THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :
L'Institut National Polytechnique de Toulouse (INP Toulouse)

Présentée et soutenue par :

Clément Weisbecker

Le 28 Octobre 2013

Titre :

Improving multifrontal solvers by means of algebraic Block Low-Rank representations

Amélioration des solveurs multifrontaux à l'aide de representations algébriques rang-faible par blocs

Ecole doctorale et discipline ou spécialité :
ED MITT : Domaine STIC : Sûreté du logiciel et calcul haute performance

Unité de recherche :
IRIT - UMR 5505

Directeurs de Thèse :
Patrick R. Amestoy (INPT-ENSEEIHT-IRIT)
Alfredo Buttari (CNRS-IRIT)

Rapporteurs :
Timothy A. Davis (University of Florida)
Esmond G. Ng (Lawrence Berkeley National Laboratory)

Autres membres du jury :
Cleve Ashcraft (Livermore Software Technology Corporation)
Olivier Boiteau (Electricité De France)
Iain S. Duff (Rutherford Appleton Laboratory)
Guillaume Sylvand (European Aeronautic Defence and Space company)

to my grand-parents Julia and Lucien
to my friend Marthe

Résumé

Nous considérons la résolution de très grands systèmes linéaires creux à l'aide d'une méthode de factorisation directe appelée méthode multifrontale. Bien que numériquement robustes et faciles à utiliser (elles ne nécessitent que des informations algébriques : la matrice d'entrée A et le second membre b, même si elles peuvent exploiter des stratégies de prétraitement basées sur des informations géométriques), les méthodes directes sont très coûteuses en termes de mémoire et d'opérations, ce qui limite leur applicabilité à des problèmes de taille raisonnable (quelques millions d'équations). Cette étude se concentre sur l'exploitation des approximations de rang-faible dans la méthode multifrontale, pour réduire sa consommation mémoire et son volume d'opérations, dans des environnements séquentiel et à mémoire distribuée, sur une large classe de problèmes.

D'abord, nous examinons les formats rang-faible qui ont déjà été développé pour représenter efficacement les matrices denses et qui ont été utilisées pour concevoir des solveur rapides pour les équations aux dérivées partielles, les équations intégrales et les problèmes aux valeurs propres. Ces formats sont hiérarchiques (les formats $\mathcal{H}$ et HSS sont les plus répandus) et il a été prouvé, en théorie et en pratique, qu'ils permettent de réduire substantiellement les besoins en mémoire et opération des calculs d'algèbre linéaire. Cependant, de nombreuses contraintes structurelles sont imposées sur les problèmes visés, ce qui peut limiter leur efficacité et leur applicabilité aux solveurs multifrontaux généraux.

Nous proposons un format plat appelé Block Rang-Faible (BRF) basé sur un découpage naturel de la matrice en blocs et expliquons pourquoi il fournit toute la flexibilité nécéssaire à son utilisation dans un solveur multifrontal général, en terme de pivotage numérique et de parallélisme. Nous comparons le format BRF avec les autres et montrons que le format BRF ne compromet que peu les améliorations en mémoire et opération obtenues grâce aux approximations rang-faible. Une étude de stabilité montre que les approximations sont bien contrôlées par un paramètre numérique explicite appelé le seuil rang-faible, ce qui est critique dans l'optique de résoudre des systèmes linéaires creux avec précision. Ensuite, nous expliquons comment les factorisations exploitant le format BRF peuvent être efficacement implémentées dans les solveurs multifrontaux. Nous proposons plusieurs algorithmes de factorisation BRF, ce qui permet d'atteindre différents objectifs.

Les algorithmes proposés ont été implémentés dans le solveur multifrontal MUMPS. Nous présentons tout d'abord des expériences effectuées avec des équations aux dérivées partielles standardes pour analyser les principales propriétés des algorithms BRF et montrer le potentiel et la flexibilité de l'approche ; une comparaison avec un code basé sur le format HSS est également fournie. Ensuite, nous expérimentons le format BRF sur des problèmes variés et de grande taille (jusqu'à une centaine de millions d'inconnues), provenant de nombreuses applications industrielles. Pour finir, nous illustrons l'utilisation de notre approche en tant que préconditionneur pour la méthode du Gradient Conjugué.

Mots-clés : matrices creuses, systèmes linéaires creux, méthodes directes, méthode multifrontale, approximations rang-faible, équations aux dérivées partielles elliptiques.

Abstract

We consider the solution of large sparse linear systems by means of direct factorization based on a multifrontal approach. Although numerically robust and easy to use (it only needs algebraic information: the input matrix A and a right-hand side b, even if it can also digest preprocessing strategies based on geometric information), direct factorization methods are computationally intensive both in terms of memory and operations, which limits their scope on very large problems (matrices with up to few hundred millions of equations). This work focuses on exploiting low-rank approximations on multifrontal based direct methods to reduce both the memory footprints and the operation count, in sequential and distributed-memory environments, on a wide class of problems.

We first survey the low-rank formats which have been previously developed to efficiently represent dense matrices and have been widely used to design fast solutions of partial differential equations, integral equations and eigenvalue problems. These formats are hierarchical ($\mathcal{H}$ and Hierarchically Semiseparable matrices are the most common ones) and have been (both theoretically and practically) shown to substantially decrease the memory and operation requirements for linear algebra computations. However, they impose many structural constraints which can limit their scope and efficiency, especially in the context of general purpose multifrontal solvers.

We propose a flat format called Block Low-Rank (BLR) based on a natural blocking of the matrices and explain why it provides all the flexibility needed by a general purpose multifrontal solver in terms of numerical pivoting for stability and parallelism. We compare BLR format with other formats and show that BLR does not compromise much the memory and operation improvements achieved through low-rank approximations. A stability study shows that the approximations are well controlled by an explicit numerical parameter called low-rank threshold, which is critical in order to solve the sparse linear system accurately. Details on how Block Low-Rank factorizations can be efficiently implemented within multifrontal solvers are then given. We propose several Block Low-Rank factorization algorithms which allow for different types of gains.

The proposed algorithms have been implemented within the MUMPS (MUltifrontal Massively Parallel Solver) solver. We first report experiments on standard partial differential equations based problems to analyse the main features of our BLR algorithms and to show the potential and flexibility of the approach; a comparison with a Hierarchically SemiSeparable code is also given. Then, Block Low-Rank formats are experimented on large (up to a hundred millions of unknowns) and various problems coming from several industrial applications. We finally illustrate the use of our approach as a preconditioning method for the Conjugate Gradient.

Keywords: sparse matrices, direct methods for linear systems, multifrontal method, low-rank approximations, high-performance computing, parallel computing.

Acknowledgements

I warmly thank my two daily supervisors, Patrick Amestoy and Alfredo Buttari, as well as Cleve Ashcraft and Jean-Yves L'Excellent, for their precious help during these three years. They have always been very enthusiastic to answer my questions and propose new ideas. It was a great source of motivation to learn from researchers like you. Also, thank you for the incredible amount of work you put into this thesis, for the pricessless discussions we had together and for the wise advice. I want to thank Olivier Boiteau who convinced EDF to grant this work, provided real-life problems and introduced me to many interesting industrial issues. Thank you to Frank Hülsemann.

I would like to thank all the people I worked with since 2010. Firstly, thank you to Stéphane Operto, Jean Virieux and Romain Brossier for taking the time to explain me what is behind the word *geophysics* and to experiment with our solver in your applications. I am grateful to Sherry Li and Esmond Ng who hosted me twice at the Berkeley Lab. I want to thank Artem Napov and François-Henry Rouet from whom I learnt a lot about low-rank approximations. I also thank Maurice Brémond, Serge Gratton, Guillaume Joslin, Marième Ngom, Nicolas Tardieu and David Titley-Peloquin.

I want to thank the two referees of this dissertation, Tim Davis and Esmond Ng, for taking the time to review these pages and make precious remarks. I am very honoured that you travelled overseas to attend the defense. I also thank the other members of the jury, Iain Duff and Guillaume Sylvand.

I also owe many thanks to my co-workers, who made stressful days better ones, thanks to questionably funny jokes and games: François-Henry Rouet, Mohamed Zenadi, Gaëtan André and Florent Lopez. Thank you to the APO team at IRIT, more particularly thank you to Chiara Puglisi and Daniel Ruiz. Also, thank you to our dear Sylvie Armengaud (I know I was not that annoying) and Sylvie Eichen.

I want to thank my housemates in Berkeley, where I wrote most of this dissertation: Shayda, Kate, François-Henry (again!), Evan, Corinne and Elliott. Thank you to Jocelyne and Hervé Lecompte. I would like to warmly thank my closest friends outside from the lab, who have been (and are) of great support in any situation: Arno, Cloé, Maya, Lars, Aurélien, Marion, Gilles, Alice, Zacharie, Yohann, Pauline, Jéré, Charlot, Anne, Bapton, Tregouze, Benny, Louis, Astrid, Marco, Fred, Marie, Laure, Ben, Claire, Manu, Emilie and Oliver (in random order).

I am deeply grateful to my loving parents and brothers for their support during these numerous years of study. You are really part of this work.

Finally, I want to lovingly thank Charlotte for being such a great girlfriend, for all the smiles and funny faces when I was tired of working and for having been so understanding during these three years.

Contents

Chapter 1

General introduction

1.1 Context

Solving large sparse linear systems is a very important issue in academic research and in industrial applications, as a part of even more complex problems. Partial differential equations for structural dynamics, circuit design and numerous other fields lead to such sparse systems. Modern science and new knowledge in various domains tend to make the systems larger and larger. Newest applications commonly require the solution of linear systems with many millions of unknowns. This is often a keystone in numerical simulations.

The development of supercomputers has provided us with the opportunity to solve these critical problems but brought some new issues such as memory consumption, accuracy and speed, which are encountered daily by numerical analysts and mathematicians.

In this context, it is critical to adapt and improve the available methods to make these problems feasible. Solving large sparse linear systems involves many technical aspects, so we start with a survey of direct methods in Section 1.2, in which we first present them in the context of dense matrices, and then naturally come to sparse direct methods, and more particularly, multifrontal methods in Section 1.3. As a way to improve them, low-rank approximations techniques will then be introduced in Section 1.5, as well as some of the most important ways to exploit low-rank approximations for solving sparse linear systems. This introductive section ends in Section 1.6 with a presentation of the experimental environment of this work: the software involved, the test problems studied and the computational systems used.

1.2 Solving sparse linear systems with direct methods

We are interested in efficiently computing the solution of large sparse linear systems which arise from various applications such as mechanics and fluid dynamics. A sparse linear system is usually written in the form:

$$Ax = b\,, \tag{1.1}$$

where A is a sparse matrix of order n, x is the unknown vector of size n and b is the right-hand side vector of size n. In Wilkinson's definition, a sparse matrix is “any matrix with enough zeros that it pays off to take advantage of them”; in modern numerical computing, A can be extremely large (a few hundred billions of equations). It is thus critical to adapt dense algorithms to sparse objects, with the underlying objective to avoid storing the zero entries and thus spare useless operations and storage.

This kind of matrices is often encountered in physical simulations. While modeling phenomena such as heat diffusion from a source on a square surface, it is not feasible to compute the temperature on any point of the surface because there is an infinity of points in this continuous domain. Thus, the domain is discretized leading to a relatively small set of unknowns and linear equations. Moreover, due to the nature of physical phenomena (and also due to how the corresponding operators are approximated through the discretizations) these variables are likely to interact only with a few other physical variables (typically, its neighbors), which translates into many zeros in the corresponding matrix. This discretized structure is called a mesh. Note that more than one unknown can be attached to a given meshpoint so that the order of the matrix (i.e., the number of unknowns for the entire problem) may be larger than the number of meshpoints. In physical applications, assuming for simplicity that there is one unknown per mesh point, a coefficient a_{ij} of a matrix can thus be viewed as a representation of the interaction between physical variables i and j of the discretized domain (the mesh).

Two types of methods are commonly used to solve (1.1). *Iterative methods* build a sequence of iterates which hopefully converges to the solution. They have a relatively small memory consumption and provide good scalability in parallel environments but their effectiveness strongly depends on the numerical properties of the problem. *Direct methods* build a factorization of A (e.g., $A = LU$ or $A = QR$) and then solve the problem using the factorized form of A. Their better numerical stability and robustness are widely agreed upon although these techniques often have large memory and computational requirements. The choice of a method is usually not straightforward as it depends on many parameters (the matrix itself, the physical application, the computational resources available).

In this dissertation, we focus on a particular case of direct methods which is based on Gaussian elimination and called the *multifrontal method*, although iterative methods will also be discussed.

1.2.1 Dense direct methods: factorization, solution, error analysis

We focus on direct methods based on Gaussian elimination. The matrix A is factorized into the form LU (in the general case), LDL^T (in the symmetric, indefinite case) or LL^T (in the symmetric positive definite case, commonly referred to as the *Cholesky factorization*), where L is unit lower triangular and U is upper triangular (note that L is not unit in the case of the LL^T Cholesky factorization). We provide in Algorithm 1.1 a simplified sketch of the LU factorization in the dense case, i.e., the nonzero pattern of the matrix is not taken into account. We ignore numerical issues and pivoting for the sake of clarity (this will be discussed later). For this reason, we assume that diagonal entries are always nonzeros. Each step k of the factorization corresponds to the *elimination* of a pivot (i.e., a diagonal entry a_{kk}), which yields a new column of L and a new row of U. Then, the trailing submatrix is modified through a rank-one update. These two operations are denoted by Factor (F) and Update (U).

Algorithm 1.1 Dense LU factorization without pivoting.

```
1:      ▶ Input: a square matrix $A$ of size $n$; $A = [a_{ij}]_{i=1:n,j=1:n}$
2:      ▶ Output: $A$ is replaced by its $LU$ factors
3:
4: for $k = 1$ to $n - 1$ do
5:   Factor: $a_{k+1:n,k}$   $\frac{a_{k+1:n,k}}{a_{kk}}$
6:   Update: $a_{k+1:n,k+1:n}$   $a_{k+1:n,k+1:n} - a_{k+1:n,k}$   $a_{k,k+1:n}$
7: end for
```

In Algorithm 1.1, note that the strictly lower triangular (i.e., excluding the diagonal since $\ i, l_{i,j} = 1$) part of A is replaced by L and the upper triangular part of A is replaced by U. The operation and memory complexities of the dense factorization of a matrix of size n are $O(n^3)$ and $O(n^2)$ [**?**], respectively. Given the factors L and U of the matrix A, the solution of Equation (1.1) can be computed by means of two successive phases, namely the forward elimination (which solves a lower triangular system using L) and the backward substitution (which solves an upper triangular system using U). These two distinct phases are sketched in Algorithm 1.2. Note that the solution computed during the forward elimination is used as a right-hand side in the backward substitution.

Algorithm 1.2 Dense triangular solution through forward elimination and backward substitution.

```
1: Solution of $Ly = b$ for $y$
2: (forward elimination)
3:
4: $y \quad b$
5: for $j = 1$ to $n$ do
6:     for $i = j + 1$ to $n$ do
7:         $y_i \quad y_i - l_{ij} \quad y_j$
8:     end for
9: end for
```

```
1: Solution of $Ux = y$
2: (backward substitution)
3:
4: $x \quad y$
5: for $i = n$ to 1 by $-1$ do
6:     for $j = i + 1$ to $n$ do
7:         $x_i \quad x_i - u_{ij} \quad x_j$
8:     end for
9:     $x_i \quad x_i \ u_{ii}$
10: end for
```

Algorithms 1.1 and 1.2 are called *scalar* (or *point*) because operations are performed on single coefficients. They can be substantially improved by organizing the operations in such a way that most of them can be performed on blocks. This improves the efficiency of the process by means of BLAS 3 [41] operations (an operation defined between B_1 of size $m_1 \quad n_1$, $m_1 > 1$ and $n_1 > 1$, and B_2 of size $m_2 \quad n_2$, $m_2 > 1$ and $n_2 > 1$, such as a matrix-matrix product, is called a BLAS 3 operation) which are more cache-friendly and allow different levels of parallelism. However, both the memory and operations complexities are the same as in the scalar factorization. The blocked version of the dense factorization is presented in Algorithm 1.3 and will be particularly important in this dissertation as the algorithms we propose and study in Sections 2.4 and 3.4 are based on it. Note that at each loop step in Algorithm 1.3, the Factor phase is performed by means of Algorithm 1.1.

Algorithm 1.3 Dense Block LU factorization without pivoting.

```
1:         ▶ Input: a NB   NB-block matrix $A$ of size $n$; $A = [A_{I,J}]_{I=1:NB,J=1:NB}$
2:         ▶ Output: $A$ is replaced with its $LU$ factors
3:
4: for $K = 1$ to $NB$ do
5:     Factor: $A_{K,K} \quad L_{K,K}U_{K,K}$
6:     Solve (compute U): $A_{K,K+1:NB} \quad L_{K,K}^{-1} \quad A_{K,K+1:NB}$
7:     Solve (compute L): $A_{K+1:NB,K} \quad A_{K+1:NB,K} \quad U_{K,K}^{-1}$
8:     Update: $A_{K+1:NB,K+1:NB} \quad A_{K+1:NB,K+1:NB} - A_{K+1:NB,K} \quad A_{K,K+1:NB}$
9: end for
```

Forward elimination and backward substitution also have their blocked equivalence, presented in Algorithm 1.4. These algorithms will be reused in the context of the multi-frontal Block-Low Rank solution phase presented in Section 3.5.

Algorithm 1.4 Dense blocked triangular solution through forward elimination and backward substitution.

```
1: Solution of $Ly = b$ for $y$
2: (forward elimination)
3:
4: $y \quad b$
5: for $J = 1$ to $NB$ do
6:     $x_J \quad L_{J,J}^{-1} \quad x_J$
7:     for $I = J + 1$ to $NB$ do
8:         $y_I \quad y_I - L_{I,J} \quad y_J$
9:     end for
10: end for
```

```
1: Solution of $Ux = y$
2: (backward substitution)
3:
4: $x \quad y$
5: for $I = NB$ to $1$ by $-1$ do
6:     for $J = I + 1$ to $NB$ do
7:         $x_I \quad x_I - U_{I,J} \quad x_J$
8:     end for
9:     $x_I \quad U_{I,I}^{-1} \quad x_I$
10: end for
```

Once the linear system (1.1) has been solved, one usually wants to evaluate the quality of the solution. Due to the floating-point arithmetic, the representation of numbers in computers is indeed inexact, which means operations are also inexact, leading to roundoff errors. The computed solution $\widehat{x}$ is thus not the exact solution of (1.1) but it can be seen as the exact solution of a perturbed linear system written $(A + \quad A)\widehat{x} = b$. How *far* is this perturbed system from the one we wanted to solve is measured by the backward error, which corresponds to the smallest perturbation on A and on b which leads to a linear system whose $\widehat{x}$ is the exact solution [85]. This normwise backward error is defined as follows:

$$\frac{A\widehat{x} - b}{A \quad \widehat{x} \ + \ b}$$

Although the normwise backward error can be used for sparse systems, it does not take into account the fact that zeros are exact zeros, even in floating-point arithmetic. Therefore, it is possible to use a componentwise backward error defined as follows:

$$\max_i \frac{|A\widehat{x} - b|_i}{(|A||\widehat{x}| + |b|)_i}$$

Throughout this dissertation, the normwise backward error and the componentwise backward error will be referred to as the *scaled residual* (SR) and the componentwise scaled residual (CSR), respectively.

Arioli et al. [15] showed that the relative forward error $\frac{x - x}{x}$ (where $\widehat{x}$ is the computed solution and x the exact solution) is bounded by the backward error multiplied by the condition number of the matrix. The direct consequence of this statement is that a bad forward error can be caused either by a large condition number (in which case the problem is called *ill-posed* and is by nature hard to solve) or by an unstable algorithm (in which case even a well-conditioned problem can lead to poor forward error). For this reason, in order to evaluate the numerical quality of our algorithms, scaled residuals and componentwise scaled residuals will be used in this dissertation.

In practice, most significant roundoff errors are caused by adding a large value with a small value. Thus, small pivots should be avoided because they appear in the denominator of operations, increasing the original value. Many solutions have been developed to overcome this issue, such as scaling the input matrix [92] in order to balance globally the numerical values and generally improve the numerical properties of the matrix. However, in many cases, preprocessing facilities are not sufficient to ensure the numerical stability of the algorithm and other strategies have been developed to achieve this goal such as the *numerical pivoting*.

