A LOCALLY OPTIMIZED REORDERING ALGORITHM AND ITS APPLICATION TO A PARALLEL SPARSE LINEAR SYSTEM SOLVER

K. Gallivan * P. C. Hansen † Tz. Ostromsky † Z. Zlatev ‡

Abstract

A coarse-grain parallel solver for systems of linear algebraic equations with general sparse matrices by Gaussian elimination is discussed. Before the factorization two other steps are performed. A reordering algorithm is used during the first step in order to obtain a permuted matrix with as many zero elements under the main diagonal as possible. During the second step the reordered matrix is partitioned into blocks for asynchronous parallel processing (normally the number of blocks is equal to the number of processors). It is possible to obtain blocks with nearly the same number of rows, because there is no requirement to produce square diagonal blocks. The first step is much more important than the second one and has a big influence on the performance of the solver.

A straightforward implementation of the reordering algorithm will result in $O(n^2)$ operations. By using binary trees this cost can be reduced to $O(NZ \log n)$, where NZ is the number of non-zero elements in the matrix and n is its order (normally NZ is much smaller than n^2). Some experiments on parallel computers with shared memory have been performed. The results show that a solver based on the proposed reordering performs better than another solver based on a cheaper (but at the same time rather crude) old reordering whose cost is only $O(NZ)$ operations.

AMS Subject Classifications: 65F05, 65Y05
Keywords: sparse matrix, general sparsity, Gaussian elimination, drop-tolerance, reordering, binary tree, block algorithm, coarse-grain parallelism, speed-up.

*Center for Supercomputing Research and Development University of Illinois at Urbana-Champaign Urbana, Illinois, USA
e-mail: gallivan@csrd.uiuc.edu

†UNI•C, The Danish Computer Centre for Research and Education, The Technical University of Denmark, Bldg. 305, DK-2800 Lyngby, Denmark
e-mail: Per.Christian.Hansen@uni-c.dk
e-mail: Tzvetan.Ostromsky@uni-c.dk

‡National Environmental Research Institute Frederiksborgvej 399, DK-4000 Roskilde, Denmark
e-mail: luzz@sun2.dmu.min.dk

1 COARSE-GRAIN PARALLEL ALGORITHMS FOR GENERAL SPARSE MATRICES

It is difficult to develop efficient parallel methods for solving systems of linear algebraic equations $Ax = b$ where A is a general sparse matrix. There are three major reasons for that:

1. **the matrix structure is irregular,**
2. **the loops involve short vectors,**
3. **the nested loops are not well-balanced.**

While the first two reasons are very clear, the third reason deserves some more explanations. Assume, for example, that an outer loop scans the target rows (i.e. the rows that have a non-zero elements in the pivotal column) during a given stage of the Gaussian elimination and consider the inner loop. The amount of work needed to modify a target row will in general vary from one row to another, because the rows contain different numbers of non-zero elements.

The short discussion given above (more details can be found in **Zlatev [27]**) explains why there is only a limited number of parallel codes for general sparse matrices. The solution of the task (development of efficient methods for parallel computers) is considerably easier for parallel machines with shared memory than for distributed-memory machines. Therefore it is not a big surprise that there are several efficient codes for such computers (in the small set of available codes); see, for example, **Davis and Yew [7]**, **Gallivan et al., [17]**, **Gilbert [20]** and **Zlatev [27]**. The situation becomes much more difficult when parallel machines with distributed memory are to be used. The only well-described and efficient code for this class of parallel computers is the code prepared by **Van der Stappen et al., [24]**. It should be reiterated here that the matrices are assumed to be **general**. This means that there is neither an assumption that the matrix under consideration has some special property (such as symmetry and/or positive definiteness) nor some special structure (such as bandedness). If the matrix has either some special property or some special structure, then the task of developing a parallel code becomes in general much easier.

In most of the codes that were quoted above one is trying to exploit either fine-grain parallelism or medium-grain parallelism. This is a good strategy in the case where the overhead for starting parallel tasks is not very large. However, if the overhead for starting parallel tasks is considerable, then algorithms based on the exploitation of coarse-grain parallelism may often perform better (or even much better) than the codes based on fine-grain parallelism or medium-grain parallelism. This explains why it is necessary to develop also such algorithms.

If coarse-grain parallelism is to be used, then the original matrix A has to be reordered before the start of the Gaussian elimination in order to obtain several relatively large blocks that can be treated concurrently. The idea is not a new one; it has been used in many applications (not only in order to obtain a parallel algorithm). An efficient reordering has been proposed by **Hellerman and Rarick [21,22]**. It has been used, with some modifications, by many other authors; see, for example **Arioli et al., [6]** or **Erisman et al., [11]**. Other preliminary reorderings have also been proposed in the literature; see, for example, **Gallivan et al., [13]**. A common feature of all these reorderings is that one always imposes a requirement to obtain **square blocks** on the main diagonal. Moreover, it is also required that the reordered matrix B is either an upper block-triangular matrix or a bordered matrix; in both cases with square blocks on the main diagonal (see the references given above or **Duff et al.,** [**8**]). For some matrices these two requirements are too restrictive. Therefore these requirements should not always be imposed. The main purpose of this paper is to show how to avoid them (when this is appropriate).

A direct solver, where one attempts to exploit coarse-grain parallelism without imposing the above two requirements is described and tested in **Gallivan et al.,** [**17**] and **Zlatev** [**27**]. This solver is based on partitioning the matrix into an upper block-triangular form with rectangular diagonal blocks. A reordering algorithm, by which as many as possible zero elements are obtained in the lower left corner of the matrix, is to be applied before the partitioning. After that the matrix must be divided into block rows, each of them containing approximately the same number of rows. If the reordering algorithm is efficient, then it is rather easy to obtain large block-rows that contain approximately the same number of rows during the partitioning (because it is allowed to use rectangular diagonal blocks). This is why we concentrate our attention on the initial reordering.

An improvement of the reordering algorithm proposed in **Gallivan et al.,** [**17**] and **Zlatev** [**27**], and its application in the solution of systems of linear algebraic equations by Gaussian elimination is discussed in this paper. This algorithm is efficiently implemented by using special graphs: binary trees with leveled nodes (see Section 3). These graphs gives us a possibility to represent readily and to modify cheaply certain partial orderings of the columns of a general sparse matrix. Different kinds of graphs have often been used in sparse matrix studies; especially in connection with symmetric and positive definite matrices (see, **George and Liu** [**18**]). The application of concepts from graph theory in connection with general matrices is not so popular, but also in this field several applications exist. These are based on bipartite graphs; see **George et al.,** [**19**]). The application proposed in this paper is different from the other applications.

The new initial reordering algorithm is introduced in Section 2. Binary trees are used, in Section 3, to prove that the complexity of the new reordering algorithm is $O(NZ \log n)$; here and hereafter n and NZ denote respectively the order of the matrix and the number of its non-zero elements. The proof is constructive, which allows us to apply it efficiently in the implementation of the new reordering algorithm in a code. The whole process of solving systems of linear algebraic equations is described in Section 4 (a more detailed description of the solution process will be given elsewhere). Some considerations about the stability and the sparsity properties of the solver are given in Section 5. Numerical results are presented in Section 6. In the last section, Section 6, conclusions are drawn and plans for future work are presented.

2 THE INITIAL REORDERING

As stated in the previous section our main purpose is to improve the performance of the solver for systems of linear algebraic equations proposed in **Gallivan et al.** [**14**] and **Zlatev** [**27**]. It is convenient first to sketch the initial reordering scheme used in the old solver. It consists of two completely separated steps: column reordering and row reordering. The following definition is needed in order to describe the column reordering.

Definition 1. The number c_j of non-zero elements in a given column j, $j = 1, 2, \ldots, n$ of the matrix A is called **the count** of this column.

When the column ordering is completed, the columns of matrix A are ordered by increasing counts:

$$j < k \quad \Rightarrow \quad c_j \leq c_k. \tag{1}$$

Let r_i be the column index of the first non-zero element in row i (i.e. $a_{ir_i} \neq 0$, but $a_{i1} = a_{i2} = \ldots = a_{ir_i - 1} = 0$). When the row ordering is completed, the rows are ordered so that the following relationship is satisfied:

$$j < k \quad \Rightarrow \quad r_j \leq r_k. \tag{2}$$

It is clear that by reordering the matrix in this way, many of the zero elements, are moved to the lower left corner of the matrix. It can easily be proved that the cost of this reordering is very low; it requires only $O(NZ)$ operations. The reordering can be used to partition the matrix into an upper block-triangular form with rectangular diagonal blocks (see Section 4). However, the number of zeros in the left lower corner of the reordered matrix may be considerably smaller than the maximal number that can be achieved. The main reason for this is the fact that the column ordering is performed before the row ordering. Denote by J_k and J_m the set of row indices of the non-zero elements in columns k and m. The worst case may appear when columns k and m have minimal numbers of non-zero elements, but $J_k \cap J_m = \emptyset$. The algorithm will be inefficient if the matrix is such that this situation appears often. An example where this happens is demonstrated by using the (6×6) matrix shown in **Fig. 1**. This matrix has two non-zero elements in each column. Therefore the column ordering will not lead to any column permutations. It is easy to see that the row ordering does not lead to row permutations either. Thus, the old ordering algorithm leaves this matrix unchanged. It is seen, however, that the first three columns have non-zero elements whose row numbers do not match (for example, the row numbers of the non-zero elements in the first column are 1 and 2, while the row numbers of the non-zero elements in the second column are 3 and 4). This leads to a rather poor result (only six zero elements in the lower left corner).

