Freie Universität Berlin
Department of Mathematics and Computer Science
Institute of Mathematics

Bachelor Thesis

Direct Solver for Large Sparse Linear Systems of Equations

Student: Haoyang Zhou
Matriculation Number: 5481976
Supervisor: Prof. Dr. Volker John
Second Reviewer: Dr. Alfonso Caiazzo

Berlin, 6 June 2023

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have acknowledged all the sources of information which have been used in the thesis. This thesis has not been submitted to any other University or Institution.

Date, Place	Haoyang Zhou

Abstract

Many application problems, especially in engineering and scientific computing, involve solving large sparse linear systems and sparse direct solvers are increasingly used for solving such problems. Iterative method is also used to solve large sparse linear systems. However, it can have difficulties converging for ill-conditioned problems, as small errors in the solution can amplify with each iteration. In such cases, sparse direct solvers may be preferred, as they aim to find an exact solution. Sparse direct solvers use factorization methods such as LU, Cholesky, or QR factorization to break down the matrix into simpler components, which can then be used to solve for the unknown vector.

Among the direct solvers, UMFPACK, MUMPS, and PARDISO are considered to be some of the most efficient and reliable solvers. This thesis aims to provide a study of these three solvers, focusing on their underlying algorithms. In Chapter 1 we focus on how large sparse matrices are generated, and in Chapter 2 we focus on how large sparse matrices are stored in the computer. Then we go through the direct method, from the overview to the specific algorithms of UMPFACK, MUMPS and PARDISO respectively, where we focus on the numerical factorization factorization. At the end, this thesis concludes with a comparison of these three direct solvers.

Contents

Chapter 1

Generation of Sparse Matrices

Many problems in applications, especially in engineering and scientific computing, involve solving large sparse linear systems. Specifically, most of these problems can be transformed into partial differential equations, and these partial differential equations can eventually be transformed by discretization into problems of solving large sparse matrices. We present here two methods for discretizing partial differential equations, the finite difference method and the finite element method, respectively, following [16]. We will see how sparse matrices are obtained from partial differential equations by these methods with some examples.

1.1 Finite Difference Method

Lemma 1.1.1 (Centered difference approximation of the second derivative [16]). *For any h a function u which is C^4 at the point x satifies:*

$$\frac{\mathrm{d}^2u(x)}{\mathrm{d}x^2} = \frac{u(x+h)-2u(x)+u(x-h)}{h^2} - \frac{h^2}{12}\frac{\mathrm{d}^4u(\xi)}{\mathrm{d}x^4} \tag{1.1}$$

where $\xi \in (x-h, x+h)$.

Proof. By Taylor's formula we obtain

$$u(x+h) = u(x) + h\frac{\mathrm{d}u}{\mathrm{d}x} + \frac{h^2}{2}\frac{\mathrm{d}^2u}{\mathrm{d}x^2} + \frac{h^3}{6}\frac{\mathrm{d}^3u}{\mathrm{d}x^3} + \frac{h^4}{24}\frac{\mathrm{d}^4u}{\mathrm{d}x^4}(\xi_+) \tag{1.2}$$

for $\xi_+ \in (x, x+h)$. Replace h with $-h$, the above equation (1.2) is transformed into

$$u(x-h) = u(x) - h\frac{\mathrm{d}u}{\mathrm{d}x} + \frac{h^2}{2}\frac{\mathrm{d}^2u}{\mathrm{d}x^2} - \frac{h^3}{6}\frac{\mathrm{d}^3u}{\mathrm{d}x^3} + \frac{h^4}{24}\frac{\mathrm{d}^4u}{\mathrm{d}x^4}(\xi_-) \tag{1.3}$$

for $\xi_- \in (x, x-h)$. After adding (1.2) and (1.3) and dividing through by h^2, the second derivative has the form

$$\frac{\mathrm{d}^2u(x)}{\mathrm{d}x^2} = \frac{u(x+h)-2u(x)+u(x-h)}{h^2} - \frac{h^2}{12}\frac{\mathrm{d}^4u}{\mathrm{d}x^4}(\xi_+ + \xi_-). \tag{1.4}$$

Applying the mean value theorem to the fourth order derivatives yields

$$\frac{\mathrm{d}^4u}{dx^4}(\xi_+ + \xi_-) = \frac{\mathrm{d}^4u(\xi)}{\mathrm{d}x^4} \tag{1.5}$$

where $\xi \in (\xi_-, \xi_+)$.

$\square$

Now we can consider the one-dimensional boundary value problem

$$-u''(x) = f(x) \text{ for } x \in (0,1) \tag{1.6}$$
$$u(0) = u(1) = 0. \tag{1.7}$$

The goal is to solve the problem on the interval $(0, 1)$. To do this, the interval can be discretized into $n+2$ points: $x_0, x_1, x_2, \ldots, x_{n+1}$. These points are equally spaced with a distance of $h = \dfrac{1}{n+1}$ between them. The points x_0 and x_{n+1} correspond to the boundary points 0 and 1, respectively.

Next the exact solution $u(x_i)$ at each point x_i will be approximated as a value u_i. The second derivative $u''(x_i)$ can be approximated using the centered difference approximation (1.1) to the equation (1.6) at the point x_i. The unknowns u_i, u_{i-1}, u_{i+1} are then related to each other such that

$$-u_{i-1} + 2u_i - u_{i+1} = h^2 f_i, \tag{1.8}$$

where f_i is defined as $f_i \equiv f(x_i)$. Thus the obtained matrix of the linear system is of the form

$$\begin{bmatrix} 2 & -1 & & & & \\ -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 2 & -1 \end{bmatrix}_{n\times n}.$$

It can be seen that the resulting matrix is a sparse matrix with zero elements in all positions except the subdiagonal, diagonal and superdiagonal. In fact, it is a feature of the finite difference method to obtain such a matrix with non-zero elements on only a few entries.

1.2 Finite Element Method

Consider this simple problem,

$$-\left(\frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2}\right) = f \quad \text{in } \Omega \tag{1.9}$$
$$u = 0 \quad \text{on } \Gamma, \tag{1.10}$$

where Ω is a bounded open domain in $\mathbb{R}^2$ and Γ its boundary.

Define

$$a(u,v) \equiv \int_\Omega \nabla v \cdot \nabla u \; dx = -\int_\Omega \left(\frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2}\right) v \; dx \tag{1.11}$$

$$(f,v) \equiv \int_\Omega fv \; dx. \tag{1.12}$$

With Green's formula we have the equivalence

$$-\left(\frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2}\right) = f \tag{1.13}$$

$$\Leftrightarrow \; -(\Delta u)v = fv \tag{1.14}$$

$$\Leftrightarrow \; -\int_\Omega (\Delta u) v \; dx = \int_\Omega fv \; dx \tag{1.15}$$

$$\Leftrightarrow \; -(\Delta u, v) = (f,v) \tag{1.16}$$

$$\Leftrightarrow a(u,v) = (f,v). \tag{1.17}$$

Here we use Green's formula in the last step.

We start with the function space $L^2(\Omega)$. The weak formulation involves selecting a subspace V of this function space, which consists of functions in $L^2(\Omega)$ that have zero values on the boundary Γ and possess square-integrable first-order derivatives. Then the weak formulation seeks to find a function $u \in V$ that satisfies

$$a(u,v) = (f,v) \quad \forall v \in V, \tag{1.18}$$

where the functions in the space have zero values on Γ.

To approximate the weak problem, the finite element method involves replacing the subspace V with a finite-dimensional space that consists of functions defined as low-degree polynomials on small pieces of the original domain. To be specific, we can approximate the original domain by taking the union of m triangles K_i, denoted as Ω', i.e.,

$$\Omega' = \cup_{i=1}^m K_i. \tag{1.19}$$

The finite-dimensional space can then be defined as the space of all functions that are piecewise linear and continuous on the polygonal region Ω'. An example is shown in Figure 1.1. In the context of our concrete problem here, it is also required for the functions to vanish on the boundary Γ, which means their values are zero on the boundary.

We want to find the basis $\{\phi_i\}_i$ of V', so that our original approximate problem can turn into finding a function $u \in V'$ that satisfies

$$a(u,\phi_i) = (f,\phi_i) \quad \forall i. \tag{1.20}$$

In addition, our desired approximate solution u can be rewritten in the basis $\{\phi_i\}$ as

$$u = \sum_{i=1}^n \xi_i \phi_i(x). \tag{1.21}$$

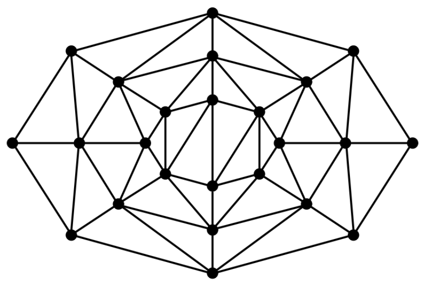


Figure 1.1: Finite element triangulation of a domain [16]

Consequently, we just need to solve the equations

$$\sum_{j=1}^{n} a(\phi_i, \phi_j)\xi_i = (f, \phi_i) \quad \forall i. \tag{1.22}$$

The equation forms a linear system of equations $Ax = b$ with

$$A = (a(\phi_i, \phi_j))_{ij}. \tag{1.23}$$

One of our desired basis $\{\phi_i\}_i$ of V' satisfies the following conditions:

$$\phi_j(x_i) = \begin{cases} 1, & \text{if } x_i = x_j, \\ 0, & \text{if } x_i \neq x_j, \end{cases} \tag{1.24}$$

where x_j, $j = 1, ..., n$ are the nodes of the triangulation.

Note that the matrix A is sparse with the basis, because the non-zero entries occur only when the basis functions ϕ_i and ϕ_j are associated with vertices that belong to the same triangle.

Chapter 2

Storage Schemes of Sparse Matrices

To conserve storage space for sparse matrices, it is desirable to store only the non-zero elements. In this section, we will keep following [16] to examine the common storage schemes for sparse matrices, which include the coordinate format, compressed sparse row format, modified sparse row format, and Ellpack-Itpack format. The first three formats can be utilized for general matrices, while the last one is more appropriate for matrices with a small number of non-zero elements per row. In the following discussion, the total number of non-zero elements is denoted as N_z, and the matrix size is $n \times n$.

2.1 Coordinate Format

The data structure of coordinate format consists of three arrays:

1. A real array AA of length N_z containing all the values of the nonzero elements of the matrix in any order,
2. an integer array JR of length N_z where JR[k] is the row index of AA[k] and
3. another integer array JC of length N_z where JC[k] is the column index of AA[k].

2.2 Compressed Sparse Row Format

Compressed sparse row format is more popular than the coordinate scheme. The data structure of the format consists also of three arrays:

1. A real array AA of length N_z containing all the values of the nonzero elements of the matrix row by row, from row 1 to n,
2. an integer array JA of length N_z where JR[k] is the column index of AA[k] and

3. another integer array IA of length $n+1$ where IA[k] is the pointer to the beginning of k-th row in the array AA for $k = \{1, ..., n\}$, and IA[$n+1$] containing the number IA[1]+N_z.

This format needs less memory compared to the coordinate format, because the third array is shorter than in the coordinate format. A variation of the format is to store the columns rather than rows. The corresponding schema is referred to as compressed sparse column format.

2.3 Modified Sparse Row Format

Modified sparse row format is another variation of compressed sparse row format. The data structure of the format consists of two arrays:

1. A real array AA containing the diagonal elements of the matrix in the first n positions in order, followed by the non-zero off-diagonal elements of the matrix arranged row by row, starting from position $n+2$, and
2. an integer array JR containing the pointer to the beginning of each row in AA after the diagonal element is removed and in the first $n+1$ position and JR[k] represents the column index of AA[k] starting at $k = n+2$.

The rationale for storing diagonal and off-diagonal elements separately is that the diagonal elements of many matrices are frequently accessed and are typically non-zero.

