Nonzero Structure Analysis*

Aart J.C. Bik and Harry A.G. Wijshoff
High Performance Computing Division
Department of Computer Science, Leiden University
P.O. Box 9512, 2300 RA Leiden, the Netherlands
{ajcbik,harryw}@cs.leidenuniv.nl

Abstract

Because the efficiency of sparse codes is very much dependent on the size and structure of input data, peculiarities of the nonzero structures of sparse matrices must be accounted for in order to avoid unsatisfying performance. Usually, this implies retargeting a sparse application to specific instances of the same problem. However, if characteristics of the input data are collected at compile-time and used in the data structure selection and code generation by a compiler that converts dense programs into sparse programs automatically, the complexity of sparse code development can be greatly reduced, and an efficient way for this retargeting results. Such a 'sparse compiler' requires an analysis engine, which is the topic of this paper.
Index Terms: Nonzero Structures, Restructuring Compilers, Sparse Matrices.

1 Introduction

Automatic analysis of nonzero structures of sparse matrices is useful for a number of reasons. First, it can provide a programmer with useful insights about characteristics of a range of sparse matrices for which an application must be developed, so that these characteristics can be accounted for. This requires, of course, access to a representative set of sparse matrices. Second, an analysis engine is very helpful for a 'sparse compiler', proposed in [4, 6]. This compiler automatically transforms a program that operates on dense matrices into a program that exploits the sparsity of certain matrices. In this manner, the burden of sparse code generation is placed on the compiler, while the programmer only has to deal with the dense code that is both qualitative and quantitative much simpler [8]. Usually more optimizations are enabled because, in general, information that is obtained by analysis of dense codes is much more accurate [3]. Clearly, the characteristics of the sparse matrices must drive such a compiler. An analysis engine is required to obtain nonzero structure information that can be used in the data structure selection and code generation. Since analysis time contributes to compile-time, it is important to have an efficient analyzer. Finally, once the techniques of the sparse compiler have been well developed, a more powerful approach can be taken. The compiler generates multiple versions of the code that have been optimized for several nonzero structures that are likely to occur, and inserts the analysis code in the resulting program. At run-time, the outcome of the analysis determines which version of the code is the most suited for a particular matrix. This approach has as major advantage that the nonzero structure of a matrix does not have to be known at compile-time. However, since analysis is performed at run-time, it is even more important to have an efficient analyzer, since no gain is obtained, if the savings in execution time in an optimized version are outweighted by the analysis time.

In the context of LU-factorization, a priori orderings, usually expressed in terms of a relabeling on the underlying graph [9, 14], are often applied to matrices in order to confine so-called fill-in to certain regions in the matrix. For example, application of Tarjan's algorithm [22] can be used to obtain a block triangular form [9], while application of the well-known Cuthill-McKee algorithm [7] is often used to reduce the bandwidth of a matrix. So-called local strategies, like application of the Markowitz criterion [9], try to meet the minimum fill-in objective by selection of a pivot at each stage for which some fill-in related properties are satisfied. These methods, however, are not considered in this paper, i.e. we merely try to detect a certain structure in a (possibly reordered) matrix. A necessary extension of the sparse compiler, however, must deal with reorderings, for example, by automatic insertion of such routines. This makes all sparsity related issues completely transparent to the programmer.

In this paper we present some efficient analysis techniques that have been implemented for the prototype compiler MT1 [5], which is currently used to implement the techniques of automatic sparse code generation. In section 2, important characteristics of nonzero structures are identified, followed by a discussion in section 3 of how some of these nonzero structures can be detected efficiently. In section 4 the implementation of these techniques is discussed, illustrated with some compile-time techniques that become possible if nonzero structure information is available. Finally, conclusions and topics for further research are stated in section 5.

*Support was provided by the Foundation for Computer Science (SION) of the Netherlands Organization for the Advancement of Pure Research (NWO) and the EC Esprit Agency DG XIII under Grant No. APPARC 6634 BRA III. THIS PAPER HAS BEEN ACCEPTED FOR ICS 94.

2 Nonzero Structures

In this section, we explore some important characteristics of the nonzero structure of a sparse $n \times n$ matrix A, i.e. $Nonz_A = \{(i,j) | a_{ij} \neq 0\}$ (see e.g. [9, 18, 23, 24, 25]).

2.1 Band Forms

The **band** form of a matrix is defined by the **lower** and **upper semi-bandwidth**, which are two integers $b_1 \geq 0$ and $b_2 \geq 0$ that contain the *minimum* values for which constraint $(a_{ij} \neq 0) \Rightarrow (-b_1 \leq j - i \leq b_2)$ is satisfied,[1] which means that all nonzeros are confined to a band. If $b_1 = b_2$ holds, $b_1 + b_2 + 1$ is referred to as the **bandwidth** of this band. Although the semi-bandwidths are defined for arbitrary matrices (e.g. $b_1 = b_2 = n - 1$ for a full matrix), matrices are usually only called band matrices if the semi-bandwidths are relatively small. If $(-b_1 \leq j - i \leq b_2) \Rightarrow (a_{ij} \neq 0)$ also holds, the matrix is a full band matrix [24], while zero elements might appear in the band otherwise. Some special kinds of band matrices can be distinguished. If $b_1 = b_2 = 0$, the matrix is in **diagonal** form, while for $b_1 = b_2 = 1$ the matrix is called **tridiagonal**. A form that is closely related to storage formats, consists of the so-called **variable band matrices** [9, 13, 16, 17], where lower and upper semi-bandwidths are determined per diagonal element, which gives rise to a skyline as defined in section 3.1.

2.2 Triangular Forms

A matrix is in **lower triangular** form, if $(a_{ij} \neq 0) \Rightarrow (j \leq i)$. For a *unit* lower triangular matrix, $a_{ii} = 1$ for all $1 \leq i \leq n$ hold as well. If $(a_{ij} \neq 0) \Rightarrow (j < i)$ holds, the matrix is called **strictly lower triangular**. A lower triangular matrix can be seen as a special band matrix for $b_2 = 0$ and relatively large $b_1 > 0$ (note that $b_1 = n - 1$ does not necessarily hold if the triangular part is not full). For relatively small $b_2 > 0$ the matrix is in so-called **band lower triangular** form. The corresponding upper triangular forms are defined similarly.