	1	2	3	4	5	6
1	■	0	0	0	0	■
2	■	0	0	■	0	0
3	0	■	0	■	0	0
4	0	■	0	0	■	0
5	0	0	■	0	■	0
6	0	0	■	0	0	■

Figure 1
The original sparsity pattern of a matrix;
the old reordering algorithm preserves this structure.

Not only is the main drawback of the old algorithm revealed by this simple example, but the example also shows how the reordering can be made more efficient. It is worthwhile to carry out the following operations. When the first column with a minimal count is chosen (and brought to position 1 by permutations), the counts of the other columns should be updated by removing all non-zero elements from the rows that have non-zero elements in that column. Then the second column is to be chosen (among the columns with minimal count) and to be brought by permutations to position 2. The counts of the other columns are again to be updated by removing all non-zero elements from the rows that have non-zero elements in the second column. The process must be continued in this manner. The most important feature of this new algorithm is that after choosing a column with a best count (and permuting it to the appropriate position) the counts of some columns are reduced. A more rigorous description of the algorithm will be given below. It is worthwhile to illustrate here the effect of this reordering by using the same matrix as in the example shown in **Fig. 1**. The result obtained by the new ordering is shown on **Fig. 2**. It is immediately seen that the number of zero elements in the lower left corner is considerably greater. It will be illustrated in Section 6 that the new reordering algorithm gives also better results for large sparse matrices.

	1	4	2	5	3	6
1	■	0	0	0	0	■
2	■	■	0	0	0	0
3	0	■	■	0	0	0
4	0	0	■	■	0	0
5	0	0	0	■	■	0
6	0	0	0	0	■	■

Figure 2
An optimal reordering of the matrix from Fig. 2.
The column permutation sequence is given above the matrix.
The reordered matrix is obtained without row interchanges.
This reordering is produced by the new algorithm.

The examples demonstrated on **Fig. 1** and **Fig. 2** show that better results can be achieved by using a new reordering algorithm. Two definitions are needed in order to formulate more rigorously the new algorithm.

Definition 2. Assume that the first k columns of the reordered matrix have been selected (and brought to the first k positions by permutations). Let $i, i \in 1, 2, \ldots, n$ be an arbitrary row number. Consider a column j, $\;j \in \{k+1, k+2, \ldots, n\}$. The elements in column j which have row numbers greater than i form **the active part of column j with index i.**

Remark 1. The above definition can be considered as a generalization of a definition which is sometimes used in connection with the Gaussian elimination. In the original definition i is equal to k. In the new reordering algorithm (see below) i will in general be greater than k.

Definition 3. The number of non-zero element in the active part of column j with index i is called **the active count c_j^i of column j with index i.**

Now everything is prepared for the formulation of the new ordering algorithm.

Locally Optimized Reordering Algorithm (LORA)

- **Step 1 — Initialization.**
  Set $j = 1$; $i = 0$; and find the active counts c_k, $k = 1, 2, \ldots, n$, of the columns of matrix A.

- **Step 2 — Choice of the next column.**
  Find a column $k \geq j$ with minimal active count c_k^i with index i among the columns $j, \; j+1, \ldots n$ and interchange columns j and k.

- **Step 3 — Row permutations.**
  Perform row permutations such that the q rows with row numbers greater than i that have non-zero elements in column j are moved to locations $i+1, \; i+2, \ldots, i+q$.

- **Step 4 — Stopping check.**
  If $i + q = n$ then STOP.

- **Step 5 — Updating the active counts.**
  Find the active counts of the columns $j+1, j+2, \ldots, n$ with index $i+q$.

- **Step 6 — Preparation for a new column search.**
  Increase i by q, j by 1, and go to **Step 2**.

The structure of a matrix reordered by **LORA** is given in **Fig. 3**. The reordered matrix (it will always be called A when there is no danger of misunderstanding) can be divided into three parts, denoted by different shading on **Fig. 3**. These are described below:

1. The black blocks are **dense**, i.e. they contain no zero elements (these blocks are normally small and rectangular);
2. The shaded part over the black blocks and to the right of them (i.e. the upper right corner of the reordered matrix A) is **sparse**;
3. All elements that are located under the black blocks and to the left of them (i.e. in the lower left corner of the reordered matrix A) are **zeros**.

The black blocks in **Fig. 3** will be called **the dense separator** of the reordered matrix. As stated above, the new algorithm tries to reorder the matrix so that the number of zero elements under the dense separator is as large as possible. It is not obvious that the black blocks in **Fig. 3** are dense. This is proved in the following theorem.

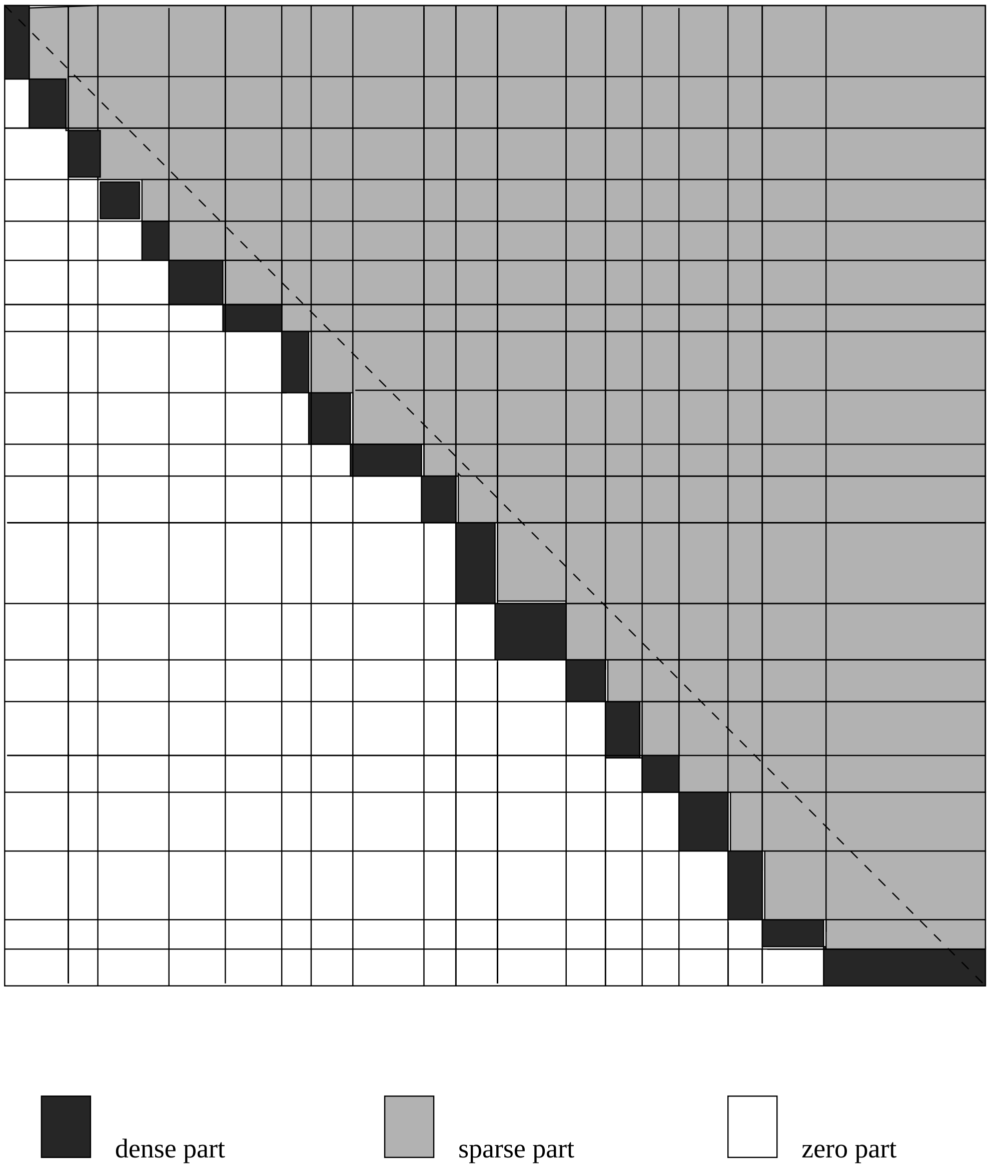

Figure 3
The structure of the reordered matrix.

Theorem 1. All blocks of the dense separator, which is obtained when the new reordering algorithm is applied, are dense matrices.

Proof: Denote by A_{mm}, $m = 1, 2, \ldots, s$, the blocks of the dense separator. Consider any of them, say A_{mm}, and assume that the first column of this matrix is column j of matrix A, while the first row is row $i + 1$. It is clear that there is no zero elements in the first column of A_{mm} (this follows from **Step 4** of **LORA**). Assume that there is at least one zero element in another column of this block; say in column k with $k > j$. Consider now the determination of the j-th column in **Step 2** of **LORA**. It is seen that column k has a smaller active count with index i than column j. Thus, if the assumption that there is a zero element in block A_{mm} is true, then column j cannot be chosen before column k. However, column j has been chosen before column k. This shows that the assumption is false, and proves the theorem. ■

The strategy by which one attempts to achieve as many as possible zero elements under the dense separator is **local**. The algorithm finds consecutively columns that are locally best. If a column j is chosen at some stage of **LORA**, then this column is the column that adds the maximal number of zeros to the zero part of the matrix (because its active count is one of the smallest; see **Step 2** in **LORA**). Therefore the algorithm **is** locally optimized. However, this local optimization does not ensure a global optimization. Examples, in which the number of zero elements that is obtained under the dense separator is not the largest possible, can be found. In Section 6 it will be shown that the algorithm performs well in practice.