1.2.2 Pivoting for accuracy

Numerical pivoting has been designed to control growth factor and prevent instability of Gaussian elimination. As said before, because most of the instability is due to the selection of small pivots, pivoting aims at avoiding choosing a pivot if it is too small. This yields a factorization where pivots are not eliminated in the natural order as it was presented in Algorithm 1.1. Instead, at step k of the algorithm (where $a_{k,k}$ is supposed to be selected as pivot), the selected pivot is $a_{k,p}$, such that $|a_{k,p}| = \max_{k \quad j \quad n} |a_{k,j}|$. This is called *partial pivoting*. Once the pivot has been found, columns k and p are switched so that $a_{k,p}$ becomes the new diagonal value and the new pivot. All such permutations are stored in order to reapply them during the solution phase. Note that for partial pivoting, the pivot research is, in the literature and in reference codes (i.e., `LAPACK`), done along the column, but we will perform it along the row, as in the equation above (it mainly depends on the storage of the rowwise or columnwise storage of the matrix, aiming at enhancing memory access efficiency). This pivoting strategy is often enough to ensure the numerical stability of the algorithm, although it is in theory not stable. If still not satisfying, one can perform *complete* pivoting where any entry in the whole trailing submatrix can be chosen as a pivot. This strategy is rarely used because it performs more data movements to bring the pivot to the diagonal position by means of rows and columns permutations. More importantly, it requires globally $O(N^3)$ comparisons instead of $O(N^2)$ in case of partial pivoting, which means that the cost of pivoting is roughly comparable to the cost of factorizing the matrix (.

Another pivoting strategy which is sometimes used (especially in sparse factorizations) is the *static pivoting*. As opposed to previous strategies, static pivoting does not perform any permutation: when a pivot is considered too small, it is artificially set to a given value. For instance, if $|a_{k,k}| < \overline{\quad_{mach}}\ ||A||$, then $a_{k,k} \quad \overline{\quad_{mach}}\ ||A||$, where $_{mach}$ is the machine precision. This strategy is simpler, more scalable but also less reliable [10].

Note that for symmetric matrices, pivoting is performed similarly and we want to maintain the symmetric structure of the matrix. Regular pivots as well as *two-by-two* pivots [24, 25, 44] are used, which leads to a block diagonal D matrix in the LDL^T factorization (*two-by-two* pivots are 2 2 diagonal blocks on D).

1.2.3 Graphs, ll-in and dependencies

Sparse matrices being a particular case of dense ones, all the previously presented algorithms and analysis can be applied to sparse matrices too. However, this is not efficient in practice as many computations would be done with zero values. It is possible to take advantage of these zero values in order to avoid useless operations. The main issue is then to maintain the number of zero values as high as possible during the factorization process, because less zero values leads to more storage and more operations. This phenomenon (the creation of new nonzero values) is called *ll-in* and can be conveniently modeled by graphs.

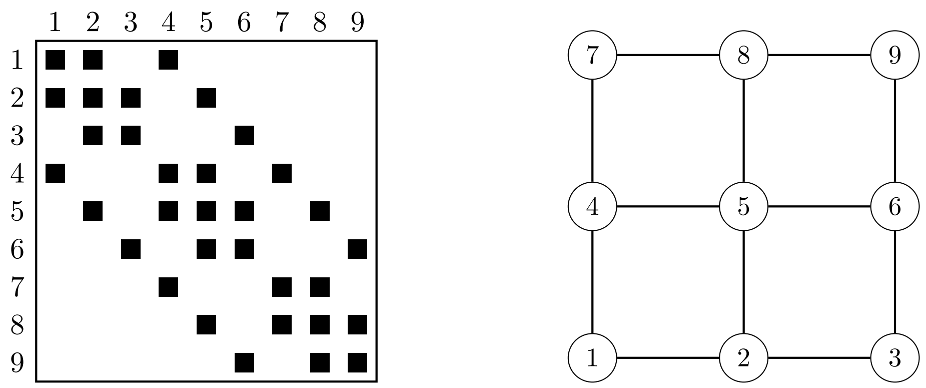
In all this section, the matrix is assumed structurally symmetric (i.e., $a_{i,j} = 0 \quad a_{j,i} = 0$) for the sake of clarity, although more general results exist for the unsymmetric case. To a structurally symmetric sparse matrix A can be associated an undirected graph $\mathcal{G}$ which represents its pattern i.e., where the non-zeros are. This graph is called an adjacency graph and is formalized in Definition 1.1.

Definition 1.1 - Adjacency graph.
The adjacency graph of a structurally symmetric matrix A is a graph $\mathcal{G}(A) = (V, E)$ with

n vertices such that:

- There is a vertex $v_j \in V$ for each row (or column) j of A.
- There is an edge $v_i, v_j \in E$ if and only if $a_{ij} = 0$, for $i = j$.

An illustration of the notion of adjacency graph is given in Figure 1.1.



(a) Initial matrix.

(b) Adjacency graph.

Figure 1.1: The original matrix and its corresponding adjacency graph.

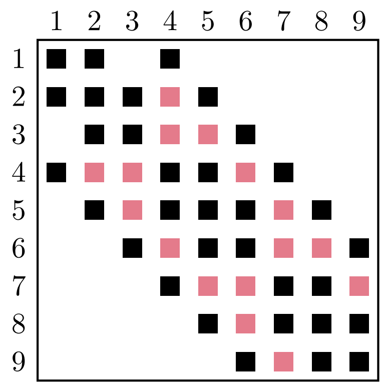
The elimination of a variable k in A through Gaussian elimination has the following effects on its adjacency graph:

1. vertex v_k is removed from $\mathcal{G}(A)$ along with all the incident edges.
2. edges are added between any pair of neighbors of v_k (if not already present). A new edge in the graph corresponds to a new non-zero value: a fill-in.

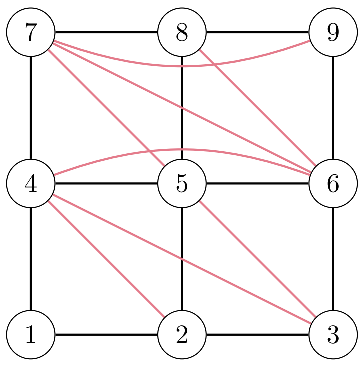
This is illustrated in Figure 1.2. For instance, the elimination of $a_{1,1}$ creates new non-zero values in $a_{2,4}$ and $a_{4,2}$. Equivalently, vertices (2) and (4) were originally not connected but since they both are connected to (1), a new edge has to be added. Then, the process continues and these new nonzero values yield themselves new nonzero values (for instance, $a_{3,4}$ becomes non-zero because $a_{2,4}$ itself became non-zero before).

In this example, 16 new values turn non-zero in the factors, representing 67% more non-zero than in the original matrix. Many techniques have been developed to reduce the fill-in which occur during sparse factorizations. One of the most used and famous one is certainly the nested dissection [51], especially for large scale systems.

The idea is to divide the original graph $\mathcal{G}(A)$ into s subgraphs $\mathcal{G}_1, \ldots, \mathcal{G}_s$, disconnected by a vertex separator $\mathcal{S}$ at their interface, so that no edge exists between any vertex of $\mathcal{G}_i$ and any vertex of $\mathcal{G}_j$, i, j, $i = j$. The process is then recursively applied to both $\mathcal{G}_1$, ..., $\mathcal{G}_s$, as shown in Figure 1.3(a) where we assume $s = 2$. Two levels of nested dissection are performed in this example. Note that throughout this study, we will now always assume $s = 2$ for the sake of clarity (this is also often the case in practice). The nested dissection yields a separator tree, which is a graph such that each node p is associated with a separator interfacing $\mathcal{G}_{p_1}$ and $\mathcal{G}_{p_2}$ and has two children i_1 and i_2, associated with the separator of $\mathcal{G}_{p_1}$ and $\mathcal{G}_{p_2}$, respectively. The root of the separator tree is the first separator.

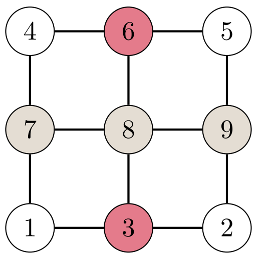


(a) Filled matrix.

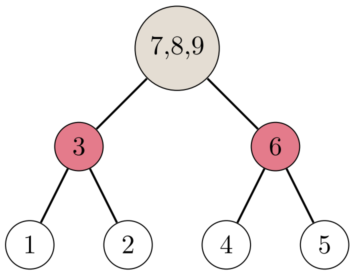


(b) Adjacency graph with ll-in.

Figure 1.2: The filled matrix and its corresponding adjacency graph. In red are represented the new edges and non-zeros.



(a) The two rst steps of nested dissection applied on the graph from Figure 1.1(b). Gray vertices are vertices from the top level separator. Vertices 3 and 6 are two separators of level 2.

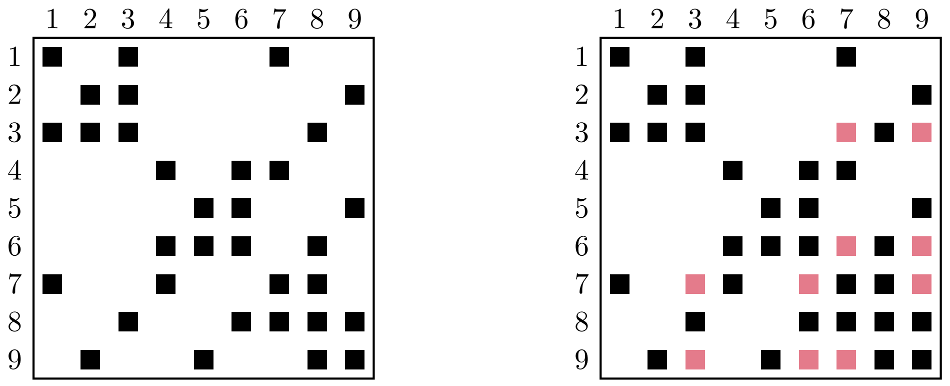


(b) The corresponding separator tree. The root is the top level separator.

Figure 1.3: Example of nested dissection of a 3 3 regular grid and its corresponding separator tree.

The matrix is then reordered according to this partition (the reordered matrix will be referred to as $\tilde{A}$): variables 1, 2, 4 and 5 come first (they are leaves), then the variables 3 and 6 come (separators of level 2), and finally the variables 4, 5 and 6 come (top level separator). This is illustrated in Figure 1.4(a). This ordering scheme has been theoretically shown to optimally decrease fill-in during factorization for regular 2D graphs [51] and more generally for almost any planar graph [75]. In practice, it is efficient on a wider class of graphs. When the factorization of $\tilde{A}$ is performed, the fill-in is indeed substantially reduced. Figure 1.4(b) shows the filled form of $\tilde{A}$ where only 10 new non-zeros have been created. This represents 37% less than in the natural order (the order of Figure 1.1). Consequently, less operations are also needed to compute the factors (because no operation is performed on zero values). This is due to the fact that when eliminating variable 1, fill-in only occurs between variables 3 and 7 (because it has no edge with any other variable). Then once variables 1, 2, 4 and 5 have been eliminated, no more fill-in occurs.

The direct consequence is that the elimination of variable 1 has no influence on variables 2, 4 and 5. However, it has an influence on the variables 3 and 7. Thus, variable


(a) Matrix reordered after one step of nested dissection.

(b) Fill-in within factorization of the matrix reordered after one step of nested dissection.

Figure 1.4: The effect of two steps of nested dissection on the original matrix and its filled form.

1 has to be eliminated before those latter two variables. This illustrates the existence of dependencies between the variables, which are well modeled by the separator tree already presented in Figure 1.3(b): the elimination of a variable i can only influence variables in separators located on the path from i to the root in the separator tree. It is possible to exploit this property in order to efficiently compute the factorization of a sparse matrix and this is what the multifrontal method (and also other methods) does.

1.3 The multifrontal method

The multifrontal method aims at solving large sparse linear systems of equations. It has been first designed in 1983 by Duff and Reid [42; 43], and represents the dependencies between the elimination of variables by means of the *elimination tree*, which was introduced in 1982 by Schreiber [90] in the context of symmetric matrices. In this dissertation, we will often use terminology related to the nested dissection to conveniently present the multifrontal method, although this applies in any context. This also allows for clear explanations in Chapter 3, which presents a low-rank multifrontal solver. Because the theory related to the multifrontal method will not be used for our purpose, we will avoid an excessively technical presentation here. For this same purpose, the multifrontal method is presented for symmetric matrices but it can be generalized to general matrices. For detailed theory and presentation about the multifrontal method, one can refer to Eisenstat and Liu [45; 46] for unsymmetric matrices, to Liu [77] for a survey on symmetric matrices or to L'Excellent [71] for implementation, parallelism and numerical issues.

1.3.1 Elimination tree and frontal matrices

As stated before, the multifrontal method heavily relies on the elimination tree. Definition 1.2 formalizes this idea.

Definition 1.2 - Elimination tree (symmetric case); from Liu [77].
Given a symmetric matrix A and its lower triangular factor $L = (l_{ij})_{1 \quad i,j \quad n}$, the elimination tree of A is a graph with n vertices such that p is the parent of a node j if and only

if

$$p = \min\ i > j : l_{ij} = 0$$

The elimination tree corresponding to the matrix of Figure 1.4(a) (which will be referred to as simply A in the rest of this section) is illustrated in Figure 1.5.

6 is *parent* of 5
4 is *child* of 6
4 and 5 are *siblings*
9 is the *root*
1, 2, 4 and 5 are *leaves*
{3,1,2} is a subtree rooted at 3

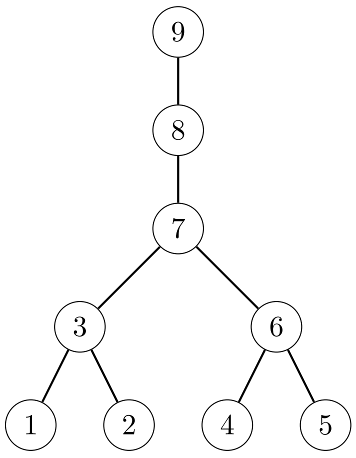
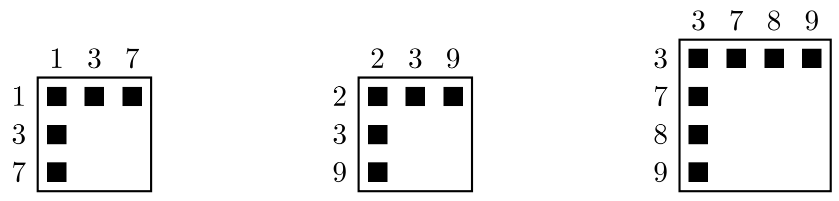


Figure 1.5: An example of elimination tree based on matrix from Figure 1.4(a) and some graph related vocabulary.

We know from Schreiber [90] that the elimination tree represents the dependencies between the elimination of variables: a variable i can be eliminated only when all its descendants have been eliminated, i.e., the traversal of tree must be topological and is in practice in postorder: a topological order is a scheme where a parent is numbered after its children; a postorder is a topological order where the nodes of each subtree are numbered consecutively. From the opposite view, when variable i is eliminated, some of its ascendants are updated. The idea behind the multifrontal method is to perform the eliminations and store the updates in dense matrices called *frontal matrices* or *fronts*.

Let us take the example of A. The elimination of variable 1 updates the variables 3 and 7. The rest of the matrix is not needed as the subtree rooted at 4 is independent from 1, thus we form the corresponding frontal matrix as shown in Figure 1.6(a). A similar behavior occurs for the elimination of variable 3 in Figure 1.6(c).



(a) Front associated with variable 1.

(b) Front associated with variable 2.

(c) Front associated with variable 3.

Figure 1.6: Three of the frontal matrices involved in the factorization of matrix A.

The elimination of variable 1 yields a 2 2 Schur complement which contains the updates related to variables 3 and 7. Similarly, elimination of variable 2 yields a 2 2 Schur complement which contains the updates related to variables 3 and 9. Those

Schur complements are called *contribution blocks* (CB). They are kept in memory (see Section 1.3.3) until they are not needed anymore. Once 1 and 2 have been eliminated, no more update related to variable 3 will be created elsewhere. Thus, the updates in related contribution blocks are collected, summed with the corresponding values coming from the original matrix and a new frontal matrix, associated with variable 3, is formed: this is called the *assembly*. Variable 3 is for this reason called *fully-summed*, as all the contributions related to variable 3 have been summed together, and no more contribution will be created. Then, the process continues: the elimination of variable 3 yields a new 3   3 contribution block which contains the updates related to variables 7, 8 and 9, and so on. The variables corresponding with to the contribution blocks are called the *non fully-summed* variables, because they are not ready to be eliminated.

In practice, the nodes of the elimination tree are amalgamated to form *supernodes* so that a frontal matrix usually has more than only 1 fully-summed variable. *Regular* amalgamation merge variables which have the same column structure (in the filled matrix) together in the same front. For instance, in Figure 1.4, variables, 7, 8 and 9 have the same column structures and can thus be amalgamated. This improves the efficiency of the process as BLAS 3 operations can be performed. Then, instead of eliminating only one single variable, several variables are eliminated within each frontal matrix. To further improve efficiency, amalgamation can also be relaxed by amalgamating columns which have not necessarily the same structure. This increases the number of zero entries stored (and potentially the fill-in) but also enhances the efficiency of operations. The resulting tree, called an *assembly tree*, is pictured in Figure 1.7. The root is obtained through regular amalgamation while other nodes have not been amalgamated. All of the eliminations performed within a node (amalgamated or not) will be called the *partial factorization* of the corresponding frontal matrix, as not all the variables are eliminated. Note that there is not only one way to amalgamate nodes and many strategies may be defined in order to enhance different aspects of the factorization (such as parallelism).

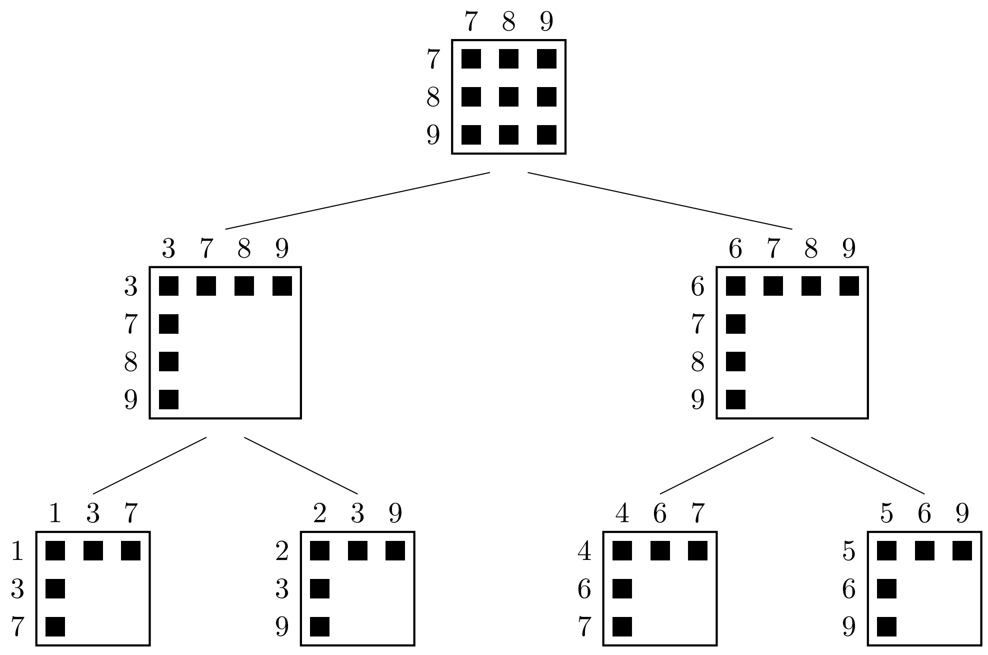
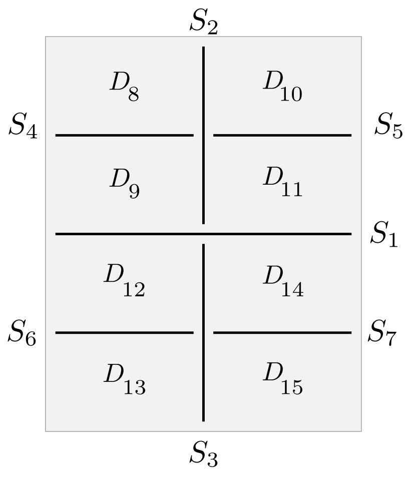
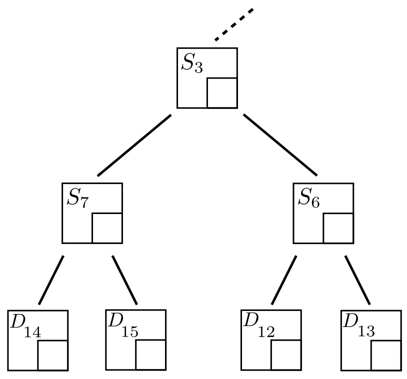


Figure 1.7: Assembly tree based on the nested dissection of A.