Example 2.3.1. Consider the matrix

$$\mathrm{A} = \begin{bmatrix} 4 & 0 & 0 & 8 \\ 0 & 6 & 3 & 7 \\ 5 & 0 & 1 & 0 \\ 0 & 2 & 0 & 9 \end{bmatrix}.$$

In coordinate format the matrix A can be stored as

$$\begin{aligned} \mathrm{AA} &= [3, 6, 1, 7, 2, 8, 5, 4, 9] \\ \mathrm{JR} &= [2, 2, 3, 2, 4, 1, 3, 1, 4] \\ \mathrm{JC} &= [3, 2, 3, 4, 2, 4, 1, 1, 4] \end{aligned}$$

For the same matrix, the compressed sparse format is

$$\begin{aligned} \mathrm{AA} &= [4, 8, 6, 3, 7, 5, 1, 2, 9] \\ \mathrm{JA} &= [1, 4, 2, 3, 4, 1, 3, 2, 4] \\ \mathrm{IA} &= [1, 3, 6, 8, 10] \end{aligned}$$

and the modified version is

$$\begin{aligned} \mathrm{AA} &= [4, 6, 1, 9, *, 8, 3, 7, 5, 2] \\ \mathrm{JR} &= [6, 7, 9, 10, 11, 4, 3, 4, 1, 2] \end{aligned}$$

where the star denotes an unused location.

2.4 Ellpack-Itpack Rormat

The assumption in this schema is that there are at most N_d nonzero elements per row, where N_d is small. The data structure consists of two rectangular arrays of dimension $n \times N_d$:

1. A rectangular array COEF of size containing the nonzero elements of A where each row of the array will correspond to a row of the matrix, and if a row has fewer nonzero elements than N_d, the remaining spaces in the array will be filled with zeroes, and
2. another rectangular array JCOEF where JCOEF$[i, j]$ represents the column index of COEF$[i, j]$ on the matrix and the zero elements also correspond to certain but somewhat arbitrary column numbers.

Example 2.4.1. Consider the matrix

$$\mathrm{A} = \begin{bmatrix} 4 & 0 & 0 & 0 \\ 0 & 6 & 0 & 7 \\ 5 & 0 & 1 & 0 \\ 0 & 2 & 0 & 9 \end{bmatrix}.$$

The corresponding Ellpack-Itpack storage scheme can be shown as

$$\mathrm{COEF} = \begin{bmatrix} 4 & 0 \\ 6 & 7 \\ 5 & 1 \\ 2 & 9 \end{bmatrix}, \quad \mathrm{JCOEF} = \begin{bmatrix} 1 & 2 \\ 2 & 4 \\ 1 & 3 \\ 2 & 4 \end{bmatrix}.$$

Here, the position of the zero element in the first row of the COEF corresponding to JCOEF can be filled with any index column other than 1, i.e. 2, 3 and 4.

Chapter 3

Sparse Direct Solvers

A sparse direct solver is a numerical algorithm for solving sparse linear systems. Unlike iterative solvers, which use an iterative approach to achieve convergence to the solution, direct solvers compute the solution of a linear system directly. Nowadays, direct solvers are used in many different fields, including scientific computing, engineering, and finance. They are particularly useful in cases where the linear system being solved is very large, or where ill-conditioned matrix can cause problems for iterative methods. In this chapter, we will first broadly look at phases of this method and fill-in reducing ordering, and then focus on an important structure, the supernodal structure.

3.1 Introduction

The goal is to factorize a large sparse matrix using Gaussian elimination. However, due to its size, it is more efficient to partition the large sparse matrix into several small dense matrices and perform factorization of each of these small dense matrices individually. These dense matrices are called supernodes, which are connected to each other by an elimination tree. Each supernode corresponds to a row and a column that needs to be factorized. How supernodes and supernodal elimination tree are generated, we can see in Section 3.3. With supernodes our computation is much more efficient, because with many processors, factorization of some supernodes can be performed simultaneously. The factorization of a supernode has an impact on the supernodes that are not factorized, so we use updates along with it. How to perform these updates is the main difference between the algorithms of direct solvers, and is the focus of this thesis. There are broadly three types of data movements for the updates, namely left-looking, right-looking and multifrontal algorithms, see Section 3.4. In addition to updates, we need to preprocess the matrix before factorization so that the matrix has as little *fill-in* as possible after factorization, and has good numerical stability.

Definition 3.1.1 (Fill-in [4])**.** Assume that a matrix A can be factorized as $A = LU$, where L is the lower triangular matrix and U is the upper triangular matrix. Then the fill-in after the factorization of matrix A is the introduction of new non-zero entries in $L + U$ whose corresponding entries in A are zero.

A typical sparse solver consists of four distinct steps :

1. Ordering: A fill-in reducing algorithm is applied to the matrix.
2. Symbolic analysis: The nonzero pattern of the factors are determined, and the elimination tree is thus obtained.
3. Numerical factorization: The numerical values of L and U factors are computed.
4. Forward and back substitution: The iterative refinement and a complete solution to the linear system of equations is performed with the factors.

It's worth noting that in the case of general unsymmetric systems, additional pivoting strategies may be required to ensure numerical accuracy. Consequently, the solver may combine the first three steps 1, 2, and 3 to accommodate these strategies.

In the numerical factorization phase, the corresponding decomposition methods differ for different types of matrices. For a certain type of matrix, the decomposition method used for it is provided in Table 3.2.

Shape	Characteristic	Decomposition
Square	Symmetric positive definite	Cholesky decomposition
	Symmetric Indefinite	$\mathrm{LDL}^{\mathrm{T}}$ decomposition
	Unsymmetric	LU decomposition
Rectangular		QR decomposition

Table 3.1: Different types of the decomposition

Since the object of this thesis is a general unsymmetric matrix, we use LU decomposition by default.

3.2 Fill-In Reducing Ordering

In this section we will follow [9] to overview three of the fill-in reducing orderings, namely approximate minimum degree ordering, a priori column ordering and nested dissection ordering. For more detailed descriptions see [1], [11] and [13].

To understand approximate minimum degree ordering, let's begin with the Markowitz criterion. In each elimination stage of Gaussian elimination, the Markowitz criterion considers an active submatrix of size $(n-k+1) \times (n-k+1)$, where n is the size of the original matrix and k represents the current elimination stage. In the active submatrix of size $(n-k+1) \times (n-k+1)$, the number of entries in row i and the number of entries in column j are denoted respectively by $r_i^{(k)}$ and $c_j^{(k)}$. The Markowitz criterion aims to select the entry $a_{ij}^{(k)}$ that maximizes the expression

$$(r_i^{(k)} - 1)(c_j^{(k)} - 1) \tag{3.1}$$

from the entries of the active submatrix that are not too small numerically. This is done because if $a_{ij}^{(k)}$ is the only nonzero element in its rows and columns in the submatrix, taking it as the next pivot does not produce a fill-in. Note that to implement the Markowitz criterion, it is usually necessary to know the latest sparse pattern of the reduced submatrix at each elimination stage.

In the context of symmetric matrices, the minimum degree ordering is a kind of Markowitz ordering strategy, where the degree refers to the number of non-zero entries in a column or row of the matrix being factorized. For a symmetric matrix, choosing the diagonal element as the pivot does not make the factorization unstable, so the diagnoal element $a_{ii}^{(k)}$ that best satisfies the Markowitz criterion can be chosen as the next pivot at the kth stage. Note that it is not necessary to explicitly update the sparse pattern at each stage. Instead, it relies on storing the degrees of the columns or rows and updating only the degrees that change during the pivoting steps. This approach significantly reduces the computational cost associated with degree updates. Additionally, an approximate minimum degree algorithm can be employed to further enhance the ordering. The approximate minimum degree algorithm provides better approximation results while still maintaining low computational cost.

For unsymmetric matrices, numerical pivoting is not so simple. To solve this problem, let's examine the patterns that emerge when performing Gaussian elimination without pivoting on the normal matrix $\mathrm{N} = \mathrm{A}^{\mathrm{T}}\mathrm{A}$ at first.

Theorem 3.2.1 (Pattern of the normal matrix [9])**.** *Suppose the rows of matrix* A *have been rearranged in a way that ensures* A *has pivots on its diagonal. Then when performing the Gaussian elimination on matrix* A*, the pattern of non-zero entries and their locations in each pivot row of* A *is a subset of the pattern observed in the corresponding pivot row during the reduction process of the normal matrix* $\mathrm{N} = \mathrm{A}^{\mathrm{T}}\mathrm{A}$ *of* A*.*

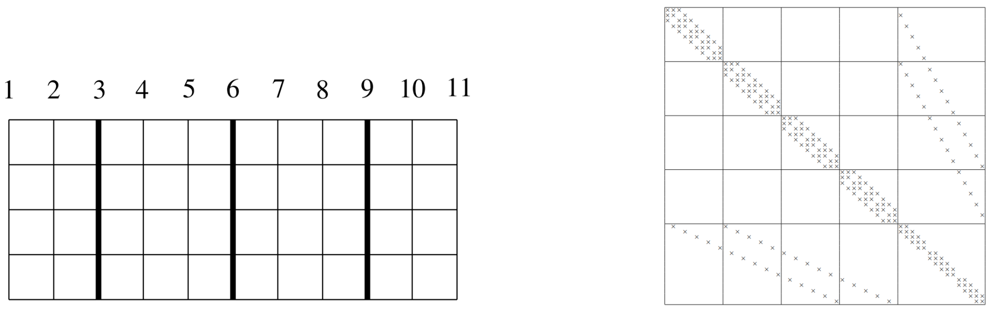
Theorem 3.2.1 follows that if the reduction process of the normal matrix $\mathrm{N} = \mathrm{A}^{\mathrm{T}}\mathrm{A}$ of A retains its sparsity, the resulting matrix of matrix A after reduction still contains a large number of zero entries. The theorem is useful because it allows us to study the pattern of the pivots in the symmetric normal matrix N, which can often be easier to work with than the original unsymmertric matrix A.

To summarise, we can construct a good fill-in ordering with the approximate minimum degree ordering and then apply it as a column ordering for A.

Now we turn to another type of fill-in ordering, namely nested dissection ordering. Nested dissection is an extension of one-way dissection including removal of a set of nodes from a graph of a symmetric graph and dissection of the graph into small pieces. A good dissection will make the matrix into diagonal blocks with borders and fill-in will then be restricted to these diagonal blocks and their borders, see Figure 3.1. We can further dissect these blocks, level by level, which is why this ordering is called nested dissection.

3.3 Supernodal Elimination Tree

Assume that A is an invertible matrix where none of the diagonal elements is zero, and that it can be decomposed as A = LU, where L is the upper triangular matrix and U



(a) A grid cut in the column 3 and 6 (b) The resulting matrix

Figure 3.1: An example of one-way dissection [9]

is the lower triangular matrix.

Definition 3.3.1 (Associated direct graph [3])**.** For a matrix $\mathrm{A} \in \mathbb{R}^{n\times n}$, the associated directed graph of A is (V, E), where $V = \{1, ..., n\}$ and $E = \{(i, j) \mid a_{ij} \neq 0\}$, and will be denoted by $G_{\mathrm{d}}(\mathrm{A})$.

Definition 3.3.2 (Undirect graph [3])**.** For a matrix $\mathrm{A} \in \mathbb{R}^{n\times n}$, the undirected graph of A is given by $G_d(|\mathrm{A}| + |\mathrm{A}|^{\mathrm{T}})$, and will be denoted by $G(\mathrm{A})$.

Definition 3.3.3 (Filled graph [3])**.** Given $\mathrm{A} = \mathrm{LU}$ with the aforementioned assumptions, the filled graph of A is the undirect graph $G(\mathrm{L} + \mathrm{U})$ and will be denoted by $G_f(\mathrm{A})$.

With the filled graph $G(\mathrm{A})$, we already know the structure of the L and U. This means that we have completed the symbolic factorization. The question that remains is how do we get the structure of $G(\mathrm{L} + \mathrm{U})$. Theorem 3.3.4 gives us the solution.

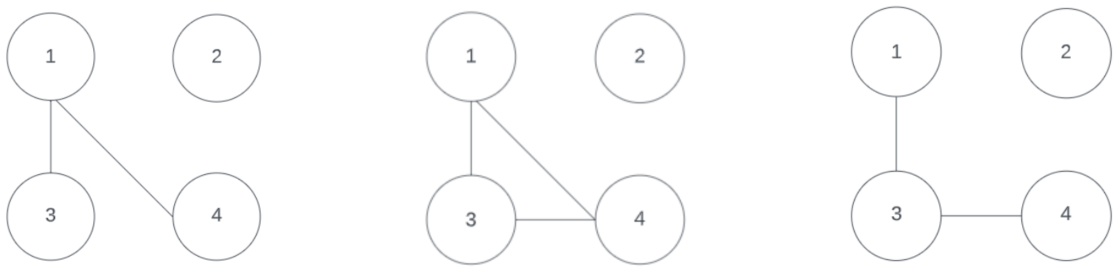
Theorem 3.3.4 (Symbolic LU factorization [3])**.** *Given* $\mathrm{A} = \mathrm{LU}$ *with the aforementioned assumptions, there exsits an edge* (i, j) *in* $G_d(\mathrm{L}+\mathrm{U})$ *if and only if there exists a path from* i *to* j *through vertices* $v_1, v_2, ..., v_k$ *where for any vertex* $v_l \in \{v_1, v_2, ..., v_k\}$, $v_l < \min(i, j)$.