However, it must also be shown that the computational cost of **LORA** is not very high. In the next section the complexity of the algorithm will be discussed. It is proved there that **LORA** can be performed in $O(NZ \log n)$ operations. This is more expensive than the cost of the old algorithm, $O(NZ)$, but it is still quite acceptable. If $NZ = O(n)$, that is usually the case for large sparse matrices, then the estimation of complexity converts to $O(n \log n)$.

3 COMPLEXITY OF THE NEW REORDERING ALGORITHM

Consider the six steps of the new ordering algorithm **LORA**. The most time-consuming steps are **Step 1**, **Step 2** and **Step 5**. **Step 1** can be performed in $O(NZ)$ operations. A single **Step 2** can be performed in $O(n)$ operations. While **Step 1** has to be performed only once, **Step 2** is to be executed about n times. Thus, when **LORA** is completed, the total number of operations in **Step 2** will be $O(n^2)$. **Step 5** is also repeatedly carried out. At a single **Step 5**, the algorithm scans the non-zero elements in the rows involved (rows $i+1, i+2, \ldots, i+q$). For each non-zero element the active count of its column (with index i) is reduced by one. It is clear that when **LORA** is completed, all non-zero elements of the matrix will be scanned (except the non-zero elements in the few last rows for which **Step 5** is not performed; see the description of the algorithm in the previous section). Thus, the total number of operations in **Step 5** is $O(NZ)$. The number of operations in the other steps of **LORA** is very small and has no influence on the total complexity of the algorithm. For sparse matrices NZ is normally much smaller than n^2 (very often the relationship $NZ = cn$, where $c < 10$, is satisfied). Therefore for large sparse matrices **Step 1** and **Step 5** are relatively inexpensive and the cost of **Step 2**, $O(n^2)$, becomes dominant for the whole algorithm. For sparse matrix algorithms this is a very considerable cost and, therefore, it is highly desirable to reduce it.

The use of binary trees in the efforts to reduce the total number of operations needed to complete **LORA** is discussed in this section. The discussion is based on a set of fundamental concepts from graph theory, which are assumed to be known. This set includes node, edge, path, root, parent, child, ancestor, descendant, tree, subtree, etc. (a good description of these concepts can be found, for example, in **Aho et al. [1,2]**).

Definition 4. A tree is called **binary tree** if each of its nodes has at most two children.

Below we introduce the important, for the efficient implementation of **LORA**, concept of a binary tree with leveled nodes. The binary tree with leveled nodes could be considered as a modification of the leftist tree discussed in **Knuth** [**23**].

Some modifications of the tree are often to be carried out (many times, consecutively) during the solution of a given problem. This will be needed, for example, if **LORA** is implemented by using binary trees. Such modifications can efficiently be performed if some extra information about the binary tree is stored and used. A special class of binary trees, the binary trees with leveled nodes, can be very useful for some special kinds of modifications. Five arrays, each of them of length n, are necessary to store the information and to update it efficiently when binary trees with leveled nodes are to be used. The contents of the j-th component of these arrays are given below:

1. $LEFT(j)$ — contains any of the two children of the j-th node; $LEFT(j) = 0$ if the corresponding child is missing.
2. $RIGHT(j)$ — contains the other child of the j-th node; $RIGHT(j) = 0$ if the corresponding child is missing.
3. $PREV(j)$ — contains the parent of the j-th node; $PREV(j) = 0$ for the root.
4. $KEY(j)$ — numerical value that determines some order in the set of nodes.
5. $LEV(j)$ — the minimal distance between the j-th node and a descendant of it with less than two children; $LEV(j) = 0$ if the j-th node has less than two children.

The arrays $LEFT, RIGHT$ and KEY are borrowed from Knuth's definition of leftist tree [**23**]. LEV in our case corresponds to $DIST - 1$ (**Knuth** [**23**]), while $PREV$ is an additional array necessary for our purposes. Now we can introduce the class of binary trees with leveled nodes, which will be applied in the implementation of **LORA**.

Definition 5. A binary tree $T = (V, KEY, LEV, PREV, LEFT, RIGHT)$ is called **binary tree with leveled nodes** if the following relations are satisfied:

1. $KEY(PREV(j)) \leq KEY(j) : \quad \forall j \in V, \quad PREV(j) \neq 0$
2. if $\; LEFT(j) \neq 0 \;$ and $\; RIGHT(j) \neq 0 \;$ then
   $LEV(j) = \min(LEV(LEFT(j), LEV(RIGHT(j)) + 1$

An example of a binary tree with leveled nodes is shown in **Fig. 4**. The relations stated by Def. 5 can easily be checked by using the information about KEY and LEV of each node, which is also given in **Fig. 4**.

The binary trees with leveled nodes preserve the main properties of the leftist trees, which are described in **Knuth** [**23**].

Property 1. The *root* is always a node with minimal KEY, i.e.

$$KEY(root) \leq KEY(j) \; , \quad \forall j \in V \; . \tag{3}$$

Property 2. In a binary tree with n leveled nodes LEV of any node is not greater than $\log n$, or more precisely:

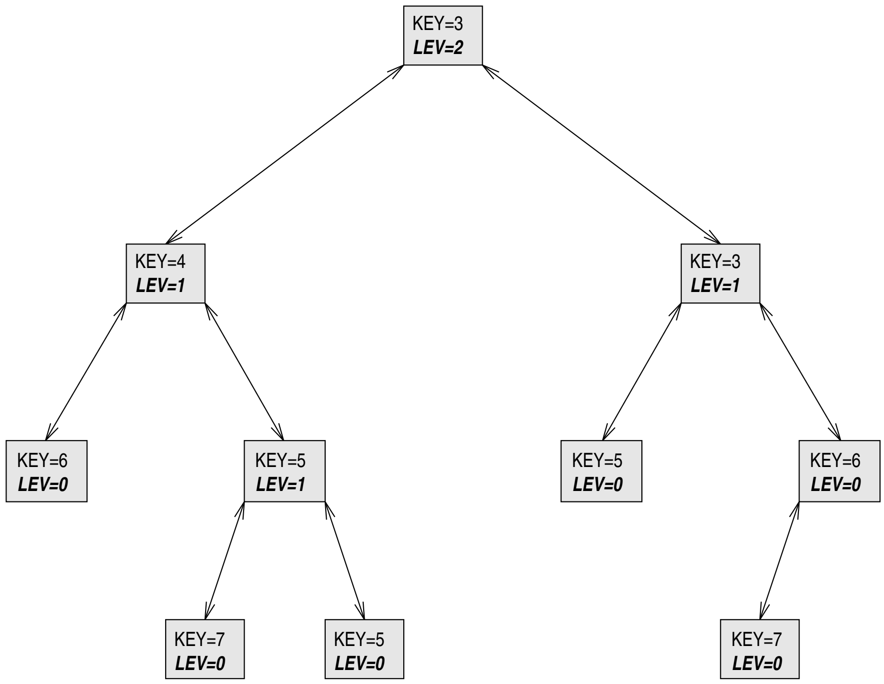

Figure 4
A binary tree with ten leveled nodes.

(4) $$LEV(j) \leq \log(n+1) - 1 \,, \quad \forall j \in V \,.$$

Property 3. The number of operations required to merge two binary trees with leveled nodes with disjoint vertex sets and with roots r_1 and r_2 is proportional to $LEV(r_1) + LEV(r_2)$.

The next propositions follow either directly from the definition of binary trees with leveled nodes or from their properties.

Proposition 1. Consider two binary trees, T_1 and T_2, with leveled nodes. Assume that the number of nodes in each of them is $O(n)$. Then T_1 and T_2 can be merged in $O(\log n)$ operations.

Proof. Follows from Property 2 and Property 3 of the binary trees with leveled nodes. ■

Proposition 2. A binary tree with leveled nodes can be created from an n-element set of vertices in $O(n)$ operations by recursively creating and merging its parts.

The complexity of an important modification (removing a node from a binary tree with leveled nodes) is proved in Theorem 2 below.

Theorem 2. Any node of a binary tree T with n leveled nodes can be removed from T in $O(\log n)$ operations. The resulting tree is a binary tree with $n - 1$ leveled nodes.

Proof. The proof is constructive and gives an $O(\log n)$ algorithm for removing a node. Let q be the node of T that is to be removed. Denote by T_q the subtree whose root is q. The removal process can be performed in three steps:

1. Cut subtree T_q and update LEV of the ancestors of q in the remainder of the tree T_r.
2. Remove the root q of the tree T_q and merge the subtrees rooted at the children of q in a binary tree with leveled nodes T'.
3. Merge the trees T_r and T' obtained at the previous two steps.

From Proposition 1 it follows that each of the last two steps has complexity $O(\log n)$. It remains to prove that the same estimation is valid for the first step. As the tree T may not be balanced, the path from q to the root may be rather long (in general $O(n)$). If we have to update LEV in all this path up to the root then the complexity of the above removing algorithm would be also $O(n)$. However, this is not necessary, because we can stop at the first node for which the old value of LEV is the same as the new one. From item 2. of Def. 5 and Property 2 it follows that the length of the path that needs updating of LEV can not exceed $\log n$. ∎

The results proved above can be applied to show that **LORA** can be completed in $O(NZ \log n)$ operations. Before the formulation of the main result, Theorem 3, it is necessary to explain how the graph theory concepts are related to the matrix concepts used in the definition of **LORA**.