In the context of nested dissection, a natural assembly tree is the one that matches exactly the separator tree pictured in Figure 1.3(b), because of how the separators are built. Indeed, extracting the fully-summed variables of each front in the assembly tree presented in Figure 1.7 would lead exactly to the separator tree of Figure 1.3(b). For this reason and in the case of nested dissection, a frontal matrix is associated with each separator and its fully-assembled variables correspond to the variables in the separator itself. Similarly, the non fully-summed variables correspond to parts of separators which are higher in the tree (for instance, variable 1 updates completely the directly upper separator, e.g., variable 3, and partly the top level separator, e.g., variable 7). Moreover, the non fully-summed variables form a border around the current separator (variables 7 and 3 around variable 1). This is illustrated on a larger example in Figure 1.8, where the nested dissection is stopped when subdomains are too small, which is what is done in practice.



(a) General nested dissection of the graph.



(b) Corresponding assembly tree where fully-summed variables of frontal matrices correspond to separators.

Figure 1.8: A general nested dissection and its corresponding assembly tree. When S_7 is eliminated, variables of S_3 and half the variables of S_1 are updated. This forms a border around S_7.

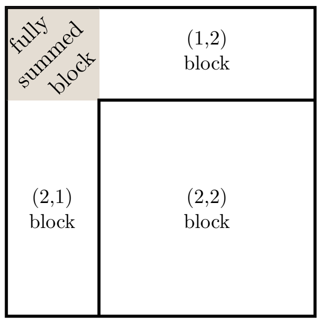
The non fully-summed variables within the frontal matrix where variables of S_7 are eliminated consist of all the variables of S_3 and half of the variables of S_1, which forms a border around S_7. For these reasons, we will often use *separator* and *fully-summed variables* without any distinction, and similarly for *border* and *non fully-summed variables.* Note that this analogy will be used even in the case where the fill-reducing permutation is not obtained with a nested dissection method, which is often the case in practice. This is justified by the fact that if we consider a subtree of the elimination tree rooted at a given node $\mathcal{N}$, the set of variables eliminated within a given branch below $\mathcal{N}$ is independent of the set of variables eliminated within any other branch below $\mathcal{N}$, meaning there is no direct edge between them in the graph. As they commonly update the fully-summed variables of $\mathcal{N}$, these latter variables can be viewed as a separator.

The generalization of the notions of elimination trees and frontal matrices to the unsymmetric case is not straightforward [34, 35, 39, 40, 45, 46, 52, 53, 49]. It requires significantly more formalism and graph theory to generalize these notions, mainly because handling interaction between the two distinct zero-patterns of the L and U factors during the elimination process is complex. In this dissertation, we will always consider that the elimination tree of the symmetrized matrix $|A| + |A|^T$ is used, which is still quite a competitive approach even if one can better take into account the unsymmetry with more sophisticated approaches (see [6, 32, 33, 61, 49, 74]). Note that in all previously referenced work for unsymmetric matrices, the task dependency graph resulting from the generalization of the elimination tree is in general not anymore a tree which significantly complicates data structures and algorithms.

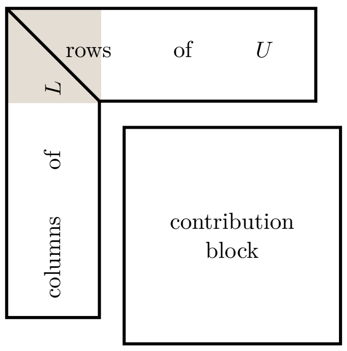
Due to the underlying tree structure, the multifrontal method is naturally suited for parallelism as two siblings can be processed on two different processes, because the two corresponding sets of variables are independent. This type of parallelism is called the *tree* parallelism. Another level of parallelism can also be exploited: the *node* parallelism, where a front is processed by different processes. This latter type is the greatest source of parallelism. We will get back to the parallelism within multifrontal methods in Section 1.6.2, in the particular case of the MUMPS multifrontal solver.

The multifrontal method is only one of the two main classes of sparse direct methods. The other class, called *supernodal*, includes several variants of the same technique which differ in how the updates are handled. In a right-looking supernodal method, the updates are applied right after a variable is eliminated. In a left-looking approach, the updates are delayed as much as possible and applied just before a variable is eliminated. For a detailed survey on parallel approaches for sparse linear systems on should consult Heath et al. [65]. The Multifrontal method can thus be viewed as in between since updates are performed after elimination (like in the right-looking approach) but stored in contribution blocks and applied later (more like in the left-looking approach). Note that most of the techniques which will be presented throughout this dissertation could also fit supernodal approaches.

The general structure of a frontal matrix, which will be used throughout this dissertation is illustrated in Figure 1.9, before and after the partial factorization. Note that the root node of the assembly tree has no contribution block.



(a) Frontal matrix after assembly and before partial factorization.



(b) Frontal matrix after partial factorization.

Figure 1.9: The general structure of a front before and after its processing. The shaded block is called the (1,1) block or the fully-summed block.

1.3.2 Threshold pivoting and delayed pivots

Because the multifrontal method is based on Gaussian elimination, just as the dense LU factorization is, the same numerical problems may occur. Thus, pivoting techniques must be used to ensure the accuracy of the solver. Because of the particular constraints due to the multifrontal method, partial pivoting as presented in Section 1.2.2 must be slightly adapted. First of all, pivoting dynamically modifies the structure of the factors which can increase the fill-in (and thus the operation count) as variables are not eliminated in the predicted order. For this reason, the standard partial pivoting is relaxed and a *partial threshold pivoting* strategy is used where a pivot $a_{k,k}$ is considered not acceptable when

$$|a_{k,k}| < u \max_{k \quad j \quad nass} |a_{k,j}|$$

where $0 \quad u \quad 1$ and *nass* is the number of fully-summed variables in the considered front. In practice, $u = 0\ 1$ or $u = 0\ 01$ is enough to ensure stability. Moreover, because only fully-summed variables can be eliminated within a given front, a pivot must be fully-summed to be selected. This means pivots must remain in the (1,1) block of the front, so that at the first step of the partial factorization of a front, any entry of the (1,1) block is candidate for being the first pivot. Note that pivoting is performed in a *quasi complete* fashion as pivots are not searched only in current row (but are not searched in the entire front though). Sometimes, priority is given to diagonal pivots.

Because a pivot cannot be chosen outside the (1,1) block, some pivots may remain unacceptable within current front. In this case, they remain uneliminated and are merged to the original non fully-summed variables as part of the contribution block. Such a variable is called a *delayed pivot* and will again be considered fully-summed in the parent frontal matrix, until it is eliminated. This further modifies the structure of the frontal matrices and increases the computational cost.

1.3.3 Memory management

Because the multifrontal method stores the updates in contribution blocks to apply them later (i.e., upper in the assembly tree), two types of memories, which are handled and behave differently, have to be distinguished:

1. the active memory which stores the active frontal matrix as well as all the contribution blocks waiting to be consumed.

2. the memory of the factors which stores the rows of U and the columns of L as frontal matrices are processed.

These two types of memory are strongly impacted by the underlying ordering used to obtain the assembly tree. Because an analysis of this behavior is not needed for our purpose, we will assume in this section that an assembly tree is given. For more details about this particular issue, one can refer to Guermouche et al. [63].

The active memory has a complex behavior. When the memory needed to process a front is allocated, it increases. Then, contributions blocks are consumed by the assembly of the current front, which makes the active memory space size decrease. Once the frontal matrix has been processed, the contribution block has to be copied and stored, so that the active memory size increases again as shown in Figure 1.10 (right-hand side arrow). The process continues and the active memory size keeps increasing and decreasing. Note that in a sequential setting, when a postorder traversal of the tree is used, the memory of the contribution blocks behaves like a stack: a node being processed right after all its children, all the contribution blocks needed for its assembly can be conveniently accessed as they are on top of the stack. This is not the case in a parallel environment as a postorder traversal of the tree is no longer ensured [5].

Because of this particular behavior, the maximum size of the active memory, called its *peak*, will often be considered to analyze the memory consumption of the multifrontal method. This peak consists of the memory for a given frontal matrix and all the contribution blocks which are stacked (and thus called the *stack*) at the moment when the frontal matrix is assembled. A tree being given, the size of the peak of active memory (as well as its global behavior) strongly depends on the postorder traversal of the tree [76] and, in general, may be bigger than the size of the factors.

The memory of the factors is simpler to analyze: it always increases during the execution and its maximal size does not depend on the tree traversal.

Figure 1.10 summarizes the typical memory usage at a given point of the multifrontal process. Note that to enhance memory efficiency, the memory space in many multifrontal codes is usually allocated once at the beginning (based on a prediction of the memory needed for the factors and the peak of active memory, which is sometimes hard to evaluate accurately in a parallel environment or when pivoting is intense) and then managed explicitly, which avoids dynamic allocations.

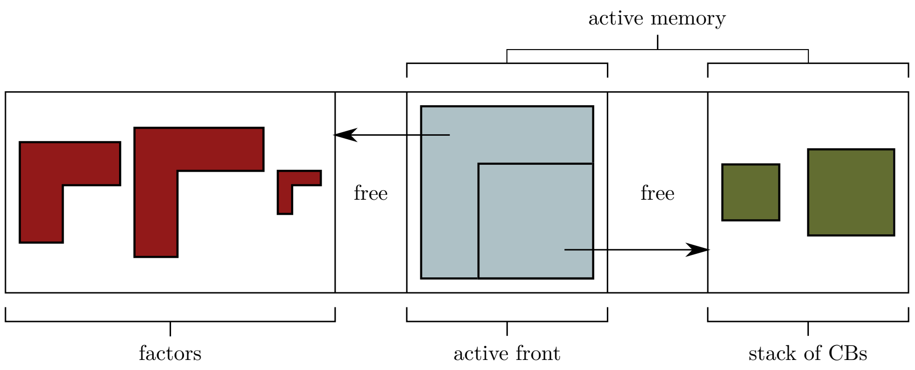


Figure 1.10: The total memory allocated statically for the multifrontal process. The two distinct types of memory are illustrated. After the active front is processed, the columns and rows of the factors computed go to the factors memory area and the new contribution block is stacked.

1.3.4 Multifrontal solution

Once all the nodes of the assembly tree have been processed, the matrix is factorized and the linear system can be solved through forward elimination and backward substitution. Throughout this dissertation, the partial factorization of front F^i (associated with the node i of the assembly tree) will be considered yielding a factor blocked as illustrated in Figure 1.11. Note that this is not the only way of blocking the factor. The part of L (U) computed within front F^i will be referred to as L^i (U^i). For convenience, we present the blockwise version of the algorithm which will be reused in Chapter 3.

The forward elimination $Ly = b$ is achieved by means of a bottom-up traversal of the assembly tree. At each node, part of the temporary solution y is computed (corresponding

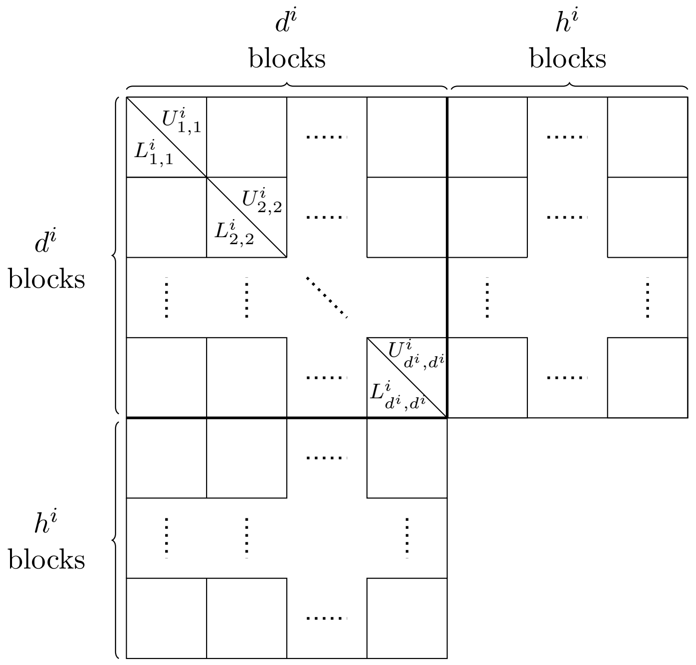


Figure 1.11: Structure of the factor computed within front F^i, ready for the solution phase.

to the fully-summed variables associated with the frontal matrix of the current node) and part of it is updated (corresponding to the non fully-summed variables associated with the frontal matrix of the current node), as Algorithm 1.5 and Figure 1.12 show. Note that because the forward elimination requires a bottom up traversal of the assembly tree, it can be done directly on the fly during the factorization.

Algorithm 1.5 Multifrontal right-looking forward elimination in front F^i i.e., $L^i y^i = b^i$ is solved. This step is performed for any node (front) of the assembly tree, from bottom to top.

```
1: for $c = 1$ to $d^i$ do
2:     $y_c \quad (L^i_{c,c})^{-1} \quad b_c$
3:     for $r = c + 1$ to $d^i + h^i$ do
4:         $b_r \quad b_r - L^i_{r,c} \quad y_c$
5:     end for
6: end for
```

Then, a top-down traversal of the assembly tree is performed for the backward substitution phase, which behaves quite similarly as the forward elimination phase. It is illustrated in Algorithm 1.6 and Figure 1.13.

Note that the forward elimination and the backward substitution are performed in a right-looking and left-looking fashions, respectively, in order to match the natural storage scheme when L is stored by columns and U by rows.

1.3.5 The three phases

Multifrontal solvers are usually organized into three convenient different phases, which already naturally appeared in the discussion above and are namely:

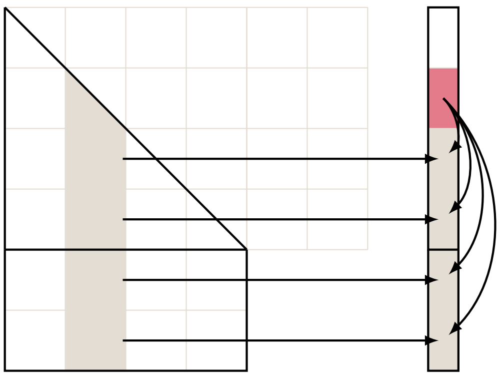


Figure 1.12: Illustration of the multifrontal right-looking forward elimination in front F^i.

Algorithm 1.6 Multifrontal left-looking backward substitution in front F^i i.e., $U^i x^i = y^i$ is solved. This step is performed for any node (front) of the assembly tree, from top to bottom.

```
1: $x^i$    $y^i$
2: for $r = d^i$ to $1$ by $-1$ do
3:     for $c = r + 1$ to $d^i + h^i$ do
4:         $x^i_r$    $x^i_r - U^i_{r,c}$  $x^i_c$
5:     end for
6:     $x^i_r$    $(U^i_{r,r})^{-1}$  $x^i_r$
7: end for
```

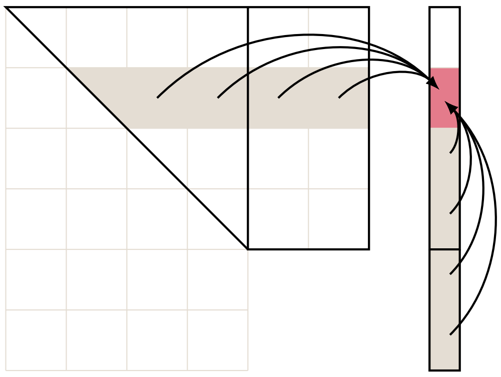


Figure 1.13: Illustration of the multifrontal left-looking backward substitution in front F_i.

Analysis. This step handles the preprocessing of the matrix. A global, fill-reducing, permutation is computed and the corresponding assembly tree is built. Other numerical pretreatments can also be performed to improve the numerical properties of the matrix and enhance the accuracy of the subsequent phases. A symbolic factorization (i.e., based on the nonzero pattern of the matrix only) is also performed to forecast the data structures of the factorization and the memory consumption.

Factorization. The assembly tree is traversed and partial factorizations of each front is performed (under LU, LL^T or LDL^T forms). The factorization step usually requires the most computations. The theoretical memory and operation complexities [51, 75] of the multifrontal factorization are given in Table 1.1, for the Poisson and Helmholtz equations discretized on both 2D and 3D meshes. These problems are presented in Section 1.6.4.1. The meshes are in theory assumed "well-behaved" [80] (they are cubic in our context) and partitioned with a nested dissection.

Problem	Mry	Ops
`PoissonN` 2D	$O(n\log(n))$	$O(n^{3\;2})$
`HelmholtzN` 2D		
`PoissonN` 3D	$O(n^{4\;3})$	$O(n^2)$
`HelmholtzN` 3D		

Table 1.1: Theoretical complexities of the multifrontal factorization. *Mry* stands for number of entries in factors. *Ops* stands for number of operations for the factorization. n is the size of the matrix. N is the size of the underlying square or cubic mesh: $n = N^2$ in 2D, $n = N^3$ in 3D.

Solution. The assembly tree is traversed twice to perform the forward elimination and the backward substitution. The accuracy of the solution may be optionally evaluated through various types of metrics (see 1.2).

1.4 Iterative methods and preconditioning

Direct methods are not the only methods to solve linear systems: iterative methods [88] can also be used. In this also widely used kind of methods, the solution is computed iteratively from an initial guess x_0. At each step, an operation $x_{k+1} = f(x_k)$ is applied, where f depends on the method, which can be for instance the Conjugate Gradient [66] or GMRES [89]. A stopping criterion is defined (based on how close to the exact solution x_k is) and the process stops when it is satisfied. In this case, the process has *converged*, but it is not always the case. To avoid non-convergence or slow convergence, some preprocessing of the matrix may be needed. This is called *preconditioning* the matrix and amounts to solving a preconditioned system $M^{-1}Ax = M^{-1}b$. When A is a symmetric positive definite matrix, the preconditioner M is split to keep this property and the preconditioned system is written as $M^{-1\;2}AM^{-1\;2}\quad M^{1\;2}x = M^{-1\;2}b$. If $M = A$, then the convergence is fast (0 iteration) but we need to solve the original system to obtain M^{-1}. The idea is thus to find a good approximate of A^{-1} which can be computed at a very low cost and makes the convergence faster. The strategy for finding it can be very problem-dependent and this is the main drawback of iterative methods, although versatile strategies have also been proposed, such as incomplete factorizations (where some off-diagonal elements are dropped, for instance, if they match a zero in the original matrix). In the context of a double precision linear system, a good and versatile, although expensive, preconditioner is A^{-1} computed in single precision with a direct solver. This strategy will be investigated with Block Low-Rank factorization (see Sections 1.6.4.3 and 5.1.2).

1.5 Low-rank approximations

This dissertation is motivated by several algebraic properties dealing with the rank of matrix, which can be exploited to efficiently represent matrices. We first recall some algebraic properties of the rank and will introduce the notion of low-rank matrix. We will show how these particular matrices can be used in order to accelerate some of the algebraic operations they are involved in. Finally, we will describe how to reorder certain full-rank matrices to obtain low-rank subblocks, and how these subblocks have been exploited in the literature. In this section, A will always be chosen in $\mathbb{R}^{m\quad n}$ but the same definitions and theorems hold for $\mathbb{C}^{m\quad n}$.

1.5.1 Rank and properties

We denote $rank(A)$ the algebraic rank of A, in contrast with the numerical rank at precision ε defined below.

Definition 1.3 - Numerical rank at precision ε.
Let $A \in \mathbb{R}^{m\quad n}$ be a matrix. The number of singular values σ_i satisfying $|\sigma_i| \geq \varepsilon$ is called the numerical rank at precision ε of A and is written $rank\ (A)$.

This work relies more on the numerical rank at precision ε than on the algebraic rank. For this reason and for sake of clarity, the notation *rank* will be used for both algebraic rank and numerical rank at precision ε when no ambiguity is possible.

We now recall some properties of the rank (see for instance [94, 57]).