2.3 Block Forms

Consider a partition of a square matrix A into submatrices A_{ij} for $1 \leq i \leq p$ and $1 \leq j \leq p$. Each A_{ii} is an $n_i \times n_i$ matrix, and is referred to as a **diagonal block**. The sizes n_i of these diagonal blocks completely determine the partition of the whole matrix: each A_{ij} is an $n_i \times n_j$ submatrix. Submatrices A_{ij} for $i \neq j$ are called **off-diagonal blocks**. If a block only consists of zero elements, this is denoted by $A_{ij} = 0$. Such blocks are referred to as **zero blocks**.

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

Given this partition, a **block banded** form can be defined with semi-bandwidths $B_1 \geq 0$ and $B_2 \geq 0$, which are the *minimum* values for which $(A_{ij} \neq 0) \Rightarrow (-B_1 \leq j - i \leq B_2)$ is satisfied. Consequently, a **block tridiagonal** form results if $B_1 = B_2 = 1$ holds, and a **block diagonal** form results if $B_1 = B_2 = 0$. Similarly, a **block lower triangular** or **block upper triangular** form is defined, if $(A_{ij} \neq 0) \Rightarrow (j \leq i)$ or $(A_{ij} \neq 0) \Rightarrow (i \leq j)$ holds respectively. 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 nonzero blocks occur in the borders, this gives rise to **singly bordered diagonal**, **doubly bordered diagonal**, and **bordered block triangular** forms.

Although, depending on which blocks are nonzero, the block form of a matrix is defined once a partition of that matrix is given, block forms corresponding to different partitions might differ in the accuracy of description of the nonzero structure. As an extreme example, a matrix is in any block form if the trivial partition $A = A_{11}$ is used. The most accurate description is obtained from a block form that is defined by a *minimum partition into that particular block form*, which means that there are no other partitions of the matrix into that block form for which the corresponding nonzero blocks have a smaller total area. For example, the areas of the nonzero blocks of a partition into block diagonal and block tridiagonal are determined by the following formulae:

$$\text{BDF: } \sum_{i=1}^{p} n_i^2 \quad \text{BTF: } \sum_{i=1}^{p} n_i^2 + 2 \cdot \sum_{i=1}^{p-1} n_i \cdot n_{i+1}$$

These numbers are likely to exceed the number of nonzero elements, since even in a minimum partition the nonzero blocks are not necessarily full.

Proposition 1 A square matrix has a *unique* minimum partition into block diagonal form.

PROOF Assume that a matrix has two *different* minimum partitions into block diagonal form. Since each diagonal element is contained in a diagonal block, at least two diagonal blocks of these partitions partially overlap or one is properly contained in the other. This implies that there is a non-trivial partition into block diagonal form of at least one of these diagonal blocks, which for the corresponding partition gives rise to a partition into block diagonal form of the *whole* matrix with fewer elements, contradicting the minimality of both partitions. △

The following obvious property holds:

Proposition 2 A partition of a square matrix into block diagonal form is minimum if and only if no corresponding diagonal block has a non-trivial partition into block diagonal form

PROOF '⇒' A non-trivial partition of a diagonal block into block diagonal form would imply the existence of a partition into block diagonal form of the whole matrix with fewer elements. '⇐' Since each diagonal block of the minimum partition into block diagonal form is properly contained in, or equal to a diagonal block of an arbitrary partition into block diagonal form, this implies that for a partition into block diagonal form where no diagonal block has a non-trivial partition into block diagonal form, each diagonal block must be equal to a diagonal block of the minimum partition into block diagonal form, so that both partitions are equal. △

Similar propositions can be given for the minimal partitions into block lower or upper triangular form. In the context of permutations, a matrix is called **reducible** if there is a symmetric permutation PAP^T with a non-trivial partition into block triangular form [14]. A permuted matrix is fully reduced, if it has a partition into block triangular form for which all diagonal blocks are irreducible, so that

[1] Minimum values reveal the most information about a matrix, because the constraints are satisfied in any matrix for the trivial semi-bandwidths $b_1 = b_2 = n - 1$. Allowing for negative semi-bandwidths would enable the specification of arbitrary bands, where $b_1 = b_2 = -\infty$ corresponds to a zero matrix.

this partition is necessarily minimal. In contrast, diagonal blocks of the minimal partition into triangular form can still be reducible.

For the previous partitions it was tacitly assumed that all diagonal blocks A_{ii} are non-empty (an 0×0 submatrix is called an **empty block**). However, in order to generalize partitions into bordered block forms and into the corresponding block forms without border, it is convenient to allow that the last diagonal block A_{pp} is empty, so that all border blocks are also empty, as is illustrated below for a partition into bordered block upper triangular form, where D and O denote diagonal and off-diagonal blocks respectively, and ϵ is used for an empty block:

$$\begin{pmatrix} D & O & O & \epsilon \\ & D & O & \epsilon \\ & & D & \epsilon \\ \epsilon & \epsilon & \epsilon & \epsilon \end{pmatrix}$$

Some matrices have several minimum partitions into bordered block form, in which case we will select the partition that corresponds to the smallest border size. For partitions into block banded form, empty diagonal blocks will be used to obtain a block banded form from a more general form, as will be further explained in section 3.4.

3 Automatic Analysis

In this section, techniques to analyze nonzero structures are presented. It is assumed that all sparse matrices are available on file in so-called coordinate scheme storage (see e.g. [9, 10, 11, 12, 29]). In this scheme, a file consists of integers n and m that contain the size of the matrix, an integer τ that indicates the number of stored elements, followed by τ unordered triples (i, j, a_{ij}) to indicate row and column indices and value of each entry. Because there is no advantage in storing particular zeros explicitly, it is assumed that all stored elements are nonzero.

3.1 Lower and Upper Skyline

The **lower skyline** of an $n \times n$ sparse matrix is defined as the sequence $\alpha_i = \max\{(i-j)|(a_{ij} \neq 0) \wedge (j < i)\}$, for $1 \leq i \leq n$, and similarly the **upper skyline** is defined as the sequence $\beta_j = \max\{(j-i)|(a_{ij} \neq 0) \wedge (i < j)\}$ for $1 \leq j \leq n$, where the maximum over an empty set is zero. This implies that the maximum horizontal and vertical distance from each entry to the main diagonal is stored per diagonal element (cf. variable band). For example, the skylines of the 15×15 sparse matrix in figure 1, where the positions of the 26 nonzeros are shown, are described as $\alpha_1 = 0, \alpha_2 = 0, \alpha_3 = 1, \ldots, \alpha_{15} = 3$ and $\beta_1 = 0, \beta_2 = 0, \beta_3 = 2, \ldots, \beta_{15} = 2$.

