SOFSEM '94 – Milovy, The Czech Republic, 27. 11. – 9. 12. 1994

Current Trends in Numerical Linear Algebra: From Theory to Practice

Z. Strakoš, M. Tůma

Institute of Computer Science, Academy of Sciences of the Czech Republic
Pod vodárenskou věží 2, CZ–182 07 Prague, The Czech Republic

Email: strakos@uivt.cas.cz tuma@uivt.cas.cz

Abstract: Our goal is to show on several examples the great progress made in numerical analysis in the past decades together with the principal problems and relations to other disciplines. We restrict ourselves to numerical linear algebra, or, more specifically, to solving $Ax = b$ where A is a real nonsingular n by n matrix and b a real $n-$dimensional vector, and to computing eigenvalues of a sparse matrix A. We discuss recent developments in both sparse direct and iterative solvers, as well as fundamental problems in computing eigenvalues. The effects of parallel architectures to the choice of the method and to the implementation of codes are stressed throughout the contribution.

Keywords: numerical analysis, numerical linear algebra, linear algebraic systems, sparse direct solvers, iterative methods, eigenvalue computations, parallel architectures and algorithms

1 Motivation

Our contribution deals with numerical analysis. What is numerical analysis? And what is its relation to computer science (computing sciences)? It is hard to give a definition. Sometimes "pure" mathematicians do not consider numerical analysis as a part of mathematics and "pure" computer scientists do not consider it as a part of computer science. Nevertheless, hardly anyone would argue against the importance of this field. Dramatic development in high technology in the last decades would not be possible without mathematical modelling and extensive use of the numerical analysis, mentioning just one important area of application.

Nick Trefethen proposes in his unpublished note [38] the following definition:

> *Numerical analysis is the study of algorithms for mathematical problems involving continuous variables*

The keywords are *algorithm* and *continuous*, the last typically means real or complex. Since real or complex numbers cannot be represented exactly on computers, the part of the business of numerical analysis must be to *study rounding errors.* To understand finite algorithms or direct methods, e.g. Gaussian elimination or Cholesky factorization for solving systems of linear algebraic equation, one have to understand the computer architectures, operation counts and the propagation of rounding errors. This example, however, does not tell you all the story. *Most mathematical problems involving continuous variables cannot be solved (or effectively solved) by finite algorithms.* A classical example - there are no finite algorithms for matrix eigenvalue problems (the same conclusion can be extended to almost anything nonlinear). Therefore the deeper business of numerical analysis is *approximating unknown quantities that cannot be known exactly even in principle.* A part of it is, of course, estimating the precision of the computed approximation.

Our goal is to show on several examples the great achievements of the numerical analysis, together with the principal problems and relations to other disciplines. We restrict ourselves to *numerical linear algebra*, or more specifically, to solving $Ax = b$, where A is a real nonsingular n by n matrix and b a real n-dimensional vector (for simplicity we restrict ourselves to real systems; many statements apply, of course, also to the complex case), and to computing eigenvalues of a square matrix A. Much of scientific computing depends on these two highly developed subjects (or on closely related ones). This restriction allows us to go deep and show things in a sharp light (well, at least in principle; when judging this contribution you must take into account our - certainly limited - expertise and exposition capability).

We emphasize two main ideas which can be found behind our exposition and which are essential for the recent trends and developments in numerical linear algebra. First, *our strong belief is that any "software $\to$ solution" without a deep understanding of mathematical (physical, technical, ...) background of the problem is very dangerous and may lead to fatal errors.* This is illustrated, e.g., on the problem of computing eigenvalues and on characterizing the convergence of the iterative methods. Second, *there is always a long way (with many unexpectably complicated problems) from the numerical method to the efficient and reliable code.* We demonstrate that especially on the current trends in developing sparse direct solvers.

2 Solving large sparse linear algebraic systems

Although the basic scheme of the symmetric Gaussian elimination is very simple and can be casted in a few lines, the effective algorithms which can be used for the really large problems usually take from many hundreds to thousands of code statements. The difference in the timings can then be many orders of magnitude. This reveals the real complexity of the intricate codes which are necessary to cope with the large real-world problems. Subsection 2.1 is devoted to the sparse direct solvers stemming from this symmetric Gaussian elimination. Iterative solvers, which are based on the approaching the solution step by step from some initial chosen approximation, are discussed in subsection 2.2. A comparison of both approaches is given in subsection 2.3.

2.1 Sparse direct solvers

This section provides a brief description of the basic ideas concerning *sparsity in solving* large linear systems by direct methods. Solving the large sparse linear systems is the bottleneck in a wide range of engineering and scientific computations. We restrict to the symmetric and positive definite case where most of the important ideas about algorithms, data structures and computing facilities can be explained. In this case, the solution process is inherently stable and we can thus avoid numerical pivoting which would complicate the description otherwise.

Efficient solution of large sparse linear systems needs a careful choice of the algorithmic strategy influenced by the characteristics of computer architectures (CPU speed, memory hierarchy, bandwidths cache/main memory and main memory/auxiliary storage). The knowledge of the most important architectural features is necessary to make our computations really efficient.

First we provide a brief description of the Cholesky factorization method for solving of a sparse linear system. This is the core of the symmetric Gaussian elimination. Basic notation can be found, e.g., in [2], [18].

We use the square root formulation of the factorization in the form

$$A = LDL^T,$$

where L is lower triangular matrix and D is diagonal matrix. Having L, solution x can be computed using two back substitutions and one diagonal scaling:

$$L\bar{y} = b \ ; \ y = D^{-1}\bar{y} \ ; \ L^T x = y.$$

Two primary approaches to factorization are as follows (we do not mention the row-Cholesky approach since its algorithmic properties usually do no fit well with the modern computer architectures).

The left-looking approach can be described by the following pseudo-code:

```
(1)  for $j = 1$ to $n$
(2)      for $k = 1$ to $j - 1$ if $a_{kj} \neq 0$
(3)          for $i = k + 1$ to $n$ if $l_{ik} \neq 0$
(4)              $a_{ij} = a_{ij} - l_{ik}a_{kj}$
(5)          end $i$
(6)      end $k$
(7)          $d_j = a_{jj}$
(8)          for $i = k + 1$ to $n$
(9)              $l_{ij} = a_{ij}/a_{jj}$
(10)         end $i$
(11) end $j$
```

In this case, a column j of L is computed by gathering all contributions from the previously computed columns (i.e. the columns to the left of the column j in the matrix) to the column j. Since the loop at the lines (3)–(5) in this pseudo-code involves two columns, j and k, with potentially different nonzero structures, the problem of matching corresponding nonzeros must be resolved. Vector modification of the column j by the column k at the lines (3)–(5) is denoted by *cmod*(j, k). Vector scaling at the lines (8)–(10) is denoted by *cdiv*(j).

The right-looking approach can be described by the following pseudo-code:

```
(1)  for $k = 1$ to $n$
(2)          $d_k = a_k$
(3)      for $i = k + 1$ to $n$ if $a_{ik} \neq 0$
(4)          $l_{ik} = a_{ik}/d_k$
(5)      end i
(6)      for $j = k + 1$ to $n$ if $a_{kj} \neq 0$
(7)          for $i = k + 1$ to $n$ if $l_{ik} \neq 0$
(8)              $a_{ij} = a_{ij} - l_{ik}a_{kj}$
(9)          end $i$
(10)     end $j$
(11) end $k$
```

In the right-looking approach, once a column k is completed, it immediately generates all contributions to the subsequent columns, i.e., columns to the right of it in the matrix. A number of approaches have been taken in order to solve the problem of matching nonzeros from columns j and k at the lines (6)-(10) of the pseudo-code as will be mentioned later. Similarly as above, operation at the lines (3)–(5) we denote *cdiv*(k) and column modification at the lines (7)–(9) is denoted by *cmod*(j, k).