Property 1.1 - Matrices operations.
Let $A \in \mathbb{R}^{m\quad n}$ and $B \in \mathbb{R}^{m\quad n}$ be two matrices. Then the following properties hold:

(i) $rank(A) \quad min(m, n)$

(ii) $rank(A + B) \quad rank(A) + rank(B)$

Now assume $B \in \mathbb{R}^{n\quad m}$. Then:

(iii) $rank(AB) \quad min(rank(A), rank(B))$

Properties (ii) and (iii) are the most interesting. (ii) implies that we have almost no control on the rank of the sum of two matrices. Assume the two matrices have same rank. Adding the two of them can either (worst case) double the rank, either (best case) dramatically decrease it, either just change it. Secondly, we have almost the opposite behavior with the product. There is no rank increase. The product of two matrices of small rank will still have a small rank.

1.5.2 Low-rank matrices

This work is based on representing matrices under a particular form, called low-rank form. This is formalized in Definition 1.4.

Definition 1.4 - [18] Low-rank form.
Let A be a $m \quad n$ matrix of numerical rank k at precision ε. The low-rank form of A, written $\widetilde{A}$, is an approximation of A under the form:

$$\widetilde{A} = X \quad Y^T,$$

where X and Y are matrices of size $m \quad k$ and $n \quad k$, respectively. The quality of the approximation is given by $A = \widetilde{A} + E$, where $\ E\ _2 \quad \varepsilon$. Computing the low-rank form of a matrix will be referred to as *demoting*.

Proof of the existence. A proof of existence of such a representation uses the Singular Value Decomposition (SVD, [57]) of A. We compute the SVD of A, which is defined for any $m \quad n$ matrix A:

$$A = U \qquad V^T$$

where $U \in \mathbb{R}^{m \quad m}$ and $V \in \mathbb{R}^{n \quad n}$ are orthogonal and $\quad \in \mathbb{R}^{m \quad n}$ is diagonal. By definition [57], $\quad = diag(\sigma_1, \sigma_2, \quad , \sigma_n)$ with $\sigma_1 \geq \sigma_2 \geq \quad \geq \sigma_n$ (each σ_i is a singular value of A). If we set (using Matlab notations) $X = U(:, 1:k)$, $Y = V(:, 1:k) \quad (1:k, 1:k)$, we have $A = XY^T + E$, where $\quad E \quad_2 \quad \varepsilon$ since all the removed singular values are smaller than ε. □

Low-rank forms are particularly suitable representations for low-rank matrices, defined in Definition 1.5.

Definition 1.5 - [18] Low-rank matrix.
Let A be a $m \quad n$ matrix of numerical rank k at precision ε and $\varepsilon_{\max}$ be a given real number. A is said to be a low-rank matrix if it has a low-rank form $\widetilde{A} \quad X \quad Y^T$ with X and Y matrices of size $m \quad k$ and $n \quad k$, respectively, which satisfies:

$$\begin{array}{ll} k(m+n) & mn \\ \varepsilon & \varepsilon_{\max} \end{array}$$

The first condition states that the low-rank form of a low-rank matrix requires less storage than the standard form. For this reason, we will sometimes use the term *compressing* instead of *demoting* with no distinction for low-rank matrices. The second condition simply states that the approximation is "good enough", which strongly depends on the application. For industrial applications, typical values of $\varepsilon_{\max}$ range from 10^{-16} (double precision) or 10^{-8} (single precision) to 10^{-2}. Small values of $\varepsilon_{\max}$ are mostly used in the context of a direct solver while larger values are used in the context of the computation of a preconditioner. This definition can be expressed similarly as Wilkinson's definition of sparse matrices (see Section 1.2): "A low-rank matrix is any matrix with enough very small singular values that it pays off to take advantage of them". Note that throughout this dissertation, the acronyms FR and LR will denote Full-Rank and Low-Rank, respectively.

As a consequence of the first condition, the complexity of basic linear algebra operations performed on low-rank matrices under low-rank forms, such as matrix-matrix products and triangular solves, is also decreased, as explained in Section 1.5.3.

The proof of existence of Definition 1.4 shows that the numerical rank at precision ε can be computed together with X and Y with a Singular Value Decomposition (SVD). This can also be done, less precisely but much faster, with a Rank-Revealing QR (RRQR) factorization. Less standard techniques have also been experimented in this context, such as Adaptive Cross Approximation (ACA, [18]) and randomized sampling [104]. Low-rank approximation techniques are thus based upon the idea to ignore E and simply represent A with its low-rank form $A = X \quad Y^T$. This is, by definition, an approximation whose precision can be controlled through the parameter ε called *low-rank threshold*. The efficiency of such a representation depends on how small is the rank of the matrix at a given precision satisfying the user.

1.5.3 Low-rank basic algebraic operations

Low-rank matrices are, by definition, a way to reduce the amount of memory needed to store matrices. Moreover, this is a convenient format to reduce the complexity of the main algebraic operations performed during the multifrontal factorization of a matrix: the matrix-matrix product and the triangular solve.

1.5.3.1 Low-rank solve

We consider a low-rank matrix $A = XY^T$ of size n n and a lower triangular matrix L. A is assumed to have rank k. A full-rank solve consists of computing the product $R = A\ L^{-T}$ which requires n^3 floating-point operations. Using the low-rank structure of A, this operation can be rewritten as $R = X\ (Y^T\ L^{-T})$. Thanks to this particular bracketing, it requires now only $3kn^2$ operations. If $k < n$ 3, the low-rank triangular solve requires less operations than its full-rank counterpart. Moreover, if one sets $\widetilde{Y}^T = Y^T\ L^{-T}$, then $R = X\widetilde{Y}^T$ is a low-rank matrix and it can be directly used for low-rank computations, although the numerical rank of R may be in reality lower than k (the low-rank form of R is in this case not optimal).

1.5.3.2 Low-rank matrix-matrix product

We consider two low-rank matrices $A_1 = X_1Y_1^T$ and $A_2 = X_2Y_2^T$. We assume that $A_1 \in \mathbb{R}^{m\ p}$ has rank k_1 and $A_2 \in \mathbb{R}^{p\ n}$ has rank k_2. The product of two matrices can also be improved in case one of them, at least, is low-rank. This is even more important than for the low-rank solve as matrix-matrix products occur more often than triangular solves in the multifrontal factorization. The full-rank matrix-matrix product $R = A_1\ A_2$ requires $2mnp$ operations. Using low-rank forms, the product becomes $R = (X_1Y_1^T)\ (X_2Y_2^T)$. Obviously, the latter bracketing must be avoided because it just recomputes the regular form of each matrix, and multiplies them, which would require even more operations than in the standard case. This would bring us back to the original matrix product. Consequently, the product Y_1^T with X_2 has to be performed first. Say $C\ \ Y_1^T\ X_2$. This leads to the situation depicted in Figure 1.14.

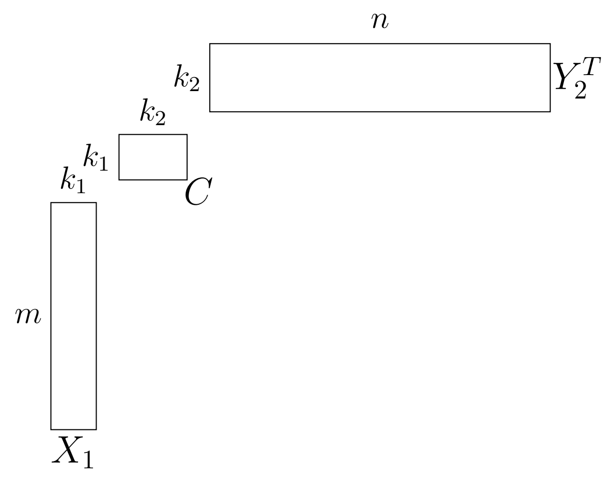


Figure 1.14: Multiplying two low-rank matrices, after the first step: $C\ \ Y_1^T\ X_2$.

At this point a block C appears which is of very small dimensions. Since we have three matrices to multiply together, two types of bracketing are possible:

right-first product:	$R = X_1\ (CY_2^T)$
left-first product:	$R = (X_1C)\ Y_2^T$

The right-first product requires $2k_1k_2(p+n) + 2k_1mn$ operations divided in three phases:

1. Computation of C: $2pk_1k_2$ ops
2. Computation of $B = C \quad Y_2^T$: $2nk_1k_2$ ops
3. Computation of $X_1 \quad B$: $2mnk_1$ ops

This result is not symmetric, i.e., the number of operations required for the left-first product is different. With the same reasoning as before, we get a cost of $2k_1k_2(p+m) + 2k_2mn$ operations. The operation requirements of each version of the low-rank matrix-matrix product are summarized in Table 1.2.

	left-first product	right-first product
# of ops	$2k_1k_2(p+m)+2k_2mn$	$2k_1k_2(p+n)+2k_1mn$

Table 1.2: Operations required for multiplying two low-rank matrices.

Since the number of operations is not symmetric, we have to decide in which case we use the right-first product, and in which case we use the left-first product. We denote C_L (C_R) the cost of a left-first product (right-first product). To decide if we use a right-first or a left-first product, we have to know the sign of $C_L - C_R$ which is given by the following equality:

$$sign(C_L - C_R) = sign \quad (k_2 - k_1) + \frac{k_1}{m}\frac{k_2}{n}(m-n)$$

This gives us a simple way to know which product should be computed, depending on the numerical properties of each operand. Note that if we assume the result to be a square matrix, i.e. $m = n$, then the sign of $C_L - C_R$ is simply given by the sign of $k_2 - k_1$.

The low-rank product, as we defined it, multiplies two matrices under low-rank forms and outputs a matrix under regular form. This means that if one wants the low-rank form of the product, one has to recompute it from the result. A mean to avoid this is to perform an incomplete product. Assume we have done the computation of C as before. The product $R = A_1 \quad A_2$ can be written $R = U_1 \quad C \quad V_2^T$. Thus there are two remaining products. The previous method proposed to compute both of them, in order to obtain R in a full-rank standard format. If we now consider doing only one of them, say the second one (it works similarly with the first one): $R = X_1 \quad (C \quad Y_2^T)$. Now we observe that the term $C \quad Y_2^T \in \mathbb{R}^{k_1 \ n}$. Since $X_1 \in \mathbb{R}^{m \ k_1}$, setting $\widetilde{Y}_2 = C \quad Y_2^T$ leads to $R = X_1 \quad \widetilde{Y}_2^T$, which is a low-rank form of R. This way, the cost of the product is lower (as we compute an incomplete one) and a low-rank form of the result is directly obtained. However, similar to the low-rank solve, this low-rank form may not be the most compact one for the result.

Multiplying two low-rank matrices A_1 and A_2 will thus decrease the required number of operations. However, we also have to consider the cost of *demoting* the two matrices, which can be quite expensive too. This will increase the number of operations. In the context of a multifrontal approach, we will compute many matrix products involving a given low-rank matrix. Then, the computational cost of demoting a matrix can be absorbed by the total savings for all the products.

1.5.4 Admissibility condition and clustering

In the general case, a given matrix is not a low-rank matrix, which means it admits no *e cient* low-rank form. However, it has been shown in Börm [20] and Bebendorf [18] that submatrices (also called *subblocks*) defined by an appropriately chosen partitioning of matrix indices, assuming the matrix arises from an elliptic partial differential equation, can be low-rank matrices. Assuming $I = \ 1, \quad , n$ is the set of row (and column) indices of A, a set of indices $\sigma \quad I$ is called a cluster. Then, a *clustering* of I is a disjoint union of clusters which equals I. $b = \sigma \quad \tau \quad I \quad I$ is called a *block cluster* based on clusters σ and τ. A *block clustering* of $I \quad I$ is then defined as a disjoint union of block clusters which equals $I \quad I$. Let $b = \sigma \quad \tau$ be a block cluster, $A_b = A_{\sigma\tau}$ is the subblock of A with row indices σ and column indices τ.

A considerable reduction of both the memory footprint and the operations complexity can be achieved if the block clustering defines blocks whose singular values have a rapid decay (for instance, exponential) in which case each block can be accurately represented by a low-rank product $X \quad Y^T$. Clearly, this condition cannot be directly used in practice to define the matrix block clustering. In many practical cases it is, however, possible to exploit the knowledge of the mathematical problem or the geometrical properties of the domain where the problem is defined in order to define an *admissibility condition*, i.e., a heuristic rule that can be cheaply checked to establish whether or not a block is (likely to be) low-rank or that can be used to guide the block clustering computation. For instance, in the case of matrices deriving from discretized elliptic PDEs one such admissibility condition is presented in Definition 1.6.

Definition 1.6 - [18, 20] Admissibility condition for elliptic PDEs.
Let $b = \sigma \quad \tau$ be a block cluster. b is admissible if

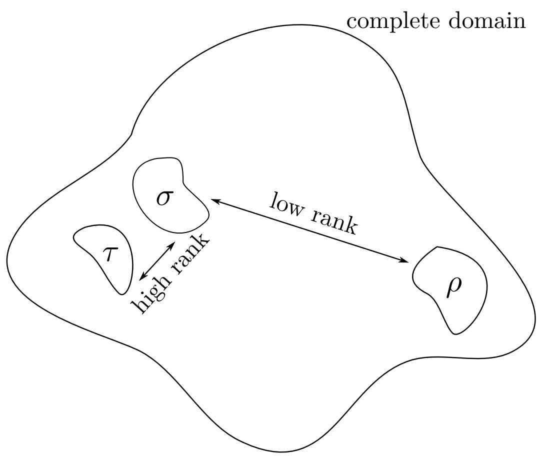
$$diam(\sigma) + diam(\tau) \quad 2 \quad dist(\sigma, \tau)\,,$$

where the *distance* dist(σ,τ) is the number of edges in the shortest path from a node of σ to a node of τ and the *diameter* diam(σ) is the largest distance between two nodes of σ. is a problem dependent parameter.

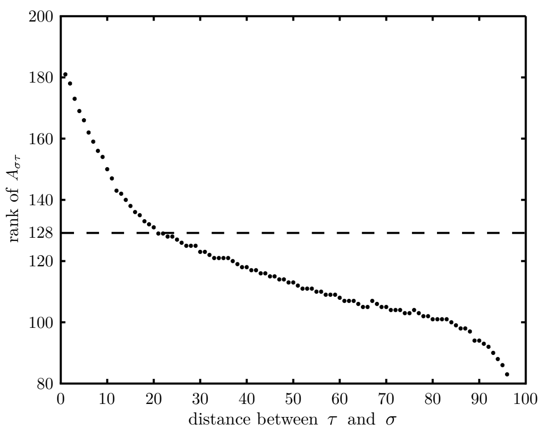
This admissibility condition follows the intuition that variable sets that are far away in the domain are likely to have weak interactions which translates into the fact that the corresponding block has a low rank; this idea is depicted in Figure 1.15(a). Figure 1.15(b) shows that the rank of a block $A_{\sigma\tau}$ is a decreasing function of the geometric distance between clusters σ and τ. This experiment has been done on a top-level separator of a 3D 128^3 wave propagation problem called `Helmholtz128` (and further described in Table 1.4), with square clusters of dimension 16 16, so that each subblock has size 256. It shows that depending on the distance between clusters, there is potential for compression which can be exploited. The dashed line at $y = 128$ shows the cutoff point where it pays to store the subblock using a low-rank representation.

An admissibility condition being given, it is possible to define an *admissible block clustering*, i.e., a block clustering with admissible blocks, which can be used to efficiently represent a dense matrix by means of low-rank approximations with an accuracy defined by the low-rank threshold ε in Definition 1.5.

This admissibility condition requires geometric information in order to properly compute diameters and distances. In the context of an algebraic solver, this is not conceivable because the only available information is the matrix. Thus, another condition must be used and a natural idea is to use the graph of the matrix. In Börm [20] and Grasedyck [59], other admissibility conditions have been studied. They only need the graph of the matrix $\mathcal{G}$ and are thus called *black box* methods. Definition 1.7 details this graph version of Definition 1.6.



(a) Strong and weak interactions in the geometric domain.



(b) Correlation between distance and full accuracy block rank.

Figure 1.15: Two illustrations of the admissibility condition for elliptic PDEs.

Definition 1.7 - [20, 59] Black box admissibility condition.
Let $b = \sigma \quad \tau$ a block cluster where σ and τ are sets of graph nodes. b is admissible if

$$diam_{\mathcal{G}}(\sigma) + diam_{\mathcal{G}}(\tau) \quad 2 \quad dist_{\mathcal{G}}(\sigma, \tau)\,,$$

where $diam_{\mathcal{G}}$ and $dist_{\mathcal{G}}$ are classical graph diameter and distance in $\mathcal{G}$, respectively, and a problem dependent parameter.

This admissibility condition may still be unpractical because, first, it is not clear how to choose the parameter in an algebraic context and, second, because computing diameters and distances of clusters of a graph may be costly. This condition, however, can be further simplified and complemented with other practical considerations that, for instance, pertain to the efficiency of basic linear algebra kernels on the X and Y matrices of the low-rank form. In order to define an efficient clustering strategy suited for our purpose and format, as well as to ensure the efficiency of the algorithms designed, another admissibility condition is proposed in Section 3.3.1.

1.5.5 Existing low-rank formats

Several matrix formats have been proposed in the literature, in order to exploit low-rank forms of submatrices. We will present first a brief historical background on low-rank formats, followed by a more extensive presentation of different widely used formats.

1.5.5.1 Brief historical background

The idea of exploiting low-rank approximations to represent matrices was first (and for a long time) motivated by the computation of the inverse of an irreducible tridiagonal matrix [50] (1950), which is a semiseparable matrix (for a definition, see Gohberg et al. [56], Vandebril et al. [97]). Based on this notion, sequentially semiseparable matrices appear and are defined as follows.

Definition 1.8 - Sequentially semiseparable matrix (SSS) [62].
Let B be a semiseparable $N \quad N$ matrix. Then there exist n positive integers $m_1, \quad , m_n,$

with $N = m_1 + \quad + m_n$, to block-partition A as

$$B = (B_{i,j}), \text{ where } B_{i,j} \in \mathbb{R}^{m_i \quad m_j} \text{ satisfies } B_{i,j} = \begin{array}{lll} D_i & & \text{if } i = j \\ X_i W_{i+1} & W_{j-1} Y_j^T & \text{if } i < j \\ P_i R_{i-1} & R_{j+1} Q_j^T & \text{if } i > j \end{array}$$

with:
$i \in [1, \quad , n-1], X_i \in \mathbb{R}^{m_i \quad k_i}, Q_i \in \mathbb{R}^{m_i \quad l_{i+1}}$,
$i \in [2, \quad , n], Y_i \in \mathbb{R}^{m_i \quad k_{i-1}}, P_i \in \mathbb{R}^{m_i \quad l_i}$,
$i \in [2, \quad , n-1], W_i \in \mathbb{R}^{k_{i-1} \quad k_i}, R_i \in \mathbb{R}^{l_{i+1} \quad l_i}$.
Note that k_i and l_i need to be small enough to make this representation efficient.

For instance, with $n = 4$, the matrix B has the form:

$$B = \begin{array}{cccc} D_1 & X_1 Y_2^T & X_1 W_2 Y_3^T & X_1 W_2 W_3 Y_4^T \\ P_2 Q_1^T & D_2 & X_2 Y_3^T & X_2 W_3 Y_4^T \\ P_3 R_2 Q_1^T & P_3 Q_2^T & D_3 & X_3 Y_4^T \\ P_4 R_3 R_2 Q_1^T & P_4 R_3 Q_2^T & P_4 Q_3^T & D_4 \end{array}$$

After many years of theoretical work on showing that the inverses of very specific tridiagonal, pentadiagonal and band matrices can be represented as semiseparable matrices, Barrett [17] proved in 1979 that the inverse of a tridiagonal matrix is a semiseparable matrix, without further assumption. In 1985, Gohberg et al. [56] proposed methods to solve semiseparable systems of equations, for strongly regular matrices. Finally, in 2002, Chandrasekaran et al. [27] generalized this result to more classes of matrices. A detailed survey on the evolution of these notions can be found in Vandebril et al. [96].

Simultaneously, the Fast Multipole Method (FMM, [60]) introduced the notion on far-field approximations from an analytical point of view, on the context of electromagnetics (*n-body* problems). This technique will strongly influence researchers to design low-rank approximations of matrices, based on well separated variables (see the notion of admissibility condition above). Although some black-box variants of the fast multipole method have been defined for specific classes of kernels [48], it still does not offer the same versatility as algebraic low-rank formats.

Many formats have been derived from the SSS format, and are widely used in many applications. Most of them are hierarchical, in the sense that a hierarchy is introduced between the approximated blocks. Within this class of formats, two subclasses have to be distinguished: independent hierarchical formats (where all the blocks are independently expressed) and nested hierarchical formats (where blocks are expressed with respect to other ones). This makes the algorithms much more complex but also more scalable. We will focus on Hierarchical $\mathcal{H}$ matrices which are a good starting point to understand independent hierarchical formats, and on HSS matrices (which is a nested hierarchical format) because it was the first to be used for implementing high-performance sparse direct solvers. Another format, called the Hierarchically Block Separable (HBS) format [54, 55] has also been used for direct solvers. As it is closely related to the HSS format, we will not further explain it.

