Combinatorial Aspects in Sparse Elimination Methods

by Matthias Bollhöfer[1], and Olaf Schenk[2]

Technical Report CS-2004-006
Department of Computer Science, University of Basel

Submitted.

[1]Dept. of Mathematics, TU Berlin, Germany, e-mail: bolle@math.tu-berlin.de

[2]Department of Computer Science, University of Basel, Klingelbergstrasse 50, CH-4056 Basel, Switzerland email: olaf.schenk@unibas.ch

Technical Report is available at
`http://www.computational.unibas.ch/cs/scicomp`

Combinatorial Aspects in Sparse Elimination Methods

Matthias Bollhöfer[*1] and **Olaf Schenk**[**2]

[1] Dept. of Mathematics, TU Berlin
[2] Dept. of Computer Science, University of Basel

Received 15 November 2004, revised 30 November 2004, accepted 2 December 2004
Published online 3 December 2004

Key words sparse direct methods, LU decomposition, sparse matrices, reordering techniques, elimination tree, maximum weight matching, supernodes.
MSC (2000) 65F05, 65F50, 05C50, 05C85.

In this paper we will give an overview on combinatorial algorithms in sparse elimination methods. Beside well established techniques that have been developed in the last twenty years a modern viewpoint of LU decomposition is presented that illustrates how the evolution of techniques in the last decade improved the performance of sparse direct solvers.

Copyright line will be provided by the publisher

1 Introduction

Sparse direct solvers are a core part of many problems in computational science and engineering. While on one hand modern computer architectures provide larger and larger memory resources and faster processors, on the other hand the need for solving large scale application problems often compensates these developments. The request for large sparse fast and memory efficient solvers has been a challenge for many years and remains an open field for further developments. In particular modern computer architectures have influenced massively the design of modern solvers similar to the BLAS–oriented concept of LAPACK [4]. In this paper, we compare and review some important combinatorial aspects and main algorithmic features for solving sparse systems and show that the algorithmic improvements of the past twenty years have reduced the time required to factor general sparse matrices by almost three orders of magnitudes. Combined with significant advances in the performance to cost ratio of computing hardware during this period, current sparse solver technology makes it possible to solve those problems quickly and easily that might have been considered by far too large until recently. Therefore, this paper reviews the basic and the latest developments for sparse direct solution methods that have lead to modern LU decomposition methods.

The paper is organized as follows. In Section 2 we will introduce basic combinatorial tools like the elimination graph and moreover the elimination tree and its generalization, the column elimination tree. The elimination tree plays a key role in understanding sparse direct solution methods and we will review several important topic which are still used for the foundation of modern fast solvers.

* M. Bollhöfer: e-mail: bolle@math.tu-berlin.de, Phone: +00 49 30 31429381, Fax: +00 49 30 31479706
** O. Schenk: e-mail: olaf.schenk@unibas.ch, Phone: +00 41 61 2671465 Fax: +00 41 61 2671461. The work was supported by the Swiss Commission of Technology and Innovation under contract number 7036.1 ENS-ES.

Copyright line will be provided by the publisher

Section 3 introduces the practical computation of sparse LU decompositions. Beside combinatorial aspects that allow to compute the factors L and U in time proportional to the number of nonzeros, the elimination tree allows to predict dense submatrices in the factorization. In this way several columns of the factors are collected in a one dense block. This is the basis for the use of higher levels of BLAS and to exploit fast computation using the cache hierarchy.

One central keypoint to improve the factorization time remains the a priori ordering of the system. Section 4 essentially reviews two reordering strategies that are based on the undirected matrix pattern.

One recent technique based on maximum weight matching that improves the diagonal dominance of the given system by scaling and unsymmetric reordering is presented in Section 5.

Finally we will demonstrate in Section 6 how the ongoing developments in sparse direct solution methods from the past until today have accelerated state–of–the–art solution techniques and still give a potential for further improvements in the future.

2 Combinatorial Aspects in Sparse LU Decompositions

We begin with some basic terminology. For a matrix $A = (a_{ij})_{i,j=1,\dots,n} \in \mathbb{R}^{n,n}$ we denote by $A_{k:l,p:q}$ the submatrix $(a_{ij})_{i=k,\dots,l,j=p,\dots,q}$ of A. Here we always assume that $1 \leqslant k \leqslant l \leqslant n$ and $1 \leqslant p \leqslant q \leqslant n$. Next we introduce some definitions from graph theory associated with a given matrix $A \in \mathbb{R}^{n,n}$. We assume that the reader is familiar with some elementary knowledge from graph theory, see e.g. [12, 19] and some simple computational algorithms based on graphs [1].

Definition 2.1 For a matrix $A \in \mathbb{R}^{n,n}$ we define the associated graph $G(A)$ by the pair (V, E), where $V = \{1, \dots, n\}$ and the set of edges $E = \{(i, j) : a_{ij} \neq 0\}$.

Whenever we refer to an *undirected* graph we mean that $(i, j) \in G(A)$ if and only if $(j, i) \in G(A)$. In this case we may also use $\{i, j\}$ instead of (i, j).

Example 2.2 We will use Figure 1 to illustrate basic combinatorial properties in sparse elimination methods. At this point we ignore the '$\circ$' in Figure 1. Then the graph in the center of Figure 1 represents the directed graph $G(A)$.

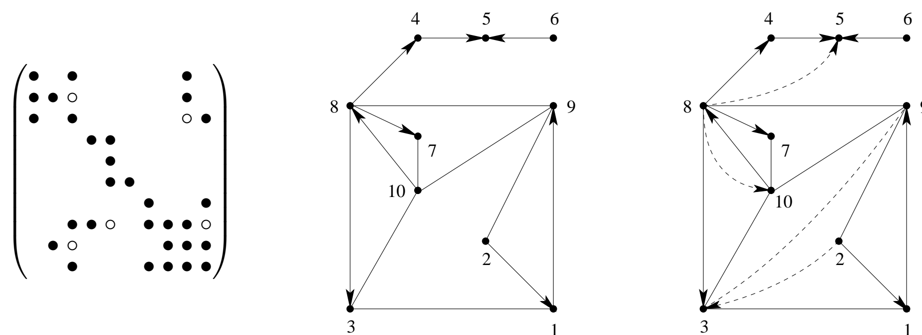

Fig. 1 Pattern and graph of a matrix before and after LU decomposition ('$\circ$' denotes fi ll-in)

As usual we will denote directed edges (i, j) by an arrow whereas if (i, j) and (j, i) exist we omit the arrow and just draw a line (see Figure 1).

Copyright line will be provided by the publisher

Definition 2.3 Let $A \in \mathbb{R}^{n,n}$. For an edge (i, j) in $G(A)$ we will write $i \overset{G(A)}{\rightarrow} j$ or simply $i{\rightarrow}j$ if the graph is clear from the context. Likewise we will write $i \overset{G(A)}{\Rightarrow} j$ or simply $i{\Rightarrow}j$ if the exists a path from i to j in $G(A)$.

2.1 The Elimination Graph

We assume that our matrix A possesses an LU decomposition (see e.g [25]). It is well–known [19], if $A = LU$, where L and $U^\top$ are lower triangular matrices, then in the generic case we will have $G(L{+}U) \supset G(A)$, i. e., we will only get additional edges unless some entries cancel by "accident" during the elimination. In the sequel we will ignore cancellations. Throughout the paper we will always assume that the diagonal entries of A are nonzero. We also assume that $G(A)$ is connected and that A is nonsingular. The following Theorem (see [22] in the references therin) shows how the factors L and U fill as by-product of the Gauss algorithm.

Theorem 2.4 *Suppose that $A = LU$ and assume that no cancellations occur in the elimination process. Then $i \overset{G(L+U)}{\rightarrow} j$ if and only if there exists a path*

$$i \overset{G(A)}{\rightarrow} x_1 \rightarrow \cdots \overset{G(A)}{\rightarrow} x_k \overset{G(A)}{\rightarrow} j$$

such that $x_1 \ldots, x_k < \min(i, j)$.

In other words, by Theorem 2.4 we get a (new) edge $i{\rightarrow}j$ for every path $i \Rightarrow j$ through vertices less than $\min(i, j)$.

Example 2.5 The effect of fill edges is illustrated by adding the fill–in entries ("$\circ$") to the matrix in Figure 1 and the inserting the additional edges in the rightmost graph of Figure 1. For example, we have a path $9 \rightarrow 2 \rightarrow 1 \rightarrow 3$ and a path $3 \rightarrow 1 \rightarrow 9$. This introduces two fill edges $(9, 3)$ and $(3, 9)$ or equivalently an undirected edge $\{3, 9\}$.

Definition 2.6 The graph $G(L + U)$ that is derived from $G(A)$ by applying Theorem 2.4 is called the *filled graph* and it will be denoted by $G^+(A)$.

A problem in general is to predict the filled graph $G^+(A)$ and the fastest known method to compute it, is Gaussian elimination. The situation simplifies if the graph is undirected.

2.2 The Elimination Tree

We now assume that $G(A)$ is undirected. In this case the "symmetrized" version of Figure 1 becomes an undirected graph (Figure 2). In the case of symmetrically structured matrices one can drastically simplify the description of the LU decomposition. The key tool is called the *elimination tree* which allows to easily predict fill–in in many ways, e. g. where are the fill edges or how many edges occur and where will we have cliques. Note that *cliques* are complete subgraphs, i. e., any two vertices are connected by an edge. This information can be exploited to derive better orderings, to compute L and U more efficiently and it allows to group suitable columns/rows of L and U into blocks ("supernodes"). Although elimination trees essentlially deal with symmetric matrices and do not consider pivoting, these techniques can be successfully transferred to the general case using the *column elimination tree* [20].

An undirected and connected graph is called a *tree*, if it does not contain any cycle and there exists precisely one node which is labeled as *root*. As usual we call a node j *parent* of i, if there exists an edge $\{i, j\}$ in the tree and j is closer to the root. We refer to i as a *child*

Copyright line will be provided by the publisher

Fig. 2 Matrix with symmetric pattern and the associated undirected graph

of j. The nodes of the subtree rooted at j are called *descendants* of j whereas j is called their *ancestor*.