Discretized operators from most of the applications such as structural analysis, computational fluid dynamics, device and process simulation and electric power network problems contain only a fraction of nonzero elements: these matrices are very *sparse*. Explicit consideration of sparsity leads to substantial savings in space and computational time. Usually, the *savings in time are more important*, since the time complexity grows quicker as a function of the problem size than the space complexity (memory requirements).

$$A = \begin{pmatrix} * & & & & * \\ & * & & & * \\ & & * & & * \\ & & & * & * \\ * & * & * & * & * \end{pmatrix} \qquad \bar{A} = \begin{pmatrix} * & * & * & * & * \\ * & * & & & \\ * & & * & & \\ * & & & * & \\ * & & & & * \end{pmatrix}$$

Figure 1.1: The arrow matrix and in the natural and reverse ordering.

For a dense problem we have CPU time proportional to n^3 while the necessary memory is proportional to n^2. For the sparse problems arising from the mesh discretization of 3D problems CPU time is proportional (typically) to n^2 and space proportional to $n^{\frac{4}{3}}$.

We will demonstrate the time differences using the following simple example from [30]:

Example 2.1 Elapsed CPU time for the matrix arising from a finite element approximation of a 3D problem (design of an automobile chassis). Dimension of a the matrix: $n = 44609$. Proportion of nonzeros: 0.1%. Time on Cray X-MP (1 processor) when considered as a full (dense) matrix: 2 days. Time on Cray X-MP when considered as a sparse matrix: 60 seconds.

Considering matrix sparsity we must care about the positions of nonzeros in the matrix patterns of A and L. For a given vector $v \in R^k$ define

$$Struct(v) = \{j \in \hat{k} | v_j \neq 0\}.$$

Usually, nonzero elements are introduced into the new positions outside the pattern of A during the decomposition. These new elements are known as *fill-in* elements. In order to reduce time and storage requirements, it is necessary to minimize the number of the fill-in elements. This can be accomplished by a combination of a good *choice of data structures used for matrix elements, matrix ordering and an efficient implementation of the pseudo-code.*

A typical example showing how the matrix ordering influences time and storage is the case of an arrow matrix in the Figure 1.1. While in the first matrix A we do not get any fill-in during the Cholesky factorization process, reverse ordering of variables provides matrix $\bar{A}$, which completely fills after the first step of the decomposition:

2.1.1 A Special sparsity structure – Profile, band and frontal schemes

A well-known way to make use of the sparsity in the Cholesky factorization is to move the nonzero elements of A into the area "around" the diagonal. Natural orderings of many application problems lead to such concentrations of nonzeros.

Define $f_i = min\ \{j \mid a_{ji} \neq 0\}$ for $i \in \hat{n}$. This locates the leftmost nonzero element in each row. Set $\delta_i = i - f_i$. The *profile* is defined by $\sum_{i=1}^{n} \delta_i$. The problem of concentrating elements around the diagonal can be thus reformulated as the problem of minimizing the profile using symmetric reordering of matrix elements.

$$\begin{pmatrix} * & * & & & * \\ * & * & * & & * \\ & * & * & * & * \\ & & * & * & * \\ * & * & * & * & * \end{pmatrix}$$

Figure 1.2: An matrix illustrating the profile Cholesky scheme. We have $f_1 = 1, f_2 = 1, f_3 = 2, f_4 = 3, f_5 = 1$.

Sometimes we use a rougher measure of the quality of the ordering - we are minimizing only *band* of the matrix - (often) defined as $\beta = max\ \delta_i$, for $i \in \hat{n}$.

$$\begin{pmatrix} * & * & * & & & \\ * & * & * & * & & \\ * & * & * & * & * & \\ & * & * & * & * & \\ & & * & * & * & * \\ & & & & * & * \end{pmatrix}$$

Figure 1.3: An example of a band matrix with $\beta = 2$.

More advanced variant of this principle relies on the *dynamical reordering* of the matrix to get the nonzeros as close to the diagonal as possible *during* the Cholesky factorization. Such an algorithm we call the *frontal* method. In this case we use in any step only the elements of a certain window which is moving down the diagonal.

For the algorithms to *reorder* matrices according to this principles see [18]. Advantages of the methods considered in this subsection are in their simplicity. To store the nonzeros we need to store only that part of the matrix

A which is covered by the elements which determine the profile or the band of A or the dynamical window of the frontal method.

Simple result (see, for instance, [18]) guarantees that all the nonzeros of L are inside the above-mentioned part determined from the pattern of A. This observation justifies the three algorithmic types mentioned in this subsection.

To implement *band* elimination we need only to store nonzero elements in a rectangular array of the size $\beta \times n$. Some problems generate matrices with a main diagonal and with one or more nonzero subdiagonals. In this case we can store also these diagonals and diagonals necessary for fill-in as a set of "long" vectors. To implement the *profile* method, we usually need even less nonzero positions and one additional pointer vector to point to the first nonzeros in the matrix rows. *Frontal* elimination needs vectors to perform row and column permutations of the system dynamically, throughout the factorization steps. More complicated implementation is compensated by the advantage of smaller working space. All these possibilities can be considered in both the right- and left-looking implementations but the differences are not, in general, very large since all these models are based on similar principles.

Although all these three schemes are very simple to implement and also the data structures are simple (nonzero parts of rows, columns or diagonals are stored in vectors or rectangular arrays as dense pieces), they are not used very often as in recent sparse symmetric solvers.

First reason is algorithmic. General sparse schemes may have much less fill-in elements than the previous schemes would implicitly suppose. We demonstrate this fact on the previously mentioned example taken from [30]:

Example 2.2 Comparison of factor size and number of floating-point operations for the matrix arising from the finite element approximation of a 3D problem (design of an automobile chassis). Dimension of a the matrix: $n = 44609$. Memory size used and number of floating-point operations for the factor L for the frontal solver: 52.2 MByte / 25 Billion. Memory size used and number of floating-point operations for L when general sparse solver was used: 5.2 MByte / 1.1 Billion.

The second reason is architectural. Hardware *gather/scatter facilities* used in modern computers (see [26]) caused that even the simplicity of data structures for band, profile and frontal solvers are not able to guarantee competitive computation times. They behave worse than general sparse solvers even in the case when the difference in number of fill-in elements (size of the factor L) is not so dramatic.

2.1.2 General Sparse Solvers

To describe basic ideas used in today's general sparse solvers we need to introduce some terminology. Undirected graphs are useful tools in the study of symmetric matrices. A given matrix A can be structurally represented by its associated graph $G(A) = (X(A), E(A))$, where nodes in $X(A) = \{1, \ldots, n\}$ correspond to rows and columns of the matrix and edges in $E(A)$ correspond to nonzero entries.

Filled matrix $F = L + L^T$ contains generally more nonzeros than A. Structure of F is captured by the *filled graph* $G(F)$. The problem how to get structure of nonzeros of F was solved first in [33] using graph-theoretic tools to transform $G(A)$ to $G(F)$.

An important concept in sparse factorization is the *elimination tree* of L. It is defined by

$$parent(j) = min\ \{i \mid l_{ij} \neq 0, i > j\}.$$

In other words, column j is a child of column i if and only if the first subdiagonal nonzero of column j in L is in row i. Figure 1.4 shows structures of matrices A and L and the elimination tree of L.

$$\begin{pmatrix} * & & & * & \\ & * & & & * \\ & & * & * & * \\ * & & * & * & f \\ & * & * & f & * \end{pmatrix}$$

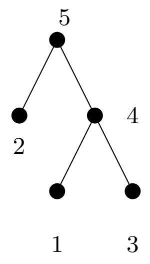

Figure 1.4: Structures of matrices A, L and of the elimination tree of L. By stars we denote original nonzeros of A, additional fill-in in L is denoted by the letter f.

Following few lemmas recall some important properties of elimination trees which will help us to understand basic principles of sparse solvers. For a deeper insight into this subject we refer to [25]. Note, that elimination tree can be computed directly from the structure of nonzeros of A in complexity nearly linear in n (see [27]).

Lemma 2.1 *If $l_{ij} \neq 0$, then the node i is an ancestor of j in the elimination tree.*

This observation provides a necessary condition in terms of the ancestor-descendant relation in the elimination tree for an entry to be nonzero in the filled matrix.