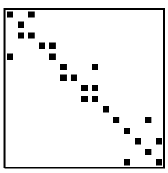

Figure 1: Example Matrix

Skyline information requires $O(n)$ storage and can be obtained in $O(\tau)$ time in a *single* pass over the stored nonzero elements in a file by execution of the following fragment, limiting the increase of computational time and storage requirements of the compiler:

```
read(n,m,nz);     /* Assume m=n holds */
allocate storage for lsky[1..n] and usky[1..n]
for k := 1, nz do
  read(i,j,aij);
  lsky[i] := max(lsky[i],(i-j));
  usky[j] := max(usky[j],(j-i));
endfor
```

The elements of the arrays *lsky* and *usky*, all initialized to zero, are used to store all α_i and β_j. The semi-bandwidths can be computed:

$$b_1 = \max_{1 \leq i \leq n} \alpha_i \quad b_2 = \max_{1 \leq j \leq n} \beta_j$$

For example, $b_1 = 4$ and $b_2 = 3$ holds for the matrix in figure 1. The bandwidths reflect the band form of a matrix and can be used to determine whether a matrix is diagonal, tridiagonal, or lower or upper triangular. However, skylines can also be used to obtain more complex characteristics of the nonzero structure in an efficient way, as is discussed in the following sections.

3.2 Block Diagonal/Triangular Form

If, in the search for a partition of a matrix into block diagonal matrix form, all diagonal blocks beyond row and column B already have been identified, then the next diagonal block can be of size k if for all $B - k < i \leq B$, constraints $\alpha_i + (B - i) < k$ and $\beta_i + (B - i) < k$ are satisfied, as is illustrated in figure 2. This gives rise to the following algorithm to construct a partition into diagonal block form in $O(n)$ time. Starting with $k = 1$ and $B = n$, a scan $i = B, \ldots, B - k + 1$ is made. At each step i, assignment $k := \max(k, \max(\alpha_i, \beta_i) + B - i + 1)$ is performed, which possibly affects the lower bound of the scan. However, if this lower bound is reached, the current $k \times k$ diagonal block is recorded, the base is decreased by k, and k is reset to 1. This method is repeated until $i = 1$. Note that in any case, a diagonal block is recorded after the final step.

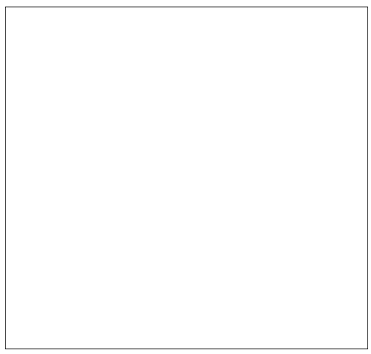

Figure 2: Valid Block in Block Diagonal Form

Proposition 3 Application of the previous algorithm to the lower and upper skyline of a matrix results in the minimum partition of that matrix into block diagonal form.

Proof By construction each entry is incorporated in a diagonal block, so it suffices to show that the minimal partition into block diagonal form results, for which proposition 2 is used. Assume that in the resulting partition, a $k \times k$ diagonal block, consisting of the elements a_{ij} for $B-k < i, j \leq B$, for $1 \leq B \leq n$, has a non-trivial partition into block diagonal form. Consideration of the right lower most diagonal block in this partition implies that there is $k' < k$, such that (1) for all $B - k' < i \leq B$, $\alpha_i + (B - i) < k'$ and $\beta_i + (B - i) < k'$ hold. If k_i denotes the value of k after step i, then (2) $k' < k_{B-k'+1}$ holds because the scan proceeded after $i = B - k' + 1$. Furthermore, since $1 \leq k' < k$, the scan proceeded after step $i = B$, and thus $k_B > 1$. Consequently, k has been adapted, which implies that (3) $k_i = \alpha_j + B - j + 1$ or $k_i = \beta_j + B - j + 1$ holds for $B - k' < i \leq j \leq B$. Combining (2) and (3) yields either $k' \leq \alpha_j + B - j$ or $k' \leq \beta_j + B - j$ for certain $B - k' < j \leq B$, contradicting (1). $\triangle$

If only β_i or α_i is used in the assignment to k, the minimum partition into respectively block lower or upper triangular forms are constructed, as can be proven similarly. Application of these algorithms to the matrix of figure 1, yields the block diagonal, block lower and upper triangular forms that are shown in figure 3. The total number of elements that are contained in the nonzero blocks of these block forms (respectively 67, 140 and 138) can be used as a measure of the description accuracy, which implies that the block diagonal form reveals the most information about the nonzero structure of this matrix.

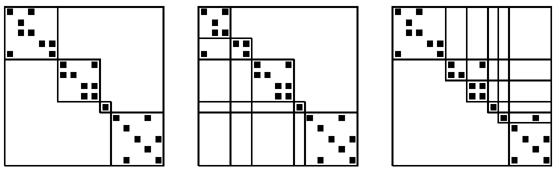

Figure 3: Block Forms

3.3 Bordered Block Form

Nonzero elements that occur in the borders of a matrix might be responsible for large values in the two skylines, so that the size of the resulting diagonal blocks increases. For the matrix in figure 4, for example, 176 elements appear in the nonzero blocks of the minimum partition into block upper triangular form, which is large in comparison with the total number of elements. Since only the trivial partition defines a block diagonal and lower triangular form, the minimal partitions into these forms even contain more elements.

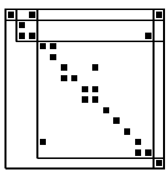

Figure 4: Block Upper Triangular Form

Therefore, it might be useful to have algorithms that also construct a minimum partition into bordered block forms. In a naive approach, such a partition is found by application of the algorithms of the previous section to the submatrix that remains for every border size, followed by selection of the partition with the fewest corresponding elements. However, even if a limited number of border sizes is considered, this approach would have unacceptable complexity [8], since a reasonable upper bound on the border size must be expressed in terms of n. Fortunately, it is also possible to obtain the best border size in $O(n)$ time.

