Chapter 5
Sparse Cholesky Solver: The Factorization Phase

The adoption of Cholesky's method owes not a little to the publicity given to it shortly after the end of World War II by British mathematicians and computer pioneers, including Alan Turing, Leslie Fox, Jim Wilkinson, and especially John Todd – Benzi (2017).

Achieving high performance for sparse direct solvers in general, and sparse Cholesky factorization, in particular, is a very well researched topic – Rennich et al. (2016)

Having considered the symbolic phase of a sparse Cholesky solver in the previous chapter, the focus of this chapter is the subsequent numerical factorization phase. If A is a symmetric positive definite (SPD) matrix, then it is factorizable (strongly regular) and (in exact arithmetic) its Cholesky factorization $A = LL^T$ exists. LDLT factorizations of general symmetric indefinite matrices are considered in Chapter 7.

5.1 Dense Cholesky Factorizations

Because efficient implementations of sparse Cholesky factorizations rely heavily on exploiting dense blocks, we first consider algorithms for the Cholesky factorization of dense matrices that can be applied to such blocks. Algorithm 5.1 is a basic left-looking algorithm. It is an in-place algorithm because L can overwrite the lower triangular part of A (thus reducing memory requirements if A is no longer required).

Writing A in the block form (1.2), the computation can be reorganized to give Algorithm 5.2. This allows the exploitation of Level 3 BLAS for the computationally intensive components (dense matrix-matrix multiplies and dense triangular solves). Here A has nb block columns, which are referred to as **panels**. Step 6 can be performed using Algorithm 5.1.

Algorithms 5.1 and 5.2 are left-looking. This means that the updates are not applied immediately. Instead, all updates from previous (block) columns are applied together to the current (block) column before it is factorized. In a right-looking

J. Scott, M. Tůma, *Algorithms for Sparse Linear Systems*, Nečas Center Series,
https://doi.org/10.1007/978-3-031-25820-6_5

ALGORITHM 5.1 In-place dense left-looking Cholesky factorization
Input: Dense SPD matrix A.
Output: Factor L such that $A = LL^T$.

1: **for** $j = 1 : n$ **do**
2: $L_{j:n,j} = A_{j:n,j}$ ▷ Only the lower triangular part of A is required
3: **for** $k = 1 : j-1$ **do**
4: $L_{j:n,j} = L_{j:n,j} - L_{j:n,k}\, l_{jk}$ ▷ Update column j using previous columns
5: **end for**
6: $l_{jj} = (l_{jj})^{1/2}$ ▷ Overwrite the diagonal entry with its square root
7: $L_{j+1:n,j} = L_{j+1:n,j} / l_{jj}$ ▷ Scale off-diagonal entries in column j
8: **end for**

ALGORITHM 5.2 In-place dense left-looking panel Cholesky factorization
Input: Dense SPD matrix A in the form (1.2) with nb panels.
Output: Factor L such that $A = LL^T$.

1: **for** $jb = 1 : nb$ **do**
2: $L_{jb:nb,jb} = A_{jb:nb,jb}$
3: **for** $kb = 1 : jb-1$ **do**
4: $L_{jb:nb,jb} = L_{jb:nb,jb} - L_{jb:nb,kb}\, L_{jb,kb}^T$ ▷ Update block column jb
5: **end for**
6: Compute in-place factorization of $L_{jb,jb}$ ▷ Overwrite $L_{jb,jb}$ with its Cholesky factor
7: $L_{jb+1:nb,jb} = L_{jb+1:nb,jb}\, L_{jb,jb}^{-T}$ ▷ Dense triangular solve
8: **end for**

approach (Algorithm 5.3), outer product updates are applied to the part of the matrix that has not yet been factored as they are generated.

The large panel updates can be split into operations involving only blocks. This is shown in Algorithm 5.4 for the right-looking approach.

The panel and block descriptions of the factorization enable efficient parallelization. The three main block operations, which are called tasks, are **factorize**(jb), **solve**(ib, jb), and **update**(ib, jb, kb). There are the following dependencies between the tasks.

factorize(jb) depends on **update**(jb, kb, jb) for all $kb = 1, \ldots, jb-1$.
solve(ib, jb) depends on **update**(ib, kb, jb) for all $kb = 1, \ldots, jb-1$, and **factorize**(jb).
update(ib, jb, kb) depends on **solve**(ib, kb), **solve**(jb, kb).