Definition 3.3.5 (Elimination tree [3])**.** The elimination tree of $\mathrm{A} \in \mathbb{R}^{n\times n}$ is given by performing depth-first search in $G_f(\mathrm{A})$ starting from vertex n and will be denoted by $T(\mathrm{A})$. When vertex m is visited, choose from its unvisited neighbours the index with largest number.

Example 3.3.6 (Computation of elimination tree)**.** The matrix A is of the symbolic form

$$\mathrm{A} = \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} \bullet & & \bullet & \bullet \\ & \bullet & & \\ \bullet & & \bullet & \\ & & & \bullet \end{bmatrix} \end{array} \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}.$$

We can get the final elimination tree by first calculating the direct graph, and then the filled graph. The corresponding undirect graph, filled graph and elimination tree of A is shown in Figure 3.2.



(a) Undirect graph of A (b) Filled graph of A (c) Elimination tree of A

Figure 3.2: Generation of the elimination tree

Because only a_{13}, a_{14} and a_{31} are nonzero entries that are not on the diagonal, the edges of the undirect graph are $(1, 3)$ and $(1, 4)$. Now we can see there is a path from vertex 3 to vertex 4 in the undirect graph, passing through vertex 1. Since 1 is smaller than both 3 and 4, by Theorem 3.3.4, the filled graph will add the edge $(3, 4)$ to the undirect graph.

Finally, we use depth-first search starting from vertex 4 on the filled graph to obtain the elimination tree. In depth-first search, we begin by selecting a starting node and then visit all of its adjacent unvisited nodes one by one. Once we have visited all of a node's neighbors, we backtrack to the node that was visited immediately before the current node and continue the process with the next unvisited neighbor, repeating this process until we have backtracked to the starting node. When we finally get back to the starting node, we have completed the traversal of the component, and we can then start to do the same for the rest of the graph. Therefore, the search result is $\{4, 3, 1, 2\}$ and the edges in the elimination tree are $(3, 4)$ and $(1, 3)$.

For structurally symmetric matrices, there is a more direct method to determine the elimination tree without generating a specific filled graph, which is obtained by Corollary 3.3.7.

Corollary 3.3.7. *[3] Let $i, k \in \{1, ..., n\}$ and $i < k$. If $a_{ik} \neq 0$, then*

1. *Third level i is a desecdents of k;*
2. *Third level $a_{jk} \neq 0$ for all nodes j between i and k in the elimination tree $T(\mathrm{A})$.*

Definition 3.3.8 (Parent nodes list [3])**.** Parent nodes list is a vector p where p_i represents the parent node of node i.

We can use Corollary 3.3.7 to get the parent node list directly, instead of generating the filled graph. Since the elimination tree can be easily described by the parent nodes list, the elimination tree can be obtained directly as well. We start from column 2 and find the nonzero a_{ij} in each column j with $i < j$, because by Corollary 3.3.7, if such a nonzero exists, then i is a descendant of j. We can illustrate this process with the matrix

$$\begin{array}{c|cccc|c} & 1 & 2 & 3 & 4 & \\ \hline & \bullet & & \bullet & \bullet & 1 \\ \mathrm{A} = & & \bullet & & & 2 \\ & \bullet & & \bullet & & 3 \\ & \bullet & & & \bullet & 4 \end{array},$$

where we start by setting $p_i = 0$. The second column does not have the nonzero we need so the p in this step is not changed. In column 3 there is a nonzero entry a_{13}, so $p_1 = 3$. We finally come to column 4 where a_{14} is a nonzero, so 1 is a descendant of 4. However, 1 is also a descendant of 3, which implies $p_4 = 3$. Therefore, the parent nodes list of the matrix A is $p = (3, 0, 4, 0)$, see Table 3.2. This process is described by Algroithm 1.

	p_1	p_2	p_3	p_4
$k = 2$	0	0	0	0
$k = 3$	3	0	0	0
$k = 4$	3	0	4	0

Table 3.2: Determining the parent nodes list of matrix A

Algorithm 1 Computation of the elimination tree [3]

Input: Structurally symmetric matrix $\mathrm{A} \in \mathbb{R}^{n\times n}$
Output: Parent nodes list p of A

```
1: let $a \in \mathbb{R}^n$ be an auxiliary vector
2: $p \leftarrow 0, a \leftarrow 0$
3: for $k = 2, ..., n$ do
4:     for all $i < k$ such that $a_{ik} \neq 0$ do
5:         while $i \neq 0$ and $i < k$ do
6:             $j \leftarrow a_i$
7:             $a_i \leftarrow k$
8:             if $j = 0$ then
9:                 $p_i \leftarrow k$
10:            end if
11:            $i \leftarrow j$
12:        end while
13:    end for
14: end for
```

By using an elimination tree, it becomes simpler to determine the fill-in of a matrix. Any fill-in entries that were added when processing node i and its descendants will also be considered as fill-in for node j. The algorithm can be summarized as Algorithm 2.

Definition 3.3.9 (Supernode in symmetric case [3])**.** Denote by $\mathcal{P}_j$ the nonzero indices of column j of P as computed by Algorithm 2. A sequence of columns $k, k+1, ..., k+s-1$ is called supernode of size s if the columns of $\mathcal{P}_j = \mathcal{P}_{j+1} \cup \{j+1\}$ for all $j = k, ..., k+s-2$.

Algorithm 2 Computation of fill-in [3]

```
Input: Structurally symmetric matrix A ∈ $\mathbb{R}^{n\times n}$
Output: Sparse strict lower triangular pattern P ∈ $\mathbb{R}^{n\times n}$ with same pattern as L and $U^T$
1: compute parent nodes list $p$ of A
2: for $j = 1, .., n$ do
3:     supplement nonzero of column $j$ of P with all $i > j$ such that $a_{ij} \neq 0$
4:     $k \leftarrow p_j$
5:     if $k > 0$ then
6:         supplement nonzeros of column $k$ of P with nonzeros of column $j$ of P greater than $k$
7:     end if
8: end for
```

Definition 3.3.10 (Supernodal elimination tree [14])**.** Given a set of supernodes, the supernodal elimination tree is the tree consisting of the supernodes, where supernode $\tilde{S}$ is the parent of supernode $S = \{j, ..., j+t\}$ if the parent p of $j+t$ in the corresponding elimination tree belongs to $\tilde{S}$.

Example 3.3.11. Let's consider a matrix A which has the following form:

$$\mathrm{A} = \begin{array}{cccccc|c} 1 & 2 & 3 & 4 & 5 & 6 & \\ \hline \bullet & & & & \bullet & \bullet & 1 \\ & \bullet & & & & \bullet & 2 \\ & & \bullet & & & & 3 \\ & & & \bullet & \bullet & \bullet & 4 \\ \bullet & & & \bullet & \bullet & & 5 \\ \bullet & \bullet & & \bullet & & \bullet & 6 \end{array}\ .$$

This matrix is structurally symmetric, so we can use Corollary 3.3.7 to compute the parent nodes list $p = (5, 6, 0, 5, 6, 0)$ and then use Algorithm 2 to compute the sparse strict lower triangular pattern, which is

$$\mathrm{P} = \begin{array}{cccccc|c} 1 & 2 & 3 & 4 & 5 & 6 & \\ \hline \bullet & & & & & & 1 \\ & \bullet & & & & & 2 \\ & & \bullet & & & & 3 \\ & & & \bullet & & & 4 \\ \bullet & & & \bullet & \bullet & & 5 \\ \bullet & \bullet & & \bullet & \bullet & \bullet & 6 \end{array}\ .$$

Next let's compute the column size of each supernode. Since $\mathcal{P}_1 = \{1, 5, 6\} \neq \{2, 6\} \cup \{2\} = \mathcal{P}_2 \cup \{2\}$, the second column cannot be grouped into the same supernode as the first column following Definition 3.3.9. Similarly, the third column cannot be grouped with the second column, and the fourth column cannot be grouped with the third column. However, columns 4, 5, and 6 can be grouped in the same supernode, because $\mathcal{P}_4 = \{4, 5, 6\} =$

$\mathcal{P}_5 \cup \{4\}$ and $\mathcal{P}_5 = \{5, 6\} = \mathcal{P}_6 \cup \{5\}$. Therefore, the supernodes of A are $\{1\}$,$\{2\}$,$\{3\}$ and $\{4, 5, 6\}$.

The left side of Figure 3.3 is the elimination tree of A, which we can get directly from the parent nodes list. Given the supernodes of A as $\{1\}$,$\{2\}$,$\{3\}$ and $\{4, 5, 6\}$, the corresponding supernodal tree then looks like the one on the right.

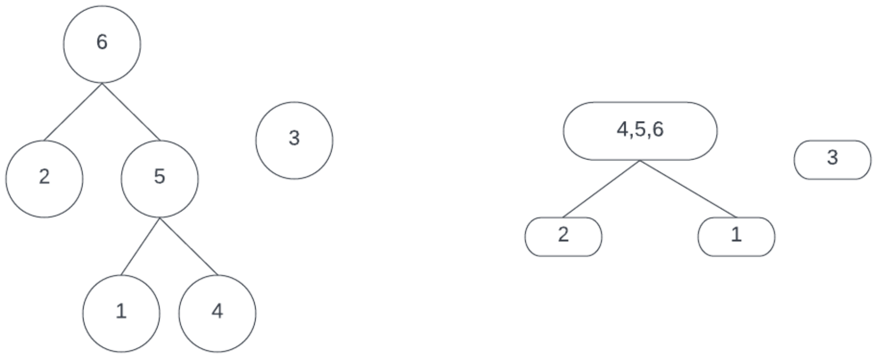


Figure 3.3: Generation of supernodal elimination tree

This definition of supernode applies specifically to symmetric matrices. However, for general unsymmetric matrices, a process called pivoting is performed at each step, resulting in the creation of a supernodal column tree, as described in [8]. In general, if it is possible to treat columns as a computational unit during the sparse LU factorization process, they are grouped together to form a supernode. Essentially, a supernode is made up of collections of columns that share the same off-diagonal sparsity pattern.

3.4 Three Patterns of Data Movements

In this section, we will see three algorithms to implement sparse direct solvers, namely left-looking, right-looking, and multifrontal algorithms, following [3]. They all require the use of an elimination tree to perform the numerical factorization. The main difference between these algorithms lies in how they organize and arrange data movement and computation while maintaining the priority relationships represented by the elimination tree. Figure 3.4 shows the patterns of data access in the these algorithms, where the circled node corresponds to the current submatrix being factorized.

The updates are applied to the current subtask by modifying the non-zero entries. In left-and right-looking algorithms, this is done through the edges of the filled graph that connect the current subtask with the previous subtasks that generated the updates. In left-looking algorithm, the updates are not immediately applied as they are generated. Instead, they are delayed and applied just before the factorization of the current submatrix, when the updates become pivotal. In contrast to it is the right-looking algorithm, in which as soon as the updates are computed in the current subtask, they are directly applied to the corresponding entries in the future subtask without delay.