Definition 2.7 Given the filled graph $G^+(A)$ of a symmetrically structured matrix we define its *elimination tree* by performing a depth–first–search in $G^+(A)$. I. e., starting from node n, at any step we are faced with some node m and its its unvisited neighbors $i_1, \dots, i_k$. From those we first pick the index j with the largest number $j = \max\{i_1, \dots, i_k\}$ and continue the search with j. A leaf of the tree is reached, when all neighbours have already been visited or have a larger number than the node itself.

For those readers that are not familiar with the depth–first–search algorithm, we refer to [1]. It is important to notice that the depth–first–search in $G^+(A)$ significantly differs from a depth–first–search in general, i. e., in the filled graph we will never encounter the case that we visit a node m such that one of its neighbours $j > m$ has not been visited yet. This follows from Theorem 2.4 for the case of undirected graphs. There are also other ways to define the elimination tree, see e. g. [35].

Example 2.8 We illustrate the depth–first–search in Figure 3. By dashed lines in the left graph of Figure 3 we denote the filled edges in $G^+(A)$. The right graph of Figure 3 is used to trace the ongoing depth–first–search. Starting at vertex 10 this is done by using solid lines for the depth–first search and dotted lines for the remaining edges. I. e. we visit the vertices in the order $10, 9, 8, 7;\ 6, 5, 4;\ 3, 2, 1$. Vertex 8 has three children $7, 6, 3$ that have to be visited and following the tie–break rule that the largest index is preferred, we first follow 7, then 6 and its descendants and finally 3 and its descendants.

The depth–first–search traverses $G^+(A)$ by first visiting a parent node followed by its children. This induces a tree rooted at n which we will denote by $T(A)$.

Example 2.9 The elimination tree $T(A)$ with root 10 and its is illustrated in the leftmost graph of Figure 4. To indicate the filled graph $G^+(A)$ we add in the middle graph of Figure 4 the remaining edges by dotted lines.

Remark 2.10 It follows immediately from the construction of $T(A)$ that the additional edges of $G^+(A)$ are only between vertices and their ancestors (so–called "back–edges"), but there are never "cross–edges" between unrelated vertices.

The elimination tree can be used to derive several conclusions based on its structure. We cite some of them in the following Theorem.

Copyright line will be provided by the publisher

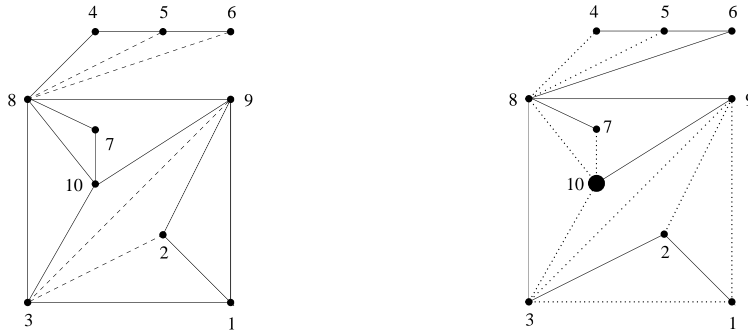

Fig. 3 Filled graph and depth–fi rst–search

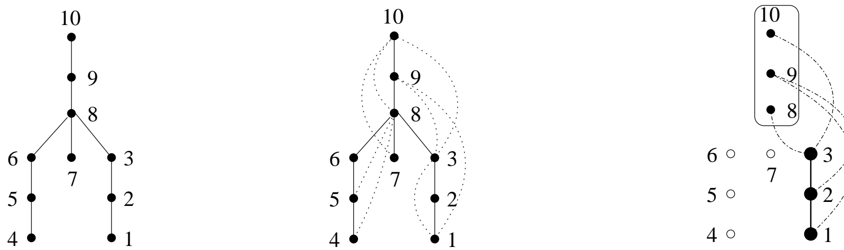

Fig. 4 Elimination tree $T(A)$ without and with remaining edges

Fig. 5 Fill–in detection for column 3 of L

Theorem 2.11 *Suppose that $A \in \mathbb{R}^{n,n}$ possesses an LU decomposition, i.e. $A = LU$, where $L = (l_{ij})_{i,j=1,\dots,n}$ and $U = (u_{ij})_{i,j=1,\dots,n}$. Let $G(A)$ denote its undirected graph and $T(A)$ the associated elimination tree.*
Then for any $j = 1,\dots,n$ the numerical values of $L_{j:n,j}$ (resp. $U_{j,j:n}$) depend only on the numerical values of $L_{s:n,s}$, $U_{s,s:n}$ such that s is a descendant of j in the elimination tree $T(A)$.

Proof. see e.g. [35] □

Remark 2.12 One major consequence of Theorem 2.11 is that we can rearrange the order of elimination and still get the same matrices L, U up to a symmetric permutation as long as children precede their parent.

Definition 2.13 Given a tree $T(A) = (V, E)$, where $V = \{1,\dots,n\}$ we call an ordering $(i_1,\dots,i_n)$ of the vertices a "topological ordering" if children always precede their parent.

Example 2.14 In Figure 4 beside the natural ordering $1, 2, \dots, 10$ one could also use e. g. $1, 4, 2, 7, 5, 6, 3, 8, 9, 10$. By Definition 2.13 this is also a topological ordering.

Further conclusions can be easily derived from the elimination tree using Theorem 2.4. For example given only the information on the elimination tree T and the pattern of A we can detect all fill–in edges.

Copyright line will be provided by the publisher

Corollary 2.15 *Using the assumptions of Theorem 2.11 we consider i, j such that $i > j$. Then there exists a (fill–in) edge $i \overset{G^+(A)}{\to} j$ if and only if there exists a common descendant k of i, j in $T(A)$ such that $a_{ik} \neq 0$.*

Based on Corollary 2.15 we can predict the nonzero structure of any column $L_{j+1:n,j}$. To do this we fix j and consider all descendants $k \leqslant j$ of j in $T(A)$. For any of these k we examine $a_{ik} \neq 0$ such that $i > j \geqslant k$. Since by Remark 2.10 we never have cross–edges, $i > j$ must be an ancestor of j in $T(A)$. Thus we will get a fill–in edge $\{i, j\}$. It follows that we will have fill–in edges $\{i, j\}$ for any vertex i that is adjacent with the subtree rooted at j.

Definition 2.16 Given a matrix $A \in \mathbb{R}^{n,n}$ that is structurally symmetric and its elimination tree $T(A)$ we consider a vertex j in $T(A)$.
Define $T[j]$ as the subtree of $T(A)$ that is rooted at vertex j.
Given an undirected graph $G = (V, E)$ and $W \subseteq V$, we denote the set of *adjacent nodes* by $\text{adj}_G(W) = \{i \in V \setminus W : \{i, k\} \in E \text{ for some } k \in W\}$.

It follows that $\text{adj}(T[j]) \equiv \text{adj}_{G(A)}(T[j])$ is the set of all i such that $i > j$ and $a_{ik} \neq 0$ for some vertex k in $T[j]$.

We reformulate the nonzero pattern of $L_{j+1:n,j}$ using the notion of adjacent nodes.

Lemma 2.17 *We consider the assumptions of Theorem 2.11. For given $i > j$ we have that $l_{ij} \neq 0$ if and only if $i \in \text{adj}(T[j])$.*

Example 2.18 We illustrate the nonzero pattern for column 3 of L using the elimination tree from Figure 4. We first consider the subtree rooted at 3. Then for any descendant k of 3 in $T[3]$ we check whether there are nonzero entries of type $a_{ik} \neq 0$ such that $i > j$. Here k corresponds to $1, 2, 3$ and the nonzero entries are a_{91}, a_{92}, a_{83} and $a_{10,3}$ (see Figure 5).
From this it follows that $\text{adj}(T[3]) = \{8, 9, 10\}$. Thus the nonzero structure below the diagonal in column 3 of consists of l_{83}, l_{93} and $l_{10,3}$. This already covers fill–in edges.

2.3 Computing the elimination tree

We have seen that many properties of the symmetrically structured LU decomposition can be derived from the elimination tree. The elimination tree itself can be easily described by a vector p of length n such that for any $i < n$, $p[i]$ denotes the parent node while $p[n] = 0$ corresponds to the root. For a given ordering of the unknowns the computation of the elimination tree is based on growing the tree step by step from the leafs up to the root. Suppose that at some step k, all $i < k$ belong to certain subtrees of $T(A)$. Let $j_1, \ldots, j_r$ denote the children of k, then at step k we merge the subtrees $T[j_1], \ldots, T[j_r]$ to one new subtree $T[k]$ rooted at k. The problem is how to find the children of k. The answer is given by Corollary 2.15. Vertex k is an ancestor of certain nodes from some subtree $T[j_i]$, if $a_{ki} \neq 0$ for a member i of $T[j_i]$. In other words, for all $i < k$ such that $a_{ki} \neq 0$ we traverse the subtree that covers i up towards to its current root j_i. Then k is the parent node of j_i. Finally all these subtrees $T[j_i]$ are rooted at children j_i of k and thus merged together at vertex k.

Example 2.19 Consider the elimination tree $T(A)$ from Figure 4. and suppose that we have already performed step $k = 1, 2, \ldots, 7$. It is easy to verify that three subtrees $T[3], T[6]$ and $T[7]$ have grown until step $k = 7$. At step $k = 8$, we have $a_{83} \neq 0$, $a_{84} \neq 0$ and $a_{87} \neq 0$. So we have to start from $i = 3, 4, 7$ and traverse their trees up towards to their roots using the parent vector $p[i]$. This situation is illustrated in Figure 6.

Copyright line will be provided by the publisher

For an algorithmic description see e.g. [35]. We note that the most expensive part of the construction of $T(A)$ is the search from a member i of subtree $T[j_i]$ towards its current root j_i. One simple way to accelerate the search consists of path compression [45].

2.4 The column elimination tree

The concept of the elimination tree so far is reduced to the case that the given matrix has an undirected graph and even more, at most diagonal pivoting is required, if any. The situation totally changes in the general case. One concept that can be read as the generalization of the elimination tree is the so called *column elimination tree* [20, 23].