ALGORITHM 5.3 In-place dense right-looking panel Cholesky factorization
Input: Dense SPD matrix A in the form (1.2) with nb panels.
Output: Factor L such that $A = LL^T$.

for $jb = 1 : nb$ **do**
  $L_{jb:nb,jb} = A_{jb:nb,jb}$
end for
for $jb = 1 : nb$ **do**
  Compute in-place factorization of $L_{jb,jb}$ ▷ Overwrite $L_{jb,jb}$ with its Cholesky factor
  $L_{jb+1:nb,jb} = L_{jb+1:nb,jb}\, L_{jb,jb}^{-T}$ ▷ Dense triangular solve
  for $kb = jb + 1 : nb$ **do**
    $L_{kb:nb,kb} = L_{kb:nb,kb} - L_{kb:nb,jb}\, L_{kb,jb}^{T}$
  end for
end for

ALGORITHM 5.4 In-place dense right-looking block Cholesky factorization
Input: Dense SPD matrix A in the form (1.2) with $nb \times nb$ blocks.
Output: Factor L such that $A = LL^T$.

for $jb = 1 : nb$ **do**
  $L_{jb:nb,jb} = A_{jb:nb,jb}$
end for
for $jb = 1 : nb$ **do**
  Compute in-place factorization of $L_{jb,jb}$ ▷ Task **factorize**(jb)
  for $ib = jb + 1 : nb$ **do**
    $L_{ib,jb} = L_{ib,jb}\, L_{jb,jb}^{-T}$ ▷ Task **solve**(ib, jb)
    for $kb = jb + 1 : ib$ **do**
      $L_{ib,kb} = L_{ib,kb} - L_{ib,jb}\, L_{kb,jb}^{T}$ ▷ Task **update**(ib, jb, kb)
    end for
  end for
end for

A dependency graph can be used to schedule the tasks. Its vertices correspond to tasks and dependencies between tasks are represented as directed edges. The result is a directed acyclic graph (DAG). A task is ready for execution if and only if all tasks with incoming edges to it have completed. DAG-driven linear algebra uses either a static or dynamic schedule based on these graphs to implement the tasks in a parallel environment. In practice, it is not necessary to explicitly compute the

task DAG: it can be constructed on-the-fly taking into account the dependencies. The task DAG allows a lot of flexibility in the order in which tasks are carried out: the left- and right-looking approaches correspond to particular restricted orderings of the tasks.

5.2 Introduction to Sparse Cholesky Factorizations

There are several classes of algorithms that implement sparse Cholesky factorizations. Their major differences relate to how they schedule the computations. This affects the use of dense kernels, the amount of memory required during the factorization as well as the potential for parallel implementations. As in the dense case, the factorization is split into tasks that involve computations on and between dense submatrices and the precedence relations among them can be captured by a task graph.

We start by extending the dense Cholesky factorizations to the sparse case in a straightforward way. In practice, it is essential for efficiency to exploit the supervariables of A and the supernodes of L. Thus, while for simplicity of the descriptions and notation, we refer to rows and columns of A and L, these typically represent block rows and block columns and, as in the above discussion of the dense block factorization algorithm, the entries of A and L are then submatrices.

The entries of L satisfy the relationship

$$L_{j+1:n,j} = \left(A_{j+1:n,j} - \sum_{k=1}^{j-1} L_{j+1:n,k} l_{jk} \right) / l_{jj} \quad \text{with} \quad l_{jj} = \left(a_{jj} - \sum_{k=1}^{j-1} l_{jk}^2 \right)^{1/2},$$

and from this we deduce the following result.

Theorem 5.1 (Liu 1990) *The numerical values of the entries in column $j > k$ of L depend on the numerical values in column k of L if and only if $l_{jk} \neq 0$.*

The theoretical background of the previous chapter based on the elimination tree $\mathcal{T}$ enables the dependencies in Theorem 5.1 to be searched for efficiently. In particular, $\mathcal{T}$ allows the row (or column) counts of L to be computed and they can be used to allocate storage for L. It can also be used to find supernodes and the resulting (block) elimination tree can then be employed to determine the (block) column structure of L. In practice, it can be beneficial to split large supernodes into smaller panels to better conform to computer caches.

Algorithms 5.5 and 5.6 are simplified sparse left- and right-looking Cholesky factorization algorithms that are straightforward sparse variants of Algorithms 5.1 and 5.4, respectively (the latter with $nb = n$, that is, without considering blocks). Here, we assume that the sparsity pattern of L has already been determined in the symbolic phase and static storage formats based, for example, on compressed columns and/or rows are used.

ALGORITHM 5.5 Simplified sparse left-looking Cholesky factorization
Input: SPD matrix A and sparsity pattern $\mathcal{S}\{L\}$.
Output: Factor L such that $A = LL^T$.

```
1:  l_ij = a_ij for all (i, j) ∈ S{L}        ▷ Filled entries in L are initialised to zero
2:  for j = 1 : n do
3:      for k ∈ {k < j | l_jk ≠ 0} do
4:          for i ∈ {i ≥ j | l_ik ≠ 0} do
5:              l_ij = l_ij − l_ik l_jk
6:          end for
7:      end for
8:      l_jj = (l_jj)^{1/2}
9:      for i ∈ {i > j | l_ij ≠ 0} do
10:         l_ij = l_ij / l_jj
11:     end for
12: end for
```