Suppose that for a certain border size b', the minimum partition of the remaining $(n - b') \times (n - b')$ submatrix into block upper triangular form is determined by application of the previous algorithm, i.e. each next diagonal block is computed by a scan $i = B, \ldots, B - k + 1$, where k is increased if necessary. After each step i, it can also be decided to discard the partition found so far, and to start the algorithm for $B = i - 1$ and $k = 1$ after a new border size $b = n - i + 1$ has been recorded. Selection of this border is only profitable if the number of extra elements that are included in the border (*loss*) is less than the number of elements that are gained by starting with a new partition (*gain*). If the new partition remains within the old diagonal block, gain consists of elements in that block only, as illustrated in figure 5.

Figure 5: Gain and Loss within Diagonal Block

However if a diagonal block of this new partition exceeds the old diagonal block, more elements are included in the gain. The loss decreases because k would have been increased to $B_I - B_V + k_V$ in the old partition, where subscripts I and V are used to denote the values of the variables in this particular run at **initiation time** ($i = n - b + 1$) and **verification time** (when i becomes $B_I - \max(k_I, B_I - B + k) + 1$) respectively, as illustrated in figure 6.

Figure 6: Gain and Loss outside Diagonal Block

If Z contains the number of elements in the zero blocks of the partition found so far, the loss and gain can be computed as $l(b,b') = Z_I + (B_V - k_V)\cdot(B_I - i_I + 1)$ and $g(b,b') = Z_V - (B_V - k_V)\cdot(i_I - B_V - 1)$ respectively, so that the improvement is determined as $I(b,b') = g(b,b') - l(b,b') = Z_V - Z_I - (B_V - k_V)\cdot(B_I - B_V)$. If the gain exceeds the loss, i.e. $I(b,b') > 0$, it is profitable to continue the algorithm with the current partition by recording this next block, while for $I(b,b') \le 0$ the old partition (corresponding to border size b') must be restored. Moreover, the next block of size $\max(k_I, B_I - B + k) + 1)$ can be recorded immediately (i.e. without any backtracking).

Because the remaining $(i_V - 1)\times(i_V - 1)$ submatrix would be further partitioned identically for *both* border sizes, $I(b,b')$, in fact, indicates the difference between the number of elements in the nonzero blocks of the partitions that would result for border sizes b and b' respectively. Consequently, although the value of $I(b,b')$ is only constructed for certain $b' < b$, this new definition reflects the fact that properties like $I(b,b') = -I(b',b)$, $I(b,b) = 0$ and, more important $I(b,b'') = I(b,b') + I(b',b'')$ hold. This latter property is exploited to consider all border sizes in a single pass over the skyline. If at a verification $I(b,b') > 0$ holds, pending verifications of b' with respect to smaller border sizes b'' can be converted into verification of b with respect to b'', since $I(b,b'') > I(b',b'')$ holds, so that in any case it is more profitable to select b than b'. If $I(b,b') \le 0$ holds, this verification can be abandoned safely, since $I(b,b'') \le I(b',b'')$ holds for all pending verifications of b' with respect to b''. No improvement can be obtained from a border size $n-i+1$ if at step i a block is recorded, and the following algorithm to construct *a minimum partition into bordered block upper triangular form* results. The border size is initially 0, corresponding to an empty last diagonal block:

```
Z:=0; B:=n; k:=1; b:=0; s:=0; p:=0;
for i:=n, 1 step -1 do
  k:=max(k,lsky(i)+B-i+1);
  if (i = B-k+1) then /* Block Detected */
    while ((s > 0) cand
                (i = sB[s]-max(sk[s],sB[s]-B+k)+1)) do
      if ( (Z-sZ[s]-(B-k)*sB[s]-B) > 0 ) then
        s:=s-1;
      else
        k:=max(sk[s],sB[s]-B+k); Z:=sZ[s];
        B:=sB[s]; b:=sb[s]; p:=sp[s]; s:=s-1;
      endif
    enddo
    Z:=Z+k*(B-k); B:=B-k; k:=1; p:=p+1; part[p]:=i;
  else /* No Block Detected */
    s:=s+1; sZ[s]:=Z; sB[s]:=B;
    sk[s]:=k; sb[s]:=b; sp[s]:=p;
    Z:=0; B:=i-1; k:=1; b:=n-i+1;
  endif
endfor
```

Five auxiliary arrays are used in a stack-like manner with pointer s to save and restore states if necessary. Array *part* with pointer p is used similarly to record the left upper corner of each diagonal block in the partition, so that an old partition can be discarded by restoring the corresponding pointer value. As soon as improvement is obtained, s is simply decremented by 1, which effectively converts pending verifications into verification of the current border size because of the formulation of the improvement function. Moreover, the old partition below row $n-b$ that still resides on the stack, can be eliminated afterwards. Although a while-loop occurs inside the loop body, this algorithm runs in $O(n)$ time because each border can only be moved to and from the auxiliary arrays once.

If the upper skyline or both skylines are considered, this algorithm determines the minimum partition into bordered block lower triangular form or doubly bordered block diagonal form respectively, because the same improvement function can be used (ignoring a factor of 2 in the latter case). For example, application of this algorithm to the matrix of figure 4, yields the bordered block forms shown in figure 7 with 113 (viz. $225 - 2\cdot 56$), 157 (viz. $225 - 68$) and 162 (viz. $176 - 14$) elements, for $I(3,0) = 56$, $I(2,0) = 68$ and $I(3,0) = 14$ respectively:

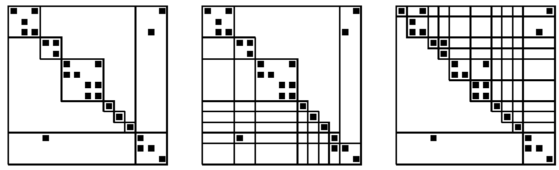

Figure 7: Bordered Block Forms

The block upper triangular form, for example, is represented in array *part* after elimination of the partition below row 12 as shown below:

12	11	10	8	6	5	4	2	1

Application of this algorithm to the matrix with 22 nonzero elements in figure 8, results in a minimum partition into bordered block upper triangular form with 166 elements in the nonzero blocks because $I(1,0) = 21$. However, the nonzero blocks of the partitions corresponding to border sizes 2 and 3 also contain 166 elements, which illustrates that a matrix does not have a unique minimum partition into a particular bordered block form. Because a border is denied if the gain is equal to the loss, all ties (in this case $I(3,2) = 0$ and $I(2,1) = 0$, and thus, $I(2,0) = 21$ and $I(3,0) = 21$) are solved in favor of the smallest associated border size.