Definition 2.20 Given a matrix $A \in \mathbb{R}^{m,n}$ we define

$$G_\cap(A) := G(A^\top A),\ G_\cap^+(A) := G^+(A^\top A),\ T_\cap(A) := T(A^\top A)$$

as the *column intersection graph*, the *filled graph* of the column intersection graph and the *column elimination tree* of A.
Denote further by $T_\cap[k]$ the subtree of $T_\cap(A)$ rooted at node k.

The most important property of the column elimination tree is that even using partial pivoting by rows, the nonzero pattern of U can be described in term of $G_\cap^+(A)$, as stated in the following theorem by [20].

Theorem 2.21 *Suppose that $A \in \mathbb{R}^{n,n}$ is nonsingular and let $PA = LU$, where P is a permutation matrix and $L, U^\top$ are lower triangular matrices. Then the columns of L can be rearranged to a lower triangular matrix $\hat{L}$ such that $G(\hat{L} + U) \subset G_\cap^+(A)$.*

Note that for L one can derive tighter bounds using a sparse symbolic QR decomposition [17]. In [21] it is shown that one can form $G(A^\top A)$ and thus $T_\cap(A)$ without explicitly computing the matrix product.
We have seen in Theorem 2.11 that the dependencies of the numerical values can be described by the hierarchy induced by the elimination tree. A similar, also weaker result can be shown in the general case using the column elimination tree [23, 9].

Theorem 2.22 *Suppose that $A \in \mathbb{R}^{n,n}$ is nonsingular and that $PA = LU$ represents the LU decomposition with some partial pivoting strategy.*
If $U_{jk} \neq 0$, then j already belongs to $T_\cap[k]$. If $L_{ik} \neq 0$, then k already belongs to $T_\cap[i]$.

We can thus conclude from the hierarchy in the column elimination tree that at most children modify their ancestors.

Example 2.23 For the unsymmetric matrix A from Figure 1 we illustrate in Figure 7 the associated matrices $A^\top A$ as well as its fill '$\circ$' in the Cholesky decomposition.

In Figure 7 we see that the column elimination tree reduces to the simple chain $1, 2, \ldots, 10$. This is the reduction to the trivial case.

3 Efficient computation of the LU decomposition

We have already seen that the compact representation of the elimination tree can be used to derive all information concerning fill–in and numerical dependencies. We now illustrate how the elimination tree can be used to improve the numerical computation of L and U. Here we

Copyright line will be provided by the publisher

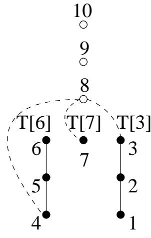

Fig. 6 $T[6]$, $T[7]$ and $T[3]$ are merged into $T[8]$

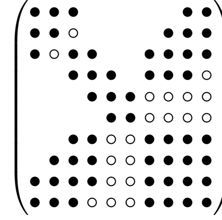

Fig. 7 Pattern of $A^\top A$ and its Cholesky factor

concentrate on the general unsymmetric case that is more challenging and more difficult than the symmetric case, since in any step pivoting is required to guarantee numerical stability. In particular the fact that pivoting and the factorization must be interlaced requires a completely different treatment than in the case without pivoting. Consider the situation when computing the LU decomposition column by column.

Algorithm 1

for $k = 1, \dots, n$

  update column k *of L and U via the equation* $A_{1:n,k} = L_{1:n,1:k-1}U_{1:k-1,k} - L_{1:n,k}u_{kk}$

  1. step. solve $L_{1:k-1,1:k-1}U_{1:k-1,k} = A_{1:k-1,k}$

  2. step. $L_{k:n,k} := A_{k:n,k}$

    for $i < k$ *such that* $u_{ik} \neq 0$

      $L_{k:n,k} := L_{k:n,k} - L_{k:n,i}u_{ik}$

  3. step. $u_{kk} = l_{kk}$

    $L_{k:n,k} = L_{k:n,k}/u_{kk}$

Note that it is easy to add pivoting to Algorithm 1 after step 2 by interchanging l_{kk} with some sufficiently large $|l_{mk}|$, where $m \geqslant k$ before u_{kk} is defined. This does not change the character of Algorithm 1. The art of efficiently computing column k of L and U consists of how the sparse forward solve $L_{1:k-1,1:k-1}U_{1:k-1,k} = A_{1:k-1,k}$ in step 1 is efficiently implemented and, a fast update of $L_{k:n,k}$ in step 2.

We will now present three techniques that address these problems and help significantly speeding up the computation of L and U.

3.1 Fast sparse forward solve based on depth–first–search

In order to solve

$$L_{1:k-1,1:k-1}U_{1:k-1,k} = A_{1:k-1,k} \tag{1}$$

efficiently, we have to predict the pattern of $U_{1:k-1,k}$. Suppose that $a_{jk} \neq 0$. Then the forward solve (1) involves $L_{1:k-1,j}$ and the nonzero pattern of $U_{1:k-1,k}$ covers that of $L_{1:k-1,j}$. Consequently for all $i < k$ such that $l_{ij} \neq 0$, we also need $L_{1:k-1,i}$ to solve (1). These columns can be determined by a backtracking strategy [24]. Initially we start with all $j < k$ such that $a_{jk} \neq 0$ and search backwards in L by a depth–first–search. This means that we have to visit column j of $L_{1:k-1,1:k-1}$ and as next index we select i such that $l_{ij} \neq 0$ and i has not been visited yet. The final order is taken in reverse post–order, i. e., whenever the search comes

Copyright line will be provided by the publisher

to an end at some node j and we start backtracking, j is added to the list. Finally the list is reverted.

Example 3.1 We illustrate this depth-first search by computing $U_{1:8,9}$ in Example 1. The result is sketched in Figure 8. We start with $j = 1, 2, 8$ one after another and backtrack until there is no node left over that has not been visited yet. In Figure 8 we sketch $L_{1:8,1:8}$ and $A_{1:8,9}$ and the arrows indicate the trace of the depth–first search.

The search starting from vertex 1 already yields the whole data structure for $U_{1:8,9}$. The search starting from node 3 or 8 does not give any new information since these vertices have already been visited. The backtracking algorithm returns the nodes $2, 8, 3, 1$ in post–order, where 2 and 8 are the nodes where one has to backtrack. So columns $1, 3, 8$ and 2 of $L_{1:8,1:8}$ are needed to compute u_{19}, u_{39}, u_{89} and u_{29}.

As indicated by Figure 8 the order of elimination returned by the depth-first-search is not necessarily the natural order where all vertices are sorted in increasing order. However, no sorting is necessary since this ordering is topological equivalent to the natural order of elimination as pointed out in [24]. The equivalent orderings depend on the tie–breaking strategy in the depth-first-search. E. g. if in Figure 8 one would have chosen l_{31} before l_{21} then precisely the natural ordering $1, 2, 3, 8$ would have been computed.

3.2 Accelerated depth–first–search using pruning

While the backtracking strategy itself an be applied to any sparse triangular solve (even if L did not arise from the LU decomposition), a further improvement can be made using the specific properties of the LU decomposition. The strategy called *symmetric pruning* [15] allows to significantly reduce the cost of the depth–first search. After elimination step k the update of the reduced matrix requires an operation like

$$A_{k+1:n,k+1:n} \to A_{k+1:n,k+1:n} - L_{k+1:n,k} U_{k,k+1:n}.$$

For all columns j such that $u_{kj} \neq 0$, the nonzero pattern of $L_{k+1:n,k}$ is stamped into the nonzero pattern of column j. Likewise the nonzero pattern of $U_{k,k+1:n}$ is stamped into any row i of the reduced matrix whenever $l_{ik} \neq 0$. Here it is necessary to ignore numerical cancellations and only to refer to the index structure. Otherwise the pattern were not inherited. This observation can be used to truncate the depth-first-search on an earlier stage. Suppose that $l_{jk} \neq 0$ and $u_{kj} \neq 0$ for some $j > k$. Then for any $i > j$, such $l_{ik} \neq 0$ we will also have that $l_{ij} \neq 0$. Thus after performing step j of the LU decomposition we can *prune* $L_{k+1:n,k}$ and ignore $L_{j+1:n,k}$ for the depth first search. l_{jk} remains as a link to refer to column $L_{j+1:n,j}$. Note further that any column need to be pruned at most once.

Example 3.2 We illustrate in Figure 9 the original factor L and the *pruned* factor L. Entries that are not considered anymore are labeled using a '$\times$'. According to Figure 1, at step $j = 3$, $l_{31}, u_{13} \neq 0$ and thus column $k = 1$ is pruned. But since there are no further nonzeros, no '$\times$' occurs. Similarly at step $j = 9$, column $k = 2, 3, 8$ are pruned resulting in one visual entry that is skipped. At step $j = 10$ one could in theory prune column $k = 7, 9$ if this were not the last step.

Although pruning is stated here for LU decomposition without pivoting, it is still applicable when being combined with partial pivoting. I. e., at step k of the LU decomposition row

Copyright line will be provided by the publisher

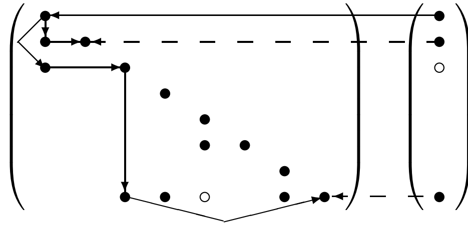

Fig. 8 Depth-first-search in $L_{1:8,1:8}$ to compute the nonzero pattern of $u_{1:8,9}$

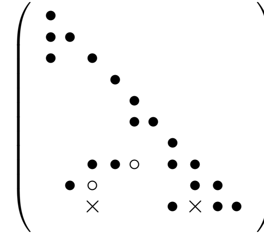

Fig. 9 Pruned lower triangular factor L

k may be interchanged with some row $m > k$. After that, pruning is applied to the permuted matrix.

3.3 The supernodal approach

The sparse LU decomposition as it is described so far deals with combinatorial support. One aspect which allows to raise efficiency and to speed up the numerical factorization will be discussed now. It is the recognition of an underlying block structure with dense submatrices, caused by the factorization and the fill. The block structure allows to collect parts of the matrix in dense blocks and to treat them commonly using higher levels of BLAS [10, 11].