1.5.5.2 Hierarchical ($\mathcal{H}$) matrices

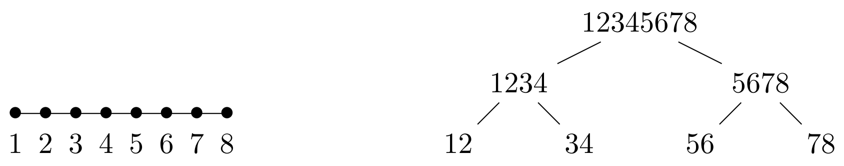
The $\mathcal{H}$-matrix [18, 20] format, where $\mathcal{H}$ stands for Hierarchical, is historically the first low-rank format for dense matrices. This format is based on an admissible block clustering which is obtained by a recursive subdivision of $I \quad I$.

A hierarchical matrix consists in a hierarchy of subblocks within the original matrix; its structure can be conveniently represented by a tree graph. This tree is commonly referred to as *cluster tree* in the literature (see [20] for instance).

Definition 1.9 - Cluster tree.
Let $\mathcal{T}(V, S, r)$ be a tree and $\mathcal{I}$ a set of indices. To each node $v \in V$ is associated a set of children $S(v)$ and a label $\widehat{v}$ (a label is a cluster associated with a node of the tree, i.e. a subset of $\mathcal{I}$, see Section 1.5.4). A cluster tree on $\mathcal{I}$, written $\mathcal{T}_{\mathcal{I}}$, is defined as follows :

- $r = \text{root}(\mathcal{T}_{\mathcal{I}})$ has the label $\mathcal{I}$, i.e. $\widehat{r} = \mathcal{I}$;
- $\quad v \in V$, $\widehat{v} = \underset{s \quad sons(v)}{} \widehat{s}$, where $\quad$ is a disjoint union and $sons(v)$ denotes all the children of node v in $\mathcal{T}$;

A cluster tree represents a recursive dissection of the original index set $\mathcal{I}$, as illustrated in Figure 1.16.



(a) Regular 1D mesh

(b) The cluster tree based on recursive bisections of the mesh

Figure 1.16: Example of a mesh and a corresponding cluster tree

This leads to the definition of a hierarchical block partition of the set $\mathcal{I}$ relying on the notion of *block cluster tree.*

Definition 1.10 - Block cluster tree.
Let $\mathcal{T}_{\mathcal{I}}$ and $\mathcal{T}_{\mathcal{J}}$ be two cluster trees. A block cluster tree $\mathcal{T}_{\mathcal{I} \quad \mathcal{J}}$ for $\mathcal{T}_{\mathcal{I}}$ and $\mathcal{T}_{\mathcal{J}}$ is defined as follows :

- $r = \text{root}(\mathcal{T}_{\mathcal{I} \quad \mathcal{J}}) = (\text{root}(\mathcal{T}_{\mathcal{I}}), \text{root}(\mathcal{T}_{\mathcal{J}}))$;
- each node $b \in \mathcal{T}_{\mathcal{I} \quad \mathcal{J}}$ has the form $b = (t, s)$, for $t \in \mathcal{T}_{\mathcal{I}}$ and $s \in \mathcal{T}_{\mathcal{J}}$, and its label satisfies $\widehat{b} = \widehat{t} \quad \widehat{s}$;
- let $b = (t, s) \in \mathcal{T}_{\mathcal{I} \quad \mathcal{J}}$. If $sons(b) = \quad$, we have

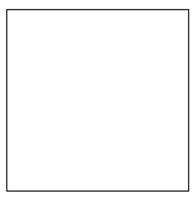
$$\text{sons}(b) = \begin{array}{lll} \quad t & \text{sons}(s) & \text{if sons}(t) = \quad, \text{sons}(s) = \quad, \\ \text{sons}(t) & \quad s & \text{if sons}(t) = \quad, \text{sons}(s) = \quad, \\ \text{sons}(t) & \text{sons}(s) & \text{otherwise} \end{array}$$

Several block cluster trees $\mathcal{T}_{\mathcal{I} \quad \mathcal{J}}$ can be built based onr $\mathcal{T}_{\mathcal{I}}$ and $\mathcal{T}_{\mathcal{J}}$. Two examples of block cluster trees constructed from cluster tree 1.16(b) are shown in Figures 1.17 and 1.18, together with each corresponding hierarchical matrix.

It is shown in Figures 1.17 and 1.18 that several different block cluster trees can be obtained from a single cluster tree. The induced hierarchical matrices are also very different. For instance, it is obvious that the hierarchical structure of Figure 1.17 will

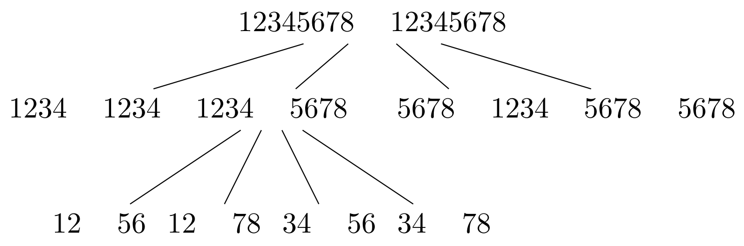
12345678 12345678

(a) A block cluster tree from cluster tree 1.16(b)

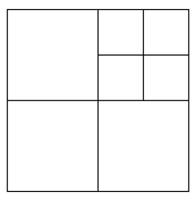


(b) The corresponding hierarchical matrix

Figure 1.17: Example 1



(a) A block cluster tree from cluster tree 1.16(b)



(b) The corresponding hierarchical matrix

Figure 1.18: Example 2

not be efficient if the original matrix is not a low-rank matrix, which is the general case. Consequently, we need more rules to build a block cluster tree which will induce an efficient hierarchical representation of the matrix. A natural idea is to use one of the admissibility conditions defined in Section 1.5.4 to define an admissible block cluster tree.

Definition 1.11 - Admissible block cluster tree.
An *admissible* block cluster tree is a block cluster tree which, for all (t,s) leaves of $\mathcal{T}_{\mathcal{I}\ \ \mathcal{J}}$, satisfies :

$$(t, s) \text{ is admissible or sons}(t) = \quad \text{or sons}(s) =$$

The definition of a $\mathcal{H}$-matrix comes straightforward from an admissible block cluster tree.

Definition 1.12 - $\mathcal{H}$-matrix.
Let A be a $n \quad n$ matrix defined on $I \quad I$. Let $T_{I\ \ I}$ be an admissible block cluster tree for a given admissibility condition. A is a $\mathcal{H}$-matrix if $\ b$ leaf of $T_{I\ \ I}$, A_b is a low-rank matrix or $|b|$ is small. Then, its $\mathcal{H}$-matrix form A can be obtained with Algorithm 1.7.

Algorithm 1.7 shows how the $\mathcal{H}$-matrix format of a matrix A, together with its admissible block cluster tree can be built, by a recursive procedure.

The result of this procedure is thus an admissible block cluster tree built in a top-down fashion, as well as its corresponding $\mathcal{H}$-matrix. Note that a set of siblings of the tree defines a block clustering of the block cluster associated with their parent node and that the final admissible clustering is defined by the leaves of the tree, which defines the structure of the $\mathcal{H}$-matrix. An example is shown in Figure 1.19, with the following

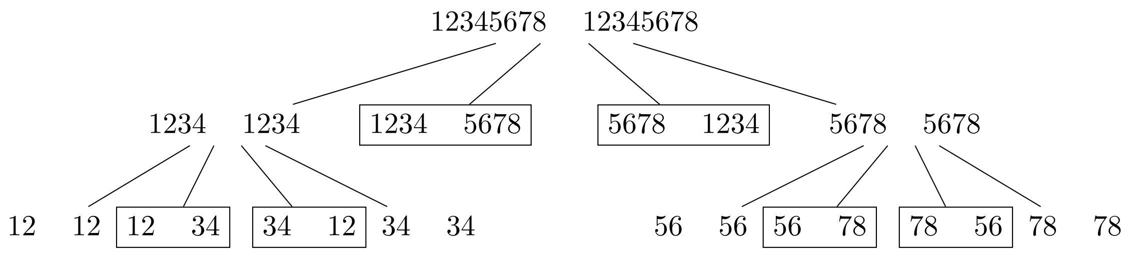
Algorithm 1.7 $\mathcal{H}$-matrix construction

Input A, a matrix defined on row and column indices I.

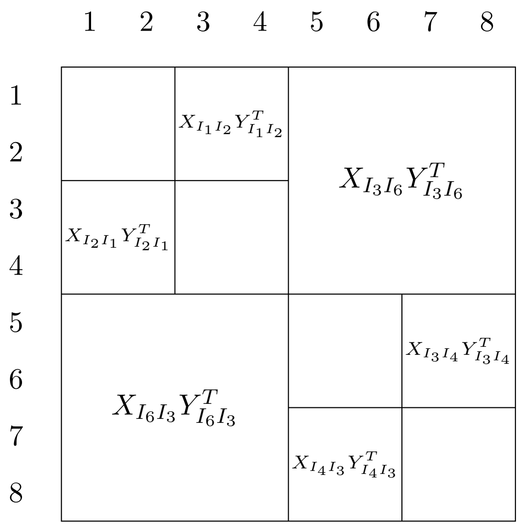
Output A, the $\mathcal{H}$-matrix form of A.

```
 1: initialize list with $[I \quad I]$
 2: while list is not empty do
 3:     remove an element $b$ from list
 4:     if $b$ is admissible then
 5:         $A_b \quad$ low-rank form of $A_b$
 6:     else if $b$ is large enough to split then
 7:         $\begin{matrix} b_1 & b_2 \\ b_3 & b_4 \end{matrix} = b$
 8:         add $b_1, b_2, b_3$ and $b_4$ to list
 9:     else
10:         $A_b \quad A_b$
11:     end if
12: end while
```

simplified admissibility condition : *admissible* *empty intersection.* In this case the admissible block cluster tree has been constructed by splitting the set of matrix indices as follows: $I = I_7 = I_3 \cup I_6 = \ I_1 \cup I_2 \ \cup \ I_4 \cup I_5$.



(a) Admissible block cluster tree from cluster tree 1.16(b). Rectangle nodes are admissible.



(b) Corresponding $\mathcal{H}$-matrix.

Figure 1.19: An admissible block cluster tree and the induced $\mathcal{H}$-matrix.

In a more general case, the structure of a $\mathcal{H}$-matrix is not necessarily as regular as in the provided example which means that the diagonal blocks may not have the same size. Moreover, different admissibility conditions can also be defined.

For more details about $\mathcal{H}$-matrices, we refer the reader to Bebendorf [18], Börm [20] or Hackbusch [64]. Note that $\mathcal{H}$-matrices can be generalized to $\mathcal{H}^2$-matrices [20], which feature nested basis.

1.5.5.3 Hierarchically Semiseparable (HSS) matrices

Although also based on a hierarchical blocking defined by a cluster tree, HSS matrices are substantially different from $\mathcal{H}$-matrices as nested basis are involved. They rely on a bottom-up traversal of a cluster tree.

Definition 1.13 - [103] HSS matrix.
Assume A is an $n \quad n$ dense matrix, and $I = \quad 1, 2, \quad, n \quad$. I is recursively split into smaller pieces following a binary tree T with k nodes, denoted by $j = 1, 2, \quad, k \quad \text{root}(T)$. Let $t_j \quad I$ be a cluster associated with each node j of T. (T is used to manage the recursive partition of A.) We say A is an *HSS form* with the corresponding postordered *HSS tree* T if :

1. T is a full binary tree in its postordering, or, each node j is either a leaf or a non-leaf node with two children j_1 and j_2 which satisfy $j_1 < j_2 < j$;

2. The index sets satisfy $t_{j_1} \cup t_{j_2} = t_j$ and $t_{j_1} \setminus t_{j_2} = \quad$ for each non-leaf node j, with $t_k \quad I$;

3. For each node j, there exist matrices $D_j, X_j, Y_j, R_j, W_j, B_j$ (called *HSS generators*), which satisfy the following recursions for each non-leaf node j:

$$D_j \quad A|_{t_j \quad t_j} = \begin{matrix} D_{j_1} & X_{j_1} B_{j_1} Y_{j_2}^T \\ X_{j_2} B_{j_2} Y_{j_1}^T & D_{j_2} \end{matrix} , X_j = \begin{matrix} X_{j_1} R_{j_1} \\ X_{j_2} R_{j_2} \end{matrix} , Y_j = \begin{matrix} Y_{j_1} W_{j_1} \\ Y_{j_2} W_{j_2} \end{matrix} ,$$

   where X_k, Y_k, R_k, W_k and B_k are not needed (since $D_k \quad A$ is the entire diagonal block without a corresponding off-diagonal block). Due to the recursion, only the D_l, X_l, Y_l matrices associated with a leaf node l of T are stored. At any other node j (excluding the root), B_j, R_j, W_j are stored and will be used to generate D_j, X_j, Y_j from D_l, X_l, Y_l (this explains why these matrices are called *generators*).

For instance, the HSS matrix $\widetilde{A}$ associated with a 4 leaves HSS tree has the form:

$$\widetilde{A} = \begin{array}{c|c|cc} D_1 & X_1 B_1 Y_2^T & X_1 R_1 B_3 W_4^T Y_4^T & X_1 R_1 B_3 W_5^T Y_5^T \\ \cline{1-2} X_2 B_2 Y_1^T & D_2 & X_2 R_2 B_3 W_4^T Y_4^T & X_2 R_2 B_3 W_5^T Y_5^T \\ \hline X_4 R_4 B_6 W_1^T Y_1^T & X_4 R_4 B_6 W_2^T Y_2^T & D_4 & \multicolumn{1}{|c}{X_4 B_4 Y_5^T} \\ \cline{3-4} X_5 R_5 B_6 W_1^T Y_1^T & X_5 R_5 B_6 W_2^T Y_2^T & X_5 B_5 Y_4^T & \multicolumn{1}{|c}{D_5} \end{array} \tag{1.2}$$

The construction of such an HSS matrix is achieved through a topological order traversal of the T tree, also referred to as *HSS tree*, picture in 1.20.

Each time a node j is visited, the block-row and block-column corresponding to the related cluster (i.e., $A_{t_j,:} = A_{t_j,(I \quad t_j)}$ and $A_{:,t_j} = A_{(I \quad t_j),t_j}$, respectively) are compressed into a low-rank form. Note that, apart from the leaves, the compressed block-row or block-column is formed by combining the result of previous compressions. Algorithm 1.8 gives a more formal description of the construction of an HSS matrix.

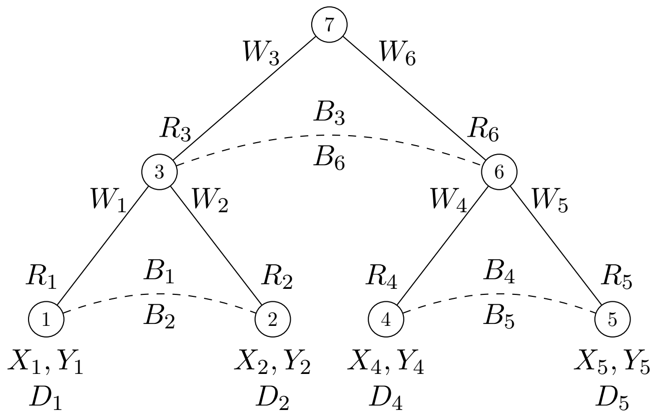


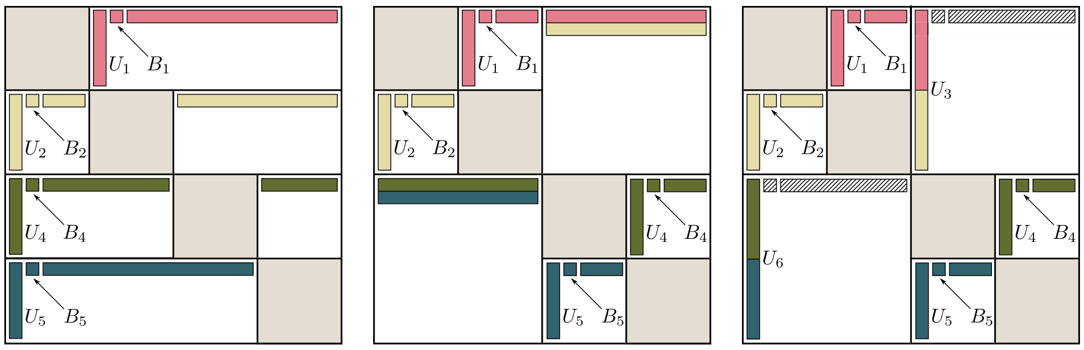
Figure 1.20: HSS tree

Algorithm 1.8 HSS matrix construction

Input A a matrix defined on row indices I and column indices J.
Input T_I a cluster tree on I with n postordered nodes
Output A the HSS matrix form of A.

```
1:  for $i = 1$ to $n$ do
2:      node $i$ is associated with a cluster $c$
3:      if $i$ is a leaf then
4:          compress block row $A_{c,:}$ and block column $A_{:,c}$ ignoring any previous bases $X, Y$:
            $A_{c,:} = X_i \widetilde{T}_{c,:}$
            $A^T_{:,c} = Y_i \widetilde{T}^T_{:,c}$
5:      else
6:          $i$ is assumed to have two children $i_1$ and $i_2$ associated with two clusters $c_1$ and $c_2$
7:          form $A_{c,:}$ and $A_{:,c}$ using $A_{c_1,:}$, $A_{:,c_1}$, $A_{c_2,:}$ and $A_{:,c_2}$ ignoring any $X, Y$ bases
8:          compress $A_{c,:}$ and $A_{:,c}$ to obtain generators $R_{c_1}$, $R_{c_2}$, $W^T_{c_1}$ and $W^T_{c_2}$ :
            $$A_{c,:} = \begin{matrix} R_{c_1} \\ R_{c_2} \end{matrix} \widetilde{A}_{c,:} \qquad A^T_{:,c} = \begin{matrix} W_{c_1} \\ W_{c_2} \end{matrix} \widetilde{A}^T_{:,c}$$
9:          identify $B_{c_1}$ and $B_{c_2}$ from $T_{c_1,:}$ and $T_{c_2,:}$, respectively. At line 7, the columns of
            $A_{c_1}$ that do not go to $A_{c,:}$ form $B_{c_1}$, and similarly for $B_{c_2}$.
10:     end if
11: end for
```

The first steps of Algorithm 1.8 are sketched in Figure 1.21, based on the HSS form of (1.2). In Figure 1.21(a), the four blockrows are compressed, yielding U_i's and B_i's and short and wide $\widetilde{V}_i^T$'s. In Figure 1.21(b), the blockrows are merged (which corresponds to move in the HSS tree from the leaves to their parent). The corresponding part of the $\widetilde{V}_i^T$'s from previous level are thus merged, and recompressed in Figure 1.21(c), yielding U_3 and U_6. Note that the colors in Figure 1.21(c) illustrate that, for instance, U_3 can be recovered from U_1 and U_2 through small generators. In practice, these blockrow compression steps are followed by blockcolumn compressions, which are performed similarly. This construction is extensively explained in Xia et al. [106] through a similar 4 4 example.



(a) First step of row compression at the leaf level l. The blockrows are entirely compressed.

(b) Second step of row compression, corresponding to the $l-1$ level (parents of the leaves). Matrices generated through compression at previous level are partly merged.

(c) After the merge, blockrows are recompressed.

Figure 1.21: Pictorial illustration of the blockrow compression steps of the construction of the HSS form of $\widetilde{A}$.

Because of the particular structure of HSS matrices, the standard LU factorization cannot be used efficiently to factorize a HSS matrix. Instead, a ULV [28, 103] factorization must be used to ensure that the HSS structure is efficiently exploited.