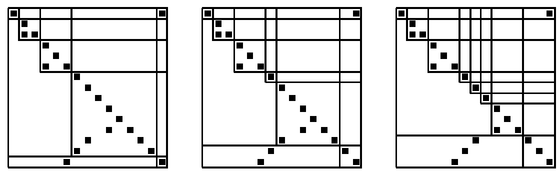

Figure 8: Different Minimum Partitions

3.4 Block Banded Form

Even if no entries occur within the borders, large diagonal blocks might result if the size of the next diagonal block is increased each time before the end of a scan has been reached. In figure 2, for example, k is increased at step i, although $k = 3$ was valid before that step. This implies that many zeros are incorporated in the partition into diagonal form. Therefore, construction of a partition into block banded form is also useful. In this section, we consider a form with identical semi-bandwidths.

Suppose that after step $i = U$, the next diagonal block would consists of a square with the left lower corner in row L. As long as during the scan $i = U, \ldots, L$, inequality $L \le i - \max(\alpha_i, \beta_i)$ holds, this next block remains valid. However, as soon as $i - \max(\alpha_i, \beta_i) < L$ holds at step i, unnecessary incorporation of zeros is avoided by recording a diagonal block that extends rows $i+1, \ldots, U$ that has one off-diagonal block, while the new square is recorded for future computations. This process is repeated until the end of the first square is reached, thereby possibly further dividing the off-diagonal blocks as is illustrated in figure 9.

Figure 9: Blocks in Block Banded Form

At the end of a square, the number of off-diagonal blocks for all diagonal blocks outside the next square is known, because these blocks cannot be divided any further. Subsequently, this next square is considered. If no next square has been recorded, the end of a real diagonal block has been reached, and the number of off-diagonal blocks for all current diagonal blocks is known. This process is repeated until the whole matrix has been partitioned, as expressed in the following algorithm:

```
next:=n+1; num:=0; s:=0; sc:=1; p:=0; pv:=1;
lc[0]:=n+1; rec[0]:= 0;
for i:=n, 1 step -1 do
  left:=i-max(lsky[i],usky[i]);
  if (left < next) then /* Initiate New Square */
    if (i+1 ≠ lc[sc-1] then
      p:=p+1; part[p]:=i+1; num:=num+1;
    endif
    s:=s+1; rc[s]:=i; lc[s]:=left; rec[s]:=num;
    left:=next;
  endif
  if (i = lc[sc]) then /* End of Current Square */
    p:=p+1; part[p]:=i+1; num:=num+1;
    sc:=sc+1;
    row:=(sc ≤ s) ? rc[sc] : i-1;
    cnt:=num-rec[sc-1]-1;
    while (part[pv] > row) do
      sky[pv]:=cnt; pv:=pv+1; cnt:=cnt-1;
    enddo
  endif
enddo
```

Arrays *lc* and *rc* are used to record the left and right corner of each square. Array *part* has the same function as in the algorithm of section 3.3. Parallel to this array is *sky*, which is used to record the number of off-diagonal blocks per diagonal block. Again, the nested while-loop does not affect the $O(n)$ time, since each diagonal block is only considered once. Finally, *sc* contains the index of the current square, *s* indicates following squares and array *rec* and variable *num* are used to determine the number of off-diagonal blocks at the end of each square. Application of this algorithm to the matrix of figure 1 and two other example matrices results in the partitions shown in figure 10.

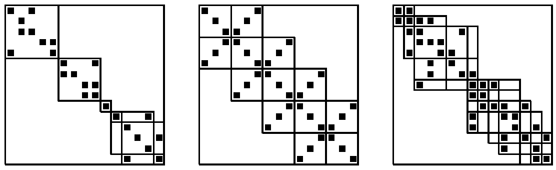

Figure 10: Block Forms

Since the semi-bandwidths in the resulting block forms vary, a completion method is required to obtain a block banded form with one bandwidth. A possible completion method is based on the observation that in a block banded form, the semi-bandwidths of the left upper diagonal blocks form a sequence $0, 1, \ldots, B-1$, followed by an arbitrary number of semi-bandwidths B. Within the partition this sequence can be 'simulated' by insertion of B empty blocks. as is illustrated below for a block tridiagonal form, where, although the semi-bandwidths are all 1, the semi-bandwidths appear to be 0,1,1,0,1:

$$\begin{pmatrix} D & O & & & & \\ O & D & O & & & \\ & O & D & \epsilon & & \\ & & \epsilon & \epsilon & \epsilon & \\ & & & \epsilon & D & O \\ & & & & O & D \end{pmatrix}$$

Consequently, it is easy to construct a partition into block banded if the semi-bandwidths form such a feasible sequence by insertion of empty blocks. For example, this method inserts a total of three empty blocks in the partition of the first matrix, while the partition of the second matrix remains unaffected. Block forms with respectively 65 and 117 elements result. For the third matrix, however, *additional* off-diagonal blocks must be inserted to obtain a feasible sequence (i.e. semi-bandwidth $0, 1, 1, 1, 2, 1, 2, 3, 2, 2, 2$ sequence is converted into $0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3$).

A partition into block banded form with 127 elements that is shown in figure 11 results. If additional off-diagonal blocks are inserted, the resulting partition is not always minimal. For example, the matrix of figure 11 also has a partition into block tridiagonal form with only 99 elements. However, it provides an easy way to obtain a block banded form of a matrix.

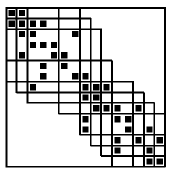

Figure 11: Block Banded Form

3.5 Statistical Information

In this section, it is explained how some statistical information about the nonzero structure of a matrix can be determined efficiently. A more advanced tool kit for statistical analysis of sparse matrices is described by Saad in [19].

Under the assumption that each nonzero element is stored exactly once in coordinate scheme, the average number of nonzero elements per row or column in an $m \times n$ sparse matrix A can be computed directly as $\frac{\tau}{m}$ and $\frac{\tau}{n}$, while the density of A is determined as $\frac{\tau}{m \cdot n}$. However, more characteristics of the nonzero structure of a matrix can be determined, if during the pass over all τ nonzero elements the total number of nonzero elements in each row, column and diagonal are accumulated in $O(n+m)$ storage for arrays $rowc[1..m]$, $colc[1..n]$ and $diagc[1-m..n-1]$, by incrementing $rowc[i]$, $colc[j]$ and $diagc[j-i]$ for each entry a_{ij}. The resulting contents of $diagc$ for the matrix of figure 1 is shown below for indices -5 through 5.