As a consequence of the LU decomposition we will encounter parts of the triangular factors that are dense or become dense by the factorization. Fortunately, in the symmetric case this situation can be read off from the elimination tree. Suppose that A is structurally symmetric. By Lemma 2.17 the nonzero pattern of $L_{k+1:n,k}$ is given by $\mathrm{adj}\ (T[k])$. I. e., the neighbours in the elimination tree rooted at k describe the fill–in pattern of $L_{k+1:n,k}$. In order to have a sequence $k, k+1, \ldots, k+s-1$ of s subsequent columns in L that form a dense lower triangular matrix, we must have

$$\mathrm{adj}\ (T[k+i]) \cup \{k+i\} = \mathrm{adj}\ (T[k+i-1]), \quad \text{for all } i = 1, \ldots, s. \tag{2}$$

Definition 3.3 A sequence $k, k+1, \ldots, k+s-1$ satisfying (2) in the elimination tree is called a *supernode*.

Let $\mathcal{I} = \mathrm{adj}\ (T[k]) \cup \{k\}$. It follows that if $k, k+1, \ldots, k+s-1$ form a supernode, then the submatrix

$$L_{\mathcal{I},k:k+s-1} \equiv (l_{ij})_{i\in\mathcal{I},j=k,\ldots,k+s-1}$$

forms a dense lower triangular matrix.

Given a fixed elimination tree $T(A)$, by Remark 2.12 any topological ordering of $T(A)$ could be used for the elimination process without changing the fill of the factors. To obtain supernodes, whenever they occur, parent nodes should immediately follow their children in the order of elimination, i. e. the vertices should be renumbered in post–order.

Example 3.4 To illustrate the use of supernodes, Figure 10 illustrates the underlying dense block structure. Note that the numbering of the vertices in $T(A)$ in Figure 4 already represents a post–ordering.

Copyright line will be provided by the publisher

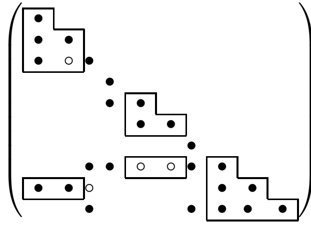

Fig. 10 Supernodes in L, symmetric case

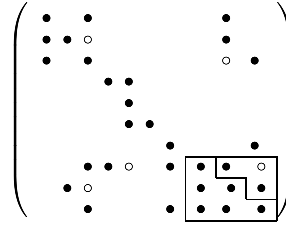

Fig. 11 Supernodes in $L + U$, general case

Supernodes allow several improvements. Certainly one can reduce the elimination tree to the block case. A supernode can be stored as one or two dense matrices. Beside the storage scheme as dense matrices, the nonzero row indices for these blocks need only be stored once. Next the use of dense submatrices allows the usage of BLAS routines [10, 11]. To see this, one can easily verify that the update process

$$L_{k:n,k} := L_{k:n,k} - L_{k:n,i} u_{ik}$$

that computes $L_{k:n,k}$ in Algorithm 1 can easily be extended to the block case. To do this suppose that columns $1, \dots, k+s-1$ are collected in p supernodes with column indices $\mathcal{K}_1, \dots, \mathcal{K}_p$. Apparently we have $\mathcal{K}_p = \{k, \dots, k+s-1\}$. Then updating $L_{k:n,\mathcal{K}_p}$ can be rewritten as

$$L_{k:n,\mathcal{K}_p} := L_{k:n,\mathcal{K}_p} - L_{k:n,\mathcal{K}_i} U_{\mathcal{K}_i,\mathcal{K}_p},$$

where the sum has to be taken over all i in the block version of the elimination tree. Depending on whether one would like to compute the diagonal block $L_{\mathcal{K}_p,\mathcal{K}_p}$ as full or as lower triangular matrix one is able to use BLAS–2 or even BLAS–3 subroutines. This allows to exploit machine–specific properties like cache and to accelerate the computation.

While the construction of supernodes is fairly easy in the symmetric case, its generalization to the general case is significantly harder. In contrast to the symmetric case we have to perform pivoting in each step of the Gaussian elimination. If we consider Algorithm 1 then we can certainly reorder the columns and maybe the rows in advance. But at each step k of the factorization, row k may be permuted with some row $m > k$. As stated by Theorem 2.22 the natural analogy consists of the column elimination tree $T_\cap(A)$. Like in the symmetric case a symmetric permutation $P^\top AP$ with respect to the postorder numbering of the column elimination tree $T_\cap(A)$ would reflect the analogous approach. Based on $T_\cap(A)$ we could decide which columns to group together into supernodes. However the prediction by $T_\cap(A)$ only gives a hull in which a dense block may occur. But this prediction includes all variations of row pivoting.

Example 3.5 In Figure 11, columns $7, 8, 9$ form a supernode if we only consider the columns of the lower triangular part. Note also that based on the prediction of the column elimination tree, column 5 and 6 would also be added to the supernode.

As sketched by Figure 11, in the unsymmetric case due to the usage of Algorithm 1 or a block variant of it, only the lower triangular part is assembled into complete supernodes while for the upper triangular part only the associated diagonal block is moved to the supernodal block of L. The remaining entries of U are kept as sparse columns.

Copyright line will be provided by the publisher

4 Symbolic symmetric reordering techniques

So far we have only discussed techniques that describe the process of Gaussian elimination. The elimination tree and other combinatorial approaches are used to efficiently compute the factorization. Now we will discuss how reordering the system before hand will reduce the fill–in and therefore speed up the factorization. At the same time less memory may be needed. We will start with a review of the classical minimum degree approach [40, 18] that tries to reduce the fill–in by selecting vertices in $G^+(A)$ first that locally produce fewer fill–in. Later on we will comment on some related methods. Globally speaking, avoiding fill-in might lead to a wider elimination tree of less depth. The second algorithm directly addresses this problem using a nested dissection strategy set up in a multilevel framework.

4.1 The minimum degree algorithm

We start with the basic idea of the minimum degree algorithm (MMD). At step k of the LU factorization, the reduced matrix is updated via

$$A_{k+1:n,k+1:n} \to A_{k+1:n,k+1:n} - L_{k+1:n,k}U_{k,k+1:n}.$$

Fill–in is at most introduced by the rank–1 update $L_{k+1:n,k}U_{k,k+1:n}$ (if we ignore cancellations). For any pair of edges (i,k) and (k,j) such that $i,j>k$ we obtain a fill–in edge (i,j). Starting with the initial graph $G(A)$ we obtain e sequence of graphs

$$G(A) = G_1(A) \to \cdots G_k(A) \to \cdots G_{n-1}(A) = G^+(A)$$

that finally lead to to the filled graph and, G_{k+1} is obtained from $G_k(A)$ via

$$G_{k+1}(A) = G_k(A) \cup \{(i,j) : i,j>k, (i,k) \text{ and } (k,j) \in G_k(A)\}.$$

In the case of structurally symmetric matrices the situation simplifies. Here we obtain $G_{k+1}(A)$ from $G_k(A)$ by replacing the set of all remaining neighbours $\mathrm{adj}_{\,G_k(A)}(k) \setminus \{1,\dots,k\}$ of k in $G_k(A)$ by its clique. I. e., any two neighbours $i,j>k$ of k in $G_k(A)$ are connected by an edge.

Definition 4.1 Given an undirected graph $G=(V,E)$ and $k\in V$ we call

$$d_k(k) = \#\mathrm{adj}_{\,G}(k) \setminus \{1,\dots,k\}$$

the *degree* of k.

To avoid too many new fill–in edges it is advantageous if k has a small degree. This is the basis of the minimum degree algorithm (MMD) that successively chooses at step k a vertex i_k such that i_k has minimum degree $d_k(i_k) = \#\mathrm{adj}_{\,G}(i_k) \setminus \{i_1,\dots,i_k\}$ in $G_k(A)$. This leads to sequence of vertices $(i_1,\dots,i_n)$. The graph that is obtained by this strategy is exactly the filled graph $G^+(P^\top AP)$, where P is the permutation matrix associated with $(i_1,\dots,i_n)$.

Example 4.2 We will use the symmetric matrix and its undirected graph from Figure 2. First we will illustrate the change of $G(A)$ when eliminating vertex $k=1$. Its neighbours are $i=2,3,9$. The associated subgraph is replaced by a clique resulting in two fill-in edges $\{2,3\}$ and $\{3,9\}$ (see Figure 12).

Copyright line will be provided by the publisher

Next we illustrate the minimum degree algorithm applied to $G(A)$. In Figure 13 this is done by relabelling the nodes in the circles. Initially $i_1 = 6$, since $d_1(6) = 1$. After that $i_2 = 5$, since again $d_2(5) = 1$. The same argument applies to $i_3 = 4$. After that, vertex 2 and 7 both have degree 2 and we choose $i_4 = 2$. Since the degree of 7 did not change, we may use $i_5 = 7$. Next $i_6 = 1$ has degree $d_6(1) = 2$ which is the smallest possible value. Note that in step $k = 1, 2, \ldots, 5$ no fill–in edges occurred. This changes at step $k = 6$, since the remaining neighbours of $i_6 = 1$ are 3 and 9. These nodes are not connected by an edge at this step. So for the first time we get a fill-in edge $\{3, 9\}$ (see Figure 13). It is easy to verify that the last 4 steps do not introduce further fill-in.
It is remarkable to see that the MMD ordering only produces 1 fill–in edge while the original ordering lead to 4 additional edges.

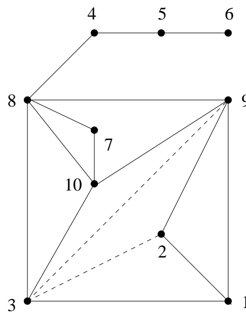

Fig. 12 Fill–in in the fi rst step of LU

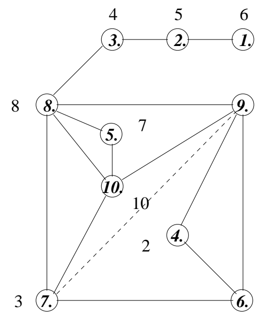

Fig. 13 Minimum degree ordering and fi ll–in

Here we have sketched the basic approach of the MMD algorithm. To raise efficiency additional techniques are added. One technique is called *mass elimination* which refers to collectively removing i_k and the set $\mathcal{I}_k$ of its neighbours j that do not lead to additional fill–in when i_k is selected. Since one can treat these nodes together the degree $d_k(i_k)$ could be replaced by the *external degree* $d_k(i_k) - \#\mathcal{I}_k$ as measure for selecting nodes. Further techniques like *incomplete degree update*, *element absorption* and *multiple elimination* as well as data structures based on cliques are used to improve the MMD algorithm. For an overview see [18].