Based on some theoretical assumptions of the maximal rank r found in the construction of the HSS form of the matrix A of order n [29, 47], it has been shown that the complexity of the HSS construction is $O(rn^2)$ [103, 106]. Based on the same assumptions, the complexity of the multifrontal HSS factorization [103] could be derived for matrices coming from the discretization of the Poisson and Helmholtz equations, in the *fully-structure* case only (when frontal matrices are always represented with HSS matrices and thus are never stored in a standard full-rank format, which requires the assembly of frontal matrices to be done in HSS format). In practice, codes which are able to run on large problems are *partially-structured* (the frontal matrices are first stored in a standard full-rank format, HSS representations are used in the L,U parts and the contribution blocks are stored in a standard full-rank format). These complexity results are summarized in Table 1.3, for both full-rank and HSS multifrontal methods on matrices coming from the 2D and 3D discretizations of the Poisson and Helmholtz equations, presented in Section 1.6.4.1. Note that in practice, the ranks observed may be higher than the theoretical ones [47, 79]. We will further develop the complexity analysis with experimental results in Sections 2.3.2 and 4.3.

For a more detailed presentation and study of HSS matrices, we refer the reader to Xia [103], Xia et al. [105] and Wang et al. [99; 101].

1.6 Experimental environment

1.6.1 HSS solvers

Our solver based on a new low-rank format (presented in Chapter 2) will be compared with two partially-structured multifrontal HSS solvers:

- Hsolver [102] is geometric (i.e., the nested dissection used to reorder the original

Problem	Rank	full-rank multifrontal		HSS multifrontal	
		Mry	Ops	Mry	Ops
`PoissonN` 2D `HelmholtzN` 2D	$O(1)$ $O(\log(N))$	$O(n\log(n))$	$O(n^{3\ 2})$	$O(n\log\log(n))$	$O(n\log(n))$
`PoissonN` 3D `HelmholtzN` 3D	$O(N)$	$O(n^{4\ 3})$	$O(n^2)$	$O(n\log(n))$	$O(n^{4\ 3}\log(n))$

Table 1.3: Theoretical complexities for full-rank and HSS multifrontal factorizations based on theoretical ranks. The HSS multifrontal is assumed to perform all assembly in HSS format. *Mry* stands for number of entries in factors. *Ops* stands for number of operations for the factorization. n is the size of the matrix. N is the size of the underlying square or cubic mesh: $n = N^2$ in 2D, $n = N^3$ in 3D.

matrix and form the assembly tree as well as the HSS trees used to represent frontal matrices in HSS forms rely on the knowledge of the underlying geometry of the mesh) and will be used in Sections 2.3.2 and 4.3 to compare the memory and operation experimental complexities. This solver has been chosen to study these aspects because results have already been published [101] and were thus available to us.

- StruMF [81] is algebraic (i.e., no knowledge on the geometry is needed) and will be used in Section 4.3.1 to compare general performance. Our solver being also algebraic, StruMF was more suitable for a comparison of the two codes. Moreover, the StruMF solver was available to us to run new experiments.

All the studies performed with HSS technologies have been done in collaboration with the Lawrence Berkeley National Laboratory (Xiaoye Sherry Li, Artem Napov and François-Henry Rouet).

1.6.2 The MUMPS solver

MUMPS [8, 9, 11, 13] is a MUltifrontal Massively Parallel sparse direct Solver, developed mainly in Bordeaux, Lyon and Toulouse (France). It started in 1996 with the European project PARASOL, first targeting distributed-memory architectures. It is inspired by the shared-memory code MA41 by Amestoy and Duff [3, 4]. MUMPS main features include MPI [93] parallel factorization and solve, handling of symmetric positive definite, general symmetric and unsymmetric matrices, multi-arithmetic capability, backward error analysis, iterative refinement [15]. The iterative refinement is an iterative process which increases the solution accuracy, as shows Algorithm 1.9.

Algorithm 1.9 Iterative refinement.

```
Input  a matrix $A$, its factors $L$ and $U$, a computed solution $\widehat{x}$ to $Ax = b$
Output  $\widehat{x}$ is replaced by a more accurate solution
1: while residual $||Ax - b|| >$ tol do
2:     $r = Ax - b$
3:     solve $Ad = r$        ▶ using the factors $L$ and $U$ computed during the factorization
4:     $x \quad x - d$
5: end while
```

MUMPS also provides many numerical features such as symmetric and unsymmetric threshold pivoting, null pivot detection, 2-by-2 pivots (symmetric), delayed pivots, scaling [12, 87]. This guarantees the robustness of the code. Work is also in progress to exploit shared-memory parallelism using OpenMP [72].

Many orderings of the matrix can be used, from a user defined permutation to standard packages or algorithms: `AMD` [7], `PORD` [91], `METIS` [68], `SCOTCH` [83] and their parallel versions `ParMETIS` [70] and `PT-SCOTCH` [30].

MUMPS relies on the elimination tree of A (or of $A + A^T$ in the unsymmetric case). Both node and tree parallelisms are exploited. Sequential nodes are referred to as *type 1* nodes. Parallel nodes are referred to as *type 2* nodes. In a parallel environment, the root node can be on demand processed using a `ScaLAPACK` [31] two-dimensional block cyclic distribution scheme; in this case, the root node is referred to as a *type 3* root. In a *type 2* node, the master process is assigned to process the fully-summed rows and is in charge of organizing computations; the non fully-summed rows are distributed following a one-dimensional row-wise partitioning, so that each slave holds a range of rows. An illustration of these tree and node parallelisms in MUMPS is given in Figure 1.22.

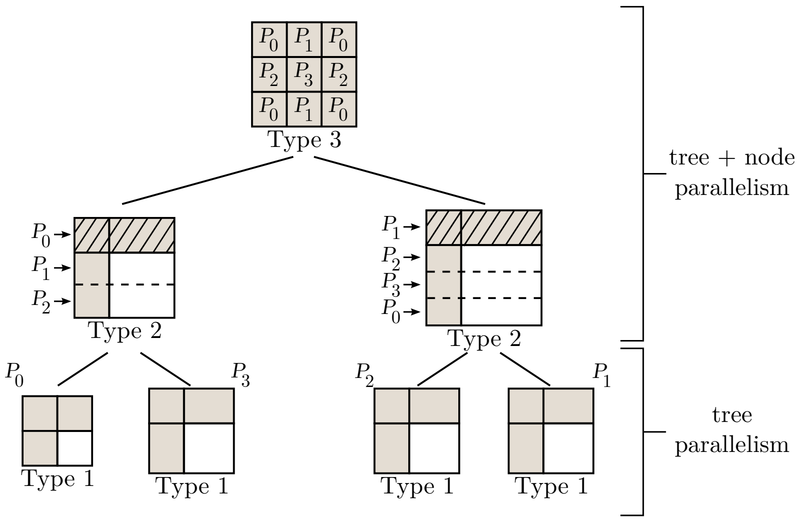
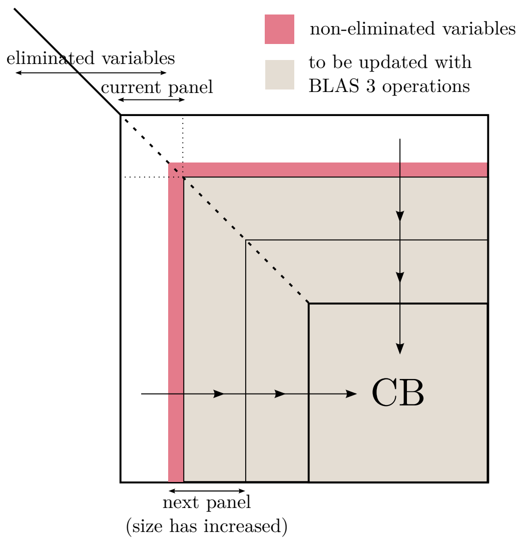
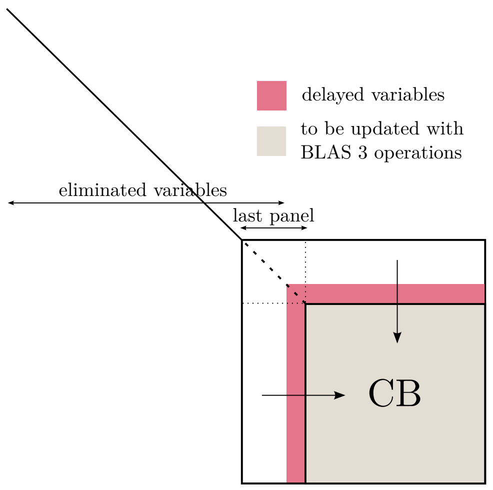


Figure 1.22: Different kinds of nodes in MUMPS (unsymmetric case). Type 3 node is a `ScaLAPACK` root distributed following a 2D block-cyclic scheme. Type 2 nodes are partitioned by blocks of rows. The master process (hatched) holds the fully-summed rows, i.e., the (1,1) block and the (1,2) block (corresponding to U_{12}). The slaves hold block rows. In this example, type 2 nodes exploit node and tree parallelism. Type 1 nodes are processed on a single process and exploit only tree parallelism. The shaded part of each front represents its fully-summed part.

Many factorization types can be performed depending on the properties of the input matrix (which can be given distributed or centralized, assembled or in elemental format): LU, LDL^T, LL^T. An out-of-core facility is also provided to reduce the memory requirements of the factorization. The factorization is performed using BLAS 2 and BLAS 3 operations [41] through a panelwise scheme (a vertical panel is a diagonal block and all the blocks below it, in which case its width is the number of columns of the diagonal block; an horizontal panel is a diagonal block and all the blocks on the right of it, in which case its width is the number of rows of the diagonal block; for the sake of simplicity, we will always use panel without any distinction) where the width of panels commonly depends on the features of the underlying architecture. Within each panel, a pivot is searched for and, if found, eliminated after permutation to bring it to the adequate position. This yields a row of U and a column of L. BLAS 2 right-looking updates are then performed within the panel. The processing of a panel ends when no more pivots can be found, which happens when either all the pivots of the current panel have been eliminated, or some pivots could not be eliminated (in which case current panel is shortened and these non-eliminated variables are postponed to the next panel). After the end of the panel processing occurs, all the variables outside the panel are updated through a BLAS 3 right-looking update. These ideas are illustrated in Figure 1.23(a) (postponed uneliminated pivots) and in Figure 1.23(b) (pivots delayed to parent front).



(a) At the end of current panel processing, some variables may not be eliminated, in which case they are postponed to the next panel, hoping they will be eliminated later during the front processing. All the trailing submatrix is updated with BLAS 3 operations using current panel s newly eliminated variables.

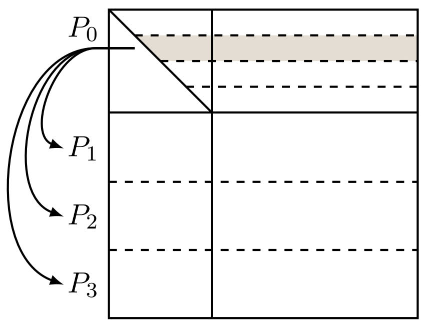


(b) At the end of the processing of the last panel, some variables may still not be eliminated, which means they cannot be eliminated within current front. They are then delayed to the parent front.

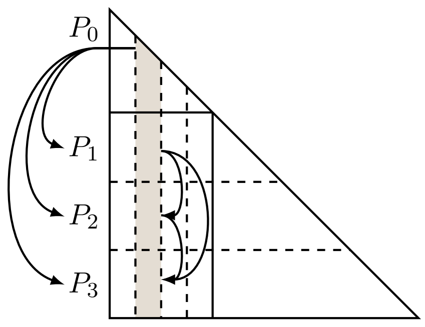
Figure 1.23: `MUMPS` sequential (type 1) factorization scheme with pivoting.

This panel scheme is used for the communication pattern in type 2 nodes, as illustrated in Figure 1.24. P_0 is the master process; other processes are slaves and receive messages from the master (unsymmetric case) or from the master and other slaves (symmetric case). When all the rows of a panel have been processed, the master updates the rows of L corresponding to fully-summed variables (without communication). It also sends data to the slaves so that they can update the rows of L corresponding to non fully-summed variables.

Our work on exploiting low-rank approximations in multifrontal methods was motivated by different common issues related to this class of solvers: the large memory consumption (the maximum size of the stack) and the computational complexity. Moreover, as problem size increase, the volume of communications tends to become more and more critical in the efficiency of solvers. Designing a low-rank format which is capable to tackle these issues while maintaining the robustness and features of a given solver is thus needed.



(a) Unsymmetric case.



(b) Symmetric case.

Figure 1.24: Communication pattern used within type 2 nodes. P_0 is the master process. In the unsymmetric case, it is the only process sending messages. In the symmetric case, slaves also send messages to other slaves.

1.6.3 Code_Aster

Among the industrial applications concerned with solving large systems, some are very critical and this is the case of the applications at Électricité De France[1]. EDF has to guarantee the technical and economical control of its means of production and transportation of electricity. The safety and the availability of the industrial and engineering installations require mechanical studies before taking any decision about their exploitation. These studies are often based on numerical simulations, which are done using Code_Aster [1], an internal software, freely distributed since 2001. Code_Aster's main goal is the analysis of structures and thermomechanics for studies and research. It offers a full range of multiphysical analysis and modeling methods that go well beyond the standard functions of a thermomechanical calculation code: from seismic analysis to porous media via acoustics, fatigue, stochastic dynamics, etc. To achieve efficient and accurate simulations, Code_Aster needs to solve larger and larger systems of equations, in many different contexts. This step is performed mainly using `MUMPS` and is usually the bottleneck of such a code, both in terms of memory and computational costs. For EDF's needs, it is thus critical to keep improving multifrontal solvers to tackle new challenging problems, without conceding the versatility and the robustness of the solver.

1.6.4 Test problems

The experiments presented throughout this dissertation were run on a set of problems coming from different physics applications. We assess the efficiency of our approach in two distinct contexts: as a direct solver (the set of problem is given in Table 1.4) and as a preconditioner (the set of problem is given in Table 1.5).

A couple of academic problems are used to evaluate the potential of our approach. Then, problems from industrial applications are used to demonstrate the effectiveness of the BLR factorization.

[1]Électricité De France, usually shortened to EDF, is the main French utility company. It is the world s largest producer of electricity, primarily from nuclear power.

1.6.4.1 Standard operators

These problems are mainly used in Chapter 4 to evaluate the influence of different parameters on the efficiency of our method, and to obtain an insight of the gains which can be obtained.

We first used the Poisson equation based on the Laplacian operator discretized on a regular cubic 3D $N \quad N \quad N$ grid with a 7-point stencil:

$$u = f \tag{1.3}$$

with $u = 1$ on the boundary (Dirichlet conditions). This yields a symmetric positive definite real matrix whose size depends on the mesh size. Matrices based on this operator discretized on a $N \quad N \quad N$ mesh will be named `PoissonN`. For instance, `Poisson128` is widely used in Chapter 4. Also, when experiments require dense matrices (in Chapter 2), we extract from these matrices their respective Schur complement associated with the top level separator of a *geometric* (i.e., computed by hands) nested dissection tree. These dense matrices are then called `geo_root_PoissonM`. Finally, when the Schur complement is associated with the top level separator of an *algebraic* (i.e., computed with `METIS`) nested dissection tree, the corresponding dense matrix is similarly called `alg_root_PoissonM`.

The second academic problem we study is a 3D finite-difference frequency-domain impedance matrix using a regular 27-point stencil discretization of the Helmholtz equation [82]:

$$- \quad - \frac{\ ^2}{v(x)^2} \quad u(x, \ ) = s(x, \ ), \tag{1.4}$$

where is the Laplacian, is the angular frequency, $v(x)$ is the seismic velocity field, and $u(x, \ )$ is called the time-harmonic wavefield solution to the forcing term $s(x, \ )$. The resulting matrix is a complex unsymmetric matrix. Matrices based on this operator discretized on a $N \quad N \quad N$ mesh will be named `HelmholtzN`. For instance, `Helmholtz128` is widely used in Chapter 4.

1.6.4.2 Direct solver context

A set of different matrices has been used in a direct solver context. Some of them come from EDF [2] industrial applications with Code_Aster. The other ones come from the University of Florida Sparse Matrix Collection [36] and were used in Xia [103]. In Table 1.4, several details about these matrices are given, and data about full-rank executions with MUMPS are also presented.

The mesh associated with the `pompe` problem is illustrated in Figure 1.25.

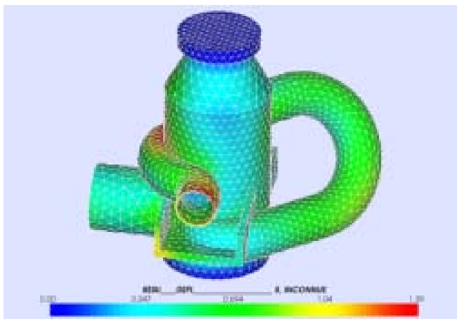


Figure 1.25: Discretized mesh associated with the `pompe` matrix.

Name	Prop.	N (10^6)	NZ (10^6)	factors storage	ops	CSR (full rank)	application field
`th-r7`	3D/sym.	8	118	128 GB	267E+12	4 10^{-14}	EDF thermal
`me-r12`	2D/sym.	134	1140	200 GB	151E+12	8 10^{-14}	EDF mechanical
`pompe`	3D/sym.	0 8	28	2 6 GB	5E+11	1 10^{-15}	EDF pump for nuclear backup circuit
`cont-300`	3D/sym.	0 18	0 53	0 2 GB	7E+9	6 10^{-8}	optimization (linear, UFL)
`kkt_power`	3D/sym.	2	8	3 8 GB	4E+12	3 10^{-10}	optimal power flow (nonlinear, UFL)
`d-plan-inco`	2D/sym.	2	21	2 6 GB	4E+11		EDF material incompressibility
`apache2`	3D/spd	0 7	5	1 3 GB	2 3E+11	1 10^{-14}	structural problem
`ecology2`	3D/spd	1	5	0 3 GB	1 8E+10	3 10^{-15}	circuit theory
`G3_circuit`	3D/spd	2	8	0 8 GB	6 7E+10	6 10^{-15}	circuit simulation
`parabolic_fem`	3D/spd	0 5	4	0 2 GB	8 5E+9	3 10^{-15}	diffusion-convection
`thermomech_dM`	3D/spd	0 2	1	0 06 GB	8 2E+8	7 10^{-16}	thermics
`tmt_sym`	3D/spd	0 7	5	0 3 GB	1 3E+10	2 10^{-15}	electromagnetics

Table 1.4: Set of problems used for the experimentations. They come from finite-difference or finite elements methods simulations. CSR = Componentwise Scaled Residual $= \max_i \frac{|\mathbf{b} - \mathbf{Ax}|_i}{(|\mathbf{b}| + |\mathbf{A}|\ |\mathbf{x}|)_i}$. All matrices use double precision real arithmetic.

Additionally to these symmetric problems, an unsymmetric complex extensive study of seismic modeling and full-waveform inversion is given in Section 5.2.

1.6.4.3 Preconditioner context

Low-rank techniques can also be efficient as preconditioners. An analyze of the performance of such preconditioners is given in Section 5.1.2 with problems presented in Table 1.5.

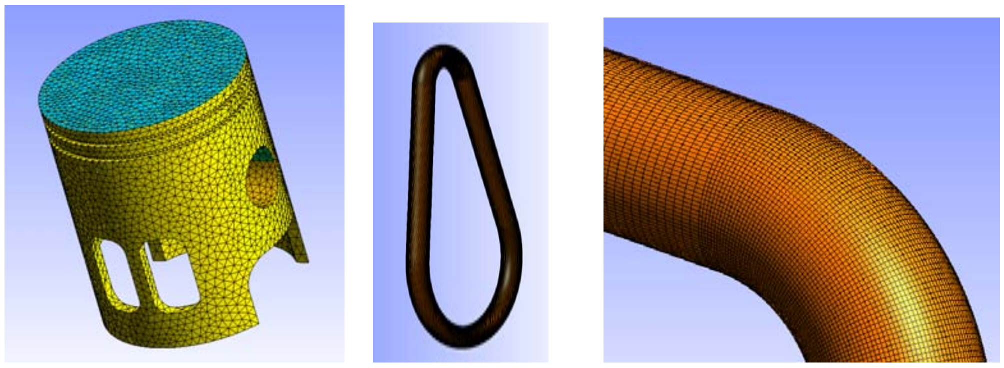
	N	NZ	application
`piston`	$1,377,531$	$54,713,045$	external pressure force on the top (see Figure 1.26(a))
`perf`	$2,016,000$	$75,850,976$	“cavity” hook subjected to internal pressure force (see Figure 1.26(b) and 1.26(b))

Table 1.5: Problems studied in the context of a BLR preconditioned conjugate gradient. The default accuracy needed for EDF applications is 10^{-6}. These problems are symmetric positive definite real double precision.