Consider a binary tree $T = (V, KEY, LEV, PREV, LEFT, RIGHT)$ with n leveled nodes. If such a tree is to be used in connection with **LORA**, then the nodes of the tree must be associated with the columns of matrix A. The active counts of column j is to be associated with the value of $KEY(j)$. These counts are modified in **LORA**. This leads to the necessity to perform modifications of $T = (V, KEY, LEV, PREV, LEFT, RIGHT)$ after the choice of each column (**Step 2** of **LORA**). Appropriate values of the active count of column j with some index i is to be used as $KEY(j)$ during the consecutive modifications of $T = (V, KEY, LEV, PREV, LEFT, RIGHT)$. These remarks about the interpretation of the matrix concepts, that are used in **LORA**, in terms of a binary tree with leveled nodes are sufficient to explain how these trees are used to implement efficiently **LORA** in a sparse solver for systems of linear algebraic equations.

Theorem 3. The new reordering algorithm **LORA** can be implemented in $O(NZ \log n)$ operations when binary trees with leveled nodes are used.

Proof. It is sufficient to prove that **Step 1, Step 2** or **Step 5** can be performed in no more than $O(NZ \log n)$ operations; the number of operations required in the remaining two steps is negligible (see also the beginning of this section). The proof of this theorem could be performed in four stages. In the first three stages the numbers of operations needed in **Step 1, Step 2** and **Step 5** respectively will be found. The results obtained during the first three stages will be used to obtain the overall complexity of the algorithm in the last stage.

Stage 1. The counts of the columns must be found before the creation of the binary tree. This can be done (as mentioned above) in $O(NZ)$ operations. A binary tree T with n leveled nodes must be formed in **Step 1**. Such a tree can be created in $O(n)$ operations (see Proposition 2). As $NZ \geq n$, the total cost of **Step 1** is $O(NZ)$.

Stage 2. Consider **Step 2**. The root of T can be chosen as a column with minimal active count (see Property 1), so the task of finding the column with minimal active count is trivial. However, it is important to prepare the tree for the continuation of the algorithm. This can be done by (i) removing the root from the tree and (ii) merging the subtrees rooted at its children. These tasks require $O(\log n)$ operations (see Theorem 2). Since **Step 2** has to be carried out about n times, the total number of operations in **Step 2** of **LORA** is $O(n \log n)$.

Stage 3. Some of the contents of array KEY is updated during **Step 5** (because the active counts of some columns change their values). This can destroy the partial order of the nodes in the tree that is induced by the contents of the array KEY. In this case the tree is to be updated. If the value of $KEY(j)$ is changed, then (i) node j is removed from the tree, (ii) the value of $KEY(j)$ is updated and (iii) the node is added to the tree (according to the updated value of $KEY(j)$). The second task is trivial, the first one can be performed in $O(\log n)$ operations (Theorem 2), the third task also requires $O(\log n)$ operations (see Proposition 1). The above three tasks are to be repeated about NZ times, which means that $O(NZ \log n)$ is the total cost of **Step 5**.

Stage 4. The operation costs, which were found in the previous three stages, as well as the fact, that the other steps of **LORA** are very cheap, show that **LORA** can be implemented in $O(NZ \log n)$ operations by using binary trees with leveled nodes. ∎

In the above considerations we assumed that $KEY(j)$ represents the active count of the j-th column. This is consistent with the first version of our reordering algorithm, described above. In this particular case a more straightforward implementation by linked lists exists, that leads to cheaper $O(NZ)$-algorithm, but not so adaptive to various KEY functions. The reason to use the algorithm based on binary trees is its applicability to great variety of KEY functions, that appear when additional criterions are applied in case of ties (to choose between several columns with best active count). Ties can be exploited to improve different characteristics of the reordering and are subject to our future research.

4 USING THE REORDERED MATRIX IN THE SOLUTION PROCESS

The reordered matrix (by the two algorithms discussed in the previous two sections) is to be used in the solution of systems of linear algebraic equations $Ax = b$. The solution process (which is based on Gaussian elimination) will be sketched in this section. The algorithm used, **Y12M3**, consists of eight steps. A simpler version of this algorithm is discussed in **Gallivan et al.** [**17**] and **Zlatev** [**27**], where **Y12M3** was used as **a direct solver**. The use of the new reordering algorithm **LORA** in the first step of the algorithm given below as well as the addition of a modified preconditioned orthomin algorithm (in an attempt to improve the accuracy of the first solution) result in considerably better results (see Section 6).

Y12M3:
a parallel sparse solver
for systems of linear algebraic equations
by Gaussian elimination

- **Step 1** — Reorder the matrix
- **Step 2** — Partition the matrix
- **Step 3** — Perform the first phase of the factorization
- **Step 4** — Perform the second phase of the factorization
- **Step 5** — Carry out a second reordering
- **Step 6** — Perform the third phase of the factorization
- **Step 7** — Find a first solution (back substitution)
- **Step 8** — Improve the first solution by a modified preconditioned orthomin algorithm

A short description of the actions that are to be performed during the eight steps of **Y12M3** is given below. It should be mentioned that the description is in fact carried out for an example where it is assumed that the matrix is to be partitioned as a 4×4 block-matrix. This is only done in order to facilitate the exposition of the results. From the context it becomes quite clear that the same kind of computations are to be carried out also in the case where other partitionings are used (i.e. where the matrix is partitioned not as a 4×4 block-matrix, but as a $q \times q$ block-matrix).

4.1. Step 1 — Reorder the matrix

Two reordering algorithms have already been discussed in the previous two sections. Results obtained by these two algorithms will be compared in Section 6.

4.2. Step 2 — Partition the matrix

The partitioning algorithm is based on the following idea. The matrix is divided into p **parts**, each of them containing a certain number of rows. In all experiments that have been performed in **Gallivan et al., [17]** and in **Zlatev [27]** as well as in the experiments that will be discussed in the next section, p is set to be equal to eight (the maximal number of processors of the **ALLIANT FX/80** computer). Such a restriction (the number of parts to be equal to the number of processors) is not necessary for the solver itself; p may also be either larger or less than the number of processors. However, the requirement for efficient computations may impose some restrictions (for some computers, for example, it may be desirable to impose a requirement that the number of parts is a multiple of the number of processors).

Assume that p is equal to the number of processors. Then each processor will receive one part of the matrix, and will carry out the work during **Step 3** on its own part. It is important to partition the matrix so that the work done (by the different processors in **Step 3**) is approximately the same. It is difficult to satisfy such a requirement. One can try to obtain parts that have approximately the same number of rows. This is the criterion that has been chosen by us. It is easy to make such a partitioning because there is no requirement for square diagonal blocks (a block is called "diagonal" if all elements under it are zeros).

One could try to partition the matrix so that the different parts contain approximately the same number of non-zero elements (instead of approximately the same number of rows). This partitioning is slightly more expensive, but for some matrices it performs better.

4.3. Step 3 — Perform the first phase of the factorization

Once the matrix is partitioned, the first phase of the factorization can be started. Each processor produces zeros under the main diagonal of the diagonal block in its part of the matrix. This is a straightforward operation and may be carried out by calling a slightly modified version of one of the subroutines described in **Gallivan et al., [17]** and **Zlatev [27]**.

Assume that the partitioning is performed with $p = 4$. The blocks obtained by such a partitioning can be depicted as in **Fig. 5**.

$$\begin{vmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ 0 & A_{22} & A_{23} & A_{24} \\ 0 & 0 & A_{33} & A_{34} \\ 0 & 0 & 0 & A_{44} \end{vmatrix}$$

Figure 5
Partitioning with $p = 4$.

Assume that A_{11} is a rectangular $q \times r$ block with $q > r$. Then an upper triangular matrix $U_{11} \in R^{r \times r}$ will be obtained after the factorization of this block. The last $q - r$ rows of A_{11} will contain zeros only. Thus, the block A_{11} can, after the factorization, be partioned into an upper triangular matrix U_{11} (containing the first r rows) and a zero matrix (containing the last $q-r$ rows). The same partitioning can be performed for the other three blocks in the first part of the matrix; A_{12}, A_{13} and A_{14} (however, the lower blocks in these three matrices will not be zero matrices like the lower block of A_{11}). The first phase of the factorization in the second and the third parts is carried out in a quite similar way. The last block, A_{44}, has another structure. In general, it is also rectangular, but the number of its columns, r, is greater than the number of its rows, q. An upper triangular matrix $U_{44} \in R^{q \times q}$ and a matrix $X_{44} \in R^{q \times (r-q)}$ (formed by the last $r-q$ columns) will be obtained after the first phase of the factorization. It is convenient to perform a similar partitioning in the whole last block-column of the matrix in **Fig. 5**. After the first phase of the factorization the matrix can be partioned as shown in **Fig. 6**.

$$
\left|
\begin{array}{ccccc}
U_{11} & U_{12} & U_{13} & U_{14} & X_{14} \\
0 & W_{12} & W_{13} & W_{14} & V_{14} \\
 & & & & \\
0 & U_{22} & U_{23} & U_{24} & X_{24} \\
0 & 0 & W_{23} & W_{24} & V_{24} \\
 & & & & \\
0 & 0 & U_{33} & U_{34} & X_{34} \\
0 & 0 & 0 & W_{34} & V_{34} \\
 & & & & \\
0 & 0 & 0 & U_{44} & X_{44}
\end{array}
\right|
$$

Figure 6

Partitioning of the matrix at the end of the first phase of the factorization. U_{ii} **$(i = 1, 2, 3, 4)$ are upper triangular matrices. The elements in the fourth part as well as the elements in the first block-rows of the other three parts will not be modified in the further calculations.**

The calculations in the different parts of the matrix can be performed concurrently. The transition from the matrix in **Fig. 5** to the matrix in **Fig. 6** can performed efficiently on four processors (or four clusters).