Lemma 2.2 *Let $T[i]$ and $T[j]$ be two disjoint subtrees of the elimination tree (rooted at i and j, respectively). Then for all $s \in T[i]$ and $t \in T[j]$, $l_{st} = 0$.*

One important problem concerning the Cholesky factorization is how to determine row structures of L. For instance, in the left-looking pseudo-code, nonzeros in a row k of L correspond to columns from $\{1, \ldots, k-1\}$ which contribute to column k.

Lemma 2.3 *$l_{ij} \neq 0$ if and only if the node j is an ancestor of some node k in the elimination tree, where $a_{ik} \neq 0$.*

This result can be used to characterize the row structure of the Cholesky factor. Define $T_r[i]$, the structure of the $i-th$ row of the Cholesky factor as follows

$$T_r[i] = \{j | l_{ij} \neq 0, j \leq i\}.$$

Then we have

$$T_r[i] \subseteq T[i].$$

Moreover, it can be shown that $T_r[i]$ is a pruned subtree of $T[i]$ and that its leaves can be easily determined.

The important corollary from these considerations is that *structure of rows in L can be easily determined using the elimination tree.*

The second important problem concerning the implementations - determination of the structures of columns of L - can be also very easily solved also using the elimination tree:

Lemma 2.4 *$Struct(L_{*j}) \equiv \{i | l_{ij} \neq \emptyset \wedge i \geq j\}$*
$= \bigcup_{\underline{k} \text{ is son } \underline{j} \text{ in } T(A)} Struct(L_{*k}) \bigcup Struct(A_{*j})$
$- \{1, \ldots, j-1\}$

This is a simple corollary of the Cholesky decomposition step and dependency relations captured by the elimination tree. This formula means that in order to get the structure of a column i in L we need only to merge structures of the column i in A with structures of sons of i in L. Consequently, the algorithm to determine the column structure of L can be implemented in $O(m)$ operations where m is a number of nonzeros in L.

The elimination tree gathers the most important structural dependencies in the Cholesky factorization scheme. Numbering of its vertices determines the order in which the matrix entries are processed by the solver. Moreover, we can renumber vertices and/or even modify the elimination tree while preserving the amount of fill-in elements in the correspondingly permuted factor L. Motivations for such changes will be described in the following subsection.

Basic structure of the general sparse left-looking solver can then be given in the following four steps (not taking into account the ordering phase):

- Form the elimination tree.
- Find the structure of columns of L (symbolic factorization).
- Allocate the static data structures for L based on the result of the previous step.
- Do numerical updates corresponding to the left-looking pseudo-code.

Implementation of the right-looking solver can be based on the similar scheme.

One of the popular type of the right-looking algorithm is the *multifrontal method* (see [12], [3]). In that method, the update step of the pseudo-code (steps (6)–(10)) is implemented using *dense matrix operations.* Updates are stored separately, usually in the separate stack working area, but some of them can make use of the final space for L. We do not discuss this method as a special case in our overview of sparse techniques, because its implementations on various computers can mix the ideas previously mentioned.

Since we can effectively compute structures of columns and rows of L, the *symbolic overhead* in the computations is rather small. But this does not mean that our computations are effective. The problem is that the choice of the algorithmic strategy has to match with the computer architecture. We will consider some issues of this kind in the following subsection.

2.1.3 Let the architecture reign

Suppose we have the usual memory hierarchy with fast and small memory parts at its top (registers, cache) and slower and bigger parts at the bottom. Usually, it is not possible to embed the large problems completely into the cache and transfer between memory hierarchy levels takes considerable time. On the other side, computations are the most effective only if the active data are as close to the top as possible. This implies that we must in some way maximize the proportion of the number of floating-point operations (flops) to number of memory references which enable us to keep the data in the cache and registers.

Basic approach how to accomplish that is to use rather operations with *blocks of data* instead of operations with matrix elements. This important observation is now commonly used in linear algebra operations (see [15]). It was introduced after the spreading of vector supercomputer architectures. The main problem with achieving the supercomputer performance on these architectures was to keep the vector functional units busy - to get enough data for them. Blocking data to use matrix-vector and matrix-matrix operations was found to be very successful.

A comparison of number of memory references and number of flops for the three types of basic operations in dense matrix computations is in the Table 1.1. We consider α to be a scalar, x, y to be vectors and B, C, D to be (square) matrices; n represents the size of the vectors and matrices.

operation	# m. r.	# flops	$\frac{\#\text{m.r.}}{\#\text{flops}}$
$y \leftarrow \alpha x + y$	$3n$	$2n$	$\frac{3}{2}$
$y \leftarrow Dx + y$	n^2	$2n^2$	$\frac{1}{2}$
$B \leftarrow DC + B$	$4n^2$	$2n^3$	$\frac{2}{n}$

Table 1.1: Comparison of number of memory references (m.r.) and number of flops for the three types of block hierarchy in dense matrix computations.

A similar principle can be applied also for the scalar computations in general sparse solvers. Note, that vector pipelining is not decisive; much more important is keeping the CPU unit as busy as possible while minimizing the data transfers. Thus, the efficient implementations of both the left-looking and right-looking algorithms require that columns of L sharing the same sparsity structure are grouped together into *supernodes.* More formally, the set of continguous columns $\{j, j+1, \ldots, j+t\}$ constitutes a supernode if $Struct(l_{*k}) = Struct(l_{*k+1}) \cup \{k\}$ for $j \le k \le j+t-1$. Note that these columns have a dense diagonal block and have identical column structure below row $j+t$. A supernode can be treated as a computational and storage unit. Following example shows the difference in timings for the sparse Cholesky right-looking decomposition using supernodes.

Example 2.3 Elapsed time for the linear system with the SPD matrix of the dimension $n = 5172$ arising from the 3D finite element discretization. Computer: 486/33 IBM PC compatible. Virtual paging system uses the memory hierarchy: registers /cache /main memory /hard disk. General sparse right-looking solver. Elapsed time without block implementation: 40 min. Elapsed time with supernodal implementation: 4 min.

Another way how to decrease the amount of communication in the general sparse solver is to try to perform the update operations in such a way that the data necessary in a step of the decomposition are as close together as possible. Elimination tree can serve as an efficient tool to describe this principle. Having an elimination tree, we can renumber some of its vertices in such a way that the Cholesky decomposition with the corresponding permutation will provide the factor L of the same size. Consider the renumbering of the elimination tree from the Figure 1.4. This elimination tree is renumbered by a so-called *postordering* (a topological ordering numbering any subtree by an interval of indices). This reordering is equivalent to the original one in the sense that it provides the same factor L.

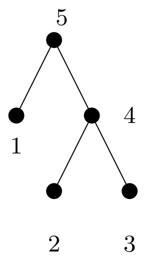

Figure 1.5: Postordering of the elimination tree of the Figure 1.4

Since the postordering numbers indices in such a way that vertices in any subtree are numbered before giving numbers to any disjoint subtree, we can expect much less data communications than in the previous case. The difference is described by the number of page faults in the virtual paging system in the following example (see [31]).

Example 2.4 Comparison of number of page faults for a matrix arising from the 9-point discretization of a regular 2D 180×180 grid (dimension $n = 32400$). Number of page faults for an level-by-level ordering of the elimination tree from the bottom to the top: 1.670.000. Number of page faults using postordering of the elimination tree: 18.000.

Another strategy to obtain the equivalent reordering that can reduce the active storage is to rearrange the sequences of children in the elimination tree. The situation is depicted on the Figure 1.6. While on the left part of the figure we have a tree with some initial postordering, on the right side we are numbering "large" children first. We are doing it in any node and recursively. For instance, considering vertex 18, we have numbered "largest" subtree of the right side elimination tree first.