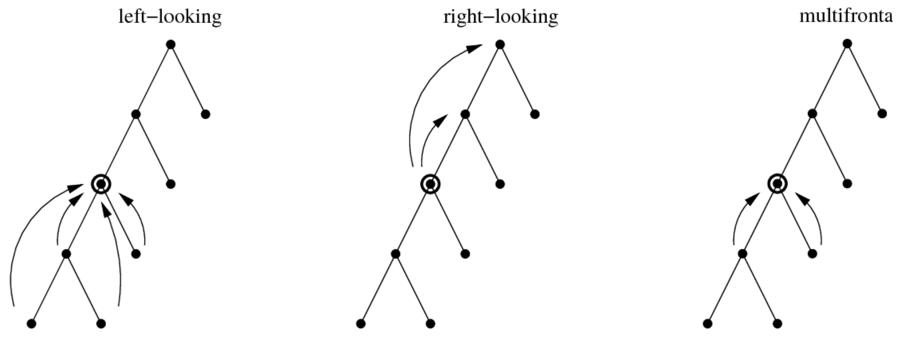


Figure 3.4: Patterns of data access [3]

The difference between left- and right-looking algorithms lies in how updates are applied during the factorization process. However, in the first two algorithms, the update is always connected to an ancestor subtask in the elimination tree via a filled graph. Unlike these two algorithms is the multifrontal algorithm, in which the updates traverse the elimination tree, passing through intermediate vertices, until they reach the appropriate ancestor column where they are applied. Each intermediate vertex in the path represents a subtask that contributes to the update transfer.

3.5 Solve Phase with the Elimination Tree

The supernodal structures will be used not only in the factorization phase, but also in the solve phase for forward and backward substitution afterwards. Let's take a general look at how the solve phase works under supernodal structures following [9].

At first we look at the solution of a linear system where the system matrix is decomposed in a product of triangular matrices in the matrix-dense case. The solution x of $\mathrm{L}\mathrm{U}x = b$ is obtained through a forward elimination step $\mathrm{L}y = b$ followed by a backward elimination step $\mathrm{U}x = y$. The system $\mathrm{L}x = c$, where L is a matrix with a block structure as shown:

$$\begin{bmatrix} \mathrm{L}_{11} & & & \\ \mathrm{L}_{21} & \mathrm{L}_{22} & & \\ \vdots & \vdots & \ddots & \\ \mathrm{L}_{m1} & \mathrm{L}_{m2} & \cdots & \mathrm{L}_{mm} \end{bmatrix}_{m\times m},$$

can be solved using a block forward-substitution algorithm, which is shown in Algorithm 3.

Similarly, the system $\mathrm{U}x = c$, where U is a matrix with a block structure as shown:

$$\begin{bmatrix} \mathrm{U}_{11} & \mathrm{U}_{12} & \cdots & \mathrm{U}_{1m} \\ & \mathrm{U}_{22} & \cdots & \mathrm{U}_{2m} \\ & & \ddots & \vdots \\ & & & \mathrm{U}_{mm} \end{bmatrix}_{m\times m},$$

can be solved using a block back-substitution algorithm, which is shown in Algorithm 4.

Algorithm 3 Block forward substitution

Input: Lower triangular matrix $\mathrm{L} \in \mathbb{R}^{n\times n}$ and vector $c \in \mathbb{R}^n$
Output: $x \in \mathbb{R}^n$ such that Lx=c

```
1: for $j = 1, j <= m, j++$ do
2:     solve $\mathrm{L}_{jj}x_j = c_j$
3:     for $i = j+1, i <= m$ do
4:         $c_i - \leftarrow c_i - \mathrm{L}_{ij}x_j$
5:     end for
6: end for
```

Algorithm 4 Block backward substitution

Input: Upper triangular matrix $U \in \mathbb{R}^{n\times n}$ and vector $c \in \mathbb{R}^n$
Output: $x \in \mathbb{R}^n$ such that $\mathrm{U}x = c$

```
1: for $j = m, j >= 1, j--$ do
2:     solve $\mathrm{U}_{jj}x_j = c_j$
3:     for $i = 1, i <= j-1$ do
4:         $c_i \leftarrow c_i - \mathrm{U}_{ij}x_j$
5:     end for
6: end for
```

In the sparse case, many of the blocks U_{kj} in the upper triangular matrix U may contain zeros or zero columns. To represent the block pivot row in the block triangular form, U_{jj} and U_{j*} are used. Here, U_{jj} represents the diagonal block, and U_{j*} represents the remaining part of the row. Then the block back-substitution consists of successively for $k = m, m-1, ..., 1$ solving

$$\mathrm{U}_{kk}x_k = c_k^{(k)} \tag{3.2}$$
$$c_j^{(k+1)} = c_j^{(k)} - \mathrm{U}_{*k}x_{*k}, \tag{3.3}$$

where x_{*k} is the part of x that corresponds to L_{*k}. The forward substitution process involves solving the system of equations in a similar way but in the forward order, starting from $k = 1$ up to $k = m$. Specifically, it solves [9]

$$\mathrm{L}_{kk}x_k = c_k^{(k)} \tag{3.4}$$
$$c_j^{(k+1)} = c_j^{(k)} - \mathrm{L}_{*k}x_{*k} \tag{3.5}$$

with with $c^{(1)} = b$.

In the solve phase of direct solvers, the tree structure is utilized to efficiently carry out the backward and forward substitution operations. We assume that the blocks of the factors are stored based on the variables that have been eliminated on each node.

During backward substitution, nodes are visited in a top-down manner, starting from the root node and proceeding to its children. As the tree is traversed, the backward substitution process updates the node vector of each visited node. Starting from the root node, the components of the solution vector x are computed and stored in the node vector.

If the node vector is preserved while processing child nodes, the child node vector x_{*k} can be derived from it. The forward substitution process is similar to the backward one, except that it visits each node before its parent node.

Chapter 4

UMFPACK

UMFPACK (Unsymmetric MultiFrontal Package) is a library developed by Tim Davis at the University of Florida, which is designed to solve large, sparse, unsymmetric linear systems of equations. In this chapter we follow [4] to go through the algorithm of UMFPACK. The version of UMFPACK studied is 4.0, which we call UMFPACK4 here. UMFPACK4 utilizes a specific method for sparse LU factorization, which involves two key techniques: column pre-ordering and a right-looking unsymmetric-pattern multifrontal numerical factorization. The algorithm proceeds by creating frontal matrices, which are dense submatrices, for each supernode. Before numerical factorization, we should permute the matrix to reduce the fill-in. In the case of unsymmmetric-pattern multifrontal algorithm here, we use column preordering strategy for fill-in reducing.

4.1 Preprocessing

One of the purpose of the prepocessing is to preseve sparsity as well as to maintain numerical stability. The matrix will be eventually factored as $\mathrm{A}' = \mathrm{PRAQ}$, where P is a row permutation matrix for reducing fill-in and preserving sparsity, Q is a column permutation matrix for preserving sparsity, R is a diagonal matrix of row scaling factors, which can be used to balance the magnitudes of the row elements and further improves the numerical accuracy. We first determine column permutation Q. Once the column permutation Q is established, the next step is to identify pivot rows. For each pivot column, specific rows, called pivot rows, are selected. The purpose of this selection is to maintain numerical stability. In addition, once the pivot columns are determined, the supernodal elimination tree can be computed, because although pivot columns can be further modified in numerical factorization, it is performed in supernode units.

First, if a pivot has a Markowitz cost of zero, it can be directly placed in the appropriate position in the L or U matrix without any further operations. As we know from Section 3.2, these are pivots satisfying $(r_i^{(k)} - 1)(c_j^{(k)} - 1) = 0$. Therefore, they are the only nonzero element in its rows or columns (or both) in the submatrix and thus can be eliminated without causing fill-in.

After the elimination of pivots with zero Markowitz cost and their placement in the LU factors, the focus shifts to the remaining submatrix S for further analysis. For a general unsymmetric matrix, UMFPACK4 uses unsymmetric strategy for preordering the rows and columns. For special matrices there are another two strategies, namely 2-by-2, and symmetric strategies, see [4]. A slightly modified version of the COLAMD algorithm [6] is utilized when applying the unsymmetric strategy. COLAMD is based on a priori column ordering which is introduced in Section 3.2. The method finds a symmetric permutation Q of the matrix $\mathrm{S}^{\mathrm{T}}\mathrm{S}$ using an approximate minimum degree method [1]. The column pre-ordering can be modified during the factorization process.

The supernodal column elimination tree is then computed. In the supernodal elimination tree, each node represents a supernode, which is a set of variables that can be eliminated together using a block Gaussian elimination.

After a supernode is generated, the algorithm will perform symbolic factorization on each supernode. In this process, the algorithm will obtain the frontal matrix corresponding to the supernode and calculate the non-zero pattern and upper bound of each frontal matrix, which represents the maximum number of non-zero elements that may appear in the numerical factorization phase. Next, we will see what a frontal matrix is.

Assume that the original matrix A can be the matrix A can be expressed as a sum of individual matrices

$$\sum_{l} \mathrm{A}^{[l]}, \tag{4.1}$$

where non-zero entries of $\mathrm{A}^{[l]}$ are limited to the specific rows and columns that pertain to the variables associated with that particular element. The basic assembly operation for constructing A is of the form

$$a_{ij} \leftarrow a_{ij} + a_{ij}^{[l]}. \tag{4.2}$$

Definition 4.1.1 (Fully summed [9]). An entry is fully summed when all contributions of the form (4.2) have been summed.

A variable can be eliminated once its rows and columns are fully summed, that is, after it has made its final appearance in the matrix $\mathrm{A}^{[l]}$. Consequently, the elimination operation is restricted to the submatrices that correspond to the rows and columns associated with the variables that have not yet been eliminated. This approach allows for intermediate computations to be performed on matrices whose size dynamically changes as variables are introduced or eliminated. To be specific, when a variable first appears in the matrix components, the size of the full matrix increases to accommodate the new variable, while the size of the matrix decreases as variables are eliminated because the rows and columns related to those eliminated variables are no longer considered.

Definition 4.1.2 (Frontal matrix [9]). Frontal matrix is the full matrix in which all intermediate work is performed.

The frontal matrices are factorized using LU, Cholesky, or QR factorization, depending on the sparsity pattern and symmetry of the matrix. The solutions of the frontal matrices are combined to form the solution of the original sparse matrix. This process involves solving triangular systems and updating the solutions of the remaining frontal matrices. A frontal matrix can be written as Figure 4.1, where F_{11} contains the fully summed rows and columns and is thus already factorized.

$$\begin{array}{ccc} & \text{fully summed columns} & \text{partly summed columns} \\ & \downarrow & \downarrow \\ \text{fully summed rows} \rightarrow & \mathbf{F}_{11} & \mathbf{F}_{12} \\ \text{partly summed rows} \rightarrow & \mathbf{F}_{21} & \mathbf{F}_{22} \end{array}$$

Figure 4.1: Partitioning of a frontal matrix [2]

Definition 4.1.3 (Contribution block [2])**.** The contribution block of a frontal matrix of the form as in Figure 4.1 is a Schur complement formed as $F_{22} - F_{12}^T F_{11}^{-1} F_{12}$.

Roughly speaking, the supernodal elimination tree provides the high-level structure of the factorization, while the frontal matrices provide the low-level details of the factorization. Figure 4.2 illustrates an example of frontal matrices in a supernodal elimination tree, assuming no pivoting is performed during the column pre-ordering and symbolic analysis or numrical factorization phase for the sake of simplicity.