One of the most costly parts in the MMD algorithm is update of the degrees. Instead of computing the exact external degree, in [2] an approximate external degree is computed that significantly saves time while producing comparable fill in the LU decomposition. The resulting algorithm is called approximate minimum degree (AMD).

The algorithms MMD as well as AMD only discuss the reduction of the fill–in for matrices with nearly symmetric pattern and when almost no pivoting is required to compute the LU factorization. In the general case this is an unrealistic scenario and different reordering strategies are required. According to Theorem 2.21 the column elimination tree provides at least a hull for the factors L and U. Moreover this even applies that partial pivoting with respect to the rows is used. A direct consequence is that reducing the fill in $G^+_\cap(A)$ can also reduce the fill in L and U independent on P. To do this one could simply compute a column permutation Q of A such that $G^+_\cap(AQ) = G^+(Q^\top A^\top AQ)$ has less fill–in. The most simple way to do that is to apply MMD algorithm of the AMD algorithm to $A^\top A$ or $A^\top + A$ and use the associated permutation matrix Q to reorder the columns of A in advance. Since in general $A^\top A$ could

Copyright line will be provided by the publisher

be significantly more dense than A, a different approach is chosen in the column approximate minimum degree approach (COLAMD). Here one starts with the pattern of A and each step k a symbolic analysis is used to compute the pattern of the reduced matrix. For details we refer to [8].

4.2 Multilevel nested dissection

Recursive multilevel nested dissection methods for direct decomposition methods were firstly introduced in the context of multiprocessing. If parallel direct methods are used to solve a sparse systems of equations, then a graph partitioning algorithm can be used to compute a fill reducing ordering that lead to a high degree of concurrency in the factorization phase. The (approximate) minimum degree (MMD) or reverse Cuthill–McKee (RCM) methods used almost exclusively in serial methods ten years ago are not suitable for parallel direct methods, as they provide very little concurrency in the parallel factorization phase. The research on graph-partitioning methods in the mid-nineteens has resulted in high-quality software packages, e.g. METIS [32], and when they are used to compute fill reorderings for sparse matrices, they produce orderings that provide more concurrency and can have substantially smaller fill than the widely uses degree-based methods such as RCM or MMD.

Since multilevel nested dissection is so closely related to graph-partitioning we will review some combinatorial aspects of it.

Definition 4.3 A *k-way graph partitioning* is defined as follows: Given a graph $G(A) = (V, E)$, with $|V| = n$, partition V into k subsets , $V_1, V_2, \ldots, V_k$ such that $V_i \cap V_j = \emptyset$ for $i \neq j$, $|V_i| = n/k$ and $\cup_i V_i = V$.

Definition 4.4 A *vertex separator* of a graph $G(A) = (V, E)$, with $|V| = n$, is a set of vertices V_s that partitions $V \setminus V_s$ into k subsets $V_1, V_2, \ldots, V_k$, such that $V_i \cap V_j = \emptyset$, $V_i \cap V_s = \emptyset$, $|V_i| \approx n/k$ for $i \neq j$. Moreover, it is assumed that $\cup_i V_i \cup V_s = V$ and that the numbers of edges $\cup_{i=1}^{k} |\{e_{is} \in V_i, s \in V_s\}|$ is minimized.

Definition 4.5 An *edge separator* $E_s \in E$ of a graph $G = (V, E)$ is a set of edges whose removal partitions V into k subsets , $V_1, V_2, \ldots, V_k$, with $V_i \cap V_j = \emptyset$, $|V_i| = n/k$ for $i \neq j$. The edges e_{ij} that connects the vertex set V_i with V_j are defined as *edge separators*.

Nested dissection recursively splits a graph $G(A) = (V, E)$ into almost equal halves by selection a vertex separator until the desired numbers of partitionings are obtained. One way of obtaining a vertex separator is to first obtain a 2-way partitioning of the graph, a so called *graph bisection*, and then to compute a vertex separator from the edge separator. The vertices of the graph are numbered such that at each level of recursion, the separator vertices are numbered after the vertices in the partitions. In general, small vertex separators results in low fill-in.

Example 4.6 To illustrate vertex separators, Figure 14 shows a 2-way partitioning with $V_1 = \{1, 2, 3\}$ and $V_2 = \{4, 5, 6\}$ and a vertex separator $V_s = \{7, 8, 9, 10\}$. The vertices in the partition are ordered before the vertices in the separator. The '$\circ$' elements indicate that no fill will occur in these zero blocks of the system.

Since recursive graph bisection is typically computational expensive, combinatorial *multilevel graph bisections* has been used to accelerate the process. The basic structure is simple.

Copyright line will be provided by the publisher

Fig. 14 A 2-way partition with vertex separators. Fill between the two partitions can only occur in the separator vertices.

During a *coarsening phase*, the size of the graph is successively decreased down to a few hundreds vertices, a *bisection* of the much smaller graph is computed; and during a *uncoarsening phase*, the bisection is successively refined as it is projected towards the original graph.

Coarsening Phase. The original graph $G = (V, E)$ is transformed in the coarsening phase into a sequence of smaller graphs $G_1, G_2, \ldots, G_m$ such that $|V_0| > |V_1| > |V_2| > ... > |V_m|$. Given the graph $G_i = (V_i, E_i)$, the coarser graph G_{i+1} can be obtained by collapsing adjacent vertices. Thus, the edge between two vertices is collapsed and a multinode consisting of these two vertices is created. This edge collapsing idea can be formally defined in terms of matchings.

Definition 4.7 A *matching* of a given graph $G(A) = (V, E)$ is a subset of edges e_{ij} such that no two of which share the same vertex. If M is a matching of G, then each edge $e_{ij} \in M$ corresponding to vertices v_i and v_j and there exists no other edge $e \in M$ that has the same vertex endpoints v_i or v_j. A matching M is called *maximal*, if no other edge from E can be added.

Thus, the next level coarser graph G_{i+1} is constructed from G_i by finding a maximal matching of G_i and collapsing the vertices being matched into multinodes. Since the goal of collapsing vertices using matchings is to decrease the size of the graph G_i , the matching should be as large as possible. The main difference between the various ordering packages is the construction of the maximal matching. One of the most popular and efficient methods is heavy edge matching [32].

Partitioning Phase. A 2-way partition P_m of the graph $G_m = (V_m, E_m)$ is computed that partitions V_m into two parts, each containing half the vertices of G_m. The partition of G_m can be obtained by using various algorithms such as spectral bisection [16] or combinatorial methods based on Kernighan-Lin variants [33]. It is shown in [32] that combinatorial methods find in general smaller edge-cut separators compared with spectral bisection for partitioning the coarse graph. However, since the size of the coarsest graph G_m is small (i.e. $|V_m| < 100$), this step takes a small amount of time.

Uncoarsening Phase. The partition P_m of G_m is projected back to G_0 by going through intermediate partitions $P_{m-1}, P_{m-2}, ..., P_1, P_0$. Each vertex v of G_{i+1} contains a distinct

Copyright line will be provided by the publisher

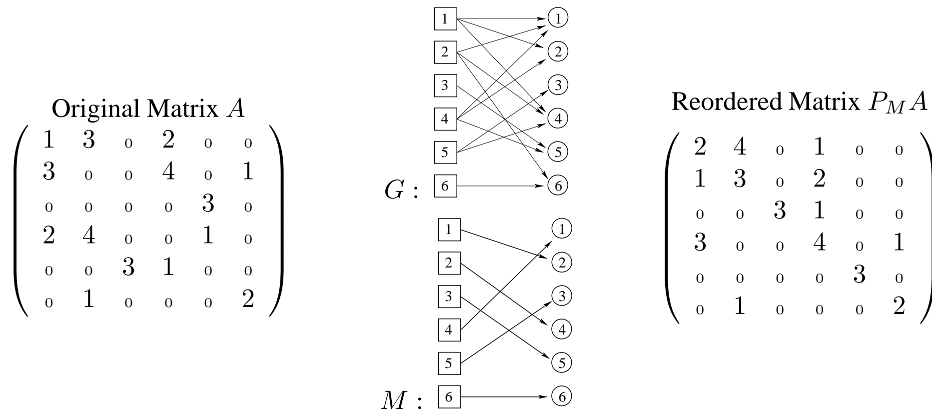

Fig. 15 Illustration of a perfect matching. Left side: original matrix A, Middle: bipartite representation $G = (V_r, V_c, E)$ of the matrix A and perfect matching M. Right side: matrix after row permutation $(P_M A)$ with permutation matrix P_M.

subset of vertices V_i^v of G_i. Obtaining P_i from P_{i+1} is done by simply assigning the set of vertices V_i^v collapsed to $v \in G_{i+1}$ to the appropriate partition in G_i. Although P_{i+1} is a local minimum partition of G_{i+1}, the projected partition P_i may not be a local minimum with respect to G_i. Since G_i is finer, it has more degrees of freedom that can be used to improve P_i, and decrease the edge-cut of the partition. Hence, it may still be possible to improve the projected partition of G_i by local refinement heuristics. The refinement is usually done by using one of the variants of the Kernighan-Lin partition algorithm [33].

5 Maximum weight matching

Numerical stability in the decomposition is typically maintained through partial pivoting, which can have a significant impact on the factorization speed. Row interchanges due to partial pivoting can also unpredictably affect the nonzero structure of the factor, thus making it impossible to statically allocate data structures. Using nonsymmetric row or the column permutations to ensure a *non-zero diagonal* or to maximize the product of the absolute diagonal values are among the techniques often used as preprocessing steps for LU factorizations in order to reduce the number of dynamic pivoting steps. The original idea on which these nonsymmetric permutations are based is to find a *maximum weighted matching* of a *bipartite graphs*. Finding a maximum weighted matching is a well known assignment problem in operation research and combinatorial analysis.