4.4. Step 4 — Perform the second phase of the factorization

During the **second phase of the factorization** zeros are produced in the blocks W_{ij} $(i = 1, 2, 3, \quad j = 2, 3, 4)$. The process is carried out **"by diagonals"**. First, zeros are produced in the blocks W_{12}, W_{23} and W_{34} by using the pivots in the blocks U_{22}, U_{33} and U_{44}, respectively. These computations can be carried out in parallel on three processors (or on three clusters of processors). When the computations with the first diagonal are completed, the computations with the second diagonal can be started. Zeros are produced in the blocks W_{13} and W_{24} by using the pivots in the blocks U_{33} and U_{44} respectively. These calculations can be carried out in parallel on two processors (or on two clusters of processors). Finally, zeros are produced in block W_{14} by using the pivots in block U_{44}. These computations are to be carried out on one processor (or on one cluster of processors).

The partitioning of the matrix after the second phase of the factorization is given in **Fig. 7**.

Only the computations in the different blocks within a diagonal can be performed concurrently. Indeed, when the computations to produce zeros in the blocks in the first diagonal are carried out, all other blocks (to the right of the blocks that are transformed to zero-blocks) are modified. Therefore, it is not possible to avoid the loss of concurrency during the second phase of the factorization.

On computers like **ALLIANT** it is better to perform the computations during the second phase of the factorization not by blocks, but "by target rows". Consider the block W_{12}. Each row of this block can be modified (using appropriate pivots in block U_{22}) independently of the other rows. The same statement holds also for the other blocks. This technique is used on the **ALLIANT** computers. It may be worthwhile to apply it also on

$$\begin{vmatrix} U_{11} & U_{12} & U_{13} & U_{14} & X_{14} \\ 0 & 0 & 0 & 0 & V_{14} \\ & & & & \\ 0 & U_{22} & U_{23} & U_{24} & X_{24} \\ 0 & 0 & 0 & 0 & V_{24} \\ & & & & \\ 0 & 0 & U_{33} & U_{34} & X_{34} \\ 0 & 0 & 0 & 0 & V_{34} \\ & & & & \\ 0 & 0 & 0 & U_{44} & X_{44} \end{vmatrix}$$

Figure 7
Partioning of the matrix at the end of the second phase of the factorization. U_{ii} $(i = 1,2,3,4)$ are upper triangular matrices. Only the elements in the blocks V_{i4}, $i = 1,2,3$, will be modified in the further computations.

some other parallel computers.

4.5. Step 5 — Carry out the second reordering

After the fourth step only the elements in the blocks V_{14}, V_{24} and V_{34} need to be modified. These blocks, if they are gathered together, form a square matrix. Moreover, it is reasonable to expect them to be rather dense. Therefore it is worthwhile to reorder the matrix again by pushing these blocks to the lower right corner of the matrix and then to switch to dense matrix technique. This is the fifth step, **the second reordering**. This is a straightforward step; the result is given in **Fig. 8**.

$$\begin{vmatrix} U^*_{11} & U^*_{12} \\ & \\ 0 & D_{22} \end{vmatrix}$$

Figure 8
Partioning the matrix after the second reordering. U^*_{11} is upper triangular (formed by the blocks U_{ij}, $i = 1,2,3,4$, $j = 1,2,3,4$). U^*_{12} is rectangular (formed by the blocks X_{i4}, $i = 1,2,3$). D_{22} is square (formed by the blocks V_{i4}, $i = 1,2,3$).

The amount of work during the fifth step is very small in comparison with the work during the other steps.

4.6. Step 6 — Perform the third phase of the factorization

The matrix D_{22} has to be factorized during this step, **the third phase of the factorization**. As mentioned above, this is done by dense matrix subroutines. The portable subroutines that are used in this step are the same as the subroutines that are used in two other packages, **Y12M1** and **Y12M2**; see **Gallivan et al.,** [**17**] and **Zlatev** [**27**].

There are several other options that could be activated if the packages used in them are available at the site where **Y12M3** is to be run. On the **ALLIANT FX/80** computer it is worthwhile to use the subroutines from package **PARALIN** (see again **Gallivan et al.,** [**17**] and **Zlatev** [**27**]. Another package of subroutines for dense matrices, developed especially for **ALLIANT FX/80**, is also available; it gives results that are similar to those obtained by package **PARALIN**. Finally, an option in which the new **LAPACK** subroutines (see **Anderson et al.,** [**4**]) are called, can be useful on all sites where package **LAPACK** is available. It should be mentioned here that **LAPACK** is already implemented on many sites and the process of implementing this package on more and more sites is continuing.

4.7. Step 7 — Find a first solution (back substitution)

The fact that both sparse matrix techniques and techniques for treatment of dense matrices are used during the factorization process should be taken into account during the back substitution. There are four factors, L_s, U_s, L_d, U_d, that are to be handled during the back substitution. The index s refers to "sparse", while the other index, index d, refers to "dense". The sparse factors L_s and U_s are trapezoidal matrices, whose dimensions are $n \times (n - n_d)$ and $(n - n_d) \times n$ respectively, where $n_d = NDENSE$ is the order of the dense matrix D_{22} which is treated during the third phase of the factorization (in Step 6). The dense factors, L_d and U_d, are triangular matrices of order n_d. The calculations are straightforward: one starts with L_s and proceeds successively with L_d, U_d and U_s. It is important to carry out the calculations in which the sparse factors are involved in parallel (especially when an iterative improvement of the first solution is to be carried out by some preconditioned iterative method and, thus, the sparse factors are to be used several times during the iterative process. Therefore the factors L_d, U_s are reordered by the algorithm proposed by **Anderson and Saad** [**5**]; see more details about the parallel back substitution in **Zlatev** [**27**].

4.8. Step 8 — Improve the first solution by a modified preconditioned orthomin algorithm

In principle, this is an optional step, but it is strongly recommended (because of stability problems; see below) to carry out this step, and this is the reason for including it in the algorithm. Different iterative methods can successfully be applied in Step 8. A modified preconditioned orthomin algorithm has been chosen. The original orthomin is proposed by **Vinsome** [**25**], a good description of the original orthomin method could be found, for example, in **Eisenstat et al.,** [**10**]). A prescribed number of Krylov vectors (say, k; very often $k = 1$ is used in the codes) is to be given before the beginning of the calculations when the original orthomin is to be applied. The code tries adaptively, during the iterations, to find a good (for the particular matrix treated) number of Krylov vectors, when the modified orthomin algorithm is used. The modified preconditioned orthomin algorithm is described in details in **Zlatev** [**27**] **- Chapter 11**.

5 STABILITY CONSIDERATIONS

Stability problems may arise when **Y12M3** is used. The stability problems will be discussed below. It is assumed that the new reordering algorithm **LORA** is used, however the same ideas can be applied to the old algorithm.

Let us consider any of the diagonal blocks. Assume that the block chosen has q_i rows and r_i columns. The code attempts to determine if there is a diagonal with non-zero elements (in the block chosen) which are large in some sense. More precisely, the code tries to find a diagonal with non-zero elements which are greater (in absolute value) than the product of 0.01 and the largest element (in absolute value) in their rows. This could be considered as a stability requirement. There is no guarantee that there is a diagonal with r_i elements which are large according to the above definition. However a diagonal that has t_i elements that satisfy the stability requirement (with $t_i \leq r_i$) can always be found. If $t_i < r_i$, then the code will move the $r_i - t_i$ columns for which the requirement is not satisfied to the end. This is a heuristic device that works pretty well in practice (see also the accuracy results presented in Section 6).

It is desirable (from the stability point of view) to have more rows than columns: if the algorithm fails to find a stable pivot in a given row, then it just proceeds with the next row. It should be mentioned that the numbers of rows in the diagonal blocks are normally not much larger than the number of columns (at least the differences between rows and columns in the diagonal blocks in the new algorithm are considerably less than the corresponding quantities in the old algorithm). Therefore the new algorithm tends to be more unstable during the first phase of the factorization than the old one.

No stability check is performed during the second phase of the factorization (it is assumed that the pivots found during the first phase are good enough also for the second phase). In this way some computations are saved and, what is even more important, the computations can easily be performed concurrently. However, this may cause problems. Therefore, it is strongly recommended to use this method together with some iterative method (see Section 4.8) even if no "small" elements are dropped during the computations. The number of rows that are processed during the second phase of the factorization by the second algorithm is as a rule less than the corresponding figure for the old algorithm. Therefore the new algorithm tends to be slightly more stable during the second phase of the factorization than the old one.

The method may lead to poor preservation of sparsity. During the first phase of the factorization the pivots are restricted to the diagonal blocks. No sparsity check is carried out during the second phase of the factorization: it is assumed again that the pivots found during the first phase are good enough (not only in regard to numerical stability, but also in regard to preserving sparsity) for the second phase. The sparsity is normally preserved well when a large drop-tolerance is used during the factorization. Therefore both the first phase of the factorization and the second phase of the factorization are normally performed by removing non-zero elements by using a large drop-tolerance (see **Gallivan et al., [14-17]**, **Zlatev [26,27]**). This means that the factors L and U may be inaccurate, and should only be used as preconditioners in some iterative method. The iterative method that is actually used is a modified preconditioned orthomin algorithm (see Section 4.8).