ALGORITHM 5.6 Simplified sparse right-looking Cholesky factorization
Input: SPD matrix A and sparsity pattern $\mathcal{S}\{L\}$.
Output: Factor L such that $A = LL^T$.

```
1:  l_ij = a_ij for all (i, j) ∈ S{L}        ▷ Filled entries in L are initialised to zero
2:  for j = 1 : n do
3:      l_jj = (l_jj)^{1/2}
4:      for i ∈ {i > j | l_ij ≠ 0} do
5:          l_ij = l_ij / l_jj
6:      end for
7:      for k ∈ {k > j | l_kj ≠ 0} do
8:          for i ∈ {i ≥ k | l_ij ≠ 0} do
9:              l_ik = l_ik − l_ij l_kj
10:         end for
11:     end for
12: end for
```

An alternative for sparse matrices held in row-wise format is to compute L one row at a time. This is sometimes called an **up-looking** factorization because rows 1 to $i-1$ are employed to compute row i ($i > 1$). The approach is asymptotically optimal in the work performed and for highly sparse matrices it is potentially extremely efficient because the entries of A are used in the natural order in which they are stored. However, it is difficult to incorporate high level BLAS.

The following relation holds for the i-th row of L

$$L_{i,1:i-1}^T = L_{1:i-1,1:i-1}^{-1} A_{1:i-1,i} \quad \text{with} \quad l_{ii}^2 = a_{ii} - L_{i,1:i-1} L_{i,1:i-1}^T.$$

The application of $L_{1:i-1,1:i-1}^{-1}$ can be implemented by solving the triangular system

$$L_{1:i-1,1:i-1} y = A_{1:i-1,i},$$

and setting $L_{i,1:i-1}^T = y$. The following result can be used to determine the sparsity pattern of y.

Theorem 5.2 (Gilbert 1994) *Consider a sparse lower triangular matrix L and the DAG $\mathcal{G}(L^T)$ with vertex set $\{1, 2, \ldots, n\}$ and edge set $\{(j \longrightarrow i) \mid l_{ij} \neq 0\}$. The sparsity pattern $\mathcal{S}\{y\}$ of the solution y of the system $Ly = b$ is the set of all vertices reachable in $\mathcal{G}(L^T)$ from $\mathcal{S}\{b\}$.*

Proof From Algorithm 3.4 and assuming the non-cancellation assumption, we see that (a) if $b_i \neq 0$, then $y_i \neq 0$ and (b) if for some $j < i$, $y_j \neq 0$ and $l_{ij} \neq 0$, then $y_i \neq 0$. These two conditions can be expressed as a graph transversal problem in $\mathcal{G}(L^T)$. (a) adds all vertices in $\mathcal{S}\{b\}$ to the set of visited vertices and (b) states that if vertex j has been visited, then all its neighbours in $\mathcal{G}(L^T)$ are added to the set of visited vertices. Thus $\mathcal{S}\{y\} = \mathcal{R}each(\mathcal{S}\{b\}) \cup \mathcal{S}\{b\}$. □

Figure 5.1 illustrates the sparsity patterns of a lower triangular matrix L and vector b together with $\mathcal{G}(L^T)$. The vertices that are reachable from $\mathcal{S}\{b\} = \{2, 4\}$ are 5 and 6 and thus $\mathcal{S}\{y\} = \{2, 4, 5, 6\}$.

Algorithm 5.7 outlines a sparse row Cholesky factorization that is based on the repeated solution of triangular linear systems. Theorem 5.2 can be used to determine the sparsity pattern of row i at Step 3, that is, by finding all the vertices that are reachable in $\mathcal{G}(L_{1:j-1,1:j-1}^T)$ from the set $\{i \mid a_{ij} \neq 0,\ i < j\}$. A depth-first search

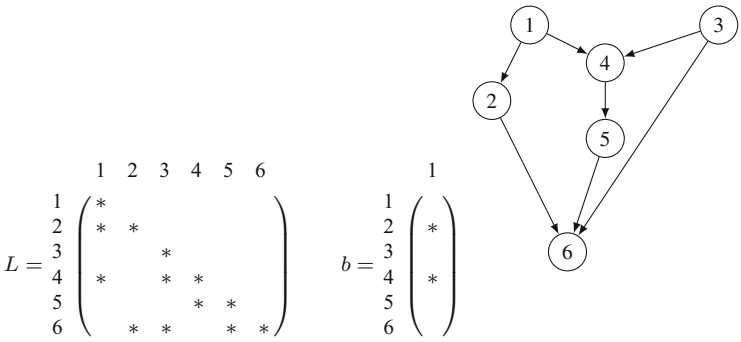

Figure 5.1 An example to illustrate L, b and $\mathcal{G}(L^T)$.

ALGORITHM 5.7 Sparse up-looking Cholesky factorization
Input: SPD matrix A.
Output: Factor L such that $A = LL^T$.