0	1	1	0	3	15	2	2	2	0	0

These arrays can be used to compute the density in a particular row i, column j, or diagonal k:

$$\frac{rowc[i]}{n}, \frac{colc[j]}{m}, \frac{diagc[k]}{\min(m, n-k) - max(1, 1-k) + 1}$$

Consequently, the number of diagonals in which elements occur can be determined by counting the number of elements in $diagc$ that are nonzero, while the number of full diagonals is determined similarly by counting the number of diagonals with density 1. Note that if the definition of semi-bandwidths is extended to arbitrary matrices, this implies that $diagc[i] = 0$ for $i - j > b_1$ and $j - i > b_2$. The percentage p of nonzero elements that are confined a band with given semi-bandwidths B_1 and B_2 is computed as follows:

$$p = \frac{1}{\tau} \cdot \sum_{k=-B_1}^{B_2} diagc[k] \times 100\%$$

On the other hand, the smallest band with bandwidth $2 \cdot B + 1$ in which $p\%$ of the nonzero elements is contained, is computed as follows:

$$B = \min \left\{ b \geq 0 \mid \sum_{k=-b}^{b} diagc[k] \geq \lceil \frac{p}{100} \cdot \tau \rceil \right\}$$

For example, for the matrix of figure 1, $B = 3$ results for $p = 90\%$, so that the corresponding bandwidth is 7. Accumulating for $B_1 = 2$ and $B_2 = 2$ yields $p \approx 85\%$. Four diagonals are used in the matrix of figure 12, while three diagonals are full.

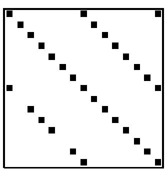

Figure 12: Statistical Example

3.6 Detection of Dense Submatrices

In [1, 26, 27, 28] the use of quad trees, well-known from image processing and computer graphics [15, 21], for representing sparse matrices is proposed. A matrix of order n is embedded in a $2^{\lceil \ln n \rceil} \times 2^{\lceil \ln n \rceil}$ matrix and padded appropriately. If the resulting matrix consists of only zeros or a single scalar, it is represented by a nil pointer or the value of that scalar respectively. Otherwise, it is represented by a tree with four subtrees (labeled 0,1,2 and 3) corresponding to left upper, right upper, left lower and right lower quadrant. This data structure provides a uniform way of representing dense and sparse matrices, while it also simplifies the implementation of algorithms that are based on matrix partitioning. For example, the sum of two matrices is assembled recursively, which terminates if one of the operands is the nil pointer, resulting in the other operand, or if two scalars have to be added. An example of a quad tree representation for a sparse matrix is shown in in figure 13. A particular element is accessed by the tree walk defined by successively accessing the subtrees defined by the next pair of bits, starting with the most significant bits in the binary representation of the row and column index, if these are numbered from 0 to $2^{\lceil \ln n \rceil} - 1$. For example, element a_{21} is stored in row 01 and column 00. It is accessed along the boldfaced path in figure 13, using the labels 00 and 10.

Figure 13: Quad Tree Representation

The same idea can be used to detect the occurrences of dense submatrices in an arbitrary sparse $m \times n$ matrix if another stop criterion is used. A matrix for which the density exceeds a given threshold λ is considered to be a dense matrix, while a zero matrix is not considered any further. Otherwise, these criterions are applied recursively to the submatrices in its quadrants, as formulated in the following algorithm. The dense blocks in an $m \times n$ matrix A are recorded by the call 'P($Nonz_A, 1, m, 1, n$)', where $Nonz_A$ indicates the index set of all nonzero elements.

procedure $P(I, i_l, i_h, j_l, j_h)$
begin
  if $(|I| > 0)$ **then**
    $a := (i_h - i_l + 1) * (j_h - j_l + 1)$
    if $(\frac{|I|}{a} < \lambda)$ **then**
      $i_c := \lfloor \frac{i_l + i_h}{2} \rfloor$; $j_c := \lfloor \frac{j_l + j_h}{2} \rfloor$;
      $P(\{(i,j) \in I | (i \leq i_c) \wedge (j \leq j_c)\}, i_l, i_c, j_l, j_c)$;
      $P(\{(i,j) \in I | (i \leq i_c) \wedge (j_c < j)\}, i_l, i_c, j_c + 1, j_h)$;
      $P(\{(i,j) \in I | (i_c < i) \wedge (j \leq j_c)\}, i_c + 1, i_h, j_l, j_c)$;
      $P(\{(i,j) \in I | (i_c < i) \wedge (j_c < j)\}, i_c + 1, i_h, j_c + 1, j_h)$;
    else
      record_block(i_l, i_h, j_l, j_h);
    endif
  endif
end procedure

Application of this algorithm takes $O(n \cdot \log n)$ time for an $n \times n$ matrix. $O(\tau)$ storage suffices for the index set, if at each call in-place sorting is applied to a global array that contains I and pointers are passed as parameters instead. For example, respectively 54 and 235 dense submatrices result for a 32×32 matrix with 331 nonzero element for $\lambda = 0.5$ and $\lambda = 1.0$ respectively, as illustrated in figure 14. This example clearly illustrates the shortcomings of this method. Although the submatrices reveal the nonzero structure of this matrix, too many submatrices might result because arbitrary divisions of the index set are made. Decreasing the threshold λ partially reduces this problem, but clusters of dense submatrices might still result in general.