6 NUMERICAL RESULTS

General sparse matrices from the well-known Harwell-Boeing set of sparse test matrices have mostly been used in the experiments (see **Duff et al., [9]**). We selected a set containing all unsymmetric square matrices whose order is greater than 900. There are 25 such matrices. Two smaller matrices were added to the set: the first of them contains many non-zero elements, the second one arises from an air pollution model. It should be noted that the same set of test matrices has been used in **Gallivan et al., [14-17]** and **Zlatev [27]**. Some information about the matrices used is given in **Table 1**.

No.	**MATRIX**	**order**	NZ	$COND$
1	sherman3	5005	20033	1.8E+ 5
2	gemat11	4929	33108	1.5E+ 6
3	gemat12	4929	33044	1.8E+ 6
4	lns_3937	3937	25407	1.2E+ 6
5	lns3937a	3937	25407	1.2E+ 6
6	saylr4	3564	22316	1.2E+ 7
7	sherman5	3312	20793	6.0E+ 3
8	or678lhs	2529	90158	2.1E+ 6
9	orsreg_1	2205	14133	8.1E+ 3
10	west2021	2021	7310	4.5E+ 7
11	hwatt_2	1856	11550	3.1E+ 5
12	hwatt_1	1856	11550	4.7E+ 4
13	west1505	1505	5414	5.8E+ 7
14	nnc1374	1374	8588	2.6E+13
15	mahistlh	1258	7682	1.7E+ 4
16	pores_2	1224	9613	3.6E+ 5
17	gre_1107	1107	5664	1.8E+ 7
18	sherman4	1104	3786	1.6E+ 3
19	gaff1104	1104	16056	3.4E+10
20	sherman2	1080	23094	2.8E+ 3
21	orsirr_1	1030	6858	1.2E+ 4
22	sherman1	1000	3750	5.5E+ 3
23	jpwh_991	991	6027	2.0E+ 2
24	west0989	989	3518	4.2E+ 7
25	pde_9511	961	4681	1.4E+ 7
26	mc_fe	765	24382	7.7E+ 1
27	steam2	600	5660	3.2E+ 0

Table 1
Matrices used in the experiments
($COND$ is a condition number estimation calculated by Y12M; see Zlatev et al. [28])

The right-hand side vectors b of the systems $Ax = b$ were created so that all components of the solution vectors are equal to 1. The code attempts to estimate the second two-norm of the error vector (**Zlatev [27]**) and stops the computation when this estimation becomes less than $ACCUR = 10^{-5}$. If this condition cannot be satisfied then, as mentioned above, the drop-tolerance is reduced and a new trial is started. The accuracy requirement was satisfied for all test-matrices for both the old algorithm and the new algorithm. In our case we are also able to calculate the exact error (because the right-hand side vectors are calculated in a special way; see above). Both the error estimation and the exact error were checked for all systems solved. Some results obtained with the new algorithm, **LORA**, are given in **Table 2**.

MATRIX		**error estimation**	**exact error**
1	sherman3	1.E-09	4.E-10
2	gemat11	8.E-08	5.E-14
3	gemat12	6.E-08	1.E-13
4	lns_3937	9.E-07	5.E-08
5	lns3937a	8.E-08	4.E-13
6	saylr4	3.E-06	7.E-11
7	sherman5	5.E-07	3.E-07
8	or678lhs	5.E-07	3.E-08
9	orsreg_1	7.E-06	7.E-06
10	west2021	1.E-09	5.E-12
11	hwatt_2	2.E-07	4.E-07
12	hwatt_1	8.E-06	3.E-05
13	west1505	2.E-10	2.E-12
14	nnc1374	2.E-10	3.E-06
15	mahistlh	4.E-06	7.E-07
16	pores_2	8.E-07	3.E-13
17	gre_1107	7.E-07	1.E-11
18	sherman4	9.E-06	1.E-06
19	gaff1104	2.E-07	1.E-08
20	sherman2	5.E-07	1.E-13
21	orsirr_1	3.E-06	5.E-06
22	sherman1	1.E-09	1.E-10
23	jpwh_991	4.E-07	5.E-08
24	west0989	1.E-06	7.E-11
25	pde_9511	5.E-07	3.E-07
26	mc_fe	1.E-05	2.E-06
27	steam2	5.E-08	2.E-16

Table 2
Error estimations and exact errors found when systems with Harwell-Boeing matrices are solved by using LORA .
The initial drop-tolerance is 0.0625 **and the accuracy required is** $1.E-5$ **.**

All experiments were carried out on an **ALLIANT FX/80** computer by using all eight processors (runs on one processor were also carried out in order to calculate the speed-up of the computational process). Therefore the code is certainly tunned for this computer. Nevertheless, the ideas are fairly general and the code will also run efficiently on other parallel computers with shared memory. It should be stressed here that the code is **portable**. This means that it will run on any other computer, although it will perhaps be inefficient when used without tunning. The parallelism is exploited mainly by calling subroutines concurrently. Therefore, it should not be very difficult to tune the code for other parallel computers with shared memory: the **ALLIANT** directives (mainly the important directive "cncall" by which subroutines are called concurrently in a loop) must be replaced with appropriate other directives that are available on the computer that is to be used.

All experiment were run by using a large drop-tolerance ($RELTOL = 0.0625$). Roughly speaking, this means that all non-zero elements that become (during the factorization) less than the product of $RELTOL$ and the largest in absolute value element in the active part of the row currently treated are dropped (not used in the further computations). The factors so calculated L and U are normally inaccurate and, therefore, they are used as preconditioners in some iterative method. The particular method used was a modified preconditioned ORTHMIN algorithm (see **Zlatev [27]**. If the preconditioned iterative method does not converge, then the drop-tolerance is reduced and new preconditioners are calculated. This procedure could be repeated several times. If several trials were made in order to obtain a sufficiently accurate solution, then the sum of the times spent in all trials is shown on the plot in **Fig. 9**. Each matrix is represented by its number in **Table 1**; the abscissa is the computing time of the old algorithm, while the ordinate is the computing time of the new algorithm.

More detailed results concerning the computing times spent in the different parts of the solution process for one of the largest Harwell-Boeing matrices, **saylr4**, are given in **Table 3**. The computing times obtained for this matrix are in some sense typical for the large matrices among the set listed in **Table 1**. The order of the dense matrix (used in the third phase of the factorization) is very small, only 6, when the new algorithm is used in connection with matrix **saylr4**. This is not typical (normally the dense matrices are considerably larger, but they are still much smaller than the corresponding dense matrices obtained when the old algorithm is used). The fact that the order of the dense matrix is only 6 shows that the new algorithm reorders very well this matrix.

Consider the ordering times. The new algorithm uses considerably more time. Of course, this is not a surprise: its cost is considerably greater than the cost of the old algorithm. It should also be mentioned here that both algorithms are in fact sequential (see the speed-ups in **Table 3**).

The factorization time for the new algorithm is less than the factorization time for the old algorithm. However, the old algorithm has a greater speed-up. This is caused by the fact that the dense matrix obtained in the third phase of factorization by the old algorithm is considerably larger than the corresponding dense matrix for the new algorithm (these matrices are of orders 691 and 6 respectively). The speed up factors for the dense factorizations are considerably greater than the speed-up factors for sparse factorization (especially when high-quality software based on the use of BLAS-3 kernels is used; see, for example, **Gallivan et al., [12,14]**). This explains why the factorization time for the new algorithm is better, but its speed-up is worse (compared with the corresponding quantities for the old algorithm).

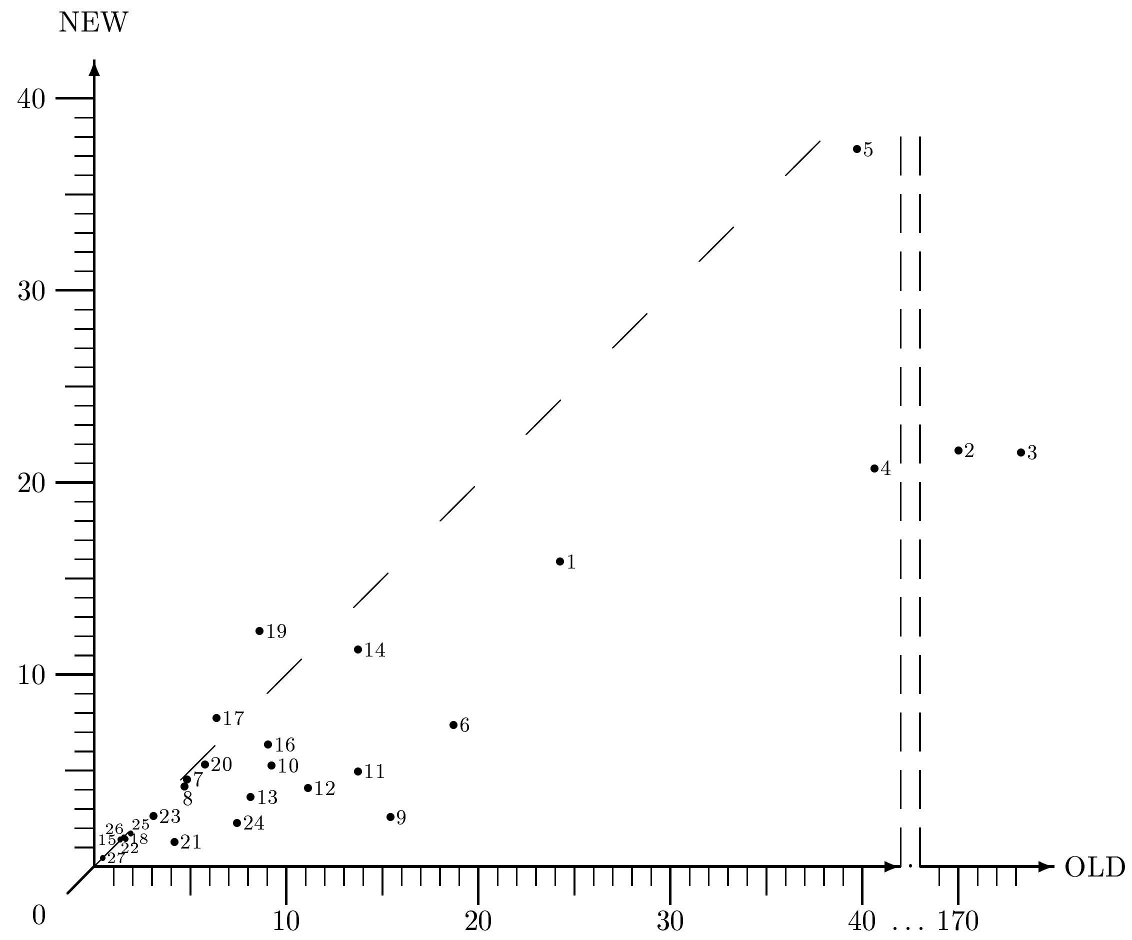

Figure 9
Comparison of the CPU times (in seconds) for the matrices from Table 1.
Each matrix is represented by its number in Table 1;
the abscissa is the computing time of the old algorithm,
while the ordinate is the computing time of the new algorithm.