1: $l_{11} = (a_{11})^{1/2}$
2: **for** $i = 2 : n$ **do**
3: Find $\mathcal{S}\{L_{i,1:i-1}\}$ ▷ Sparsity pattern of row i
4: $L^T_{i,1:i-1} = L^{-1}_{1:i-1,1:i-1} A_{1:i-1,i}$ ▷ Sparse triangular solve
5: $l_{ii} = a_{ii} - L_{i,1:i-1} L^T_{i,1:i-1}$
6: $l_{ii} = (l_{ii})^{1/2}$
7: **end for**

of $\mathcal{G}(L^T_{1:j-1,1:j-1})$ determines the vertices in the row sparsity patterns in topological order, and performing the numerical solves in that order correctly preserves the numerical dependencies. Alternatively, because nonzeros of $L_{i,1:i-1}$ correspond to the vertices in the i-th row subtree $\mathcal{T}_r(i)$ that are not equal to i, another option is to find the row subtrees using $\mathcal{T}(A)$.

5.3 Supernodal Sparse Cholesky Factorizations

The simplified schemes form the basis of sophisticated supernodal algorithms that are designed to be efficient in parallel computational environments. Consider the right-looking variant and recall that a supernode consists of one or more consecutive columns of L with the same sparsity pattern. These nonzeros are stored as a dense trapezoidal matrix (only the lower triangular part of the block on the diagonal needs to be stored and the rows of zeros in the columns of the supernode are not held). This is termed a **nodal matrix** (see Figure 5.2).

Once a supernode is ready to be factorized, a dense Cholesky factorization of the block on the diagonal of the nodal matrix is performed (one of the approaches of Section 5.1 can be used). Then a triangular solve is performed with the computed factor and the rectangular part of the nodal matrix. The next step is to iterate over ancestors of the supernode in the assembly tree. For each parent, the rows of the current supernode corresponding to the parent's columns are identified, and then the outer product of those rows and the part of the supernode below those columns formed (update operations). The resulting matrix can be held in a temporary buffer. The rows and columns of this buffer are matched against indices of the ancestors and are added to them in a sparse scatter operation. For efficiency, the updates may use panels so that the temporary buffer remains in cache.

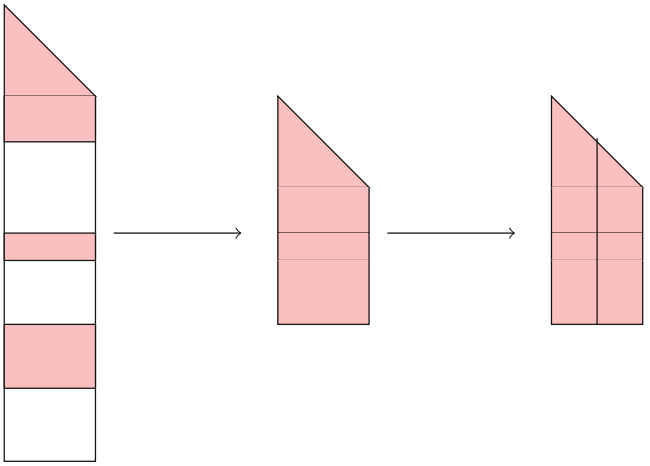

Figure 5.2 An illustration of a supernode (left), the corresponding nodal matrix (centre), and the nodal matrix with two panels (right). The shaded lower triangular part of the block on the diagonal and the shaded block rows are treated as dense.

5.3.1 DAG-Based Approach

The DAG-based approach can also be extended to the sparse case. Each nodal matrix is subdivided into blocks. The factorization is split into tasks in which a single block is revised. The key difference compared to the dense case is that it is necessary to distinguish between two types of update operations: **update_internal** performs the update between blocks in the same nodal matrix and **update_between** performs the update when the blocks belong to different nodal matrices. Thus the sparse Cholesky factorization is split into the following tasks; the first two are illustrated in Figure 5.3. In this example, the nodal matrix has two block columns that do not contain the same number of columns.

factorize_block(L_{diag}) Computes the dense Cholesky factor L_{diag} of the block on the diagonal (leftmost plot). If the block is trapezoidal, the factorization is followed by a triangular solve of its rectangular part $L_{rect} = L_{rect}L_{diag}^{-T}$ (centre plot).

solve_block(L_{dest}) Performs a triangular solve of an off-diagonal block L_{dest} of the form $L_{dest} = L_{dest}L_{diag}^{-T}$ (rightmost plot).

update_internal(L_{dest} , L_r, L_c) Performs the update $L_{dest} = L_{dest} - L_r L_c^T$, where L_{dest}, L_r and L_c belong to the same nodal matrix.

update_between(L_{dest} , L_r, L_c) Performs the update $L_{dest} = L_{dest} - L_r L_c^T$, where L_r and L_c belong to the same nodal matrix and L_{dest} belongs to a different nodal matrix.