One fundamental concept in the finding of maximal matchings is a bipartite graph $G = (V_r, V_c, E)$ of a matrix A. The two vertex sets, V_r and V_c, of the bipartite graph correspond to the rows and columns of the matrix, respectively. If there is a nonzero value in location a_{ij}, then vertex $v_i \in V_r$ from the row set is connected by an edge $e_{i,j} \in E$ to the column set $v_j \in V_c$. If, for an $n \times n$ matrix a *matching* M with maximum cardinality n is found, then we have established that the matrix is structurally nonsingular and we can use a nonsymmetric row or column permutation P_M to place a nonzero entry on each diagonal location.

Definition 5.1 A *perfect matching* is a maximal matching with cardinality n.

Copyright line will be provided by the publisher

Theorem 5.2 *When A is structurally nonsingular, there always exists a perfect matching for $G = (V_r, V_c, E)$. The perfect matching M defines an $n \times n$ permutation matrix with*

$$P_M = (p_{ij}) = \begin{cases} p_{ij} = 1 & e_{ij} \in M \\ p_{ij} = 0 & e_{ij} \notin M \end{cases}$$

Example 5.3 In Figure 15, the set of edges $M = \{(1,2),(2,4),(3,5),(4,1),(5,3),(6,6)\}$ represents a perfect maximum matching of the bipartite graph G.

The most efficient combinatorial methods for finding maximum matchings in bipartite graphs make use of an *augmenting path*. We will introduce some graph terminology for the construction of perfect matchings. If an edge e_{uv} joins a vertex u and v, we denote it as uv. If a path consists of edges $u_1u_2, u_2u_3, \ldots, u_{k-1}u_k$, where u_i all distinct, we denote it as $u_1u_2u_3 \ldots u_{k-1}u_k$. A vertex is called *free* if it is not incident to any other edge in a matching M. An *alternating path* relative to a matching M is a path $P = u_1u_2, u_2u_3, \ldots, u_s$ where its edges are alternatively in $E \setminus M$ and M. An *augmenting path* relative to a matching M is an alternating path of odd length and both of it vertex endpoints are free. If a graph $G = (V_r, V_c, E)$ is bipartite, one vertex endpoint of any other augmenting path must be in V_r and V_r. The symmetric difference, $A \oplus B$ of two edge sets is defined to be $(A \setminus B) \cup (B \setminus A)$.

These definitions and the following theorem [7] gives an constructive algorithm for finding perfect matchings on bipartite graphs.

Theorem 5.4 *If M is not a maximum matching of a bipartite graph $G = (V_r, V_c, E)$, then there exists an augmenting path P relative to M and $P \oplus M$ is a matching with size $|M| + 1$.*

According to this theorem, a combinatorial method of finding perfect matching in a bipartite graph is to seek augmenting paths. The perfect matching as discussed so far only takes the nonzero structure of the matrix into account.

There are other approaches which maximize the diagonal values in some sense and compute a *maximum weighted matching*. One possibility is e.g. to look for a matrix P_r such that the product of the diagonal values of P_rA is maximal. In other words, a permutation σ has to be found, which maximizes:

$$\prod_{i=1}^{n} |a_{\sigma(i)i}|. \tag{3}$$

This maximization problem is solved indirectly. We first reformulate it by defining a matrix $C = (c_{ij})$ with

$$c_{ij} = \begin{cases} \log a_i - \log |a_{ij}| & a_{ij} \neq 0 \\ \infty & \text{otherwise,} \end{cases}$$

where $a_i = \max_j |a_{ij}|$ is the maximum element in row i of matrix A. A permutation σ which minimizes the sum

$$\sum_{i=1}^{n} c_{\sigma(i)i}$$

also maximizes the product (3). The minimization problem is known as linear-sum assignment problem or bipartite weighted matching problem in combinatorial optimization. The problem

Copyright line will be provided by the publisher

is solved by a sparse variant of the Kuhn-Munkres algorithm. The complexity is $O(n^3)$ for full $n \times n$ matrices and $O(n\tau \log n)$ for sparse matrices with τ entries. For matrices, whose associated graph fulfill special requirements, this bound can be reduced further to $O(n^\alpha(\tau + n \log n)$ with $\alpha < 1$. All graphs arising from finite-difference or finite element discretizations meet the conditions [30]. As before, we finally get a perfect matching which in turn defines a nonsymmetric permutation.

In the solution of the assignment problem, two vectors $u = (u_i)$ and $v = (v_i)$ are generated, which can be used to scale the matrix. These vectors have the property that they fulfill the following equations:

$$u_i + v_j = c_{ij} \qquad (i,j) \in \mathcal{M}, \tag{4}$$

$$u_i + v_j \le c_{ij} \qquad \text{otherwise.} \tag{5}$$

Two diagonal matrices D_r and D_c are defined through

$$D_r = \text{diag}(d_1^r, d_2^r, \dots, d_n^r), \qquad d_i^r = \exp(u_i), \tag{6}$$

$$D_c = \text{diag}(d_1^c, d_2^c, \dots, d_n^c), \qquad d_j^c = \exp(v_j)/a_j. \tag{7}$$

With the equations (4) and (5), it can be shown that the scaled and permuted matrix $A_1 = P_r D_r A D_c$ is an I-matrix, for which holds:

$$|a_{ii}^1| = 1, \tag{8}$$

$$|a_{ij}^1| \le 1. \tag{9}$$

The permuted system can have better numerical properties, see e.g. [6, 13, 44]. Olschowka and Neumaier [38] introduced these scalings and permutation for reducing pivoting in Gaussian elimination of full matrices. The first implementation for sparse matrix problems was introduced by Duff and Koster [13]. Recent developments indicate that these nonsymmetric techniques can be transferred to the symmetric case [14, 41].

6 Numerical experiments

During the past twenty years, the algorithmic improvements discussed in the previous sections have significantly reduced the time required for the direct solution of sparse systems of linear equations. This section compares the performance of these algorithmic methods for solving general nonsymmetric and symmetric indefinite sparse systems. In particular, we demonstrate that consistently high level of performance is achieved by PARDISO [42, 43], one of the most recent of sparse direct solvers. We compare the influence of various combinatorial methods and discusses their impact on the solver performance.

In particular, we will discuss two application areas, namely *semiconductor device simulation* and *interior-point nonlinear programming*, in which the solution of large sparse linear systems represents a highly critical component.

6.1 Semiconductor Device Simulation

In 1950, Van Roosbroeck [39] introduced the drift-diffusion equations which are the commonly used model in semiconductor device simulation. The drift-diffusion equations are a

Copyright line will be provided by the publisher

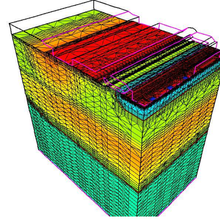

Fig. 16 3D discretization of a MOS transistor with nearly 52'000 grid points. The resulting nonsymmetric sparse linear systems of size 152'000 unknowns is shown in Figure 17.

system of three coupled, nonlinear partial differential equations (PDEs), which describe the relation between the electrostatic potential and the densities of the charge carriers in a semiconductor device. The equations of the drift-diffusion model are:

$$-\nabla \cdot (\epsilon \nabla \psi) = q\ (p - n + C)\,, \tag{10}$$

$$q\frac{\partial n}{\partial t} - \nabla \cdot \mathbf{j}_n = -q\,R, \tag{11}$$

$$q\frac{\partial p}{\partial t} + \nabla \cdot \mathbf{j}_p = -q\,R, \tag{12}$$

where ψ is the electrostatic potential, n and p are the electron and hole densities. These are the unknown variables. The other quantities are given: ϵ is the dielectrical permittivity, C the doping concentration, R the total recombination rate and q the elementary charge. The carrier current densities $\mathbf{j}_c$ in equations (11) - (12) are substituted with the equations

$$\mathbf{j}_n = q\,(D_n \nabla n - \mu_n n \nabla \psi)\,, \tag{13}$$

$$\mathbf{j}_p = -q\,(D_p \nabla p + \mu_p p \nabla \psi)\,. \tag{14}$$

Here, μ_c are the carrier mobilities and D_c are the carrier diffusivities. Typical challenges in solving the drift-diffusion equations include a large range of n and p, the steep gradients of the solution variables, and the necessity to ensure that $n, p > 0$ in the numerical computations.

Different discretizations are used to solve the drift-diffusion equations numerically and the Scharfetter-Gummel box method is typically used in modern semiconductor device simulators [31]. Regardless of the discretization a set of nonlinear equations must be solved. Often, the only possible way is to solve the nonlinear equations with the Newton method. The resulting linear systems $Ax = b$, where A is a nonsymmetric sparse $3n \times 3n$ matrix, highly ill-conditioned and significantly demanding.

Figure 16 shows a three-dimensional discretization of a semiconductor MOS transisor with 52'000 grid points and Figure 17 shows the distribution of the magnitude of elements in the resulting matrix from this semiconductor device simulation. The first picture in Figure 17 shows the 3×3 sparse block structure of a matrix resulting from the Newton linearization and discretization of the drift-diffusion equations. The diagonal blocks of A are related to the electrostatic potential ψ and the charge carrier densities n, p in a semiconductor device, respectively. The values in the off-diagonal blocks of A represent the coupling of the variables ψ, n and p. The first matrix shows all the elements, the second one only shows the largest

Copyright line will be provided by the publisher

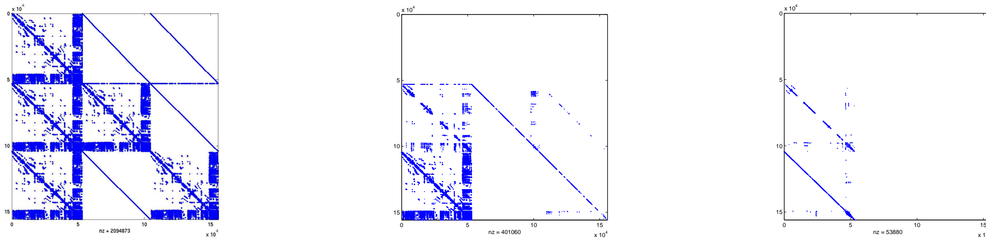

Fig. 17 Pictorial depiction of the magnitude of the elements in a matrix from semiconductor device simulation. The first picture shows the original coefficient matrix. The second picture shows only 50% of the largest absolute values, the last picture the largest 10%.