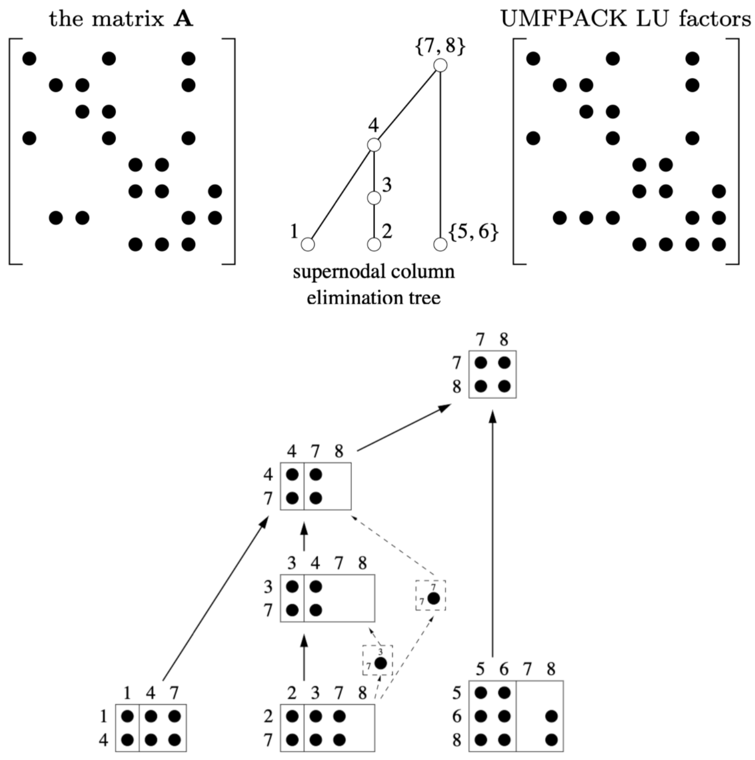
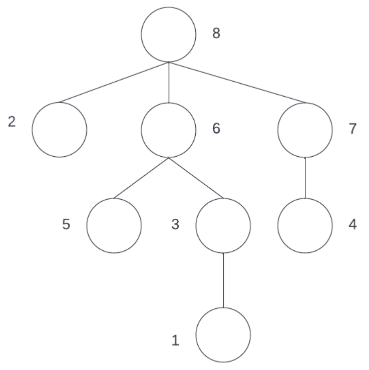


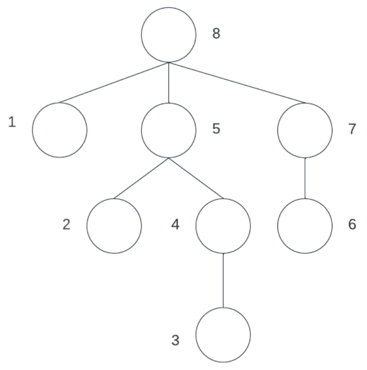
Figure 4.2: An example of frontal matrices in a supernodal elimination tree [4]

Given an supernodal elimination tree, we can reselect the order in which operations are performed. We want the generated elements required at each stage to be those most recently generated and so far unused, so we can use the stack as temporary storage on a single processor. To do this we use *post-ordering.*

Definition 4.1.4 (Post-ordering [9])**.** A post-ordering on a tree orders the node numbers such that every node is ordered ahead of its parent and the nodes in every subtree are numbered consecutively.



(a) Original



(b) After Post-ordering

Figure 4.3: An example of post-ordering

Figure 4.3 shows an example for post-ordering. With the post-ordering, each chain in the subsequent tree structure starts as a leaf, with the number of chains matching the number of leaves. Next, each frontal matrix is assigned to a unifrontal chain. For each chain, the largest frontal matrix is identified, dictating the size of the work array required for factorization, which needs to be capable of holding the largest frontal matrix and all prior pivot rows and columns from that specific chain.

4.2 Unsymmetric-Pattern Multifrontal Methods

Now let's look at the method used by UMFPACK in the numerical factorization stage, which is the unsymmetric multifrontal method. We start by looking at an example.

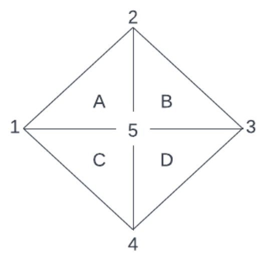


Figure 4.4: A triangulated region

Example 4.2.1. Consider a finite-element problem as in Figure 4.4, where there are three variables associated with each triangle. First we assemble the contribution from triangles

A and B. At this point the variable 2 is fully summed, so it can be eliminated in the frontal matrix of the symbolic form

$$\begin{array}{cccc|c} 2 & 1 & 5 & 3 & \\ \bullet & \bullet & \bullet & \bullet & 2 \\ \bullet & \bullet & \bullet & & 1 \\ \bullet & \bullet & \bullet & \bullet & 5 \\ \bullet & & \bullet & \bullet & 3 \end{array}.$$

Next we assemble another contribution from C. Then the variable 1 is fully summed and the corresponding matrix is of the symbolic form

$$\begin{array}{ccc|c} 1 & 5 & 4 & \\ \bullet & \bullet & \bullet & 1 \\ \bullet & \bullet & \bullet & 5 \\ \bullet & \bullet & \bullet & 4 \end{array}.$$

Note that the variables 1 and 5 in this frontal matrix retain the numerical values from the previous frontal matrix after the completion of the elimination operation. Such a value transfer can be obtained by directly extending the previous frontal matrix, or by using contribution blocks.

The above example shows the procedure of the unifrontal method. In this method, each element to be factorized corresponds to a frontal matrix. Once it has been fully summed, the dense matrix can be factorized with updates generated from previous frontal matrices.

Now we will see how the LU decomposition of a frontal matrix is performed. Here we review the algorithm of LU decomposition of A first. Algorithm 5 shows the process where U_i represents the i-th row of the matrix.

Algorithm 5 LU decomposition

Input: $A \in \mathbb{R}^{n \times n}$

Output: Lower triangular matrix L and upper triangular matrix U such that LU = A

```
1: $L \leftarrow I_n$
2: $U \leftarrow A$
3: for $i = 1, 2, ..., n$ do
4:     for $j = i + 1, i + 2, ..., n$ do
5:         $l_{ij} \leftarrow u_{ij}/u_{ii}$
6:         $U_j \leftarrow U_j - l_{ji}U_i$
7:     end for
8: end for
```

Consider a matrix

$$A = \left[\begin{array}{c|cccc} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ \hline a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{array}\right].$$

After the first step of Gaussain elimination the matrix looks like

$$A^{(1)} = \left[\begin{array}{c|cccc} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ \hline \frac{a_{21}}{a_{11}} & a_{22} - \frac{a_{21}}{a_{11}}a_{12} & a_{23} - \frac{a_{21}}{a_{11}}a_{13} & \cdots & a_{2n} - \frac{a_{21}}{a_{11}}a_{1n} \\ \frac{a_{31}}{a_{11}} & a_{32} - \frac{a_{31}}{a_{11}}a_{12} & a_{33} - \frac{a_{31}}{a_{11}}a_{13} & \cdots & a_{3n} - \frac{a_{31}}{a_{11}}a_{1n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{a_{n1}}{a_{11}} & a_{n2} - \frac{a_{n1}}{a_{11}}a_{12} & a_{n3} - \frac{a_{n1}}{a_{11}}a_{13} & \cdots & a_{nn} - \frac{a_{n1}}{a_{11}}a_{1n} \end{array}\right]$$

we denote the four blocks of A and $A^{(1)}$ respectively by W,X,Y,Z and $W^{(1)}$, $X^{(1)}$, $Y^{(1)}$, $Z^{(1)}$, so A and $A^{(1)}$ are respectively of the form

$$A = \left[\begin{array}{c|c} W & X \\ \hline Y & Z \end{array}\right]$$

and

$$A^{(1)} = \left[\begin{array}{c|c} W^{(1)} & X^{(1)} \\ \hline Y^{(1)} & Z^{(1)} \end{array}\right].$$

An observation is

$$Z^{(1)} = Z - X^{(1)}(W^{(1)})^{-1}Y^{(1)}, \tag{4.3}$$

and that's why we define the contribution block in that way. Following Algorithm 5, the first column of L and the first row of U are respectively

$$\begin{bmatrix} 1 & \frac{a_{21}}{a_{11}} & \frac{a_{31}}{a_{11}} & \cdots & \frac{a_{n1}}{a_{11}} \end{bmatrix}^T$$

and

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{14} \end{bmatrix}.$$

Removing the first element of the first vector we obtain

$$L^{(1)} := \begin{bmatrix} \frac{a_{21}}{a_{11}} & \frac{a_{31}}{a_{11}} & \cdots & \frac{a_{n1}}{a_{11}} \end{bmatrix}^T$$

Then the matrix will be stored as

$$\left[\begin{array}{c|c} \mathrm{W}^{(1)} & \mathrm{X}^{(1)} \\ \hline \mathrm{L}^{(1)} & \mathrm{Z}^{(1)} \end{array}\right] = \left[\begin{array}{c|cccc} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ \hline \dfrac{a_{21}}{a_{11}} & a_{22} - \dfrac{a_{21}}{a_{11}} a_{12} & a_{23} - \dfrac{a_{21}}{a_{11}} a_{13} & \cdots & a_{2n} - \dfrac{a_{21}}{a_{11}} a_{1n} \\ \dfrac{a_{31}}{a_{11}} & a_{32} - \dfrac{a_{31}}{a_{11}} a_{12} & a_{33} - \dfrac{a_{31}}{a_{11}} a_{13} & \cdots & a_{3n} - \dfrac{a_{31}}{a_{11}} a_{1n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \dfrac{a_{n1}}{a_{11}} & a_{n2} - \dfrac{a_{n1}}{a_{11}} a_{12} & a_{n3} - \dfrac{a_{n1}}{a_{11}} a_{13} & \cdots & a_{nn} - \dfrac{a_{n1}}{a_{11}} a_{1n} \end{array}\right]$$

after the first elimination.

In general, if A and $\mathrm{A}^{(k)}$ which is obtained after kth elimination of A are respectively of the form

$$\mathrm{A} = \left[\begin{array}{c|c} \mathrm{W} & \mathrm{X} \\ \hline \mathrm{Y} & \mathrm{Z} \end{array}\right]$$

and

$$\mathrm{A}^{(k)} = \left[\begin{array}{c|c} \mathrm{W}^{(k)} & \mathrm{X}^{(k)} \\ \hline \mathrm{Y}^{(k)} & \mathrm{Z}^{(k)} \end{array}\right],$$

where W and $\mathrm{W}^{(k)}$ are both of size $k \times k$, then the following holds:

$$\mathrm{Z}^{(k)} = \mathrm{Z} - \mathrm{X}^{(k)}(\mathrm{W}^{(k)})^{-1}\mathrm{Y}^{(k)}. \tag{4.4}$$

When these frontal matrices are factorized along the supernodal elimination tree, this method is called multifrontal method. For the right-looking unsymmetric-pattern multifrontal method, we factorize these frontal matrices in terms of chains, as the supernodal column elimination tree has been post-ordered. For each chain, we perform unifrontal method starting from the leaves and factorize upwards.

When we deal with a supernode, the first thing we need to do is to find the next pivot in it. During numerical factorization, the pivot is chosen from up to 4 candidate pivots which is within the upper bound of the frontal matrix determined during the symbolic factorization phase. A candidate pivot will be selected as the pivot if it results in the smallest increase in the number of nonzeros in the current frontal matrix during the numerical factorization. The reason why it is possible to rearrange these candidate pivot columns in the frontal matrix is that each of these columns has the same upper bound nonzero pattern as far as the symbolic analysis is concerned. In addition, certain pivot rows were identified as having the potential to be used as pivots in the numerical factorization phase during the symbolic analysis phase. Therefore, the candidate pivot rows can also be rearranged in any order within a supernode during numerical factorization without affecting the result.

Let C_i denote the set of $|C_i|$ candidate pivot columns in the ith frontal matrix and the frontal matrix hold up to n_B prior pivot rows and columns. Then the factorization process can be outlined in Algorithm 6.

Algorithm 6 UMFPACK4 numerical factorization[1][4]

Input: A sparse matrix

Output: LU data structure of the input matrix

```
1: $i \leftarrow 0, k \leftarrow 0$
2: for each chain do
3:     current frontal matrix is empty
4:     for each frontal matrix in the chain do
5:         $i \leftarrow i + 1$
6:         for $|C_i|$ iterations do
7:             $k \leftarrow k + 1$
8:             find the $k$th pivot row and column
9:             apply pending updates to the $k$th pivot column
10:            if too many zero entries in new LU part then
11:                apply all pending updates
12:                copy pivot rows and columns into LU data structure
13:            end if
14:            if too many zero in new frontal matrix then
15:                create new contribution block and place on stack
16:                start a new frontal matrix
17:            else
18:                extend the frontal matrix
19:            end if
20:            assemble contribution block into current forntal matrix
21:            scale pivot column
22:            if # pivots in current matirx $\geq n_B$ then
23:                apply all pending updates
24:                copy pivot rows and columns into LU data structure
25:            end if
26:        end for
27:    end for
28:    apply all pending updates
29:    copy pivot rows and columns into LU data structure
30:    create new contribution block and place on stack
31: end for
```

Due to the presence of numerical pivoting, some variables may not be eliminated from the frontal matrix. All the failed pivots in the previous frontal matrix will be treated as pending updates, which need to be applied to the contribution block. Once these updates are applied, the pivot can be fully eliminated and its rows and columns can be moved to the LU data structure. Pending updates should be applied in three situations:

1. When the current frontal matrix cannot be fully eliminated due to the presence of too many zero entries in the new LU part. The entries in the pivot row and column are used to eliminate the corresponding variables. However, if these entries are zero or too small, the elimination may not be effective. In such case, applying these pending updates can change the entries in the pivot row and pivot column to make the pivot fully eliminated.