Again, the tasks are partially ordered and a task DAG is used to capture the dependencies. For example, the updating of a block of a nodal matrix from a block

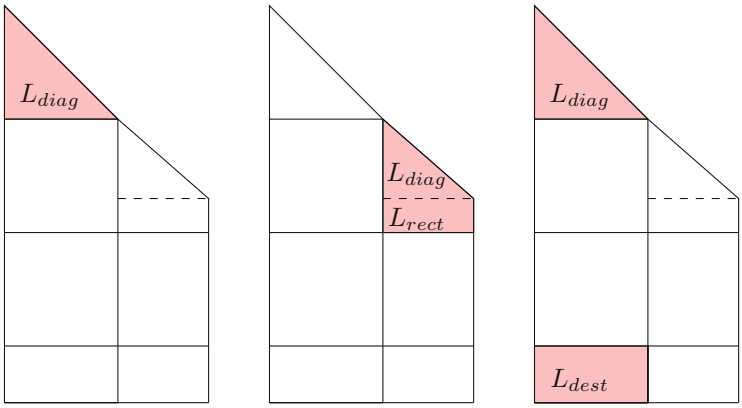

Figure 5.3 An illustration of a blocked nodal matrix with two block columns. The first block on the diagonal is triangular and the second one is trapezoidal. The task **factorize_block** is illustrated on the left and in the centre; the task **solve_block** is illustrated on the right.

column of L that is associated with a descendant of the supernode has to wait until all the relevant rows of the block column are available. At each stage of the factorization, tasks will be executing (in parallel) while others are held (in a stack or pool of tasks) ready for execution.

5.4 Multifrontal Method

The **multifrontal** method is an alternative way to compute a sparse Cholesky factorization. To discuss this popular approach, we use the following result that determines which rows and columns influence particular Schur complements using the terminology of the elimination tree.

Theorem 5.3 (Liu 1990) *Let A be SPD and let $\mathcal{T}$ be its elimination tree. The numerical values of entries in column k of the Cholesky factor L of A only affect the numerical values of entries in column i of L for $i \in anc_{\mathcal{T}}\{k\}$ $(1 \leq k < i \leq n-1)$.*

Proof From (4.1), setting $S^{(1)} = A$, for $k \geq 2$ the $(n-k+1) \times (n-k+1)$ Schur complement $S^{(k)}$ can be expressed as

$$S^{(k)} = S^{(k-1)}_{k:n,k:n} - \begin{pmatrix} l_{k,k-1} \\ \vdots \\ l_{n,k-1} \end{pmatrix} \begin{pmatrix} l_{k,k-1} & \dots & l_{n,k-1} \end{pmatrix} = S^{(k-1)}_{k:n,k:n} - L_{k:n,k-1} L^T_{k:n,k-1}. \tag{5.1}$$

Theorem 4.2 implies that all nonzero off-diagonal entries l_{ik} in column k of L explicitly used in the update (5.1) are such that $i \in anc_{\mathcal{T}}\{k\}$. Considering the

Cholesky factorization as a sequence of Schur complement updates, only columns i with $i \in anc_{\mathcal{T}}\{k\}$ can be influenced numerically by the Schur complement update in the k-th step of the factorization, and the result follows. □

The computation of subsequent Schur complements by adding individual updates as in (5.1) is straightforward; the multifrontal method employs further modifications and enhancements of this basic concept. First, because the vertices of $\mathcal{T}$ are topologically ordered, the order in which the updates are applied progresses up the tree from the leaf vertices to the root vertex. This allows the computation of $S^{(k)}$ to be rewritten as

$$S^{(k)} = A_{k:n,k:n} - \sum_{j \in \mathcal{T}(k) \setminus \{k\}} L_{k:n,j} L_{k:n,j}^T,$$

emphasizing the role of $\mathcal{T}$. In place of Schur complements, the multifrontal method uses frontal matrices connected to subtrees of $\mathcal{T}$. Assume $k, k_1, \ldots, k_r$ are the row indices of the nonzeros in column k of L. The **frontal matrix** F_k of the k-th subtree $\mathcal{T}(k)$ of $\mathcal{T}$ is the dense $(r+1) \times (r+1)$ matrix defined by

$$F_k = \begin{pmatrix} a_{kk} & a_{kk_1} & \ldots & a_{kk_r} \\ a_{k_1k} & 0 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{k_rk} & 0 & \ldots & 0 \end{pmatrix} - \sum_{j \in \mathcal{T}(k) \setminus \{k\}} \begin{pmatrix} l_{kj} \\ l_{k_1j} \\ \vdots \\ l_{k_rj} \end{pmatrix} \begin{pmatrix} l_{kj} & l_{k_1j} & \ldots & l_{k_rj} \end{pmatrix}. \tag{5.2}$$