These two problems will be used within a conjugate gradient method. All these problems are EDF test-cases for nuclear power stations: a piston with external pressure force, a cavity hook with internal pressure force and a blocking device for actuator and pump. Moreover, `perf`, is a challenging problem for EDF as no fully satisfying preconditioner has been found, i.e., the convergence is much slower than for usual EDF problems.

In EDF's process, the preconditioner is computed through a single precision multifrontal factorization of A, so that the same process can be used for almost any matrix.

When the number of iterations is low, such as for `piston`, the global process roughly requires half the memory and operation counts of a double precision multifrontal factorization, for the same accuracy. When the number of iterations is high, this method is still chosen because memory constraints are usually stronger than other constraint in EDF's context.



(a) `piston` (b) `perf001d` (c) `perf001d` (Details)

Figure 1.26: The geometries and discretization of EDF's problems studied in a BLR preconditioned conjugate gradient context.

1.6.5 Computational systems

Several computational systems have been used for the experimental sections of this dissertation:

- `Conan`, SGI machine at ENSEEIHT: 1 node with four octo-core AMD Opteron 6620 processor @3.0GHz and 512-GB memory per node. Used for some of the experiments in Chapter 4 and for the experiments in Section 5.1.
- `Hyperion`, Altix ICE 8200 machine at the Calcul en Midi-Pyrénées resource center (CALMIP): 352 nodes with two quad-core Intel Xeon 5560 processors @2.8 GHz and 32 GB memory per node. Used under the allocation 2012-p0989. Used for some of the experiments in Chapter 4.
- `Hopper`, Cray XE6 machine at the National Energy Research Scientific Computing Center (NERSC): 6384 nodes with two twelve-core AMD Opteron 6172 processors @2.1 GHz and 32 GB memory per node. Used for experiments in Sections 4.2.6.
- `Thera`, 1 node with eight opto-core AMD Opteron node with 384-GB per node. Used for experiments in Section 5.2.

Chapter 2

Block Low-Rank (BLR) format

In this chapter, we propose a new structure for exploiting the low-rank property of a matrix, called Block Low-Rank. It is based on a flat, non-hierarchical natural matrix structure. We justify why we decided not to use the $\mathcal{H}$ format, nor the HSS format and explain the main advantages of BLR in the context of an algebraic, general multifrontal solver. We also give a comparison of these three formats in terms of compression cost and efficiency in order to evaluate the quality of each format. Finally, the numerical accuracy of the BLR compression and the compression parameter ε is discussed.

2.1 De nition

The BLR format is a flat, non-hierarchical block matrix format, as defined in Definition 2.1.

Definition 2.1 - Block Low-Rank matrix (BLR).
Given an admissibility condition, let P be an admissible block clustering with p^2 clusters. Let A be an n n matrix.

$$\widetilde{A} = \begin{matrix} B_{11} & B_{12} & & B_{1p} \\ B_{21} & B_{22} & & B_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ B_{p1} & B_{p2} & & B_{pp} \end{matrix}$$

is called a “Block Low-Rank matrix” if $(\sigma, \tau) \in$ 1, 2,   , p ^{2}, there exists $k_{\sigma\tau}$ such that $B_{\sigma\tau}$ is a low-rank block with rank $k_{\sigma\tau}$.

The definition requires at least one block to be low-rank, so that the BLR representation of A always offers memory storage reduction. However, note that in practice, many (if not all) blocks are usually low-rank, which ensures the overall efficiency of the representation.

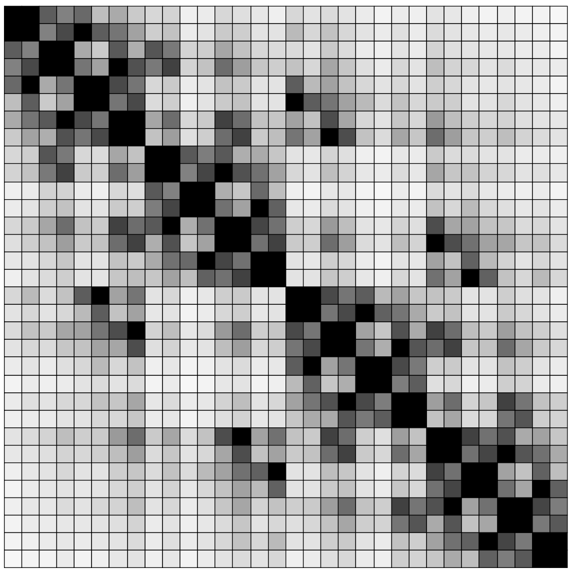
The structure of a BLR matrix is not hierarchical: a flat block matrix structure is used. BLR matrices can thus be viewed as a particular case of $\mathcal{H}$-matrices where all the subblocks have been subdivided identically, i.e., where all the branches of the block cluster tree have the same depth. However, because the basis used to approximate the $B_{\sigma\tau}$ blocks are not nested (each block is approximated independently), BLR matrices are not a particular case of HSS matrices. In Equation (2.1), an example of a 4 4 BLR matrix is given. This latter case, where only the diagonal blocks are not low-rank, occurs often in practice, as diagonal blocks represent self-self interactions.

$$\widetilde{A} = \begin{array}{c|c|c|c} D_1 & X_{12}Y_{12}^T & X_{13}Y_{13}^T & X_{14}Y_{14}^T \\ \hline X_{21}Y_{21}^T & D_2 & X_{23}Y_{23}^T & X_{24}Y_{24}^T \\ \hline X_{31}Y_{31}^T & X_{32}Y_{32}^T & D_3 & X_{34}Y_{34}^T \\ \hline X_{41}Y_{41}^T & X_{42}Y_{42}^T & X_{43}Y_{43}^T & D_4 \end{array} \tag{2.1}$$

Figure 2.1 shows the global structure of the BLR representation of a dense Schur complement of order 128 128 corresponding to the top level separator of a 128 128 128 `Laplacian` problem, with a low-rank threshold set to 10^{-14}. The numbering scheme illustrated in 2.1(a) is recursive although this is only required for $\mathcal{H}$ and HSS matrices. The diagonal blocks are full-rank. A large majority of off-diagonal blocks are low-rank. A strong correlation between distance between clusters in Figure 2.1(a) and rank in Figure 2.1 can be observed.

27	28	31	32
25	26	29	30
19	20	23	24
17	18	21	22
11	12	15	16
9	10	13	14
3	4	7	8
1	2	5	6

(a) Numbering scheme of the graph associated with the Schur complement. The numbers give the ordering of the 32 blocks of 32 16 = 512 variables within the BLR structure. Any other numbering of the clusters would give equivalent results, as the rank only depends on the interaction between two clusters.



(b) Structure of a BLR matrix. The darkness of a block is proportional to its storage requirement (the lighter a block is, the smaller is the memory needed to store it). Each block in the matrix is of size 512 512.

Figure 2.1: Illustration of a BLR matrix of a dense Schur complement of a 128 128 128 `Laplacian` problem with a low-rank threshold set up to 10^{-14}. The corresponding clustering of its 128 128 planar graph into 4 8 = 32 blocks is also given.

2.2 Motivations to use BLR in a multifrontal context

It has been shown that $\mathcal{H}$ and more extensively HSS [103] matrices can be potentially embedded in multifrontal solvers. These approaches, however, either rely on the knowledge of mathematical properties of the problem or on the geometric properties of the domain on which it is defined (i.e., the discretization mesh) or lack some of the features that are essential for a general purpose, sparse, direct solver like, for instance, robust threshold pivoting or dynamic load balancing in a parallel environment. Our objective is, instead, to exploit low-rank approximations within a general purpose, multifrontal solver which aims at providing robust numerical features in a parallel framework and where no knowledge of the problem can be assumed except the matrix itself. In such a context, hierarchical structures are likely to be too hard to handle and may severely limit the necessary flexibility:

- In a parallel multifrontal solver, frontal matrices may be statically or dynamically partitioned, in an irregular way in order to achieve a good load and memory balance; hierarchical formats may pose heavy constraints or may be complex to handle in this case.
- The HSS format achieves good compression rates only if each node of the HSS tree defines an admissible cluster. This means that the diagonal blocks cannot be permuted in an arbitrary way but have to appear along the diagonal in a specific order which depends on the geometry of the separator (see Section 1.5.5.3). This information may not be available or too complex to extrapolate in a general purpose, algebraic solver.
- In HSS matrices the compressions are achieved through SVD or RRQR decompositions on block-rows and block-columns, i.e., strongly over or under-determined submatrices. This has a twofold disadvantage. First it is based on the assumption that all the columns in a block-row (or, equivalently, rows in a block-column) lie in a small space which has been proven only for some classes of problems and may not be true in general [29]. Second, due to the shape of the data, these operations are often inefficient and difficult to parallelize; this has led researchers to consider the use of communication-avoiding RRQR factorizations or the use of randomization techniques.
- The assembly of a frontal matrix is very difficult to achieve if the contribution blocks are stored in a hierarchical, low-rank format. For this reason, intermediate full-rank representations are used in practice [99, 103, 105]. This, however, comes at a very significant cost (see Figures 2.3 and 2.5).

The analysis presented in Section 2.3, although restricted to a limited number of problems, suggests that a simpler format such as BLR delivers benefits comparable to the $\mathcal{H}$ and HSS ones; BLR, though, represents a more suitable candidate for exploiting low-rank techniques within a general-purpose, multifrontal solver as it presents several advantages over hierarchical formats:

- The matrix blocking is flat, i.e., not hierarchical, and no relative order is imposed between the blocks of the clustering. This is a very valuable feature because it simplifies the computation of the index clustering and delivers much greater flexibility for distributing the data in a parallel environment.
- The size of blocks is homogeneous and is such that the SVD or RRQR operations done for computing the low-rank forms can be efficiently executed concurrently with sequential code (for example, the LAPACK `_GESVD` or `_GEQP3` routines). This property is extremely valuable in a distributed memory environment where reducing the volume of communications is important.

- The L_{21} submatrix can be easily represented in BLR format with a block clustering induced by those of the L_{11} and L_{22} submatrices.
- Pivoting techniques seem hard and inefficient to use when factorizing a matrix stored in a hierarchical format; this has led researchers to employ different types of factorizations, e.g. a ULV factorization [28, 106], which also aims at factorizing a HSS matrix without using the original full-rank form. BLR format is more naturally suited for applying partial threshold pivoting techniques within a standard LU factorization, although it may induce a dynamic change in the BLR structure which can lead to performance loss (in terms of compression), see Section 3.4.4.
- The assembly of frontal matrices is relatively easy if the contribution blocks are in BLR format. We also have the option of switching to an intermediate full-rank format because the compression is relatively cheap (see Figures 2.3 and 2.5).

Motivated by the previous observations, we focus in this paper on the BLR format for exploiting, in an algebraic setting, low-rank techniques. The remainder of this paper describes how block clustering is performed and the main algorithmic issues of the low-rank factorization phase.

2.3 Comparison with the other formats in a dense context

A comparative study of the three formats described before ($\mathcal{H}$, HSS and BLR) is presented in this section in order to validate the potential of the BLR format in the context of the development of algebraic methods for exploiting low-rank approximations within a general purpose, multifrontal solver. We first analyze the number of operations required to represent a dense matrix in either low-rank format (called the compression cost, based on Rank-Revealing QR in this study) and compare the memory compression rates obtained on two problems of given sizes. Then, we propose an experimental complexity study for both the operation count and the number of entries in factors. This argumentation should thus be viewed globally as a two steps reasoning based on two different aspects which cannot be considered independently since each one needs the other one to make sense:

1. the compression cost and the number of entries in the given low-rank representation.
2. the complexity of the factorization based on the given low-rank representation.

2.3.1 Compression cost and memory compression

We use the $\mathcal{H}$, HSS and BLR formats for compressing the dense matrices `geo_root_Poisson128` and `geo_root_Helmholtz128` presented in Section 1.6.4.1. The geometric separator of either problem is a 128 128 surface lying in the middle of the cubic domain and the corresponding Schur complement is thus a dense matrix of order 16384. For all three formats, the low-rank threshold ε is set up to 10^{-14} and the clustering was defined by a 8 8 recursive checkerboard partitioning of the separator into blocks of size 256, using the same approach as in the 4 8 case from Figure 2.1(a); this clustering is essentially what is referred to as the *weak admissibility condition* in Börm [20, Remark 3.17]. Other block sizes have been experimented and give comparable results for both problems. These results, illustrated in Figures 2.2, 2.3, 2.4 and 2.5 are in compliance with what Wang et al. [100] observed. Note that the clustering is recursive in order to be efficient for hierarchical formats; this is not needed for BLR. Also note that this kind of clustering is used in practice for regular grids for $\mathcal{H}$ [18, 58, 21] and HSS [105, 102] applications. Because we are interested in compression cost and memory compression, no factorization is performed in this section. In terms of memory compression, Figures 2.2 and 2.4 show that the three

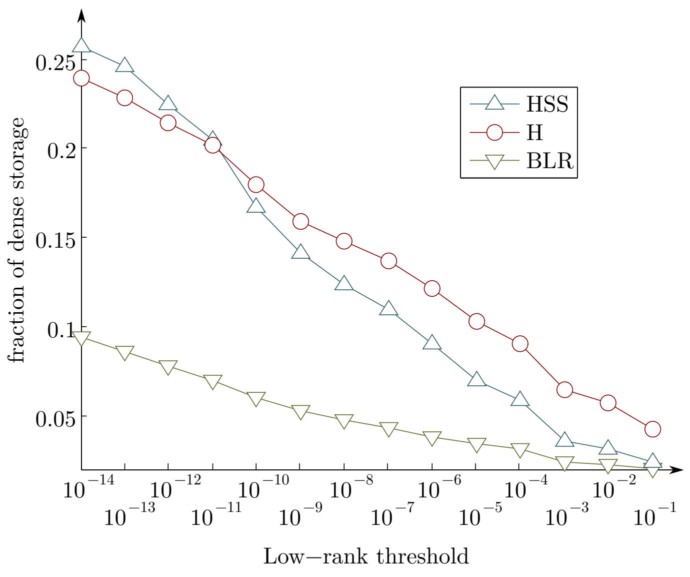


Figure 2.2: Comparison of the number of entries needed to represent `geo_root_Poisson128` as a $\mathcal{H}$, HSS and BLR matrix, expressed as a fraction of the full-rank number of entries in the original matrix. The clustering is hand-computed and geometric.

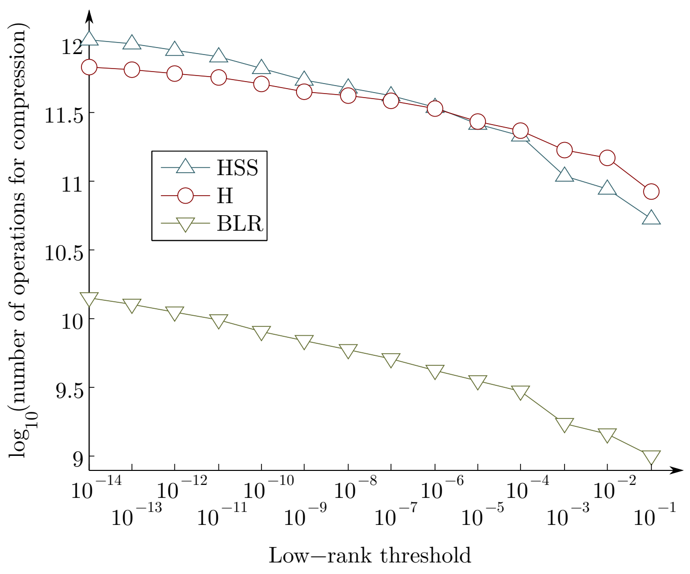


Figure 2.3: Compression cost comparison between $\mathcal{H}$, HSS and BLR formats on the `geo_root_Poisson128` problem, expressed in number of operations required. The clustering is hand-computed and geometric.

formats appear to be roughly equivalent at low precision: the truncation is so aggressive that almost no information is stored anymore. In the case of the `geo_root_Helmholtz128`

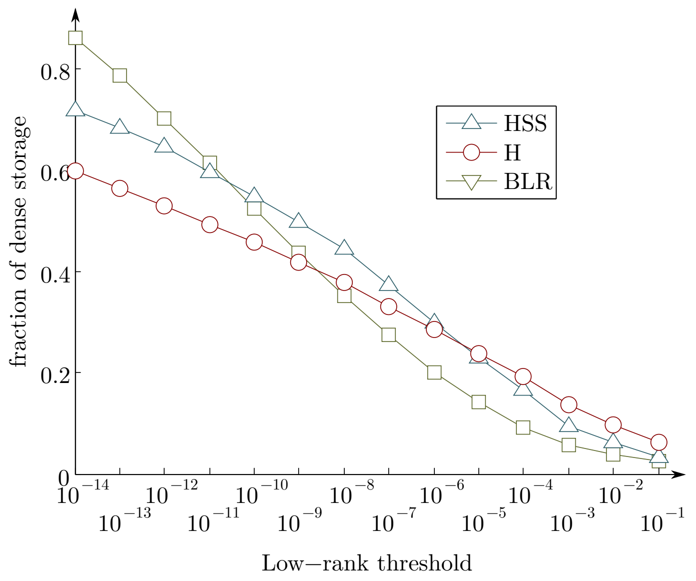


Figure 2.4: Comparison of the number of entries needed to represent `geo_root_Helmholtz128` as a $\mathcal{H}$, HSS and BLR matrix, expressed as a fraction of the full-rank number of entries in the original matrix. The clustering is hand-computed and geometric.

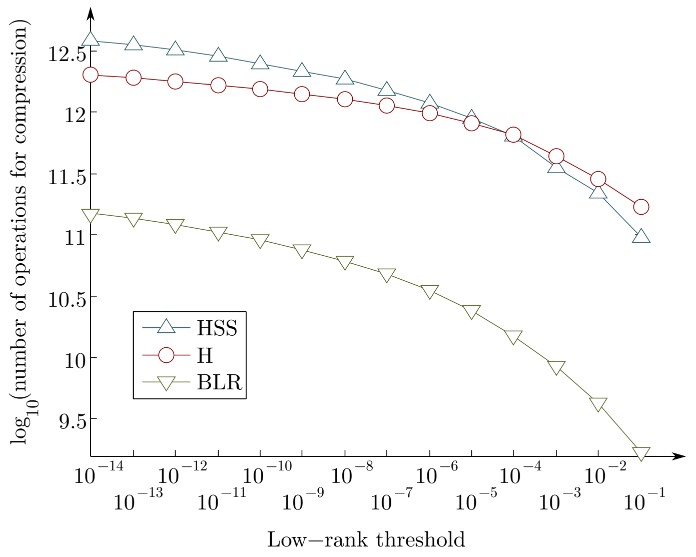


Figure 2.5: Compression cost comparison between $\mathcal{H}$, HSS and BLR formats on the `geo_root_Helmholtz128` matrix, expressed in number of operations required. The clustering is hand-computed and geometric.

matrix, the three formats provide comparable gains with BLR being slightly worse than the other two at high accuracy and better when the approximation threshold is bigger than 10^{-10}. For the `geo_root_Poisson128` matrix, BLR is consistently better than the hierarchical formats although all three provide, in general, considerable gains. These preliminary results suggest that the BLR format delivers gains that are comparable to those obtained with the hierarchical ones on the test problems.

Figures 2.3 and 2.5 show the cost of computing the $\mathcal{H}$, HSS and BLR formats starting from a standard dense matrix. This is computed as the cost of the partial RRQR factorizations; because for $\mathcal{H}$ and HSS matrices these operations are performed on much larger blocks, the conversion to BLR format is much cheaper with respect to the cases of hierarchical formats and also with respect to the cost of an LU factorization of the same front. The cost of a full-rank factorization is indeed 10 (`geo_root_Helmholtz128` problem) or 100 (`geo_root_Poisson128` problem) times larger than the BLR compression cost at full accuracy ($\varepsilon = 10^{-14}$), so the factorization remains dominant. This is one of the reasons which led researchers to investigate other compression methods such as randomized sampling [104] and adaptive cross approximations [18] in the context of hierarchical formats. Moreover, because a truncated rank-revealing QR factorization is used to compress the blocks, the compression cost decreases when the accuracy decreases. This property is extremely useful as it allows the switching from full-rank to low-rank and *vice* versa at an acceptable cost compared to the dense LU factorization cost.

2.3.2 Experimental dense complexity

The results of the previous section, which seem to make BLR format the most suitable for our purpose, have to be counterbalanced by a study on the memory and operations complexity which can be achieved thanks to these techniques. This will be done only for the HSS format (and more particularly with the Hsolver package, see Section 1.6.1) as no results on experimental dense complexity (for theoretical complexity, see Section 1.5.5) for $\mathcal{H}$ matrices could be found in the literature.