2. When the number of pivots in the current matrix reaches a threshold value due to the size of working array.

3. When the factorization reaches the last frontal matrix in the chain. Under this situation, the corresponding pivot rows and columns will be copied into LU data structure and contributes block are placed on stack, regardless of the number of zero elements added to the frontal matrix.

Finally let's turn to see how the frontal matrix to be used to process pivot k is created. Denote the frontal matrix for eliminating pivot $k-1$ by i. Following Algorithm 6, if the contribution block of the frontal matrix i has too many zero elements, then this contribution block is stacked and a new frontal matrix is created. Otherwise, the frontal matrix i can be simply extended to include the rows and columns of the pivot k. After this it is possible to assemble the contribution blocks from the stack into this new or extended frontal matrix. The reason for controlling the number of zero entries is that non-zero elements need to participate in calculations in operations such as matrix multiplication, while zero entries do not contribute to calculations and waste computational resources. This step is therefore taken to balance computational efficiency and accuracy.

[1]This version of Algorithm 6 differs from the one included in my submitted bachelor thesis. The current algorithm is the correct version.

Chapter 5

MUMPS

MUMPS (MUltifrontal Massively Parallel sparse direct Solver) is a solver developed at École Nationale Supérieure des Mines de Saint-Étienne in France. To begin with, the comparison [12] between UMFPACK and MUMPS is provided in the following:

UMFPACK

Fill reducing ordering: Column approximate minimum degree ordering

Pivoting strategy: Threshold pivoting implemented by row-exchanges

Numerical factorization: Unsymmetric-pattern multifrontal

Target architecture: Serial

MUMPS

Fill reducing ordering: Approximate minimum degree ordering based on $\mathrm{A} + \mathrm{A}^{\mathrm{T}}$

Pivoting strategy: Threshold pivoting implemented by row-exchanges

Numerical factorization: Symmetric-pattern multifrontal

Target architecture: Distributed-memory parallel

It has shown that MUMPS runs on a distributed memory parallel computer, which is different from a serial computer where each instruction is executed in a predetermined sequence. A distributed memory parallel computer is a system that uses multiple computers. These computers are connected together, each with its own processor and memory, allowing multiple tasks to be processed simultaneously. Therefore, in contrast to UMFPACK, MUMPS is a parallel solver that prioritizes computational efficiency. It also utilizes multifrontal methods but goes beyond that by capitalizing on parallelism derived from both the sparsity structure of the matrix A and the dense factorization kernels. To address this factor, a fully asynchronous algorithm based on a multifrontal approach with distributed dynamic scheduling of tasks is designed. In this section we will follow [2] to take a look at the parallel process.

5.1 Analysis Phase

In the beginning of preprocessing an ordering based on the symmetric pattern $\mathrm{A} + \mathrm{A}^\mathrm{T}$ is performed. In addition to fill-in reducing, this phase also includes partial pivoting and scaling. According to [2], the matrix should be scaled at first, because it has been shown that rearranging the rows or columns of the matrix before factorization, with the objective of maximizing the magnitude of the diagonal entries, can significantly decrease the need for pivoting during the factorization process. When all preprocessing options are activated, the matrix is modified and factorized into the form $\mathrm{PD}_r\mathrm{AQD}_c\mathrm{P}^\mathrm{T}$, where P is a permutation matrix applied symmetric, Q a column permutation and D_r and D_c are diagonal matrices for scaling.

In distributed memory parallel computing, there exists a process with a specific role and function, referred to as a "host". The relationship between the host and other processors is a parent-child relationship and the host is responsible for coordinating and managing the work of the other processors. The fill-in reducing is undertaken by the host, which perfroms an approximate minimum degree ordering by default based on the symmetrized matrix pattern $\mathrm{A} + \mathrm{A}^\mathrm{T}$. In addition, the host is responsible for symbolic factorization and computing a mapping that assigns these nodes to the different processors in the distributed system. The mapping is based on minimizing the cost of communication between programs in the factorization and solve phases, and balancing the memory and computation required by these processors. Once the processes receive information about the allocation from the host, they estimate the size of the work array based on this information, so that it is sufficient to handle computational tasks, as well as dynamic tasks that may arise temporarily during the decomposition phase, assuming that there is no significant occurrence of unexpected fill-in caused by numerical pivoting.

5.2 Numerical Factorization Phase

MUMPS uses a symmetric-pattern multifrontal algorithm guided by the elimination tree corresponding to the symmetric structure of $\mathrm{A} + \mathrm{A}^\mathrm{T}$ instead of column elimination tree used in unsymmetric-pattern one, which is the difference between these two patterns. Figure 5.1 shows an example of the elimination tree used in symmetric-pattern multifrontal algorithm. By symmetrized pattern, we mean the pattern of the matrix $\mathrm{A} + \mathrm{A}^\mathrm{T}$, which allows the matrix to be unsymmetric. The solution to the unsymmetric matrix can then be found by transforming the solution of the corresponding symmetric matrix. The symmetric-pattern multifrontal method was introduced before unsymmetric-pattern one and was originally developed to handle symmetric matrices. For very unsymmetric matrices, using elimination tree will incur more overhead due to unnecessary dependencies and extra zeros in the symmetric frontal matrices, so the performance may not be as good as when solving symmetric matrices or when using the unsymmetric-pattern multifrontal method. However, in some cases, the symmetric-pattern multifrontal method may still be used as a viable alternative, especially if the matrix is not highly unsymmetric.

Let's focus on how distributed scheduling affects this numerical factorization phase. In the factorization phase, a supernode can be treated by one or more processes. For

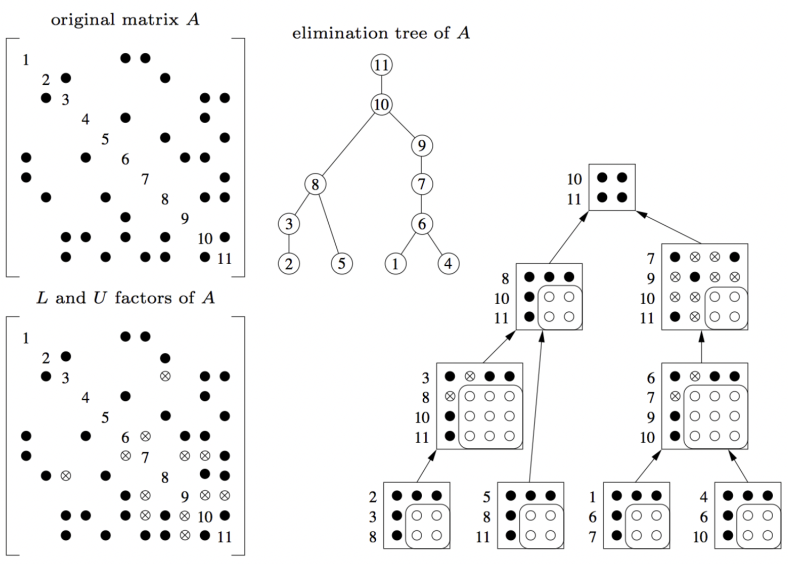


Figure 5.1: An example of frontal matrices in an elimination tree [5]

the different number of processes corresponding to nodes, nodes are divided into three categories.

Definition 5.2.1 (Node of type 1 [2])**.** A node is of type 1, if it is treated by only one process.

Definition 5.2.2 (Node of type 2 [2])**.** A node is of type 2, if it has a large contribution block and its rows are partitioned into blocks where each block is treated by a process .

Definition 5.2.3 (Node of type 3 [2])**.** A node is of type 3 is the root node which is very large and is partitioned in a two-dimensional block cyclic way, where each block is treated by a process.

An example of a supernodal elimination tree processed by distributed system is shown in Figure 5.2. Note that if the size of the root node is smaller than the value of some computer parameter defined inside MUMPS, the root node will be considered as node of type 2.

When a node is processed by more than one processors at the same time, these processors will be divided into one master, which is chosen by the host, and multiple slaves. The master is responsible for handling fully summed rows. It performs pivotal operations and carries out numerical factorization. In contrast, the slave processors are tasked with performing updates on the partly summed rows. In the following we will see how the assignment is done for nodes of type 2 and of type 3.

1. Nodes of type 2: Denote the node that needs to be factorized by S_p. At execution time, the master of node S_p receives information from the masters of its son nodes describing the structure of the contribution blocks of the corresponding son nodes. Based on this information, the master of node S_p decides the exact structure of the

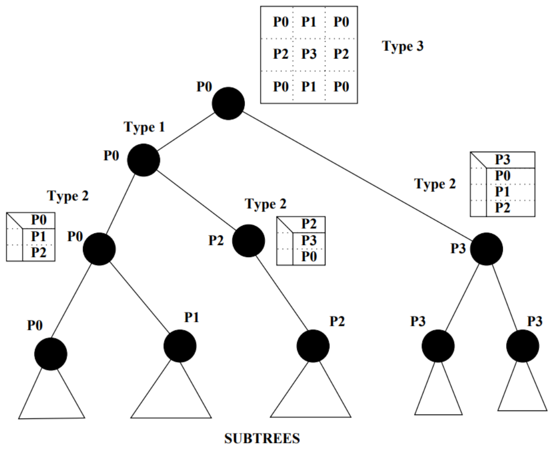


Figure 5.2: Distribution of the computations of a supernodal elimination tree over the four processors P0, P1, P2 and P3 [2]

frontal matrix. Then it decides which slaves will participate in the factorization of this node. The master of the son nodes receive the information assigned from the master of the parent node S_p and then send the entries in its contribution block to these slaves. Next the frontal matrix can be processed in parallel.

2. Node of type 3: Before the factorization, the frontal matrix of the root node is divided and assigned to the available processes in a two-dimensional grid, which completely determines the assignment of these slaves. As soon as the required data, such as matrix entries and contribution blocks from son nodes, becomes available during the factorization phase, it is assembled. Each son node has certain variables that have not been fully processed. The master of the root node collects the index information of all the delayed variables from its son nodes. Based on the collected index information, the master process constructs the final structure of the frontal matrix associated with the root node. Then, the symbolic information is is broadcast from the master process to all the slave processes involved in the factorization. Using this information, the slaves can make final adjustments to its own structures, and then perform calculations in parallel.

It can be seen that the root node of type 3 is completely static in assigning slaves and are fixed before the factorization. In contrast, the assignment of node of type 2 is dynamic, and which slaves will participate in the factorization of this node is determined during the decomposition according to the specific structure of the frontal matrix.

However, the elimination of fully summed rows, performed by a single processor, can potentially hinder the scalability or efficiency of the overall processing when dealing with nodes of type 2 in the algorithm. To overcome this problem, the node can be divided into smaller parts. Taking efficiency into consideration, a node in the algorithm should only be considered for dividing into smaller parts if its distance from the root is not greater than the calculated value, $\log_2(\text{NPROCS} - 1)$, where NPROCS denots the number of the processors.

We have an initial node consisting of NFRONT elements, with NPIV being the number of pivots within this node. If NFRONT $-$ NPIV/2 is too large, the node will be divided into smaller parts. If the numbers of flops performed by the master and by a slave satisfy some relationships, the node can be further split into a son node of size NFRONT with NPIV/2 pivots, and a parent node of size NFRONT $-$ NPIV, with (NPIV/2) pivots. If the split nodes are still very large, then continue to repeat this operation to split them, until the matrices are not so large anymore.

5.3 Solution Phase

In the solution phase, the host processor shares the right-hand side vector b with the other processors. Each processor utilizes the distributed factors calculated during the factorization phase to compute its portion of the solution vector x. The distribution of the factors would not be changed as generated in the factorization phase. Finally, the individual parts of the solution vector are assembled on the host processor to obtain the complete solution vector.

Chapter 6

PARDISO

PARDISO (Parallel Direct Sparse Solver for Irregularly Structured Matrices) was developed by the Parallel Algorithms and Numerical Software (PANOS) group at the University of Basel, Switzerland. It is an implemention of left-right looking method. Instead of distributed memory system used by MUMPS, the target technique for PARDISO is a shared memory system. Distributed memory and shared memory refer to how memory is shared and managed in a parallel computing system. In a distributed memory system, each processor has its own private memory, and data must be explicitly passed between processors, while processors can directly access and manipulate the same data in a shared memory system. In order to minimize synchronization and cache conflicts in shared memory systems, two-level scheduling methods are employed. Following [18], [17] and [19], we introduce the left-looking algorithm at first, followed by the left-right-looking method and the two-level scheduling. The left-right-looking algorithm is obtained by adding pipelining parallelism to the left-looking algorithm. We will see how pipelining parallelism is performed and how cache conflicts can be avoided.