One step of the Cholesky factorization of F_k can be written as

$$F_k = \begin{pmatrix} l_{kk} & 0 & \ldots & 0 \\ l_{k_1k} & & & \\ \vdots & & I & \\ l_{k_rk} & & & \end{pmatrix} \begin{pmatrix} 1 & & & \\ & & & \\ & & V_k & \\ & & & \end{pmatrix} \begin{pmatrix} l_{kk} & l_{k_1k} & \ldots & l_{k_rk} \\ 0 & & & \\ \vdots & & I & \\ 0 & & & \end{pmatrix} \tag{5.3}$$

$$= \begin{pmatrix} l_{kk} \\ l_{k_1k} \\ \vdots \\ l_{k_rk} \end{pmatrix} \begin{pmatrix} l_{kk} & l_{k_1k} & \ldots & l_{k_rk} \end{pmatrix} + \begin{pmatrix} 0 & & & \\ & & & \\ & & V_k & \\ & & & \end{pmatrix}, \tag{5.4}$$

where V_k is termed a **generated element** (it is also sometimes called an **update matrix** or a **contribution block**). The name "generated element" is because the multifrontal method has its origins in the simpler **frontal method**, which uses a single frontal matrix. The frontal method was originally proposed for problems arising in finite element problems to avoid the need to explicitly construct the system matrix A; it was later generalized to non-element problems. It works with a single frontal matrix and has less scope for parallelisation compared to the multifrontal method; it is no longer widely used.

Equating the last r rows and columns in (5.2) and (5.4) yields

$$V_k = -\sum_{j \in \mathcal{T}(k)} \begin{pmatrix} l_{k_1 j} \\ \vdots \\ l_{k_r j} \end{pmatrix} \begin{pmatrix} l_{k_1 j} & \dots & l_{k_r j} \end{pmatrix}. \tag{5.5}$$

Assume that c_j $(j = 1, \dots, s)$ are the children of k in $\mathcal{T}$. The set $\mathcal{T}(k) \setminus \{k\}$ is the union of disjoint sets of vertices in the subtrees $\mathcal{T}(c_j)$. Each of these subtrees is represented in the overall update by the generated element (5.5). Thus, F_k can be written in an recursive form using the generated elements of the children of k as follows

$$F_k = \begin{pmatrix} a_{kk} & a_{kk_1} & \dots & a_{kk_r} \\ a_{k_1 k} & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{k_r k} & 0 & \dots & 0 \end{pmatrix} \boxplus V_{c_1} \boxplus \dots \boxplus V_{c_s}. \tag{5.6}$$

Here, the operation $\boxplus$ denotes the addition of matrices that have row and column indices belonging to subsets of the same set of indices (in this case, $k, k_1, \dots, k_r$); entries that have the same row and column indices are summed. This is referred to as the **extend-add operator**.

Adding a row and column of A and the generated elements into a frontal matrix is called the **assembly**. A variable is **fully summed** if it is not involved in any rows and columns of A that have still to be assembled or in a generated element. Once a variable is fully summed, it can be eliminated. A key feature of the multifrontal method is that the frontal matrices and the generated elements are compressed and stored without zero rows and columns as small dense matrices. Integer arrays are used to maintain a mapping of the local contiguous indices of the frontal matrices to the global indices of A and its factors. Symmetry allows only the lower triangular part of these matrices to be held. Algorithm 5.8 outlines the basic multifrontal method.

ALGORITHM 5.8 Basic multifrontal Cholesky factorization

Input: SPD matrix A and its elimination tree.
Output: Factor L such that $A = LL^T$.

1: **for** $k = 1 : n$ **do**
2: Assemble the frontal matrix F_k using (5.6) ▷ Only the lower triangle is needed
3: Perform a partial Cholesky factorization of F_k using (5.3) to obtain column k of L and the generated element V_k
4: **end for**

ALGORITHM 5.9 Multifrontal Cholesky factorization using the assembly tree
Input: SPD matrix A and its assembly tree.
Output: Factor L such that $A = LL^T$.

1: $nelim = 0$ ▷ $nelim$ is the number of eliminations performed
2: **for** $kb = 1 : nsup$ **do** ▷ $nsup$ is the number of supernodes
3: Assemble the frontal matrix F_{kb}; let l be the number of fully summed variables in F_{kb}
4: Perform a block partial Cholesky factorization of F_{kb} to obtain columns $nelim + 1$ to $nelim + l$ of L and the generated element V_{kb}
5: $nelim = nelim + l$
6: **end for**

We have the following observation.

Observation 5.1 *Each generated element V_k is used only once to contribute to a frontal matrix $F_{parent(k)}$. Furthermore, the index list for the frontal matrix F_k is the set of row indices of the nonzeros in column k of the Cholesky factor L.*

In practical implementations, efficiency is improved by using the assembly tree (Section 4.6) because it allows more than one elimination to be performed at once. This is outlined in Algorithm 5.9. Here kb is used to index the frontal matrix on the kb-th step ($1 \le kb \le nsup$).