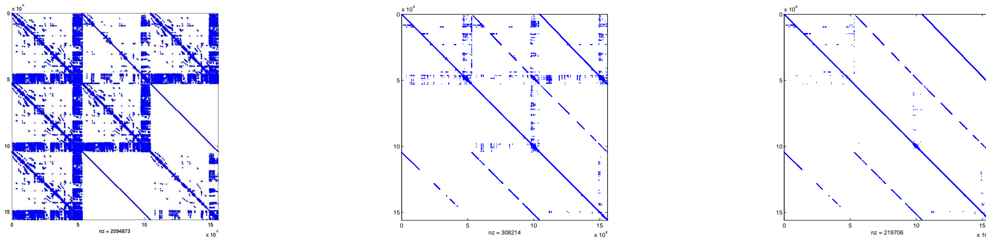

Fig. 18 Influence of nonsymmetric permutations on the matrix from semiconductor device simulation shown in Figure 17. The first picture shows the original coefficient matrix that has been permuted rowwise using a nonsymmetric weighted matching algorithm. The second picture only shows 50% of the largest absolute values, the last picture the largest 10%. Note that a majority of the largest values have migrated to the block diagonal.

50% elements in absolute value, and the last one shows the largest 10% elements. Due to the numerical properties of the carrier densities, the largest 10% elements, which are in the off-diagonal blocks, can be several orders of magnitude larger than the elements in the diagonal blocks. These extremely large entries in the off-diagonal blocks are one main reason for the difficulties in applying preconditioned iterative methods in semiconductor device simulation.

There are two main approaches to solving the nonsymmetric sparse linear systems from Newton's method using sparse direct solver technology. The first approach uses partial pivoting during elimination which dynamically applies expensive pivot searches during the Gaussian process. The second approach, which is now commonly used in nearly all state–of–the art direct solvers, uses nonsymmetric graph weighted matchings from Section 5 as a kind of static pivoting before applying the elimination process. This category of reorderings consist of permuting only the columns or the rows of the matrix in order to address the issue of avoiding poor pivots in Gaussian elimination.

Figure 18 shows the effect of applying the weighted nonsymmetric matching on the distribution of the magnitude of elements in the transistor matrix from semiconductor device simulation shown in Figure 17. The matrices in Figure 18 corresponds to those in Figure 17,

Copyright line will be provided by the publisher

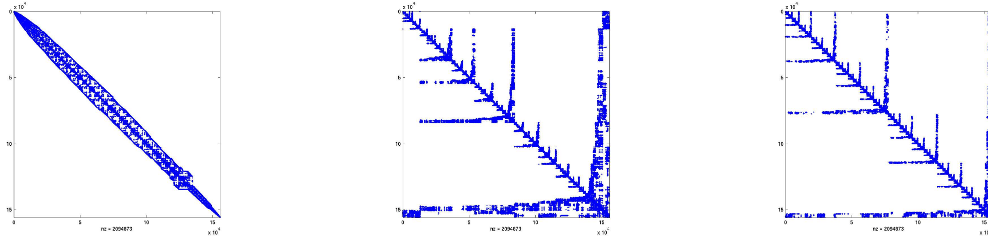

Fig. 19 The influence of various orderings algorithms applied to a semiconductor device simulation matrix from Figure 18. The pictures show the structure after a symmetric permutation with bandwidth reduction using reverse Cuthill-McKee (left), or using minimum-degree (middle) and multi-level nested dissection with METIS (right).

but scaled and permuted with nonsymmetric weighted matchings. Compared with the original matrix, the largest elements of the matrix now lie on the diagonal which has a primary advantage that less pivoting search might be necessary during the elimination process.

Fill-in preserving methods are equally important and the goal of symmetric reorderings is to reduce the fill-in during Gaussian elimination. Several approaches such as graph-degree based methods (RCM, MMD, AMD) or recursive nested dissections methods (METIS) can be used. Figure 19 shows the distribution of the non-zeros using a symmetric permutation of both columns and rows of the original device simulation matrix from Figure 18 based on the structure of $A + A^T$. The reverse Cuthill-McKee (RCM) is among the most common techniques used to enhance the effectiveness of sparse Gaussian elimination. This reordering is designed to reduce the envelope of the matrix as shown in the first picture of Figure 19 and has been widely used in finite-element structural analysis 10 years ago. One other reordering geared specifically toward reducing fill-in is the minimum-degree method MMD [18] described in Section 4. The second picture shows a structural distribution of the system after applying the MMD permutation. Finally, the last picture represents the semiconductor device simulation system that is symmetrically permuted with recursive multi-level nested dissection from the METIS package [32]. Is is clearly visible that the overall quality of the fill-in that will occur during Gaussian elimination is best in terms of fill for the METIS reordering. Furthermore, the elimination tree produced by METIS exhibits more concurrency so that a process to subtree mapping lead to good load balancing while additionally minimizing interprocess communication during a parallel factorization. The fill-in produced by METIS is one order of magnitude better compared to RCM as shown in Table 1.

We have used a recent eight processor Itanium Intel Server with a processor frequency of 2.6 GHz for the numerical experiments in this paper. Furthermore, we used the PARDISO solver package[1], a suite of publicly available parallel sparse linear solvers. The code was compiled by *g77* and *gcc* with the -O3 optimization option and linked with the Automatically Tuned Linear Algebra Software ATLAS library[2] for the basic linear algebra subprograms optimized for Intel architectures. The basic components of the solver were changed in such a way that the specific changes emulate the typical behaviour of a sparse direct solver using the

[1] `http://www.computational.unibas.ch/cs/scicomp/software/pardiso`
[2] `https://sourceforge.net/projects/math-atlas`

Copyright line will be provided by the publisher

Factorization Method	Col — Col	Sup — Sup	Sup — Sup	Sup — Sup
BLAS	Level-1	Level-3	Level-3	Level-3
Ordering	RCM	RCM	METIS	METIS + MATCH
# Processors	1 CPU	1 CPU	1 CPU	8 CPUs
Year	$\approx$ 1990	1993	1998	2003
Memory in GB for LU	11.5	11.5	1.2	1.3
Time in sec. for LU	17527	1'527	107	17

Table 1 Influence for various algorithmic choices on the factor memory requirement and factorization time for the semiconductor device simulation matrix from Figure 17. The Table show the year of invention as well as the ordering and factorization method.

algorithmic components displayed in Table 1. A common practice fifteen years ago was to apply reverse Cuthill McKee (RCM) and a sparse factorization method based e.g. on a Level-1 BLAS left-looking column-by-column update. The second column of Table 1 shows the performance of such as direct solver on a modern Intel Itanium processor. The method compensates 11.5 GB main memory for the LU factors resulting in a factorization time of 17'527 seconds on one processor. Replacing the column-by-column factorization with a Level-3 Blas supernode-by-supernode algorithm [5, 37] increases the performance of the solver by over one order of magnitude due to efficient memory hierarchy utilization of the method as shown in the third column of Table 1. The next column displays the influence of the recursive multilevel nested dissection from METIS [32] to the three-dimensional semiconductor device simulation matrix. We see that the METIS ordering is of high quality and produces a factor with only 10% of nonzeros compared to RCM. As a result, the new factorization method needs only 1.5 GB and compute the factor in about 107 seconds. Another level of accuracy and parallel scalability has been added by combining the METIS reordering with a nonsymmetric weighted matching ordering on shared–memory multiprocessing architectures [42]. The last column shows the default option of PARDISO and it indicates that the algorithmic improvements of the past few years have reduced the time required to factor general sparse matrices by almost three order of magnitude.

6.2 Interior-Point Optimizations

Our second example will show the progress that has been made in the direct solution of sparse symmetric indefinite systems. We present a symmetric indefinite augmented system from a interior point filter line-search algorithm for large scale nonlinear programming. The growing interest in efficient optimization methods has led to interior point optimization or barrier methods in which the solution of a highly indefinite system plays one central rule. We consider a primal dual barrier method to solve nonlinear optimization problems of the form

$$\min_{x\in\mathbb{R}^n} \quad f(x) \tag{15}$$

$$s.t. \quad c(x) = 0 \tag{16}$$

$$x \geq 0 \tag{17}$$

where the variable $x \in \mathbb{R}^n$, and the objective function $f: \mathbb{R}^n \to \mathbb{R}$ and the equality constraints $c: \mathbb{R}^n \to \mathbb{R}^m$ with $m \leq n$, are assumed to be twice continuously differentiable. In barrier

Copyright line will be provided by the publisher

Factorization Method	Col — Col	Sup — Sup	Sup — Sup	Sup — Sup
BLAS	Level-1	Level-3	Level-3	Level-3
Ordering	RCM	RCM	METIS	METIS + MATCH
Pivoting	BK	BK	BK	SBK
# Processors	1 CPU	1 CPU	1 CPU	1 CPU
Year	≈ 1990	1993	1998	2004
Memory in B for LU	1.52	1.52	0.22	0.20
Time for LU	3496.1	315.	21.3	3.41

Table 2 Influence for various algorithmic choices on the factor memory requirement and factorization time for a interior point optimization matrix with 74'000 unknowns. The table shows the year of invention as well as the ordering and factorization method.

methods [46], the algorithms typically compute in interior point algorithms [46], the following sparse symmetric augmented systems has to be solved for the search directions

$$\begin{bmatrix} W_k & \nabla c(x_k) \\ \nabla c(x_k)^T & 0 \end{bmatrix} \begin{pmatrix} \Delta x_k \\ \Delta \lambda_k \end{pmatrix} = - \begin{pmatrix} \nabla \varphi_\mu(x_k) + \nabla c(x_k)\lambda_k \\ c(x_k) \end{pmatrix} \tag{18}$$

where W_k denotes the Hessian $\nabla^2_{xx}\mathcal{L}_\mu(x_k, \lambda_k)$ of the Lagrangian function

$$\mathcal{L}_\mu(x, \lambda) = \varphi_\mu(x) + c(x)^T \lambda.$$

In Table 2 we compare the factorization times using the same notation as in Table 1. The abbreviation BK represents a factorization method based on (1×1) or (2×2) Bunch and Kaufman pivoting, which is necessary for solving symmetric indefinite systems. SKB in the last column represents a method that uses symmetric weighted graph matchings combined with supernode Bunch and Kaufman pivoting [41]. The progress that has been made due to new algorithmic improvements in the area of symmetric indefinite systems is again visible.