6.1 Preprocessing

Let $\mathrm{A}_1 \leftarrow \mathrm{D}_r\mathrm{P}_r\mathrm{A}\mathrm{D}_c$ according to [18]. P_r is the row permutation matrix which is generated using the maximal matching algorithm [10] to maximize the absolute value of the product of the diagonal entries in the matrix $\mathrm{P}_r\mathrm{A}$. D_r and D_c are diagonal scaling matrices, which are are chosen such that the diagonal entries of A_1 have an absolute value of 1, and all its off-diagonal entries have an absolute value that is less than or equal to 1.

After this fill-in reducing $\mathrm{A}_2 \leftarrow \mathrm{P}_{fill} \cdot \mathrm{A}_1 \cdot \mathrm{P}_{fill}^{\mathrm{T}}$ can be proceeded, where P_{fill} can be any fill-reducing ordering based on the structure of $\mathrm{A}_1 + \mathrm{A}_1^{\mathrm{T}}$, e.g. nested dissection.

For numerical pivoting, unlike UMFPACK and MUMPS, PARDISO performs block supernode diagonal pivoting, where rows and columns of a supernode can be interchanged without causing an increase in the overall fill-in. However, there may be situations where the factorization algorithm encounters a supernode that cannot be successfully factored using the regular supernode pivoting strategy alone. In such cases, PARDISO employs a pivot perturbation strategy. This strategy introduces small perturbations to the pivot values to make the factorization process feasible. To be specific, the growth of diagonal

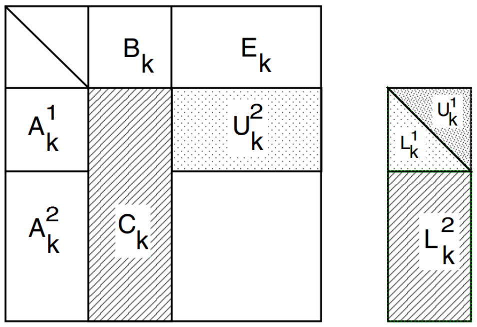


Figure 6.1: A block partitioning [18]

element is controlled to keep pivots from getting too small. If the absolute value of a diagonal element l_{ii} is less than a constant threshold of $\alpha = \epsilon \cdot ||\mathrm{A}_2||_\infty$, where ϵ is the machine precision, then we will turn it into $sign(l_{ii})\epsilon \cdot ||\mathrm{A}_2||_\infty$.

6.2 Left-Right-Looking Strategy

For LU factorization here, the form of JIK-SDOT factorization algorithm is chosen. A detailed description of all possible forms of dense LU factorization can be found in [7]. At the kth step of the elimination process, a block partitioning is depicted in Figure 6.1. C_k and U_k^2 are respectively a block of columns of L and a block of rows of U. The blocks above and to the left of them are the parts that have been factorized. Then the computation of C_k and U_k^2 requires the following operations [18]:

1. External factorization:

$$\mathrm{C}_k \leftarrow \mathrm{C}_k - \begin{bmatrix} \mathrm{A}_k^1 \\ \mathrm{A}_k^2 \end{bmatrix} \mathrm{B}_k \tag{6.1}$$

$$\mathrm{U}_k^2 \leftarrow \mathrm{U}_k^2 - \mathrm{A}_k^1 \mathrm{E}_k \tag{6.2}$$

2. Internal factorization:

$$\mathrm{L}_k^2 \leftarrow \mathrm{L}_k^2 (\mathrm{U}_k^1)^{-1} \tag{6.3}$$

$$\mathrm{U}_k^2 \leftarrow (\mathrm{L}_k^1)^{-1} \mathrm{U}_k^2 \tag{6.4}$$

   where L_k^1 and U_k^1 are obtained by factorization in the right part of Figure 6.1.

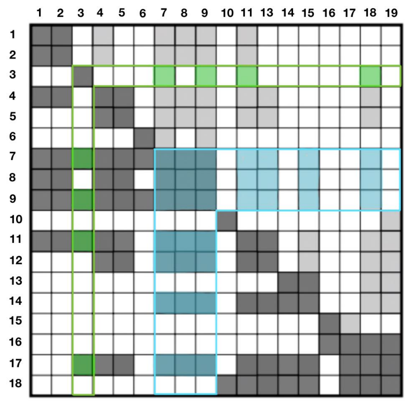
In the left-looking algorithm, all the external factorizations required by a node are performed together. A node corresponds to a set in which all nodes related to its external factorization are stored. When it is the turn for node J to be factorized, these external factorizations are applied sequentially. When all the external factorization is completed, the internal factorization is started. After it performs an external factorization with K, node K will be passed to the corresponding set of the next node next(J,K) which needs an external factorization with K. Similarly, after node J performs an internal factorization, J will be passed to the corresponding set of the next node next(J,K) which needs an external factorization with J. The operator next is defined explicitly as [18]

$$\text{next}(\text{J}, \text{K}) = \{\text{Q} \mid i \in \text{Q}, i \in \min\{l_{i,k} \neq 0, i > j, j \in \text{J}, k \in \text{K}\}\} \tag{6.5}$$
$$\text{next}(\text{J}, \text{J}) = \{\text{Q} \mid i \in \text{Q}, i \in \min\{l_{i,j} \neq 0, i > j, j \in \text{J}\}\}. \tag{6.6}$$

The set corresponding to each node is initially the empty set, which is expanded during the preceding node factorization as just described.

In the left-right-looking algorithm, all the external factorizations required by a node will no longer only be done together. When a node is factorized, all nodes affected by it receive the message that it has been factorized, and can then perform their respective external updates. This is the case of pipelining parallelism, which works with many processors. Algorithm 7 outlines the process, starting from leaves of the elimination tree and proceeding towards the root. In this algorithm, a panel is a subset of a supernode. By decomposing a supernode into panels, the factorization algorithm can be parallelized and executed more efficiently.

Example 6.2.1. Consider a matrix A with the L+U structure and supernodal elimination tree in Figure 6.2. Before internal factorization of S(7, 8, 9), the external factorization of



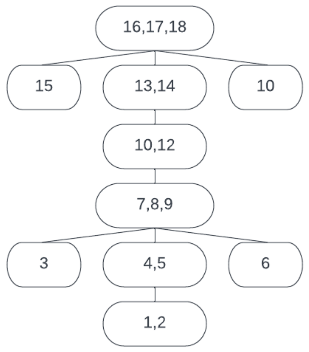


Figure 6.2: L+U structure [17] and supernodal elimination tree of matrix A

S(7, 8, 9) with S(1, 2), S(3), S(4, 5) and S(6) should be performed first. For example after proforming the external factorization of supernode S(7, 8, 9) with supdernode S(3), the block of columns of L and a block of rows of U are respectively

$$\begin{bmatrix} a_{7,7} & a_{7,9} \\ a_{9,7} & a_{9,9} \\ a_{11,7} & a_{11,9} \\ a_{17,7} & a_{17,9} \end{bmatrix} - \begin{bmatrix} a_{3,7} & a_{3,9} \end{bmatrix} \begin{bmatrix} a_{7,3} \\ a_{9,3} \\ a_{11,3} \\ a_{17,3} \end{bmatrix} \tag{6.7}$$

$$\begin{bmatrix} a_{7,11} & a_{7,18} \\ a_{9,11} & a_{9,18} \end{bmatrix} - \begin{bmatrix} a_{3,11} & a_{3,18} \end{bmatrix} \begin{bmatrix} a_{7,3} \\ a_{9,3} \end{bmatrix}, \tag{6.8}$$

Algorithm 7 Left-right-looking algorithm [17]

Input: Sparse matrix A

Output: LU data structure of the input matrix A

```
1:  $\mathrm{L} \leftarrow 0,\ \mathrm{U} \leftarrow 0$
2:  scatter nonzero entries from A into L and U
3:  **for** $\mathrm{J} = 1, ..., \#\mathrm{panels}$ **do**
4:      $S_{\mathrm{J}} \leftarrow \emptyset$
5:  **end for**
6:  $\mathrm{Q} = \{\text{leaves of the elimination tree}\}$
7:  **for** $\mathrm{P} = 1, ..., \#\mathrm{processes}$ **do**
8:      **while** $\mathrm{Q} \neq \emptyset$ **do**
9:          lock-1
10:         $\mathrm{Q} \leftarrow \mathrm{Q} \setminus \{\mathrm{J}\}$
11:         unlock-1
12:         **while** $\exists$ supernode K in $S_{\mathrm{J}}$ **do**
13:             lock-2-A
14:             $S_{\mathrm{J}} \leftarrow S_{\mathrm{J}} \setminus \{\mathrm{K}\}$
15:             unlock-2-A
16:             perform external factorization of J with K
17:         **end while**
18:         perform internal factorization of J
19:         lock-2-B
20:         **for** $\mathrm{I} = \mathrm{J}, ..., \#\mathrm{panels}$ **do**                ▷ Right-looking
21:             $\mathrm{Q} \leftarrow \mathrm{next}(\mathrm{I}, \mathrm{J})$
22:             $S_{\mathrm{Q}} \leftarrow S_{\mathrm{Q}} \cup \{\mathrm{K}\}$
23:         **end for**
24:         unlock-2-B
25:     **end while**
26: **end for**
```

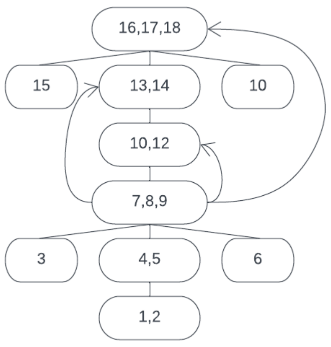


Figure 6.3: Data movement of the right-looking phase

where $a_{i,j}$ represents the current value of the entries in the location (i, j) before the operation. The internal factorization includes the operations

$$\begin{bmatrix} a_{11,7} & a_{11,8} & a_{11,9} \\ a_{12,7} & a_{12,8} & a_{12,9} \\ a_{14,7} & a_{14,8} & a_{14,9} \\ a_{17,7} & a_{17,8} & a_{17,9} \end{bmatrix} \begin{bmatrix} a_{7,7} & a_{7,8} & a_{7,9} \\ & a_{8,8} & a_{8,9} \\ & & a_{9,9} \end{bmatrix}^{-1} \tag{6.9}$$

$$\begin{bmatrix} 1 & & \\ a_{8,7} & 1 & \\ a_{9,7} & a_{9,8} & 1 \end{bmatrix}^{-1} \begin{bmatrix} a_{7,11} & a_{7,13} & a_{7,15} & a_{7,18} \\ a_{8,11} & a_{8,13} & a_{8,15} & a_{8,18} \\ a_{9,11} & a_{9,13} & a_{9,15} & a_{9,18} \end{bmatrix}. \tag{6.10}$$

After all the factorization of S$(7, 8, 9)$ is performed, its contributions will be sent to S$(10, 12)$, S$(13, 14)$, S$(16, 17, 18)$ without wait. The factorization of these supernodes with S$(7, 8, 9)$ can then start by different processes. The data movement is shown in Figure 6.3.

The previous left-right-looking algorithm is a one-level scheduling algorithm, where the one-level scheduling represents a centralized pool of tasks that is globally accessible to all processes. In order to avoid synchronization and cache conflicts as much as possible, a two-level scheduling method is introduced. Two-level scheduling aims to maximize cache utilization by assigning each process a subtree that fits within the cache size available to that process. By ensuring that the data accessed by a process remains within its cache, the likelihood of cache conflicts is reduced. Literally, the two-level scheduling method is divided into two levels, namely first level and second level. In the second level we have a pool of tasks global to all processes, while in the first one the pool of tasks is local to each process. Figure 6.4 and Algorithm 8 outlines the process.