As an example, consider the matrix and its assembly tree given in Figure 4.10. The $nsup = 5$ supernodes are $\{1, 2\}, 3, 4, 5, \{6, 7, 8, 9\}$ and so variables 1 and 2 can be eliminated together on the first step. Assembling rows/columns 1 and 2 of the original matrix, the frontal matrix F_1 and generated element V_1 have the structure

$$F_1 = \begin{array}{c} \\ 1 \\ 2 \\ 8 \\ 9 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 8 & 9 \end{array} \\ \begin{pmatrix} * & * & * & * \\ * & * & * & * \\ * & * & & \\ * & * & & \end{pmatrix} \end{array}, \quad V_1 = \begin{array}{c} \\ 8 \\ 9 \end{array} \begin{array}{c} \begin{array}{cc} 8 & 9 \end{array} \\ \begin{pmatrix} f & f \\ f & f \end{pmatrix} \end{array},$$

where f denotes fill-in entries (only the lower triangular entries are stored in practice). Similarly,

$$F_2 = \begin{array}{c} \\ 3 \\ 4 \\ 8 \end{array} \begin{array}{c} \begin{array}{ccc} 3 & 4 & 8 \end{array} \\ \begin{pmatrix} * & * & * \\ * & * & * \\ * & * & * \end{pmatrix} \end{array}, \quad V_2 = \begin{array}{c} \\ 4 \\ 8 \end{array} \begin{array}{c} \begin{array}{cc} 4 & 8 \end{array} \\ \begin{pmatrix} * & * \\ * & * \end{pmatrix} \end{array}.$$

The frontal matrix F_3 and generated element V_3 are given by

$$F_3 = \begin{array}{c} \\ 4 \\ 7 \\ 8 \end{array} \begin{array}{c} \begin{array}{ccc} 4 & 7 & 8 \end{array} \\ \begin{pmatrix} * & * & * \\ * & * & \\ * & & * \end{pmatrix} \end{array} \leftrightarrow V_2, \quad V_3 = \begin{array}{c} \\ 7 \\ 8 \end{array} \begin{array}{c} \begin{array}{cc} 7 & 8 \end{array} \\ \begin{pmatrix} * & f \\ f & * \end{pmatrix} \end{array}.$$

Then

$$F_4 = \begin{array}{c} \\ 5 \\ 7 \\ 8 \end{array} \begin{array}{c} \begin{array}{ccc} 5 & 7 & 8 \end{array} \\ \begin{pmatrix} * & * & * \\ * & * & \\ * & & * \end{pmatrix} \end{array}, \quad V_4 = \begin{array}{c} \\ 7 \\ 8 \end{array} \begin{array}{c} \begin{array}{cc} 7 & 8 \end{array} \\ \begin{pmatrix} * & f \\ f & * \end{pmatrix} \end{array},$$

and, finally, with $kb = 5$ we have

$$F_5 = \begin{array}{c} \\ 6 \\ 7 \\ 8 \\ 9 \end{array} \begin{array}{c} \begin{array}{cccc} 6 & 7 & 8 & 9 \end{array} \\ \begin{pmatrix} * & & * & * \\ & * & & * \\ * & & * & \\ * & * & & * \end{pmatrix} \end{array} \leftrightarrow V_4 \leftrightarrow V_3 \leftrightarrow V_1.$$

An important implementation detail is how and where to store the generated elements. The partial factorization of F_{kb} at supernode kb can be performed once the partial factorizations at all the vertices belonging to the subtree of the assembly tree with root vertex kb are complete. If the vertices of the assembly tree are ordered using a depth-first search, the generated elements required at each stage are the most recently computed ones amongst those that have not yet been assembled. This makes it convenient to use a stack. This affects the order in which the variables are eliminated but in exact arithmetic, the results are identical.

Nevertheless, the memory demands of the multifrontal method can be very large. Not only is it dependent on the initial ordering of A but the ordering of the children of a vertex in the assembly tree can significantly affect the required stack size. Some implementations target limiting stack storage requirements. An attractive feature of the multifrontal method is that the generated elements can be held using auxiliary storage (in files on disk) to restrict the in-core memory requirements, allowing larger problems to be solved than would otherwise be possible.

5.5 Parallelism Within Sparse Cholesky Factorizations

Sparse Cholesky factorizations use supernodes and task graphs (the assembly tree for the multifrontal method) to control the computation. The number of rows and columns in a supernode typically increases away from the leaf vertices and towards

the root of the task graph because a supernode accumulates fill-in from its ancestors in the task graph. As a result, tasks that are relatively close to the root tend to have more work associated with them. On the other hand, the width of the task graph shrinks close to the root. In other words, a typical task graph for sparse matrix factorization tends to have a large number of small independent tasks close to the leaf vertices, but a small number of large tasks close to the root. An ideal parallelization strategy that would match the characteristics of the problem is as follows. Initially, assign the relatively plentiful independent tasks at or near the leaf vertices to parallel threads or processes. This is called **task** or **tree level** parallelism; it is influenced by the ordering of A. As tasks complete, other tasks become available and are scheduled similarly. This continues while there are enough independent tasks to keep all the threads or processes busy. When the number of available parallel tasks becomes too small, the only way to keep the latter busy is to assign more than one to a task. This is termed **node level** parallelism. The number of threads or processes working on individual tasks should increase as the number of parallel tasks decreases. Eventually, all threads or processes are available to work on the root task. The computation corresponding to the root task is equivalent to factoring a dense matrix of the size of the root supernode.