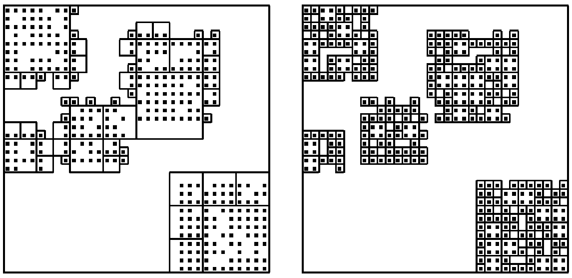

Figure 14: Dense Submatrices ($\lambda = 0.5$ and $\lambda = 1.0$)

4 Implementation and Motivation

The algorithms presented in this paper have been implemented in the analysis engine of the prototype compiler MT1 [5]. First, the skylines of a sparse matrix are determined in a single pass over the file. In fact, only the index set is needed, since actual values are not used in the analysis. Subsequently, the semi-bandwidths are determined, and if $b_1 = 0$ and $b_2 = 0$, or $b_1 = 0$, or $b_2 = 0$ holds, a diagonal, upper or lower triangular form are recorded. Otherwise, the minimum partitions into bordered block forms and block banded form are constructed, and the partition with the fewest number of elements in the nonzero blocks is selected. If this number does not exceed a certain threshold, expressed as a fraction f of the size of the matrix, the corresponding form is used as characterization of this matrix, while a general sparse matrix is assumed otherwise. Finally, some statistics are determined for the matrix and recorded for later use in the compiler. Consequently, analysis of an $n \times n$ matrix takes $O(\tau + n)$ time, or $O(\tau + n \cdot \log n)$ if a search

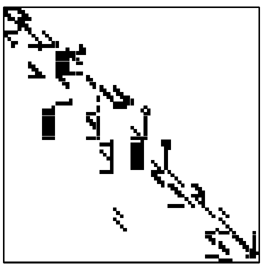

Figure 15: impcol_e

for dense submatrices is performed. An overview of the results of the analysis is prompted to the user. For example, for matrix **impcol_e** of the Harwell-Boeing Sparse Matrix Collection [10], of which the nonzero structure is illustrated in figure 15, the following output results:

```
Size                  :     225   x     225
#Entries              :    1308
Density               :       0.026
Av.#Entries p. row    :       5.81
Av.#Entries p. col.   :       5.81

Semi-Bandwidths       :      92 <->      35
            (B90)     :     129
            (P22)     :      11.93
#Used Diagonals       :     120
#Full Diagonals       :       0

Type                  : Block Banded Form
#Elements (D/L/U/B)   :   50289 34552 49778 27931
       (bs/bs/bs/B)   :      56     0   142    13
       (#blocks)      :       2    15    12    49
#Dense Blocks         :     556   (Density: 0.50)
```

Figure 16: Output for impcol_e

Entry `B90` and `P22` denote the bandwidth in which 90% of all nonzero elements is contained, and the percentage of nonzero elements within a band with bandwidth 5 . ($B_1 = 2$ and $B_2 = 2$). The total number of elements in the computed block diagonal, block lower and upper triangular and block banded form (`D/L/U/B`) is given, together with the border size or semi-bandwidth respectively (`bs/B`), and the total number of diagonal blocks in the partitions, excluding the diagonal block in the border. The partition of the block form that is selected as characterization of the matrix is made available to the compiler in a data structure that is similar to the array used in the construction algorithms.

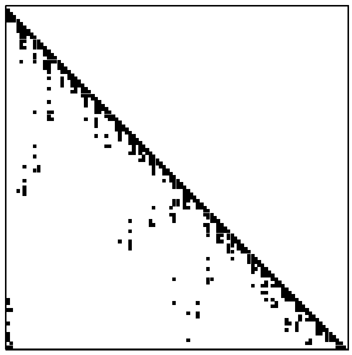

Figure 17: jagmesh2

The results of analysis on **jagmesh2**, illustrated in figure 17, are given in figure 18. The matrix is lower triangular and no information about block forms is determined.

Finally, we illustrate how nonzero structure information can be used by a sparse compiler, as described in [4, 6]. In case nonzero structures are *static*, i.e. fill-in does not occur, the only concern of the compiler is to select a data structure

```
Size                 :    1009  x   1009
#Entries             :    3937
Density              :      0.004
Av.#Entries p. row   :      3.90
Av.#Entries p. col.  :      3.90

Semi-Bandwidths      :    991 <->      0
            (B90)    :    101
            (P22)    :     50.39
#Used Diagonals      :    244
#Full Diagonals      :      1

Type                 : Lower Triangular Form
#Dense Blocks        :    2035   (Density: 0.50)
```

Figure 18: Output for jagmesh2

according to the nonzero structure characteristics. Since in some cases, explicit storage of some zeros results in storage schemes with less overhead, set E_A, for which $Nonz_A \subseteq E_A \subseteq \{1 \ldots m\} \times \{1 \ldots n\}$ is used to denote the index set of explicit stored elements, referred to as **entries**. However, for *dynamic* data structures the compiler must determine which properties are preserved during execution, so that only those properties are exploited. The results of nonzero structure analysis can be given in terms of constraints of the form $a_{ij} \neq 0 \Rightarrow P(i,j)$ where $P(i,j)$ is some predicate. For example, for a lower triangular matrix, this predicate has the form $j \leq i$. If a data structure is selected by the compiler such that $(i,j) \in E_A \Rightarrow P(i,j)$ still holds, contra position $\neg P(i,j) \Rightarrow (i,j) \notin E_A$ can be used to reduce the iteration set [6]. For example, since in the following dense fragment for matrix-vector multiplication, condition $(i,j) \in E_A$ is associated with the assignment statement to indicate the instances that must be executed, the compiler can transform this fragment as shown below if $(i,j) \in E_A \Rightarrow (-4 \leq j-i \leq 5)$ holds:

```
DO I = 1, M
  DO J = 1, N
    Y(I) = Y(I) + A(I,J) * X(J)
  ENDDO
ENDDO
```

First, a unimodular transformation [2] is applied so that all elements a_{ij} for $j - i = \mathtt{J}$ are accessed in one iteration of the *outermost* loop:

```
DO J = 1 - M, N - 1
  DO I = MAX(1,1-J), MIN(M,N-J)
    Y(I) = Y(I)+ A(I,J+I) * X(J+I)
  ENDDO
ENDDO
```

Subsequently, iteration space reduction is applied, based on the nonzero structure characteristics:

```
DO J = -4, 5
  DO I = MAX(1,1-J), MIN(M,N-J)
    Y(I) = Y(I)+ A(I,J+I) * X(J+I)
  ENDDO
ENDDO
```

Statistical information about the density in the remaining diagonal that are accessed can be used subsequently to select sparse or dense storage for these diagonals, so that one of the following versions result (see [6] for further details of code generation):

sparse band:

```
DO J = -4, 5
  DO AD = ALOW(J+5), AHIGH(J+5)
    I = AIND(AD)
    Y(I) = Y(I) + AVAL(AD) * X(J+I)
  ENDDO
ENDDO
```

dense band:

```
DO J = -4, 5
  DO I = MAX(1,1-J), MIN(M,N-J)
   Y(I) = Y(I) + ADNS(I-MAX(1,1-J)+1,6-J) * X(J+I)
  ENDDO
ENDDO
```