In the first level, a one-level left-right-looking algorithm is performed for all mutually disjoint subtrees. The update information of each subtree is synchronized only to the root node of the subtree and then each subtree is dynamically mapped to a process. To achieve this, locks are used in this algorithm. For example, when a process acquires a lock-2-a lock, it can perform operations related to the subtree, while other processes must wait for

Algorithm 8 Two-level scheduling with a left-right-looking algorithm [17]

Input: A sparse matrix

Output: LU data structure of the input matrix

```
 1: $Q_S \leftarrow \{$local independent subtrees$\}$          ▷ Beginning of the first level
 2: while $Q_S \neq \emptyset$ do
 3:     lock-1
 4:     remove a subtree T from $Q_S$
 5:     unlock-1
 6:     lock-2-a
 7:     factorize T with left-right-looking algorithm
 8:     unlock-2-a
 9: end while
10: $Q_R \leftarrow \{$root nodes$\}$          ▷ Beginning of the second level
11: while $Q_R \neq \emptyset$ do
12:     lock-2-b
13:     remove a root node J from $Q_R$
14:     unlock-2-b
15:     if J recieves updates from previous node K then
16:         if process has to wait then
17:             lock-2-c
18:             put J back to $Q_R$
19:             unlock-2-c
20:             go to the beginning of second level
21:         else
22:             perform external factorization of J with K
23:         end if
24:     end if
25:     perform internal factorization of J
26:     lock-2-d
27:     perform right-looking of J
28:     unlock-2-d
29: end while
```

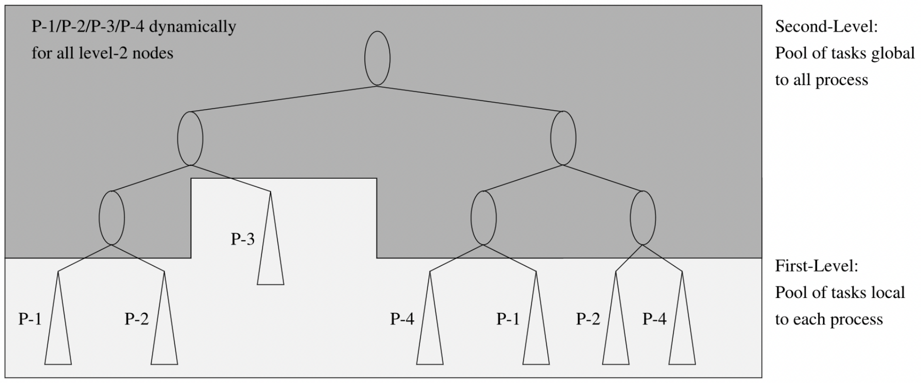


Figure 6.4: A mapping of a hypothetical elimination tree with the two-level scheduling over four processors [19]

the release of that lock to perform the corresponding operations. Because of this, synchronization is limited to the local scope of each subtree, and only the root node needs to be involved in the synchronization process. In addition, each process operates independently on its assigned subtree, minimizing the need for synchronization and contention for shared resources. The first-level scheduling is also called process-to-subtree scheduling, since all nodes in a subtree are computed by a single process.

Once all the subtrees are factorized, we move to the second level, which performs a one-level left-right looking algorithm on all the root nodes of the subtrees, which is sufficiently large. The second level scheduling is also called process-to-process scheduling of the root supernodes. These root nodes are of type 2 and stored in the global centralized queue Q_R, which allows any process to handle the available nodes. If a process assigned to a particular node J needs to wait for other nodes to complete their factorization, the current node J is put back into the queue Q_R. This ensures that the process can immediately start working on another available node, rather than being idle while waiting for dependencies.

Chapter 7

Comparisons

In this chapter, we will see comparisons of the actual performance of these three solvers following [15]. The three direct solvers use different ordering packages, and we will first take a look at their choices of ordering packages. The problems that these three solvers are solving are three-dimensional Navier-Stokes finite element formulations, see [15].

These experiments were conducted on a sequential 64-bit machine with 16 GB of RAM, as direct solvers have high memory requirements. In particular, the calculations were performed on a Windows machine equipped with an Intel Xeon processor. It is worth noting that when dealing with large three-dimensional problems, in addition to utilizing a 64-bit machine with a substantial amount of RAM, out-of-core solvers and parallel solvers can also be employed. These additional approaches provide alternative strategies for solving such problems efficiently.

7.1 Ordering

For unsymmetric matrices, UMFPACK employs the CHOLAMD ordering method as its default choice. MUMPS provides a range of inbuilt ordering packages like AMD, QAMD and AMF, along with the ability to interface with external ordering methods like METIS and PORD, enabling users to choose the most suitable ordering strategy for their specific problem. The results presented in Table 7.1 indicate that METIS produces the most optimal outcomes, in both CPU time and memory usage. PARDISO includes two ordering methods within the solver, which are minimum degree ordering and METIS ordering. Table 7.2 shows that using METIS ordering improves the solver's efficiency significantly. Based on these results, the author of [15] used METIS ordering for all subsequent runs of the MUMPS and PARDISO solver, while used CHOLAMD for UMFPACK solver.

7.2 Computation Time and Memory Requirement

Table 7.3 shows the comparison of computational time and memory requirement of different solvers. The computational time is split into 4 phases, namely matrix assembly, analysis, numerical factorization and solve, where matrix assembly refers to the process of

Ordering	#dof's	Cpu time (sec)	Memory (GB)
AMD	89373	142.8	4.06
QAMD	89373	142.75	4.04
AMF	89373	105.7	3.28
PORD	89373	86.6	3.18
METIS	89373	59.01	3.02

Table 7.1: Performance of different orderings for MUMPS solver for $30 \times 30 \times 30$ grid [15]

Ordering	#dof's	Cpu time (sec)	Memory (GB)
MD	89373	162	2.78
METIS	89373	60	1.42

Table 7.2: Performance of different orderings for PARDISO solver [15]

creating the system matrix that represents the equations to be solved.

	Cpu time (sec)					Memory (MB)
Solver	Matrix assembly	Analysis	Numerical factorization	Solve	Total	
UMFPACK	4	0.84	154.8	1.1	160.74	5520
MUMPS	4	1.51	53.3	0.58	59.39	3020
PARDISO	4	2.19	52.6	0.47	59.26	1420

Table 7.3: Computational time and memory requirements for $30 \times 30 \times 30$ grid [15]

The numerical factorization phase is found to be the most time-consuming step for all solvers according to Table 7.3. In addition, UMFPACK takes around twice as long as MUMPS or PARDISO to complete the computation, while MUMPS and PARDISO have comparable performance in terms of computation time.

In terms of memory requirements, PARDISO requires the smallest amount of memory among all solvers, while UMFPACK requires the largest amount. Specifically, UMFPACK requires about 4 times more memory than PARDISO. To summarize, among the solvers considered, PARDISO is the best choice in terms of memory usage.

Bibliography

[1] Patrick R. Amestoy, Timothy A. Davis, and Iain S. Duff. “An Approximate Minimum Degree Ordering Algorithm”. In: *SIAM Journal on Matrix Analysis and Applications* 17.4 (1996), pp. 886–905. DOI: 10.1137/S0895479894278952. URL: https://doi.org/10.1137/S0895479894278952.

[2] Patrick R. Amestoy et al. “A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling”. In: *SIAM Journal on Matrix Analysis and Applications* 23.1 (2001), pp. 15–41. DOI: 10.1137/S0895479899358194. URL: https://doi.org/10.1137/S0895479899358194.

[3] Matthias Bollhöfer et al. “State-of-the-Art Sparse Direct Solvers”. In: *Parallel Algorithms in Computational Science and Engineering*. Ed. by Ananth Grama and Ahmed H. Sameh. Cham: Springer International Publishing, 2020, pp. 3–33. ISBN: 978-3-030-43736-7. DOI: 10.1007/978-3-030-43736-7_1. URL: https://doi.org/10.1007/978-3-030-43736-7_1.

[4] Tim Davis. “A column pre-ordering strategy for the unsymmetric-pattern multifrontal method. ACM Trans”. In: *ACM Trans. Math. Softw.* 30 (June 2004), pp. 165–195. DOI: 10.1145/992200.992205.

[5] Timothy A. Davis, Sivasankaran Rajamanickam, and Wissam M. Sid-Lakhdar. “A survey of direct methods for sparse linear systems”. eng. In: *Acta numerica* 25 (2016), pp. 383–566. ISSN: 0962-4929.

[6] Timothy A. Davis et al. “A Column Approximate Minimum Degree Ordering Algorithm”. In: *ACM Trans. Math. Softw.* 30.3 (Sept. 2004), pp. 353–376. ISSN: 0098-3500. DOI: 10.1145/1024074.1024079. URL: https://doi.org/10.1145/1024074.1024079.

[7] Michel J. Daydé and Iain S. Duff. “Level 3 Blas in Lu Factorization On the Cray-2, Eta-10P, and Ibm 3090-200/Vf”. In: *International Journal of High Performance Computing Applications* 3 (1989), pp. 40–70.

[8] James W. Demmel et al. *A Supernodal Approach to Sparse Partial Pivoting.* Tech. rep. USA, 1995.

[9] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices.* Oxford University Press, Jan. 2017. ISBN: 9780198508380. DOI: 10.1093/acprof:oso/9780198508380.001.0001. URL: https://doi.org/10.1093/acprof:oso/9780198508380.001.0001.

[10] Iain S. Duff and Jacko Koster. “The Design and Use of Algorithms for Permuting Large Entries to the Diagonal of Sparse Matrices”. In: *SIAM Journal on Matrix Analysis and Applications* 20.4 (1999), pp. 889–901. DOI: `10.1137/S0895479897317661`. URL: `https://doi.org/10.1137/S0895479897317661`.

[11] Anshul Gupta. “Fast and Effective Algorithms for Graph Partitioning and Sparse-Matrix Ordering”. In: *IBM J. Res. Dev.* 41.1–2 (Jan. 1997), pp. 171–183. ISSN: 0018-8646. DOI: `10.1147/rd.411.0171`. URL: `https://doi.org/10.1147/rd.411.0171`.

[12] Anshul Gupta. “Recent Advances in Direct Methods for Solving Unsymmetric Sparse Systems of Linear Equations”. In: *ACM Trans. Math. Softw.* 28.3 (Sept. 2002), pp. 301–324. ISSN: 0098-3500. DOI: `10.1145/569147.569149`. URL: `https://doi.org/10.1145/569147.569149`.

[13] George Karypis and Vipin Kumar. “A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs”. In: *SIAM Journal on Scientific Computing* 20.1 (1998), pp. 359–392. DOI: `10.1137/S1064827595287997`. URL: `https://doi.org/10.1137/S1064827595287997`.

[14] Joseph W. H. Liu. “The Multifrontal Method for Sparse Matrix Solution: Theory and Practice”. In: *SIAM Rev.* 34 (1992), pp. 82–109.

[15] M. P. Raju and S. K. Khaitan. “High Performance Computing of Three-Dimensional Finite Element Codes on a 64-bit Machine”. In: *Journal of Applied Fluid Mechanics* 5.2 (2012), pp. 123–132. ISSN: 1735-3572. DOI: `10.36884/jafm.5.02.12174`. URL: `https://www.jafmonline.net/article_1311.html`.

[16] Yousef Saad. *Iterative Methods for Sparse Linear Systems.* Second. Society for Industrial and Applied Mathematics, 2003. DOI: `10.1137/1.9780898718003`. URL: `https://epubs.siam.org/doi/abs/10.1137/1.9780898718003`.

[17] O. Schenk, K. Gärtner, and W. Fichtner. “Efficient Sparse LU Factorization with Left-Right Looking Strategy on Shared Memory Multiprocessors”. In: *BIT Numerical Mathematics* 40.1 (Mar. 2000), pp. 158–176. ISSN: 1572-9125. DOI: `10.1023/A:1022326604210`. URL: `https://doi.org/10.1023/A:1022326604210`.

[18] Olaf Schenk and Klaus Gärtner. “Solving Unsymmetric Sparse Systems of Linear Equations with PARDISO”. In: *Computational Science — ICCS 2002.* Ed. by Peter M. A. Sloot et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 355–363. ISBN: 978-3-540-46080-0.

[19] Olaf Schenk and Klaus Gärtner. “Two-level dynamic scheduling in PARDISO: Improved scalability on shared memory multiprocessing systems”. In: *Parallel Computing* 28.2 (2002), pp. 187–197. ISSN: 0167-8191. DOI: `https://doi.org/10.1016/S0167-8191(01)00135-1`. URL: `https://www.sciencedirect.com/science/article/pii/S0167819101001351`.