The multifrontal method is often the formulation of choice for highly parallel implementations of sparse matrix factorizations. This is because of its natural data locality (most of the work of the factorization is performed in the dense frontal matrices) and the ease of synchronization that it permits. In general, each supernode is updated by multiple other supernodes and it can potentially update many other supernodes during the course of the factorization. If implemented naively, all these updates may require excessive locking and synchronization in a shared-memory environment or generate excessive message-traffic in a distributed environment. In the multifrontal method, the updates are accumulated and channelled along the paths from the leaf vertices of the assembly tree to its root vertex. This gives a manageable structure to the potentially haphazard interaction among the tasks.

In Section 1.2.4, bit compatibility was discussed. While different orderings of the children of a vertex in the assembly tree do not affect the total number of floating-point operations that are performed in the multifrontal method, in finite-precision arithmetic changing the order of the assemblies into the frontal matrices can lead to slightly different results. Given that the number of children is typically small and that large matrices can be partitioned such that summations can be safely performed in parallel, the overhead in the multifrontal method of enforcing a defined order of the summation is relatively small. By contrast, in the supernodal approach, for each data block a number of matrices equal to the block dependencies are summed. Given the relatively large numbers (several thousand) for many nodes, an enforced order may be detrimental to efficiency.

5.6 Notes and References

Exploiting panels and blocks in both left- and right-looking Cholesky factorization algorithms is extremely important. The development of sparse supernodal factorizations for uniprocessors and multiprocessors in the 1990s is discussed by Ng & Peyton (1993a,b); Rothberg & Gupta (1993) presents an early comparison of various types of block Cholesky factorizations. PaStiX of Hénon et al. (2002) is a parallel left-looking supernodal solver that is primarily designed for positive definite systems. Rotkin & Toledo (2004) introduce a hybrid left-looking/right-looking algorithm and Rozin & Toledo (2005) show that no sparse numerical factorization is uniformly better than the others. An up-looking approach, which is fast in practice for very sparse matrices, is employed in the widely used CHOLMOD solver of Chen et al. (2008). The package `HSL_MA87` implements a sparse DAG-based Cholesky factorization for shared-memory architectures; further details of the approach can be found in Hogg et al. (2010).

The multifrontal algorithm has its origins in the simpler frontal method of Irons (1970), which was developed by the civil engineering community from the 1960s onwards to solve the linear systems that arise within finite element methods. At a time when the main memory of even the most powerful computers was extremely limited, the frontal method was heavily influenced by the need to minimize the memory requirements of the linear solver. It was initially designed for SPD banded linear systems and was subsequently extended to nonsymmetric problems by Hood (1976) and to the symmetric indefinite case by Reid (1981); Duff (1984) generalizes the approach to non-element problems. The frontal method proceeds by alternating the assembly of the finite elements into a single dense frontal matrix with the elimination and update of variables. Once variables have been eliminated they are no longer needed during the factorization and so they are removed from the frontal matrix and stored elsewhere (for example, not in main memory but on an external disk) until needed during the solve phase. This frees up space to accommodate the next element to be assembled. Because the frontal method does not use the assembly tree, the frontal matrix can be much larger than those in the multifrontal method, leading to higher operation counts but also allowing the use of BLAS with larger block sizes. Efficient implementations were developed up until the late 1990s. For example, by Duff & Scott (1996, 1999), who provide a package `MA62` for SPD problems in element form that employs a single array of length n, exploits Level 3 BLAS, and holds the computed factors on disk; a coarse-grained parallel version is also available, see Duff & Scott (1994) and Scott (2001).

The frontal method and the work of Speelpenning (1978) on the so-called generalized element method led to the development by Duff & Reid (1983) of the multifrontal method for solving general symmetric systems (including systems in element form). A detailed matrix-based explanation is given in Liu (1992). The method is implemented in some of the most important sparse direct solvers. The MUMPS (2022) package, which has been actively developed over many years, provides a state-of-the-art distributed memory general-purpose multifrontal solver

that uses shared-memory parallelism within each MPI process. Other important parallel multifrontal solvers are HSL_MA97 (Hogg & Scott, 2013b) and WSMP (2020), while the serial package MA57 of Duff (2004) (which superseded the original and perhaps most well-known multifrontal solver MA27 of Duff & Reid, (1983)) remains very popular. An attractive feature of HSL_MA97 is that it computes bit-compatible solutions. HSL_MA77 of Reid & Scott (2009) is designed to minimize memory requirements by allowing the factors and the multifrontal stack to be efficiently held outside of main memory (an option that is also offered by MUMPS). In common with earlier frontal solvers, HSL_MA77 allows the user to input the system matrix in element form (that is, A is not explicitly assembled for problems coming from finite element applications but is input one element at a time).

The use of GPUs is well-suited to a multifrontal or supernodal factorization because these approaches rely on regular block computations within dense submatrices. Implementing the multifrontal method (including for symmetric indefinite matrices) on GPU architectures is discussed in Hogg et al. (2016), while Lacoste et al. (2012) and Rennich et al. (2016) present GPU-accelerated supernodal factorizations. Discussion of the use of GPUs within direct solvers is included in the comprehensive survey of Davis et al. (2016).