In the second fragment, E_A consists of the whole band that is stored in a two-dimensional dense array **ADNS**, while in the first fragment, arrays **ALOW** and **AHIGH** are used to point to the start addresses of sparse vectors that are stored conform the diagonals in two parallel arrays **AVAL** and **AIND**. Since appropriate initialization code is generated by the compiler, the exact structure E_A along the band will only be constructed at run-time.[2] Clearly, this code is more suited for this particular matrix than the code that would result for e.g. sparse row-wise storage of a general sparse matrix, shown below. The overhead in this code is clearly more intrusive, because many short sparse vectors are traversed.

```
DO I = 1, M
  DO AD = ALOW(I), AHIGH(I)
    J = AIND(I)
    Y(I) = Y(I) + AVAL(AD) * X(J)
  ENDDO
ENDDO
```

Similarly, block forms, stored in arrays like *part* can be used to transform fragments into the corresponding block algorithms, while statistical information is used to determine whether sparse or dense storage is used for all nonzero blocks of the partition. If the nonzero structure changes during execution, i.e. fill-in occurs, a *dynamic* data structure must be used to account for these changes. In such cases, the compiler must determine which properties are preserved during program execution, which requires an advanced form of program analysis.

5 Conclusions

In this paper we have presented techniques to analyze the nonzero structures of sparse matrices in an efficient way. More research is necessary into techniques to use this information in the data structure selection and code generation phase of a sparse compiler, and in program analysis techniques that determine which properties are preserved in case fill-in occurs. The results from this research can also be used for the automatic generation of analysis code in sparse codes that select a particular routine that is optimized for specific nonzero structures. Other issues, such as reordering of matrices in order to reduce the fill-in, must also be accounted for.

Acknowledgements

The authors would like to acknowledge helpful discussions with Arnold Niessen.

[2] This is a strong argument for user annotations for nonzero structures in case only certain properties of the nonzero structure are known in advance, but the exact nonzero structure is not available.

References

[1] S. Kamal Abdali and David S. Wise. Experiments with quadtree representation of matrices. In G. Goos and J. Hartmanis, editors, *Lecture Notes in Computer Science 358*, pages 96–108. Springer-Verlag, 1988.

[2] U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publishers, Boston, 1993.

[3] Aart J.C. Bik and Harry A.G. Wijshoff. Advanced compiler optimizations for sparse computations. In *Proceedings of Supercomputing 93*, pages 430–439, 1993.

[4] Aart J.C. Bik and Harry A.G. Wijshoff. Compilation techniques for sparse matrix computations. In *Proceedings of the International Conference on Supercomputing*, pages 416–424, 1993.

[5] Aart J.C. Bik and Harry A.G. Wijshoff. MT1: A prototype restructuring compiler. Technical Report no. 93-32, Dept. of Computer Science, Leiden University, 1993.

[6] Aart J.C. Bik and Harry A.G. Wijshoff. On automatic data structure selection and code generation for sparse computations. In *Proceedings of the Sixth International Workshop on Languages and Compilers for Parallel Computing*, pages 57–75, 1993. Lecture Notes in Computer Science, No. 768.

[7] Elizabeth Cuthill. Several strategies for reducing the bandwidth of matrices. In Donald J. Rose and Ralph A. Willoughby, editors, *Sparse Matrices and Their Applications*, pages 157–166. Plenum Press, New York, 1972.

[8] I.S. Duff. A sparse future. In I.S. Duff, editor, *Sparse Matrices and their Uses*, pages 1–29. Academic Press, London, 1981.

[9] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices*. Oxford Science Publications, 1990.

[10] I.S. Duff, Roger G. Grimes, and John G. Lewis. Sparse matrix test problems. *ACM Transactions on Mathematical Software*, Volume 15:1–14, 1989.

[11] I.S. Duff and J.K. Reid. Some design features of a sparse matrix code. *ACM Transactions on Mathematical Software*, pages 18–35, 1979.

[12] Alan George and Joseph W. Liu. The design of a user interface for a sparse matrix package. *ACM Transactions on Mathematical Software*, Volume 5:139–162, 1979.

[13] Alan George and Joseph W. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall Inc., 1981.

[14] Frank Harary. Sparse matrices and graph theory. In J.K. Reid, editor, *Large Sparse Sets of Linear Equations*, pages 139–150. Academic Press, 1971.

[15] Donald Hearn and M. Pauline Baker. *Computer Graphics*. Prentice-Hall International, 1986.

[16] A. Jennings. A compact storage scheme for the solution of symmetric linear simultaneous equations. *The Computer Journal*, Volume 9:281–285, 1966.

[17] A. Jennings and A.D. Tuff. A direct method for the solution of large sparse symmetric simultaneous equations. In J.K. Reid, editor, *Large Sparse Sets of Linear Equations*, pages 97–104. Academic Press, 1971.

[18] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes*. Cambridge University Press, Cambridge, 1986.

[19] Youcef Saad. SPARSKIT: a basic tool kit for sparse matrix computations. CSRD/RIACS, 1990.

[20] Youcef Saad and Harry A.G. Wijshoff. Spark: A benchmark package for sparse computations. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 239–253, 1990.

[21] Hanan Samet. Connected component labeling using quadtrees. *Journal of the ACM*, Volume 28:487–501, 1981.

[22] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Computing*, pages 146–160, 1972.

[23] Reginal P. Tewarson. Sorting and ordering sparse linear systems. In J.K. Reid, editor, *Large Sparse Sets of Linear Equations*, pages 151–167. Academic Press, 1971.

[24] Reginal P. Tewarson. *Sparse Matrices*. Academic Press, New York, 1973.

[25] M. Veldhorst. *An Analysis of Sparse Matrix Storage Schemes*. PhD thesis, Mathematisch Centrum, Amsterdam, 1982.

[26] David S. Wise. Representating matrices as quadtrees for parallel processing. *Information Processing Letters*, Volume 20:195–199, 1985.

[27] David S. Wise. Parallel decomposition of matrix inversion using quadtrees. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 92–99, 1986.

[28] David S. Wise and John Franco. Costs of quadtree representation of nondense matrices. *Journal of Parallel and Distributed Computing*, Volume 9:282–296, 1990.

[29] Zahari Zlatev. *Computational Methods for General Sparse Matrices*. Kluwer Academic Publishers, 1991.