More precisely, the new ordering is based on the structural simulation of the Cholesky decomposition. Active working space in each step can be determined for various renumberings of children of any node. In this way, the recursive structural simulation can determine a new renumbering permuting the subtrees corresponding to the tree nodes in order to minimize the active working space without changing the elimination tree. Consider, for instance, vertex 18 in the elimination trees. Decision, in which order we will process its sons (and, of course, all its subtrees considering postordered elimination tree) is based on the recursively computed value of the temporary active working space. The natural idea is to start with processing of "large" subtrees. Both the children rearrangements are postorderings of the original elimination tree. The actual size of the working space depends directly on whether we are minimizing working space for the multifrontal method or for some left-looking or right-looking out-of-core (out-of-cache) solver.

If we are looking for the equivalent transformations of the elimination tree in order to minimize the active working space, we can not only shrink vertices into supernodes and renumber elimination tree. We can also to change the whole structure of the elimination tree. Theoretical basis of this transformation is described in [29].

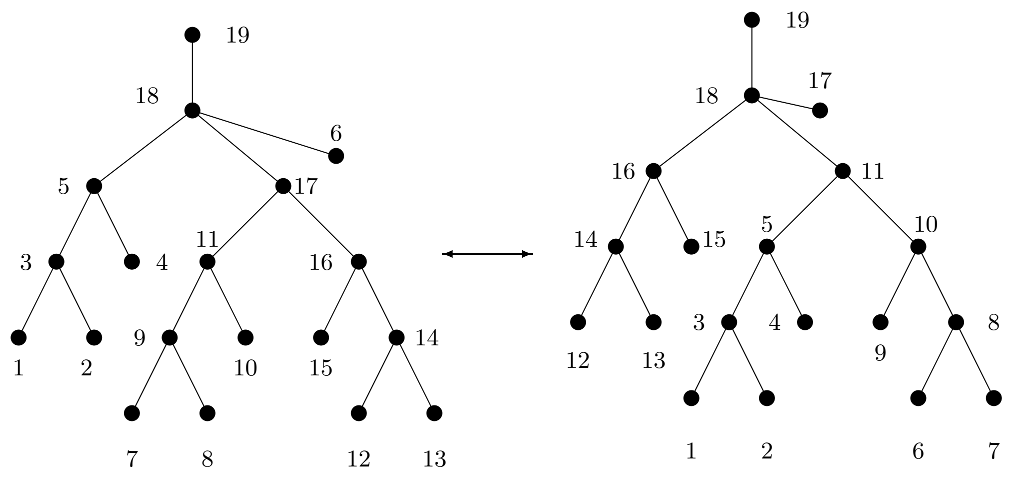

Figure 1.6: Rearranging of the children sequences in the elimination tree.

Instead of the balanced elimination tree as provided by some matrix reorderings minimizing fill-in elements we can use unbalanced elimination tree which can in practice decrease the active working space about the size up to 20%.

Balanced and unbalanced elimination trees are schematically depicted on the Figure 1.7.

In practice, all these techniques deal with the supernodes rather than with the individual entries.

Balanced and effective use of the described techniques is a difficult task strongly depending on the computer architecture for which the solver is constructed. For computers with relatively quick (possibly vectorizable) floating-point operations and slow scalar arithmetics, one can effectively merge into the supernodes more vertices despite the fact that the resulting structure of L would have additional nonzeros (see [5]). On the other hand, sometimes it is necessary to construct smaller supernodes by breaking large blocks of vertices into pieces (see [35]). This is the case of workstation and also of some PC implementations. Elimination tree rearrangements provide an a posteriori information for the optimal partitioning of blocks of vertices.

Computer architecture is the prominent source of information for implementing any general sparse solver. Without knowledge of the basic architectural features and technical parameters we are not able even to decide which combination of the left-looking and right-looking techniques is optimal. There are many open problems in this area. Theoretical investigation leads often to the directly applicable results.

2.2 Recent development in iterative solvers: steps towards a black box iterative software?

A large amount of black box software in the form of mathematical libraries as LAPACK (LINPACK), NAG, EISPACK, IMSL etc. and general sparse solvers as MA27, MA42, MA48, UMFPACK, etc. have been developed and are widely used in many applications.

Users can exploit this software with high confidence for general problem classes. Concerning systems of linear algebraic equations, codes are based almost entirely on *direct* methods. Black box *iterative* solvers would be highly desirable and practical - they would avoid most of the implementation problems related to exploiting the sparsity in direct methods. In recent years many authors devote a lot of energy into the field of iterative methods and a tremendous progress have been achieved. For a recent survey we refer e.g. to [14] and [8]. Sometimes the feeling is expressed that this progress have already established a firm base for developing a black box iterative software. This is, however, very far from our feeling.

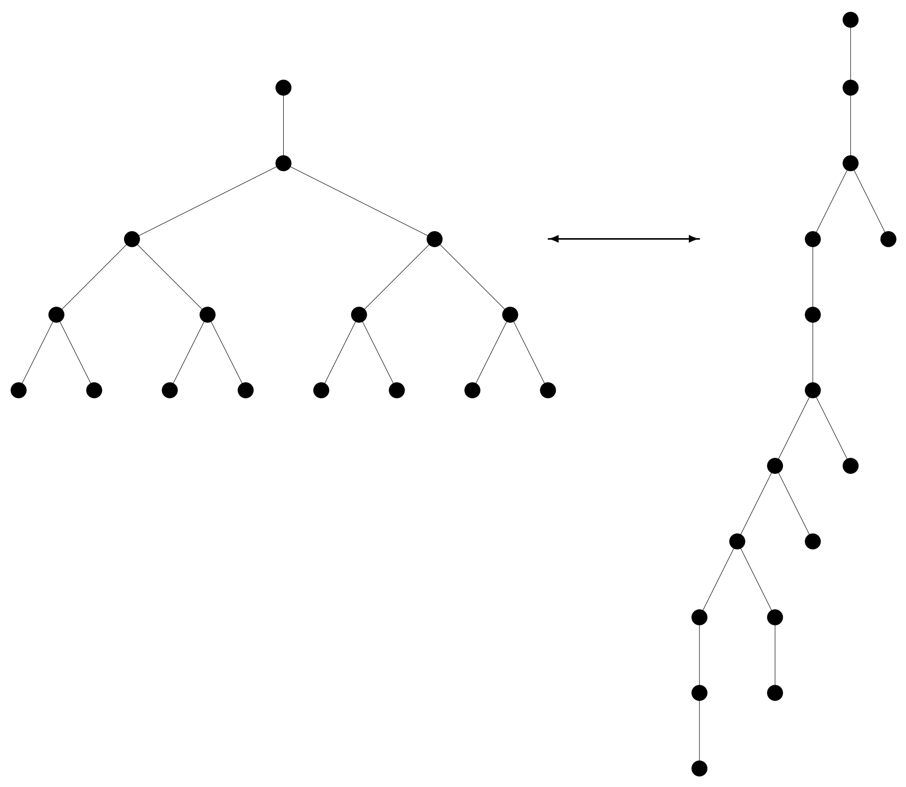

Figure 1.7: Balanced and unbalanced elimination trees.

Strictly speaking, we do not believe that any good (fast, precise, reliable and robust) black box iterative software for solving systems of linear algebraic equations will be available in the near future. This section describes several good reasons supporting our opinion. Many iterative methods have been developed; for the excellent surveys of the classical results we refer to [39], [41], [7] and [22]. For historical reasons we recall briefly the basic iterative methods and then turn to the state-of-the-art: the Krylov space methods.

The best known iterative methods - the linear stationary iterative methods of the first kind - are characterized by the simple formula

$$x^{(k)} = Bx^{(k-1)} + c$$

where $x^{(k)}$ is the current approximation to the solution at the k-th step, ($x^{(0)}$ is given at the beginning of computation), the n by n matrix B and vector c characterize the method. Any method of this type is *linear* because $x^{(k)}$ is given as a linear function of the previous approximations; it is *of the first kind* because the iteration formula involves the information just about the one previous step and it is *stationary* because neither B nor c depend upon the iteration step k. Everyone knows examples as the Richardson method, the Jacobi method, the Gauss-Seidel method, the Successive Overrelaxation method (SOR) and the Symmetric Successive Overrelaxation method (SSOR). We are not going to repeat the formulas or the theory of these methods here, that can be found elsewhere. Instead, we recall the very well known fact, that these simple methods (especially SOR and SSOR) may show an excellent performance while carefully tuned to a specific problem, but their performance is very problem - sensitive. This lack of robustness avoid their general use for a wide class of problems.