Stage	Old algorithm		New algorithm	
	Time	Speed-up	Time	Speed-up
Ordering	1.07	1.1	1.88	1.1
Factorization	5.07	5.2	0.64	3.8
Solution	12.57	4.1	4.87	5.3
Total time	**18.71**	**4.2**	**7.39**	**4.1**

Table 3
Computing times and speed-ups for the matrix saylr4.

The situation changes a little when the solution times are considered. During the solution phase the new algorithm is both less time consuming and its speed-up is greater. Now the sparse solution process is much better parallelizable than the dense solution process. This is due to the efficient Anderson-Saad algorithm used in the back substitution of the sparse part; see **Anderson and Saad** [**5**] or **Zlatev** [**27**]. The performance during the solution parts of both the old and the new algorithms may perhaps be improved by applying the algorithm proposed in **Alvarado et al.,** [**3**].

The total time for the new algorithm is much better than the total time for the old one, while the speed-up for the old one is slightly better.

The speed-ups obtained by the new algorithm when the matrix **saylr4** and two other matrices are run on different number of processors are shown in **Fig. 10**. It is seen that the speed-ups grow in a rather regular way when the number of processors is increased.

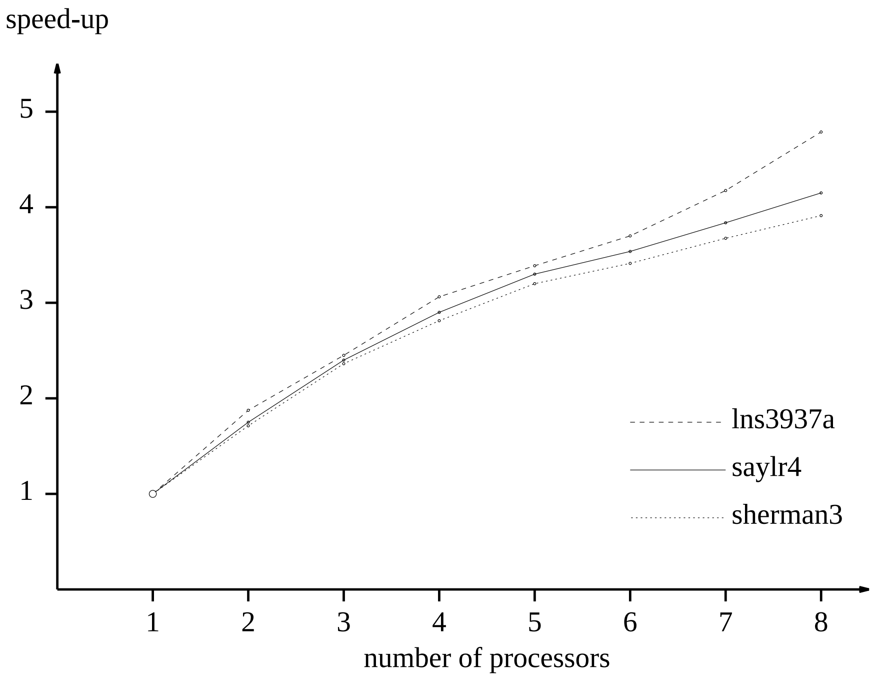

Figure 10
Speed-ups for three Harwell-Boeing matrices.

It is illustrative to show how the two algorithms, the old one and **LORA**, partition the matrix **saylr4** which is used in the experiments in this section. The corresponding sizes of the blocks are given in **Fig. 11**. It is clearly seen that while the diagonal blocks are long rectangulars when the matrix is partitioned by the old algorithms, the diagonal blocks obtained when **LORA** is used are close to square matrices. It must be emphasized here that this matrix, **saylr4**, is one of the matrices for which the new algorithm, **LORA**, produced a very good partitioning. It should also be emphasized, however, that the new algorithm produces nearly always better partitioning than the old reordering algorithm; and very often the partioning obtained by **LORA** is much better than that produced by the old algorithm.

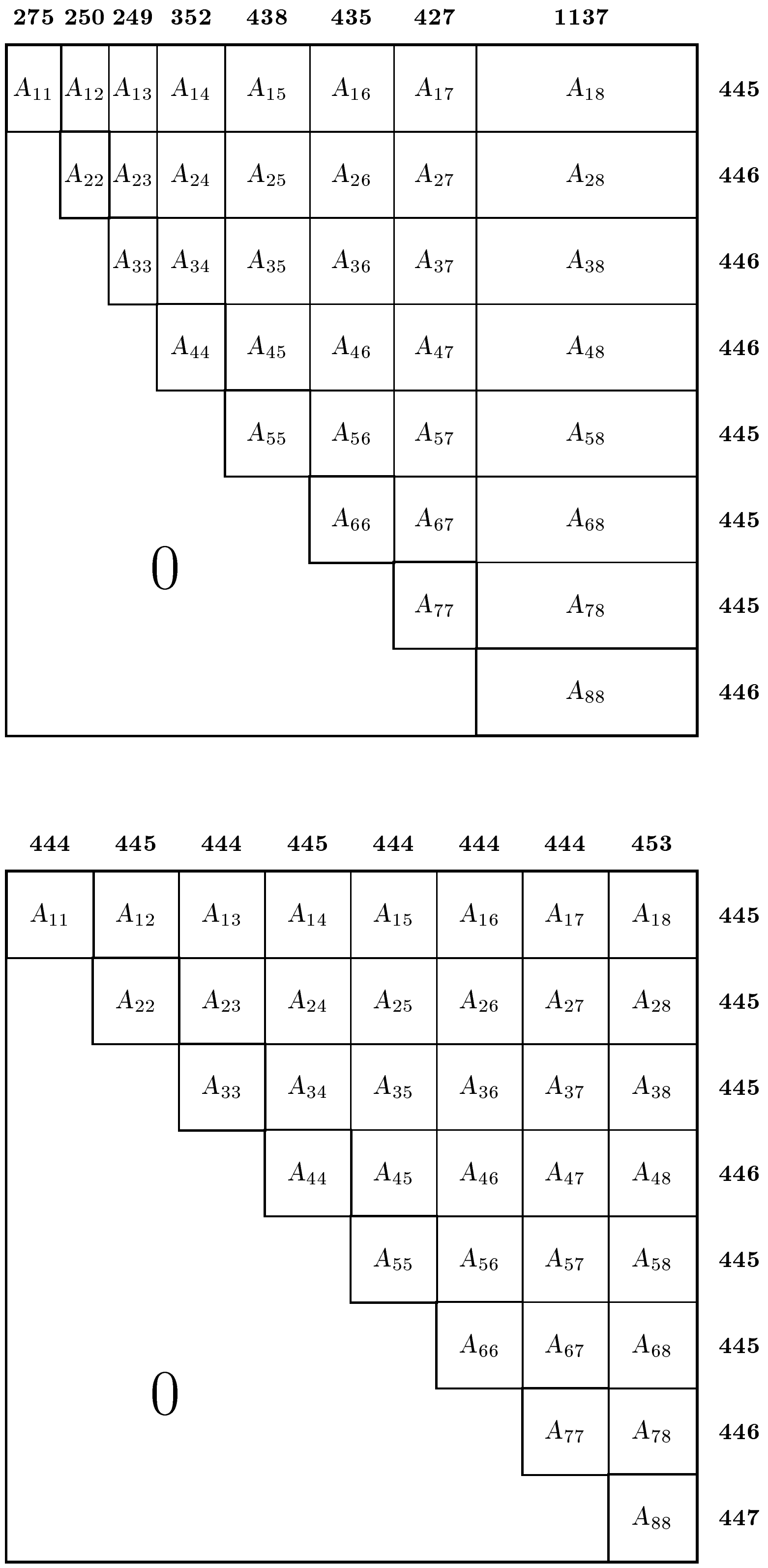

Figure 11
Partitioning of the matrix saylr4 into eight parts after reordering by the old algorithm (above) and by the new algorithm, LORA (below).

Some runs with very large matrices (created by one of the generators for general sparse matrices, **CLASSF2**, described in **Zlatev [27]**) have also been carried out. Numerical results are given in **Table 4**. It is seen that the speed-up increases with increasing the order of the matrix. It is also seen that the accuracy requirements (which are the same as for the Harwell-Boeing matrices) are satisfied.

