their Uses, I. S. Duff, ed., Academic Press, London, 1981, pp. 1–29.
[12] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices*, Oxford Science Publications, Oxford, 1990.
[13] I. S. Duff, R. G. Grimes, and J. G. Lewis, *Sparse matrix test problems*, ACM Trans. Math. Software, 15 (1989), pp. 1–14.
[14] A. George and J. W. H. Liu, *An implementation of a pseudoperipheral node finder*, ACM Trans. Math. Software, 5 (1979), pp. 284–295.
[15] A. George and J. W. H. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1981.
[16] K. J. Mann, *Inversion of large sparse matrices: Direct methods*, in Numerical Solutions of Partial Differential Equations, J. Noye, ed., North-Holland, Amsterdam, 1982, pp. 313–366.
[17] S. Pissanetsky, *Sparse Matrix Technology*, Academic Press, London, 1984.
[18] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes*, Cambridge University Press, Cambridge, 1986.
[19] R. P. Tewarson, *Sorting and ordering sparse linear systems*, in Large Sparse Sets of Linear Equations, J. K. Reid, ed., Academic Press, London, 1971, pp. 151–167.
[20] R. P. Tewarson, *Sparse Matrices*, Academic Press, New York, 1973.
[21] M. Veldhorst, *An Analysis of Sparse Matrix Storage Schemes*, Ph.D. thesis, Mathematisch Centrum, Amsterdam, 1982.
[22] R. Wait, *The Numerical Solution of Algebraic Equations*, Wiley, Chichester, 1979.
[23] Z. Zlatev, *Computational Methods for General Sparse Matrices*, Kluwer, Dordrecht, 1991.