Nonstationary iterative methods differ from stationary methods in that the parameters of the formula for computing the current approximation depend on the iteration step. Consequently, these methods are more robust; in many cases they are characterized by some minimizing property. In the last years most of the effort in this field is devoted into the Krylov space methods.

Krylov space methods for solving linear systems start with an initial guess $x^{(0)}$ for the solution and seek the k-th approximate solution in the linear variety

$$x^{(k)} \in x^{(0)} + K_k(A, r^{(0)})$$

where $r^{(0)} = b - Ax^{(0)}$ is the initial residual and $K_k(A, r^{(0)})$ is the k-th Krylov space generated by $A, r^{(0)}$,

$$K_k(A, r^{(0)}) = \text{span}\,\{r^{(0)}, Ar^{(0)}, \ldots, A^{k-1}r^{(0)}\}.$$

Then, the k-th error and the k-th residual are written in the form

$$\begin{aligned} x - x^{(k)} &= p_k(A)(x - x^{(0)}) \\ r^{(k)} &= p_k(A)r^{(0)}, \end{aligned}$$

where $p_k \in \mathcal{P}_k$, $\mathcal{P}_k$ denotes the set of polynomials $p(\lambda)$ of the degree at most k satisfying $p(0) = 1$. Based on that, Krylov space methods are also referred to as polynomial methods. An enormous variety of the Krylov space methods exists, including the famous conjugate gradient method (CG), the conjugate residual method (CR), SYMMLQ, the minimal residual method (MINRES), the biconjugate gradient method (BiCG), the quasiminimal residual method (QMR), the generalized minimal residual method (GMRES), giving just a few names.

In the rest of this subsection we will concentrate on the Krylov space methods and show several principial questions which must be satisfactorily answered prior to constructing any good black box iterative solver.

First question can be formed as: *What characterizes convergence of the Krylov space method?* This is certainly a key question. Without giving an answer one can hardly build up theoretically well justified preconditioned methods. For the Hermitian and even the normal systems (i.e. the systems characterized by the Hermitian or normal matrix) the answer is: the rate at which a Krylov space method converges is determined by the eigenvalue distribution and the initial approximation (initial error, initial residual). It can be clearly seen by substituting the unitary eigendecomposition of the the matrix A, $A = U\Lambda U^*$, $U^*U = UU^* = I$, I is the identity matrix, into the polynomial formulation of the methods. For nonnormal matrices, hovewer, no unitary eigendecomposition exist. Despite that, many authors extend intuitively the feeling that the spectrum of the coefficient matrix plays decisive role in the characterization of convergence even in the nonnormal case. This is actually a very popular belief which is (at least implicitly) present in discussions of the experimental results in many papers. This belief is, however, *wrong!*

As an example we can take the GMRES method. GMRES approximations are chosen to minimize the Euclidean norm of the residual vector $r^{(k)} = b - Ax^{(k)}$ among all the Krylov space methods, i.e.,

$$\| r^{(k)} \| = \min_{u \in x^{(0)} + K_k(A, r^{(0)})} \| b - Au \| \, .$$

Residual norms of successive GMRES approximations are nonincreasing, since the residuals are being minimized over a set of expanding subspaces. Notice that due to the minimizing property, the size of GMRES residuals forms a lower bound for the size of the residual of any other Krylov space method. In other words, GMRES shows how small residual can be found in the variety

$$r^{(0)} + AK_k(A, r^{(0)}).$$

If GMRES performs poorly, then any other Krylov space method performs poorly as well (measured by the size of the residual). But the size of the residual - that is the only easy-to-compute convergence characteristic.

A key result was proven in [21] (for the other related results see also [20]). It can be reformulated in a following way. Given a nonincreasing positive sequence $f(0) \geq f(1) \geq \ldots f(n-1) \geq 0$, and a set of nonzero complex numbers $\{\lambda_1, \ldots \lambda_n\}$, there is a n by n matrix A having eigenvalues $\{\lambda_1, \ldots \lambda_n\}$, and an initial residual $r^{(0)}$ with $\| r^{(0)} \| = f(0)$ such that the GMRES algorithm applied to the linear system $Ax = b$, with initial residual $r^{(0)}$, generates approximations $x^{(k)}$ such that $\| r^{(k)} \| = f(k)$, $k = 1, 2, \ldots, n-1$. In other words, *any nonincreasing convergence curve* *can be obtained with GMRES applied to a matrix having any desired eigenvalues!* The results of [20] and [21] demonstrate clearly that eigenvalues alone *are not* the relevant quantities in determining the behavior of GMRES for nonnormal matrices. *It remains an open problem to determine the most appropriate set of system parameters for describing the GMRES behavior.*

A second question is: *What is the precision level which can be achieved by iterative methods and which stopping criteria can be effectively used?* The user will always ask: Where to stop the iteration? Stopping criteria should guarantee a small error. If error is considered as a distance to the true solution (measured e.g. in the Euclidean norm), then the question is too hard - one usually has no tools to estimate directly this so called *forward* error. The other possibility is to consider the error in the *backward* sense, i.e., consider the approximation $x^{(k)}$ to the solution x as the exact solution of a perturbed system

$$(A + \Delta A)x^{(k)} = b + \Delta b,$$

and try to make the perturbations ΔA and Δb as small as possible. It is well known, see [24], that for a given $x^{(k)}$ such a minimal perturbations, measured in the Euclidean resp. spectral norms, exist, and their size is given by

$$\begin{aligned} & \min\{\nu : (A + \delta A)x^{(k)} = b + \Delta b, \\ & \parallel \Delta A \parallel / \parallel A \parallel \leq \nu, \\ & \parallel \Delta b \parallel / \parallel b \parallel \leq \nu\} = \\ = \; & \parallel b - Ax^{(k)} \parallel / (\parallel A \parallel \parallel x^{(k)} \parallel + \parallel b \parallel). \end{aligned}$$

Consequently, to guarantee a small backward error, it is sufficient to use the stopping criteria based on the value of

$$\parallel b - Ax^{(k)} \parallel / (\parallel A \parallel \parallel x^{(k)} \parallel + \parallel b \parallel).$$

The problem seems to be solved, because the residual is always available and the spectral norm of A can be approximated e.g. by the easily computable Frobenius norm. A careful reader will, however, raise a question about *rounding errors*. This is a real crucial point. Without a careful rounding error analysis we cannot say anything about the size of the ultimate (or "final") residual in practical computations. Consequently - we cannot predict a priori the precision level on which the iteration should be stopped! For more details we refer to [11] and [36].

One can form many other questions of similar importance. As stated earlier, a good iterative black box software must be fast, precise, reliable and robust. In all these attributes it must compete with highly effective (sparse) modern direct codes. We have discussed here some troubles related to the first two attributes. Even from the short characterization of iterative methods given above it is clear that the third and fourth attributes cause also a lot of problems which are unresolved yet (lacking in space we are not going into details here). Based on that, *we do not believe in constructing competitive black box iterative solvers in the near future.*

2.3 Direct or iterative solvers?

In this section we will first give some considerations concerning complexity of direct and iterative methods for the solution of linear systems arising in one special but important application (see [40]). Then we will state objections against the straightforward generalization of this simple case.

The matrix for our simple comparison arises from the self-adjoint elliptic boundary-value problem on the unit cube in 2D or 3D. The domain is covered with a mesh, uniform and equal in all 2 or 3 dimensions with mesh-width h. Discretizing this problem we get the symmetric and positive definite matrix A of the dimension n and the right-hand side vector b.

Consider the *iterative* method of conjugate gradients applied to this system. Considering exact arithmetics, the error reduction per iteration is $\sim \frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}$, where κ is the condition number of A defined as $\kappa = ||A|| ||A^{-1}||$.