6.3 On recent numerical evaluations of sparse direct solvers

In recent years a number of solvers for the direct solution of large sparse linear systems of equations have been developed. This includes sequential solvers that are primarily designed for only nonsymmetric systems as well as solvers that offer a family of different decompositions methods on modern architectures ranging from shared–memory to distributed–memory multiprocessing computers. The available choice can make it difficult to users to know which solver is the most appropriate for their applications. Since a numerical evaluation of all these different solvers is by far beyond the scope of this paper, we therefore refer to recent investigations. The interesting reader can find some evaluation results for nonsymmetric linear systems in [29, 42]. In addition, [26] presents direct solver performance evaluations for symmetric positive and indefinite linear systems.

Copyright line will be provided by the publisher

The solver packages PARDISO[3] [42, 41], TAUCS[4] [36], and WSMP[5] [28, 29] represent freely available, reliable and efficient high-performance implementations of sparse direct solver technology ranging from the desktop to high-end multiprocessing architectures. Furthermore, [26] lists a large number of other options that might be considered.

7 Conclusions

In this paper, we reviewed important combinatorial aspects in sparse LU decompositions and demonstrated that these combinatorial methods alone have significantly improved the state–of–the–art of the direct solution of sparse linear systems of equations. Our numerical examples reveal that recent sparse direct solvers can be up to three order of magnitude faster than the best factorization solver fifteen years ago. Furthermore, these solvers offer significant scalability on multiprocessing architectures that can be utilized to solve large–scale problems even faster [3, 27, 34, 42]. Therefore, it can be concluded that there has been a significant progress in sparse direct solver algorithms and software recently. Combinatorial methods in all phases of the sparse direct solution process have contributed to these performance gains. These include the use of (non-)symmetric maximum weighted matching to permute large magnitude elements close to the matrix diagonal, symmetric pruning strategies, nested-dissection based fill-reducing permutation applied symmetrically to rows and columns, and Level-3 BLAS static pivoting techniques. Combined with a significant progress in the performance to cost ratio of computing hardware during this period, current sparse solver technology makes it it now possible to solve those large-scale problems quickly and easily that might have been considered by far too large for direct solver until recently.

References

[1] A. Aho, J. Hopcroft, and J. Ullman. *Data structures and algorithms*. Addison-Wesley, 1983.

[2] P. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Appl.*, 17(4):886–905, 1996.

[3] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Comput. Methods in Appl. Mech. Engrg.*, 184:501–520, 2000.

[4] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide, Second Edition*. SIAM Publications, 1995.

[5] C. Ashcraft, R. Grimes, J. Lewis, B. Peyton, and H. Simon. Progress in sparse matrix methods for large sparse linear systems on vector supercomputers. *Internat. J. Supercomputing Applic.*, 1:10–30, 1987.

[6] M. Benzi, J. Haws, and M. Tuma. Preconditioning highly indefi nite and nonsymmetric matrices. *SIAM J. Sci. Comput.*, 22(4):1333–1353, 2000.

[7] C. Berge. Two theorems in graph theory. In *Proceedings of National Academy of Science*, pages 842–844, USA, 1957.

[3] The solver has been developed at the Computer Science Department of the University of Basel. It is available at `http://www.computational.unibas.ch/cs/scicomp/software/pardiso` and it is also included into Intel's Math Kernel Library Version 7.0.

[4] The solver has been developed at the Computer Science Department of Tel-Aviv University and offers out-of core solutions capabilities. It is available at `http://www.tau.ac.il/~stoledo/taucs`

[5] The solver has been developed at the Mathematical Department of the IBM Thomas Watson Research Center and is available at `http://www-users.cs.umn.edu/~agupta/wsmp.html`.

Copyright line will be provided by the publisher

[8] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng. A column approximate minimum degree ordering algorithm. Technical Report TR-00-015, Univ. of Florida, 2000.
[9] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Anal. Appl.*, 20(3):720–755, 1999.
[10] D. Dodson and J. G. Lewis. Issues relating to extension of the basic linear algebra subprograms. *ACM SIGNUM Newslett.*, 20:2–18, 1985.
[11] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. Issues relating to extension of the basic linear algebra subprograms. *ACM SIGNUM Newslett.*, 20:2–18, 1985.
[12] I. S. Duff, A. Erisman, and J. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, 1986.
[13] I. S. Duff and J. Koster. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM J. Matrix Anal. Appl.*, 20:889–901, 1999.
[14] I. S. Duff and S. Pralet. Strategies for scaling and pivoting for sparse symmetric indefi nite problems. Technical Report TR/PA/04/59, CERFACS, Toulouse, France, 2004.
[15] S. Eisenstat and J. W. H. Liu. Exploiting structural symmetry in a sparse partial pivoting code. *SIAM J. Sci. Comput.*, 14:253–257, 1993.
[16] M. Fiedler. A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory. *Czecheslovak Mathematical Journal*, 25(100):619–633, 1975.
[17] A. George and M. T. Heath. Solution of sparse linear least squares problems using Givens rotations. *Linear Algebra Appl.*, 34:69–83, 1980.
[18] A. George and J. W. H. Liu. The evolution of the minimum degree ordering algorithm. *SIAM Review*, 31:1–19, 1989.
[19] J. A. George and J. W. Liu. *Computer Solution of Large Sparse Positive Defi nite Systems*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1981.
[20] J. A. George and E. Ng. An implementation of Gaussian elimination with partial pivoting for sparse systems. *SIAM J. Sci. Statist. Comput.*, 6(2):390–409, 1985.
[21] J. A. George and E. Ng. Symbolic factorization for sparse Gaussian elimination with partial pivoting. *SIAM J. Sci. Statist. Comput.*, 8(6):877–898, 1987.
[22] J. R. Gilbert. Predicting structure in sparse matrix computations. *SIAM J. Matrix Anal. Appl.*, 15(1):162–79, 1994.
[23] J. R. Gilbert and E. Ng. Predicting structure in nonsymmetric sparse matrix factorizations. In J. A. George, J. R. Gilbert, and J. W. H. Liu, editors, *Graph Theory and Sparse Matrix Computation*. Springer-Verlag, 1993.
[24] J. R. Gilbert and T. Peierls. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM J. Sci. Statist. Comput.*, 8:862–874, 1988.
[25] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, third edition, 1996.
[26] N. Gould, Y. Hu, and J. Scott. A numerical evaluation of sparse direct solvers for the solution of large sparse, symmetric linear systems of equations. Technical report, Rutherford Appleton Laboratory, 2004. to appear.
[27] A. Gupta. A shared- and distributed-memory parallel sparse direct solver. *J. of Future Generation Computer Systems*. Submited.
[28] A. Gupta. WSMP: Watson sparse matrix package (Part-II: direct solution of general sparse systems. Technical Report RC 21888 (98472), IBM T. J. Watson Research Center, Yorktown Heights, NY, November 20, 2000.
[29] A. Gupta. Recent advances in direct methods for solving unsymmetric sparse systems of linear equations. *ACM Trans. Math. Softw.*, 28(3):301–324, September 2002.
[30] A. Gupta and L. Ying. On algorithms for fi nding maximum matchings in bipartite graphs. Technical Report RC 21576 (97320), IBM T. J. Watson Research Center, Yorktown Heights, NY, October 25, 1999.
[31] Integrated Systems Engineering AG. *DESSIS*$_{-\mathrm{ISE}}$ *Reference Manual*. ISE Integrated Systems Engineering AG, 2004.

Copyright line will be provided by the publisher

[32] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[33] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 29(2):291–307, 1970.

[34] X. S. Li and J. W. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Math. Softw.* Accepted, in press.

[35] J. W. Liu. The role of elimination trees in sparse factorization. *SIAM J. Matrix Anal. Appl.*, 11(1):134–172, 1990.

[36] O. Meshar and S. Toledo. An out-of-core sparse symmetric indefinite factorization method. *ACM Trans. Math. Softw.* Submitted. Under revision.

[37] E. Ng and B. Peyton. Block sparse Cholesky algorithms on advanced uniprocessor computers. *SIAM Journal on Scientific Computing*, 14:1034–1056, 1993.

[38] M. Olschowka and A. Neumaier. A new pivoting strategy for gaussian elimination. *Linear Algebra and its Applications*, 240:131–151, 1996.

[39] W. V. Roosbroeck. Theory of flow of electrons and holes in germanium and other semiconductors. *Bell Syst. Tech. J.*, 29:560–607, 1950.

[40] D. J. Rose. A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. In R. C. Read, editor, *Graph Theory and Computing*. Academic Press, 1972.

[41] O. Schenk and K. Gärtner. On fast factorization pivoting methods for sparse symmetric indefinite systems. Technical Report CS-2004-005, Department of Computer Science, University of Basel, 2004. Submitted.

[42] O. Schenk and K. Gärtner. Solving unsymmetric sparse systems of linear equations with pardiso. *J. of Future Generation Computer Systems*, 20(3):475–487, 2004.

[43] O. Schenk, K. Gärtner, and W. Fichtner. Efficient sparse LU factorization with left-right looking strategy on shared memory multiprocessors. *BIT*, 40(1):158–176, 2000.

[44] O. Schenk, S. Röllin, and A. Gupta. The effects of unsymmetric matrix permutations and scalings in semiconductor device and circuit simulation. *IEEE Transactions On Computer-Aided Design Of Integrated Circuits And Systems*, 23(3), 2004.

[45] R. E. Tarjan. Data structures and network algorithms. In *CBMS–NSF Regional Conference Series in Applied Mathematics*, volume 44, 1983.

[46] A. Wächter and L. T. Biegler. On the implementation of a primal-dual interior point filter line search algorithm for large-scale nonlinear programming. Technical report, IBM T. J. Watson Research Center, Yorktown, NY, March 2004.

Copyright line will be provided by the publisher