Matrix parameters		**Results**				
order	NZ	**time on 1 pr.**	**time on 8 pr.**	**speed-up**	**error estimation**	**exact error**
10000	50110	49.26	15.15	3.25	4.E-07	3.E-08
20000	100110	160.11	41.58	3.85	3.E-07	2.E-08
50000	250110	868.13	189.65	4.53	3.E-07	2.E-08
100000	500110	3301.90	673.27	4.90	2.E-07	1.E-08
200000	1000110	12876.13	2522.35	5.10	2.E-07	8.E-09

Table 4
Results obtained by running LORA on systems created by using one of the generators for general sparse matrices from Zlatev [27].

7 CONCLUDING REMARKS AND PLANS FOR FUTURE WORK

The main conclusion from all experiments that have been carried out (not only the experiments discussed in the previous section) is that the new algorithm performs better than the old one (although the time needed to perform the preliminary ordering by the new algorithm is always greater than the corresponding time spent by the old algorithm). Thus, it is worthwhile to spent some more time in order to obtain more zero elements under the dense separator. However, the reordering algorithm should not be very expensive. The new algorithm, which has an arithmetic cost of $O(NZ \log n)$ operations, seems to be a nearly optimal choice.

The parts of the code that could be run in parallel were identified and written as separate subroutines. On ALLIANT these subroutines are called concurrently by using the compiler directive "cncall". It is believed that the same strategy can also be used on other computers with shared memory (in many cases it should be enough to replace the directive "cncall" by an appropriate directive on computer that is to be used). However, some modifications in the code and/or some reorganization of the computations may be needed on other computers. This means that some runs on other computers are to be carried out, and we intend to do this in the near future.

The portability of the code is a crucial issue when parallel machines are to be used. In general, it is possible to increase considerably the efficiency of the code by applying some specific features of the parallel machine that is used. However, if this is done, then it is rather difficult to run the code on another machine. On the other hand, if the code is portable, but the parts of it that can be run in parallel are well separated in subroutines that can be run concurrently, then both the efficiency will be quite satisfactory and it will be relatively easy not only to run it on other computers, but also to obtain a good performance by using minimal efforts to tune it. Therefore, the second strategy has been chosen. The code is portable and has been run on some other computers (as, for example, on CONVEX).

If the matrix of the system solved is very large, then usually there many columns with the same number of non-zero elements. This means that normally there will be a large set of columns that have minimal active count on **Step 2** of the new reordering algorithm. In the present implementation no attempt to find the most suitable one among them is made (the column that happened to be found first is chosen). However, it may be appropriate to select (or to try to select) the most suitable one. Some attempts to design an algorithm where such a strategy is implemented are carried out now.

There are several other parts of the code that could be improved. As an illustration only, it should be mentioned that the strategy for switching to dense matrix technique can be improved. Some ideas used in the codes described in **Zlatev [27]** can be applied in the efforts to find out when it is best to switch to dense matrix technique. Some work in this direction has already been started.

ACKNOWLEDGEMENTS

The work of the Danish representatives was partially supported by the BRA III Esprit project APPARC (# 6634) and by a Danish Government Scholarship (# 1992-9222-1).

The work of K. A. Gallivan was partially supported by the National Science Foundation (Grant # CCR-9120105).

REFERENCES

1. **A. V. Aho, J. E. Hopcroft and J. D. Ullman:** "The design and analysis of computer algorithms". Addison-Wesley, Reading, Ma., USA, **1976**.

2. **A. V. Aho, J. E. Hopcroft and J. D. Ullman:** "Data structures and algorithms". Addison-Wesley, Reading, Ma., USA, **1983**.

3. **F. L. Alvarado, A. Pothen and R. Schreiber:** "Highly parallel sparse triangular solution". Report No. CS-92-09, Department of Computer Science, The Pennsylvania State University, University Park, PA 16802, USA, **1992**.

4. **E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov and D. Sorensen:** "LAPACK: Users' Guide" SIAM (Society for Industrial and Applied Mathematics, Philadelphia, **1992**.

5. **E. Anderson and Y. Saad:** "Preconditioned conjugate gradient methods for general sparse matrices on shared memory machines". In: **"PARALLEL PROCESSING**

FOR SCIENTIFIC COMPUTING" (G. Rodrigue, ed.), pp. 88-92. Society for Industrial and Applied Mathematics, Philadelphia, **1989**.

6. **M. Arioli, I. S. Duff, N. I. M. Gould and J. K. Reid:** "Use of the P^4 and P^5 algorithms for in-core factorization of sparse matrices". SIAM J. Sci. Statist. Comput., **11(1990)**, 913-927.

7. **T. A. Davis and P.-C. Yew:** "A nondeterministic parallel algorithm for general unsymmetric sparse LU factorization". SIAM J. Matrix. Anal. Appl., **3(1990)**, 383-402.

8. **I. S. Duff, A. M. Erisman and J. K. Reid:** "Direct methods for sparse matrices". Oxford University Press, Oxford-London, **1986**.

9. **I. S. Duff, R. G. Grimes and J. C. Lewis:** "Sparse matrix test problems". ACM Trans. Math. Software, **15(1989)**, 1-14.

10. **S. C. Eisenstat, H. C. Elman and M. H. Schultz:** "Variational methods for nonsymmetric systems of linear equations". SIAM J. Numer. Anal., **20(1983)**, 345-357.

11. **A. M. Erisman, R. G. Grimes, J. G. Lewis and G. W. Poole, Jr.:** "A structurally stable modification of Hellerman-Rarick's P^4 algorithm for reordering unsymmetric sparse matrices". SIAM J. Numer. Anal., **22(1985)**, 369-385.

12. **K. A. Gallivan, W. Jalby and U. Meier:** "The use of BLAS3 in linear algebra on a parallel processor with hierarchical memory". SIAM J. Sci. Statist. Comput., **8(1987)**, 1079-1084.

13. **K. A. Gallivan, B. Marsolf and H. Wijshoff:** "A large-grain parallel sparse system solver". In: **"PROCEEDINGS OF THE SIAM CONFERENCE ON PARALLEL PROCESSING FOR SCIENTIFIC COMPUTING"**, pp. 23-28. SIAM, Philadelphia, **1991**.

14. **K. A. Gallivan, R. J. Plemmons and A. H. Sameh:** "Parallel algorithms for dense linear algebra computations". SIAM Rev., **32(1990)**, 54-135.

15. **K. A. Gallivan, A. H. Sameh and Z. Zlatev:** "Solving general sparse linear systems using conjugate gradient-type methods". In: **"PROCEEDINGS OF THE 1990 INTERNATIONAL CONFERENCE ON SUPERCOMPUTING**, June 11-15 1990, Amsterdam, The Netherlands". ACM Press, New York, **1990**.

16. **K. A. Gallivan, A. H. Sameh and Z. Zlatev:** "A parallel hybrid sparse linear system solver". Computing Systems in Engineering, **1(1990)**, 183-195.

17. **K. A. Gallivan, A. H. Sameh and Z. Zlatev:** "Parallel direct methods for general sparse matrices". Preprint No.9. **"NATO ASI ON COMP. ALG. FOR SOLVING LINEAR EQUATIONS: THE STATE OF THE ART**, University of Bergamo, Italy, **1990**.

18. **J. A. George and J. W. Liu:** "Computer solution of large sparse positive definite systems". Prentice-Hall, Englewood Cliffs, N. J., **1981**.

19. **J. A. George,J. W. Liu and E. Ng:** "Row ordering schemes for sparse Givens rotations". Lin. Alg. Appl., **61(1984)**, 55-81.

20. **J. R. Gilbert:** "An efficient parallel sparse partial pivoting algorithm". Report No. 88/45052-1. Chr. Michelsen Institute, Department of Science and Technology, Centre for Computer Science, Fantoftvegen 38, N-5036 Fantoft, Bergen, Norway, **1988**.

21. **E. Hellerman and D. C. Rarick:** "Reinversion with the preassigned pivot procedure". Programming, **1(1971)**, 195-216.

22. **E. Hellerman and D. C. Rarick:** "The partitioned preassigned pivot procedure (P^4)". In: **"SPARSE MATRICES AND THEIR APPLICATIONS"** (D. J. Rose and R. A. Willoughby, eds.), pp. 67-76. Plenum Press, New York, **1972**.

23. **D. Knuth:** "The art of computer programming". Vol. 3, pp. 151-152. Addison-Wesley, Reading, **1973**.

24. **A. F. van der Stappen, R. H. Bisseling and J. G. G. van der Vorst:** "Parallel sparse LU decomposition on a mesh network of transputers". SIAM J. Matrix Anal. Appl., **14(1993)**, to apear.

25. **P. K. W. Vinsome:** "Orthomin, an iterative method for solving sparse sets of simultaneous linear equations". In: **"PROCEEDINGS OF THE FOURTH SYMPOSIUM ON RESERVOIR SIMULATION"**, pp. 140-159. Society of Petroleum Engineers of AIME, **1976**.

26. **Z. Zlatev:** "Use of iterative refinement in the solution of sparse linear systems". SIAM J. Numer. Anal., **19(1982)**, 381-399.

27. **Z. Zlatev:** "Computational methods for general sparse matrices". Kluwer Academic Publishers, Dordrecht-Toronto-London, **1991**.

28. **Z. Zlatev, Ph. Vu, J. Waśniewski and K. Schaumburg:** "Condition number estimators in a sparse matrix software". SIAM J. Sci. Statist. Comput., **7(1986)**, 1175-1186.