Relation among the condition number κ and problem dimension as follows: for a 3D problem we have $\kappa \sim h^{-2} \approx n^{\frac{2}{3}}$, for a 2D problem we have $\kappa \sim h^{-2} \approx n$.

Then, for the error reduction below the level ϵ, we have that

$$\left(\frac{1-\frac{1}{\sqrt{\kappa}}}{1+\frac{1}{\sqrt{\kappa}}}\right)^j \approx (1-\frac{2}{\sqrt{\kappa}})^j \approx exp(\frac{-2j}{\sqrt{\kappa}}) < \epsilon \Longrightarrow$$
$$j \sim -\frac{log\ \epsilon}{2}\sqrt{\kappa}.$$

Assume the number of flops per iteration to be $\sim fn$ (f is a small integer standing for the average number of nonzeros per row and the overhead introduced by the iterative scheme). Then the number of flops for the convergence below the level ϵ is proportional to $fnj \sim n^{\frac{4}{3}}$ for 3D problems and $\sim fnj \sim n^{\frac{3}{2}}$ for 2D problems.

It is known, that many preconditioners (see [10]) are able to push the condition number of the system down to $O(h^{-1})$. Then the number of flops per reduction to ϵ is given by $\sim n^{\frac{7}{6}}$ and $\sim n^{\frac{5}{4}}$ for 3D and 2D problem, respectively.

Consider now a *direct* method. Using effective ordering strategies, we can have for the matrix mentioned above number of operations $\sim n^2$ and the size of the fill-in $\sim n^{\frac{4}{3}}$ in 3 dimensions. For 2D problem the corresponding numbers are $\sim n^{\frac{3}{2}}$ for the number of operations and $\sim n\ log\, n$ for the fill-in size. The corresponding estimates and their relation to practical problems can be found in [34] and [1]. Back substitution can be done in $\sim n^{\frac{4}{3}}$ operations for the 3D problem and in $n\ log\, n$ operations for the 2D problem.

If we have to solve one system at a time then for large ϵ (small final precision) or very large n, iterative methods may be preferable. Having more complicated mesh structure or more right-hand sides, direct methods can be usually preferable up to very large matrix dimensions. Iterative methods are usually more susceptible to instabilities (or slowing down the convergence) in finite precision arithmetics. Moreover, notice that the additional effort due to the computation and use of the preconditioner are not reflected in the asymptotic formulas. For many large problems we need sophisticated preconditioners which increase substantially the computational effort described for the model problem.

The amount of memory needed for computations makes also an important defference. This is usually much smaller for the iterative methods. On the other side, we can use space of the size $O(n)$ for the sparse Cholesky factorization (see [13]) when only one right-hand side is present.

Summarizing, sparse direct solvers would win as a general purpose codes up to the very large size of the problems n. For specific applications, or extremely large n, the iteration with preconditioning might be a better or even the only alternative. This conclusion represents the state of knowledge in the early 90's and are, of course, subject to change depending on the progress in the field.

3 Computing eigenvalues - a principal problem!

To show how hopeless and dangerous might be a naive "software $\rightarrow$ solution" approach without understanding the "nature" of the problem we consider an "elementary" problem - computing eigenvalues $\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ of a n by n matrix A.

We will not specify the algorithm; just suppose that it is backward stable, i.e., the computed approximations $\{\mu_1, \mu_2, \ldots \mu_n\}$ are *exact* eigenvalues of the matrix $\overline{A}$ which is just a slight perturbation of the matrix A,

$$\overline{A} = A + E, \qquad \| E \| \leq \delta,$$

where δ is small (proportional to the machine precision). That is the best we can hope for. A question is, however, how close are $\{\mu_1, \ldots, \mu_n\}$ to $\{\lambda_1, \ldots, \lambda_n\}$. We define the (optimal) matching distance between the eigenvalues of A and $\overline{A}$ as

$$\mathrm{md}(A, \overline{A}) = \min_{\pi}\{\max_{i} |\lambda_{\pi(i)} - \mu_i|\}$$

where π is taken over all permutations of $\{1, \ldots, n\}$. Using a naive approach, one might say: eigenvalues are continuous functions of the matrix coefficients. Therefore we can expect that for $\overline{A}$ close to A, the corresponding eigenvalues will be also close to each other and $\mathrm{md}(A, \overline{A})$ will be small.

The last conclusion is, of course, wrong! For a general matrix A, there is no bound on the $\mathrm{md}(A, \overline{A})$ linear in $\| E \|$, i.e., we can not guarantee anything reasonable about the precision of the computed eigenvalue approximations based on the size of the backward error. Even worse, for any small δ and any large ω, one can find matrices $A, \overline{A}$ such that $\| E \| = \| \overline{A} - A \| \leq \delta$ and $\text{md}(A, \overline{A}) \geq \omega$. Any small perturbation of the matrix (any small backward error) may in principle cause an arbitrary large perturbation of eigenvalues! In other words – *even the best software gives you, in general, no guarantee on the precision of the computed results.* This is certainly a striking statement.

Fortunately, there is an important class of matrices for which the situation is more optimistic. We recall the following theorem (for details see, e.g., [37]):

Theorem 3.1 *Let A be normal. Then* $\text{md}\,(A, \overline{A}) \leq (2n-1) \| E \|$.

For Hermitian matrices even stronger results can be proven. We see that for normal matrices the size of the backward error essentially determine the precision of the computed eigenvalue approximations.

To summarize, for normal matrices any good (i.e. backward stable) method will give us what we want - good approximation to the true eigenvalues. For highly nonnormal matrices, however, the computed approximation may be very far from the true eigenvalues even if the best software is used.

In this context please notice that many times authors just plot the computed eigenvalue approximations and declare it as the true eigenvalues without paying any attention to the normality (or other relevant properties) of their matrices.

4 Sparse linear solvers: parallelism in attack or in defense?

To show the difficulties related to parallel implementations of linear solvers, we concentrate here on sparse direct solvers. Description of the parallel implementation of the iterative methods is much more simple and can be found elsewhere.

Dense matrix computations are of such basic importance in scientific computing that they are usually among the first algorithms implemented in any new computing environment. Sparse matrix computations are equally as important, but both their performance and their influence on computer system design have tended to lag those of their dense matrix counterparts. One could add that for the greater complexity and irregularity, sparse matrix computations are more realistic representatives of typical scientific computations, and therefore more useful as benchmark criteria, than the dense matrix computations, that usually played this role.

Despite the difficulty with sparse matrix computations on the advanced computer architectures, some noticeable success has been achieved in attaining very high performance (see [9]) and the needs of sparse matrix computations have had notable effect on computer design (indirect addressing with gather/scatter facilities). Nevertheless, it is ironic that sparse matrix computations contain more inherent parallelism than the corresponding dense matrix computations, yet typically show significantly lower efficiency on today's parallel architectures.

Roughly speaking, the most widely available and commercially successful parallel architectures fall into three classes : shared-memory MIMD computers, distributed-memory MIMD architectures and SIMD computers. Some machines have an additional level of parallelism in the form of vector units within each individual processor. We will concentrate on the general and widely applicable principles which can be used in wide variations of these computing environments.

In the sparse Cholesky decomposition we can analyze the following levels of parallelism (see [28]):

- *Large-grain* parallelism in which each computational task is the completion of all columns in a subtree of the elimination tree.
- *Medium-grain* parallelism in which each task correspond to one simple cycle of column update *cmod* or column scaling *cdiv* operation in the left- and right-looking pseudo-codes.
- *Fine-grain* parallelism in which each task is a single floating-point operation or a multiply-add pair.

Fine-grain parallelism can be exploited in two distinctly different ways:

1. Vectorization of the inner cycles on vector processors.
2. Parallelizing the rank-one update in the right-looking pseudo-code.

Vectorization of the operations is one of the basic tools to improve effectiveness of the sparse solvers using array processors, vector supercomputers or RISC processors with some pipelining. Efficient vectorization was a very strong argument to promote band, profile and frontal solvers when first vector processors appeared. As noted above, except for special types of discretized partial differential equations, they are not very important now, and other concepts are used for the Cholesky decomposition of general matrices. This is caused by enormous work which was done in the research of direct sparse solvers, by gather/scatter facilities in today's computers for scientific computing and by the high-speed scalar arithmetics in workstations.

Hardware gather/scatter facilities can usually reach no more than 50% of the performance of the corresponding dense vector operations. No wonder, that the use of dense vectors and/or matrices in the inner cycles of the Cholesky decomposition is still preferable. The above-mentioned multifrontal implementation of the right-looking algorithm widely exploits this idea. The structure of the elimination tree enables to deal only with those elements which correspond to nonzeros in the factor L.

To obtain better performance using vector functional units, we usually strive to have dense vectors and matrices of the sufficiently large dimensions (we are not going into the careful analysis of the situation which is usually quite more complex). Thus, forming *supernodes* is usually highly desirable since it helps to increase the dimension of the submatrices processed in the inner cycle.

The problem of parallelizing rank-one update is a difficult one, and research on this topic is still in its infancy (see [19]). Note, that the right-looking algorithm presents for SIMD machines much better alternative that the column left-looking approach. When rows and columns of the sparse matrix A are distributed to the rows and columns of a grid of processors, the algorithm of sparse Cholesky decomposition is *scalable*. (by a scalable algorithm we call an algorithm that maintains efficiency bounded away from zero as the number p of processors grows and the size of the data structures grows roughly linearly in p, see [19]). Up to date, however, even this method has not achieved high efficiency on a highly parallel machine. With this note we left the realm of the highly-parallel and massively-parallel SIMD machines aside and we will turn to the parallelism exploited in the most popular parallel implementations: large-grain and medium-grain left-looking algorithms and multifrontal codes.

Let us turn now to the problem of *medium-grain* algorithms. Of the possible formulations of the sparse Cholesky algorithms, left-looking algorithm is the simplest to implement. It is shown in [16], that the algorithm can be adapted in a straightforward manner to run efficiently in parallel on shared-memory MIMD machines.

Each column j corresponds to a task

$$\begin{aligned} Tcol(j) = \{cmod(j,k) | k \in Struct(l_{*j})\} \\ \cup \{cdiv(j)\}. \end{aligned}$$

These tasks are given to a task queue in the order given by some possible rearrangement of columns and rows of A, i.e., given by some renumbering of the elimination tree. Processors obtain column tasks from this simple "pool of tasks" in this order. The basic form of the algorithm has two significant drawbacks. First, the number of synchronization operations is quite high. Second, since the algorithm does not exploit supernodes, it will not vectorize well on vector supernodes with multiple processors. The remedy is to use the supernodes to decrease also the synchronization costs.

Algorithms for distributed-memory machines are usually characterized by the a priori distribution of the data to the processors. In order to keep the cost of the interprocessor communication at acceptable levels, it is essential to use local data locally as much as possible. The distributed *fan-out* (see [17]), *fan-in* (see [6]), *fan-both* (see [4]) and *multifrontal* algorithm (see overview [23]) are typical examples of the implementations. All these algorithms are designed in the following framework:

- They require assignment of the matrix columns to the processors.
- They use the column assignment to distribute the medium-grained tasks in the outer loop of left- or right-looking sparse Cholesky factorization.

The differences among these algorithms stem from the various formulations of the sparse-Cholesky algorithm.

The *fan-out* algorithm is based on the right-looking Cholesky algorithm. We will denote the $k-th$ task performed by the outer loop of the algorithm (lines (3)–(10) of the right-looking pseudo-code) by $Tsub(k)$, which is defined by

$$Tsub(k) = \{cdiv(k)\} \\ \cup\{cmod(j,k)|j \in Struct(l_{*k})\}.$$

That is, $Tsub(k)$ first forms l_{*k} by scaling of the $k-th$ column, and then perform all column modifications that use the newly formed column. The fan-out algorithm partitions each task $Tsub(k)$ among the processors. It is a data-driven algorithm, where the data sent from one processor to another represent the completed factor columns. The outer loop of the algorithm for a given processor regularly checks the message queue for the incoming columns. Received columns are used to modify every column j owned by the processor for which $cmod(j,k)$ is required. When some column j is completed, it is sent immediately to all the processors, by which it is eventually used to modify subsequent columns of the matrix.

The *fan-in* algorithm is based on the left-looking Cholesky pseudo-code. It partitions each task $Tcol(j)$ among the processors in a manner similar to the distribution of tasks $Tsub(k)$ in the fan-out algorithm. It is a demand-driven algorithm where the data required from a processor p_a to complete the $j-th$ column on a given processor p_b are gathered in the form of results $cmod(j,k)$ and sent together. Communication costs incurred by this algorithm are usually much lower than by the historically older fan-out algorithm.

The *fan-both* algorithm was described as an intermediate parametrized algorithm partitioning both the subtasks $Tcol(j)$ and $Tsub(j)$. Processors are sending both the aggregate column updates and completed columns.

The *distributed multifrontal* algorithm partitions among the processors the tasks upon which the sequential multifrontal method is based:

1. Partial dense right-looking Cholesky factorization for the vertices of independent subtrees of the elimination tree.
2. Medium-grain or large-grain subtasks of the partial Cholesky factorizations of the dense matrices.

The first source of the parallelism is probably the most natural. Its theoretical justification is given by the Lemma 1.2. Independent branches of the elimination tree can be eliminated independently. Towards the root, number of independent tasks corresponding to these branches decreases. Then tasks corresponding to partial updates of the right-looking pseudo-code near to the root can be partitioned among the processors. Combining these two principles we obtain the core of the distributed-memory (but also of the shared-memory) multifrontal method. All the parallel right-looking implementations of the Cholesky factorization are essentially based on the same principles.

Large-grain parallelism present in the contemporary implementations is usually very transparent. We do not care very much about the mapping relation column–processor. The actual architecture provides hints how to solve this problem. Using for instance hypercube architectures, the subtree-subcube mapping is very natural and we can speak about this type of parallelism.

Shared-memory vector multiprocessors with a limited number of processing units represent a similar case. The natural way is to map rather large subtrees of the elimination tree to the processor. The drawback in this situation can be the memory management, since we need more working space than if purely scalar computations are considered.

So far we have concentrated on the issues related to the *numerical factorization.* We left out the issues of the symbolic decomposition, initial ordering and making other symbolic manipulations in parallel. Although in case of scalar computations the numerical phase timing is usually dominant, in the parallel environment it is not always so. The following example shows the proportion of the time spent in the symbolic and numeric phases of the sparse Cholesky factorization. It is taken from [31].

Example 4.1 Comparison of ordering and numeric factorization time for the matrix coming from structural mechanics. Dimension $n =$ 217918, number of nonzeros in A is 5.926.567, number of nonzeros in L is 55.187.841. Ordering time was 38s, right-looking factorization time was 200s.

Optimizing parallel performance of the symbolic manipulations (including matrix ordering, computation of row and column structures, tree manipulations, ...) is an important challenge. For the overview of the classical techniques see [23].

Number of parallel steps to compute the Cholesky decomposition is determined by the height of the elimination tree. But, while in the one processor case we preferred the unbalanced form of the elimination tree (see Figure 1.7), the situation is different now. Unbalanced tree induces more parallel steps. Therefore, for the parallel elimination, the balanced alternative seems to be better. On the other side, the cumulative size of working space is in this case higher.

Also the problem of the renumbering of the elimination tree is casted into another light in the parallel case. The level-by-level renumbering and balanced children sequences of the elimination tree are the objects of further research.

Even in the "scalar" case, all the implementations are strongly connected with computer architectures. It is only natural that, in the parallel environment, where some features of the computing facilities (e.g. , communication) provide even more variations, this dependence is also more profound.

Naive hopes that with more processors we could avoid in some extent the difficulties faced in scalar Cholesky decomposition came to an disappointment. We are still trying to find better algorithmic alternatives which make both the scalar and parallel computations more effective.

5 Concluding remarks

The world of numerical linear algebra is developing very fast. We have tried to show that many problems which are considered by the numerical analyst working in this area are very complicated and still unresolved despite the fact, that most of it has been formulated a few decades or even one or two hundred years ago.

Among the current trends in this field we want to point out the strong movement towards the *justification of the numerical algorithms.* A method or algorithm programmed to a code should not only give some output, but it should guarantee an exactly defined relation of the computed approximation to the unknown true solution or warn the user about the possible incorrectness of the result. Our intuition must be checked carefully by formal proofs or at least by developing theories offering a deep insight into a problem. Attempts to solve problems without such insight may fail completely. Another trend can be characterized by the proclamation: *There is no simple general solution to the difficult tasks as, e.g., solving large sparse linear systems. Considering parallel computers, things are even getting worse.* A combination of different techniques is always necessary, most of which use very abstract tools (as the graph theory etc.) to achieve very practical goals. There is where the way from theory to practice is very short.

References

[1] Agrawal, A. - Klein, P. - Ravi, R.: Cutting down on fill using nested dissection: provably good elimination orderings, in: George, A. - Gilbert, J.R. - Liu, J.W.H., eds. : Graph Theory and Sparse Matrix Computation, Springer, 1993, 31–55.

[2] Aho, A.V. - Hopcroft, J.E. - Ullman, J.D.: Data Structures and Algorithms, Addison - Wesley, Reading, MA, 1983.

[3] Ashcraft, C.: A vector implementation of the multifrontal method for large sparse, symmetric positive definite linear systems, Tech. Report ETA-TR-51, Engineering Technology Application Division, Boeing Computer Services, Seattle, Washington, 1987.

[4] Ashcraft, C.: The fan-both family of column-based distributed Cholesky factorization algorithms, in: George, A., Gilbert, J.R., Liu, J.W.H., eds. : Graph Theory and Sparse Matrix Computation, Springer, 1993, 159–190.

[5] Ashcraft, C. - Grimes, R.: The influence of relaxed supernode partitions on the multifrontal method, ACM Trans. Math. Software, 15 (1989), 291–309.

[6] Ashcraft, C. - Eisenstat, S. - Liu, J.W.H.: A fan-in algorithm for distributed sparse numerical factorization, SIAM J. Sci. Stat. Comput., 11 (1990), 593–599.

[7] Axelsson, O.: Iterative Solution Methods. Cambridge University Press, Cambridge, 1994.

[8] Barrett, R. - et al.: *Templates for the solution of linear systems: Building blocks for iterative methods.* SIAM, Philadelphia, 1994

[9] Browne, J. - Dongarra, J. - Karp, A. - Kennedy, K. - Kuck, D.: 1988 Gordon Bell Prize, IEEE Software, 6 (1989), 78–85.

[10] Chandra, R.: Conjugate gradient methods for partial differential equations, PhD. Thesis, Yale University, 1978.

[11] Drkošová, J - Greenbaum, A. - Rozložník, M. - Strakoš, Z.: *Numerical stability of the GMRES method.* Submitted to BIT, 1994.

[12] Duff, I.S. - Reid, J.: The multifrontal solution of indefinite sparse symmetric linear equations, ACM Trans. Math. Software, 9 (1983), 302–325.

[13] Eisenstat, S.C. - Schultz, M.H. - Sherman, A.H.: Software for sparse Gaussian elimination with limited core memory, in: Duff, I.S. - Stewart, G.W. eds., Sparse Matrix Proceedings, SIAM, Philadelphia, 1979, 135–153.

[14] Freund, R. - Golub, G. - Nachtigal, N: *Iterative solution of linear systems.* Act. Numerica 1, 1992, pp 1-44.

[15] Gallivan, K.A. - Plemmons, R.J. - Sameh, A.H.: Parallel algorithms for dense linear algebra computations, SIAM Review, 32 (1990), 54-135.

[16] George, A. - Heath, M. - Liu, J.W.H.: Solution of sparse positive definite systems on a shared-memory multiprocessor, Internat. J. Parallel Programming, 15 (1986), 309–325.

[17] George, A. - Heath, M. - Liu, J.W.H.: Sparse Cholesky factorization on a local-memory multiprocessor, SIAM J. Sci. Stat. Comput., 9 (1988), 327–340.

[18] George, A. - Liu, J.W.H.: Computer Solution of Large Sparse Positive Definite Systems, Prentice-Hall, Englewood Cliffs, N.J., 1981.

[19] Gilbert, J. - Schreiber, R.: Highly parallel sparse Cholesky factorization, Tech . Report CSL-90-7, Xerox Palo Alto Research Center, 1990.

[20] Greenbaum, A. - Strakoš, Z.: *Matrices that generate the same Krylov residual spaces.* In: Recent Advances in Iterative Methods, G. Golub et.al. eds., Springer-Verlag, New York, 1994.

[21] Greenbaum, A. - Pták, V. - Strakoš, Z.: *Any nonincreasing convergence curve is possible for GMRES* (in preparation)

[22] Hageman, L. - Young, D.: *Applied iterative methods.* Academic Press, New York, 1981

[23] Heath, M.Y. - Ng, E. - Peyton, B.W.: Parallel algorithms for sparse linear systems, in: Gallivan, K.A. - Heath, M.T. - Ng, E. - Ortega, J.M. - Peyton, B.W. - Plemmons, R.J. - Romine, C.H. - Sameh, A.H. - Voigt, R.G.: Parallel Algorithms for Matrix Computations, SIAM, Philadelphia, 1990, 83–124.

[24] Higham, N.J. - Knight, P.A.: Componentwise error analysis for stationary iterative methods, in preparation.

[25] Liu, J.W.H.: The role of elimination trees in sparse factorization, SIAM J. Matrix Anal. Appl. 11 (1990), 134-172.

[26] Lewis, J. - Simon, H.: The impact of hardware gather/scatter on sparse Gaussian elimination, SIAM J. Sci. Stat. Comput., 9 (1988), 304–311.

[27] Liu, J.W.H.: A compact row storage scheme for Cholesky factors using elimination trees, ACM Trans. Math. Software, 12 (1986), 127–148.

[28] Liu, J.W.H.: Computational models and task scheduling for parallel sparse Cholesky factorization, Parallel Computing, 3 (1986), 327–342.

[29] Liu, J.W.H.: Equivalent sparse matrix reordering by elimination tree rotations, SIAM J. Sci. Stat. Comput., 9 (1988), 424–444.

[30] Liu, J.W.H.: Advances in direct sparse methods, manuscript, 1991.

[31] Liu, J.W.H.: Some practical aspects of elimination trees in sparse matrix factorization, presented at the IBM Europe Institute, 1990.

[32] Ng, E.G. - Peyton, B.W.: Block sparse Cholesky algorithms on advanced uniprocessor computers, SIAM J. Sci. Comput., 14 (1993), 1034–1056.

[33] Parter, S.: The use of linear graphs in Gaussian elimination, SIAM Review, 3 (1961), 364–369.

[34] Pissanetzky, S.: Sparse Matrix Technology, Academic Press, 1984.

[35] Rothberg, E. - Gupta, A.: Efficient sparse matrix factorization on high-performance workstations - exploiting the memory hierarchy, ACM Trans. Math. Software, 17 (1991), 313–334.

[36] Rozložník, M. - Strakoš, Z.: *Variants of the residual minimizing Krylov space methods.* Submitted to BIT, 1994

[37] Stewart, G.W. - Sun, J.: *Matrix perturbation theory.* Academic Press, Boston, 1990

[38] Trefethen, N.: *The definition of numerical analysis.* Dept. of comp. Sc., Cornell Univ., 1991

[39] Varga, R.: *Matrix iterative analysis.* Prentice-Hall Inc., NJ, 1961

[40] Van der Vorst, H.: Lecture notes on iterative methods, 1992, manuscript.

[41] Young, D.: *Iterative solutions of large linear systems.* Academic Press, New York, 1971