For HSS, we exploited the operation counts given in Wang et al. [101] for the 3D Helmholtz problem in the dense case to obtain an experimental complexity with 5 N N N meshes ($N = 100$, $N = 200$, $N = 300$, $N = 400$ and $N = 500$). To obtain a dense matrix, they perform a geometric nested dissection of the domain and compute in full-rank the dense Schur complement associated with the top level separator. The dense matrices obtained are thus of size $n = N^2$ and correspond to what is called `geo_root_HelmholtzN` in this dissertation. The HSS representations (obtained with a low-rank threshold $\varepsilon = 10^{-3}$) of these matrices are then factorized using a ULV factorization. We did a tentative fit on the published data and obtained a $O(n^{2\ 2})$ complexity. Although the maximum numerical rank should, in theory, also be taken into account in the complexity, this result gives a good insight of the behavior of the method. No results for the memory nor for other problems could be found in the literature.

For BLR, the matrices `geo_root_PoissonN`, `alg_root_PoissonN` and `geo_root_HelmholtzN`, presented in Section 1.6.4.1, are considered. The BLR factorization algorithm used to process these matrices is sketched in Algorithm 2.1 and is performed with a low-rank threshold $\varepsilon = 10^{-4}$.

Note that this algorithm is a slight modification of the standard block LU factorization (Algorithm 1.3) where the standard `Update` phase is split blockwise in order to use low-rank products. The interest of such a BLR factorization will be discussed later in the context of the multifrontal BLR solver in Chapter 3. Although Algorithm 2.1 is sufficient for this experimental complexity study, more algorithms will also be proposed in Chapter 3. Then, the same tentative fit procedure as for the HSS case is used to derive experimental memory and operation complexities.

Figures 2.6 and 2.7 show the memory and operation (respectively) experimental complexities of the BLR dense factorization of the `geo_root_PoissonN` and `alg_root_PoissonN` matrices. Figures 2.8 and 2.9 show the memory and operation (respectively) experimental complexities of the BLR dense factorization of the `geo_root_HelmholtzN` matrix

Algorithm 2.1 Dense BLR LU factorization.

```
1:      ▶ Input: a B   B-block matrix $A$ of size $n$; $A = [A_{I,J}]_{I=1:\mathrm{B},J=1:\mathrm{B}}$
2:      ▶ Output: $A$ is replaced with its $LU$ factors
3:
4:  for $I = 1$ to B do
5:    Factor: $A_{I,I}$     $L_{I,I}U_{I,I}$
6:    for $J = I+1$ to B do
7:      Solve (compute U): $A_{I,J}$     $L_{I,I}^{-1}$   $A_{I,J}$
8:      Compress: $A_{I,J}$     $X_{I,J}$   $Y_{I,J}^T$        ▶ Low-Rank compression
9:    end for
10:   for $K = I+1$ to B do
11:     Solve (compute L): $A_{K,I}$     $A_{K,I}$   $U_{I,I}^{-1}$
12:     Compress: $A_{K,I}$     $X_{K,I}$   $Y_{K,I}^T$        ▶ Low-Rank compression
13:   end for
14:   for $J = I+1$ to B do
15:     for $K = I+1$ to B do
16:       Update: $A_{K,J}$     $A_{K,J} - X_{K,I}$   $(Y_{K,I}^T X_{I,J})$   $Y_{I,J}^T$        ▶ Low-Rank updates
17:     end for
18:   end for
19: end for
```

(results with `alg_root_HelmholtzN` could not be obtained given the much larger amount of computations needed to obtain them, compared to `alg_root_PoissonN`). In all four figures, the theoretical complexities of the full-rank LU factorization are also given. Note that the theoretical full-rank memory and operation complexities do not depend on the shape of the associated separator.

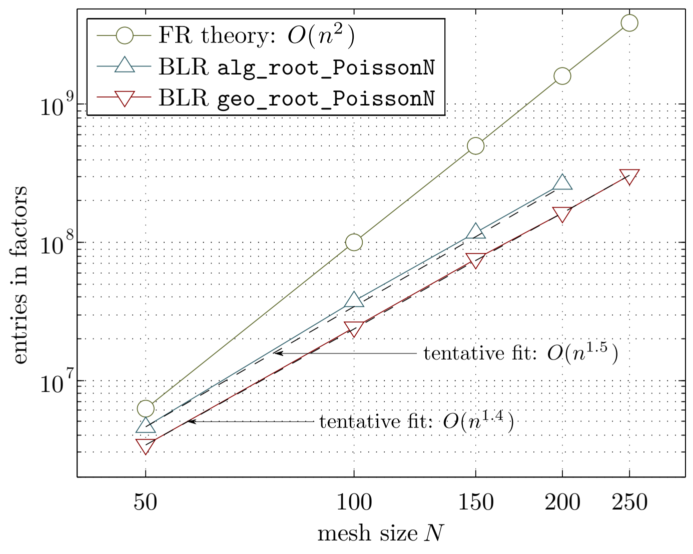


Figure 2.6: Experimental memory complexity for the BLR factorization of matrices `geo_root_PoissonN` and `alg_root_PoissonN` based on a N N N, where N is referred to as the mesh size and $n = N^2$. Note that the FR theoretical complexity does not depend on whether the associated separator is algebraic or geometric.

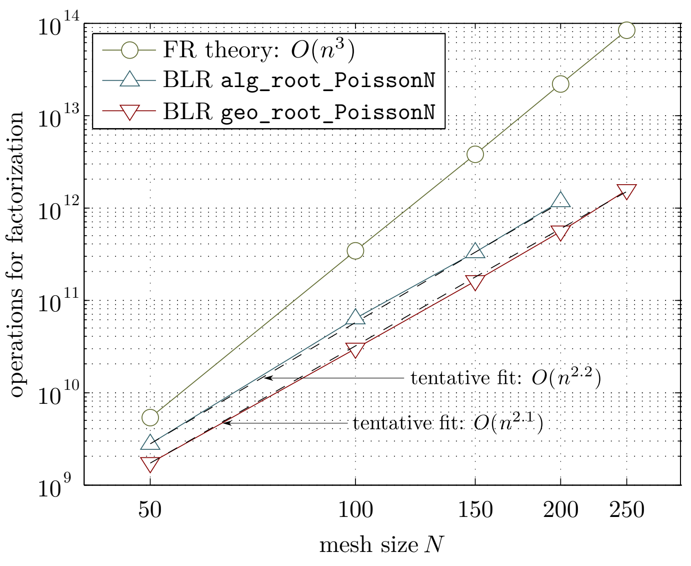


Figure 2.7: Experimental operation complexity for the BLR factorization of matrices `geo_root_PoissonN` and `alg_root_PoissonN` based on a $N \quad N \quad N$, where N is referred to as the mesh size and $n = N^2$. Note that the FR theoretical complexity does not depend on whether the associated separator is algebraic or geometric.

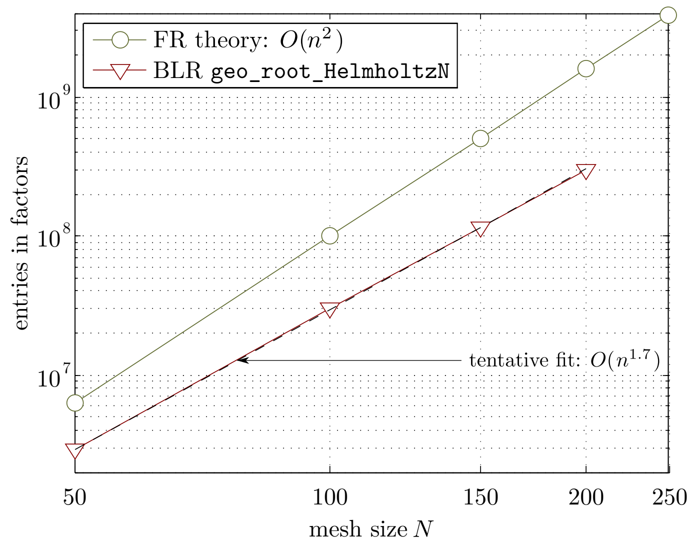


Figure 2.8: Experimental memory complexity for the BLR factorization of matrices `geo_root_HelmholtzN` based on a $N \quad N \quad N$, where N is referred to as the mesh size and $n = N^2$. Note that the FR theoretical complexity does not depend on whether the associated separator is algebraic or geometric.

The complexities obtained are always at least one order less than the corresponding full-rank complexity, both in memory and in operations. Comparing `geo_root_PoissonN` and `alg_root_PoissonN`, we can observe that the complexities are worse with the algebraic separator ($O(n^{1\;5})$ in memory, $O(n^{2\;2})$ in operations) than with the geometric one ($O(n^{1\;4})$ in memory, $O(n^{2\;1})$ in operations), because the regularity of the lat-

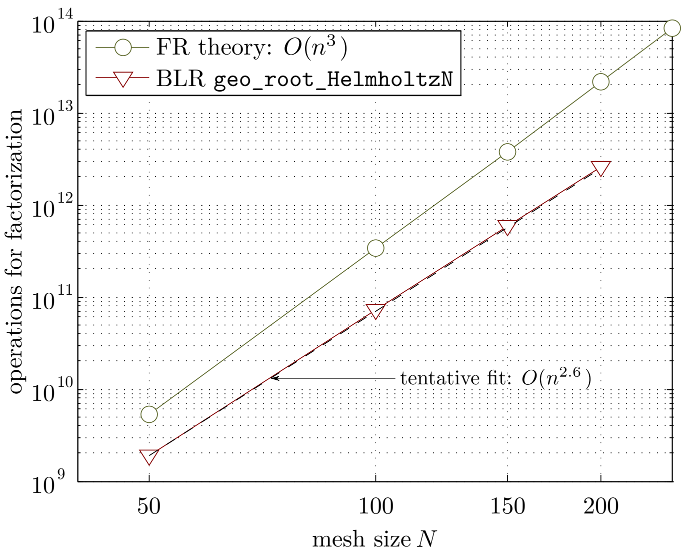


Figure 2.9: Experimental operation complexity for the BLR factorization of matrices `geo_root_HelmholtzN` based on a $N \quad N \quad N$, where N is referred to as the mesh size and $n = N^2$. Note that the FR theoretical complexity does not depend on whether the associated separator is algebraic or geometric.

ter one enhances admissibility and thus compression. The complexities obtained with `geo_root_HelmholtzN` are worse than for `geo_root_PoissonN`, both in memory ($O(n^{1\ 7})$) and complexity ($O(n^{2\ 6})$). This latter complexity $O(n^{2\ 6})$ can be compared with the HSS obtained from Wang et al. [101], which is better ($O(n^{2\ 2})$). Intuitively, it is unsurprising that hierarchical formats have better complexities since the block size naturally increases with the size of the problem. However, BLR has a lower compression cost and offers more flexibility, which are critical in the context of a general algebraic solver. In Chapter 4, we will also show that BLR allows for a good BLAS 3 efficiency and will compare our BLR multifrontal solver with the HSS partially-structured solver StruMF in Section 4.3.1. We summarize in Table 2.1 the memory and operation complexities for BLR (on all studied problems) and HSS (for `geo_root_HelmholtzN` only).

Problem	Hsolver		BLR		FR (theory)	
	mry	ops	mry	ops	mry	ops
`geo_root_HelmholtzN`		$O(n^{2\ 2})$	$O(n^{1\ 7})$	$O(n^{2\ 6})$	$O(n^2)$	$O(n^3)$
`geo_root_PoissonN`			$O(n^{1\ 4})$	$O(n^{2\ 1})$	$O(n^2)$	$O(n^3)$
`alg_root_PoissonN`			$O(n^{1\ 5})$	$O(n^{2\ 2})$	$O(n^2)$	$O(n^3)$

Table 2.1: Summary of the experimental complexities obtained with dense matrices with both HSS and BLR. *mry* stands for number of entries in factors. *ops* stands for number of operations for the factorization. n is the size of the matrix.

2.4 Numerical stability and low-rank threshold

In this section, the numerical stability of the Block Low-Rank format is discussed in the context of the LU factorization. This study is inspired by what Demmel et al. [38] did for standard LU with roundoff errors. We study the Algorithm 2.1 presented in the previous section, in which the low-rank compression is done after the Solve phase. As in the reference paper, no pivoting is performed. For the other variants of the algorithms presented in Chapter 3, equivalent, but slightly different results, could be obtained [26]. In this study, we will use the matrix norm defined by:

$$||A|| = \max_{i,j} |a_{i,j}|$$

The aim of this study is to propose a bound of the relative residual res$(L, U) = \frac{A-LU}{A} = \frac{A}{A}$, where $\widehat{L}$ and $\widehat{U}$ are the LU factors of A computed with Algorithm 2.1, and to analyse it in order to understand the numerical behavior of Algorithm 2.1. The choice of the low-rank threshold is then discussed, based on these conclusions. In Demmel et al. [38], assumptions on the numerical behavior of standard BLAS 3 operations are first stated (u denotes the unit roundoff, c_1, c_2, c_3 are polynomial constants):

1. If $A \in \mathbb{R}^{m \quad n}$ and $B \in \mathbb{R}^{n \quad p}$ then the computed approximation $\widehat{C}$ to $C = AB$ satisfies
$$\widehat{C} = AB + \quad C$$
where $|| \quad C|| \quad c_1(m, n, p)u||A||||B|| + O(u^2)$
2. The computed solution $\widehat{X}$ to the triangular systems $TX = B$, where $T \in \mathbb{R}^{m \quad m}$ and $B \in \mathbb{R}^{m \quad p}$ and satisfies
$$T\widehat{X} = B + \quad B$$
where $|| \quad B|| \quad c_2(m, p)u||T||||\widehat{X}|| + O(u^2)$
3. The block level LU factorization is done in such a way that the computed LU factors of $A_{11} \in \mathbb{R}^{r \quad r}$ satisfy
$$\widehat{L}_{11}\widehat{U}_{11} = A_{11} + \quad A_{11}$$
where $|| \quad A_{11}|| \quad c_3(r)u||\widehat{L}_{11}||||\widehat{U}_{11}|| + O(u^2)$

For conventional BLAS 3 implementations, these assumptions hold [67].

Then and under these assumptions, a bound of res(L, U) was proposed in the case of a (full-rank) block factorization with blocks of size r:

$$|| \quad A|| \quad u \quad (n,r)||A|| + \quad (n,r)||\widehat{L}||||\widehat{U}|| \quad + O(u^2), \tag{2.2}$$

where (n, r) and (n, r) are polynomial constants depending on c_1, c_2 and c_3, and u the unit roundoff, with $n = r$, $(n, r) = 1 + \quad (n - r, r)$ and $(r, r) = 0$.

2.4.1 Bound for the relative residual of a dense Block Low-Rank LU factorization

We use the same techniques to derive a bound for the factorization based on Algorithm 2.1. Assumptions 1, 2 and 3 are still considered valid, and a fourth one is added, assuming the compression is performed using a Singular Value Decomposition (which will be assumed throughout this bound study):

4. If $C \in \mathbb{R}^{m \quad n}$, $X \in \mathbb{R}^{m \quad k}$ and $Y \in \mathbb{R}^{n \quad k}$ then the low-rank approximation $\widehat{C}$ to C at numerical accuracy ε satisfies
$$\widehat{C} = XY^T + \quad C$$
where $|| \quad C|| \quad \varepsilon$

We propose in Theorem 2.1 a bound for the dense BLR factorization, under assumptions 1 to 4.

Theorem 2.1
Let $\widehat{L}$ and $\widehat{U}$ be the LU factors of $A \in \mathbb{R}^{n \quad n}$ computed with Algorithm 2.1, u the unit roundoff, r the size of each of the B blocks, $\varepsilon_{l_{ij}}$ the low-rank threshold used for block (i, j) of L and $\varepsilon_{u_{ij}}$ the low-rank threshold used for block (i, j) of U. Then, under assumptions 1, 2 and 3, holds:

$$r \in \mathbb{N}^*, \quad n \in \mathbb{N}^*, r \quad n,$$
$$|| \quad A|| \quad u \quad (n,r)||A|| + \quad (n,r) \quad ||\widehat{L}||||\widehat{U}|| + \max(\varepsilon_L)||\widehat{U}|| + \max(\varepsilon_U)||\widehat{L}|| + \max(\varepsilon_L)\max(\varepsilon_U)$$
$$+ \, r\max \quad \max(\varepsilon_U)||\widehat{L}||, \max(\varepsilon_L)||\widehat{U}|| \quad + O(u^2) \tag{2.3}$$

where is a constant depending on c_1, c_2 and c_3 (see assumptions) and $\quad n \; = \; \quad r$, $(n,r) = 1$, $\quad (r,r) = 0$. ε_U denotes the set of low-rank thresholds used for the compressions of the blocks of U, so that $\max(\varepsilon_U) = \max\limits_{j>i}(\varepsilon_{i,j})$, and similarly for L.

We first propose a proof of the theorem and will analyze it afterwards.

Proof. The proof is by induction on B, the number of blocks of size r.
We first show that (2.3) holds for B= 1. Then, we only compute the LU factorization of $A_{1,1} = A$ and thus $\max(\varepsilon_L) = \max(\varepsilon_L) = 0$. According to assumption 3, we have $\widehat{L}\widehat{U} = A + \quad A$ where:

$$|| \quad A|| \quad u||\widehat{L}||||\widehat{U}|| + O(u^2) = u \quad (r,r)||A|| + \quad (r,r) \quad ||\widehat{L}||||\widehat{U}|| + \max(\varepsilon_L)||\widehat{U}||$$
$$+ \max(\varepsilon_U)||\widehat{L}|| + \max(\varepsilon_L) \quad \max(\varepsilon_U)$$
$$+ \, r\max \quad \max(\varepsilon_U)||\widehat{L}||, \max(\varepsilon_L)||\widehat{U}|| \quad ,$$
$$\text{since} \quad (r,r) = 0, \; \max(\varepsilon_L) = \max(\varepsilon_L) = 0$$

This proves the theorem for B= 1, and for all $n \in \mathbb{N}$.

Let us assume (2.3) holds for a given B (and for all n) and then show that it holds for B+1 (and for all n). We thus consider a 2 2 matrix $A \in \mathbb{R}^{n \quad n}$ with $A_{11} \in \mathbb{R}^{r \quad r}$, $A_{21} \in \mathbb{R}^{n-r \quad r}$, $A_{12} \in \mathbb{R}^{r \quad n-r}$ and $A_{22} \in \mathbb{R}^{n-r \quad n-r}$. The objective is to find a bound on $|| \quad A_{11}||$, $|| \quad A_{21}||$, $|| \quad A_{12}||$ and $|| \quad A_{22}||$. The latter one will be given by the induction hypothesis. Then, we will conclude $|| \quad A|| \quad \max(|| \quad A_{11}||, || \quad A_{21}||, || \quad A_{12}||, || \quad A_{22}||)$. We assume that A_{12} and A_{21} are not subdivided for the sake of clarity, noting that the proof would work the same way. For the same reason, the "$O(u^2)$" will be omitted. The proof follows the same scheme as in Demmel et al. [38]. We will denote with a hat symbol^the computed blocks without low-rank approximation, and with a tilde~symbol the computed blocks after low-rank approximation. We will intensively use the bound $||AB|| \quad p||A||||B||$, where $A \in \mathbb{R}^{m \quad p}$ and $B \in \mathbb{R}^{p \quad n}$. It will be used without any comment. Note that it is the best such bound [38]. Bounding $|| \quad A_{11}||$ is straightforward as it entirely relies on assumption (3):

$$|| \quad A_{11}|| \quad c_3(r)u||\widehat{L}_{11}||||\widehat{U}_{11}|| \tag{2.4}$$

Then, we want to bound $|| \quad A_{21}||$ and $|| \quad A_{12}||$.

The factors $\widehat{L}_{21}$ and $\widehat{U}_{12}$ have to be computed first, which yields, according to assumption (2):

$$\begin{aligned} &\widehat{L}_{11}\widehat{U}_{12} = A_{12} + \quad \widehat{A}_{12} \\ &|| \quad \widehat{A}_{12}|| \quad c_2(r, n-r)u||\widehat{L}_{11}||||\widehat{U}_{12}|| \end{aligned} \tag{2.5}$$

and:

$$\begin{aligned} &\widehat{L}_{21}\widehat{U}_{11} = A_{21} + \quad \widehat{A}_{21} \\ &|| \quad \widehat{A}_{21}|| \quad c_2(r, n-r)u||\widehat{U}_{11}||||\widehat{L}_{21}|| \end{aligned} \tag{2.6}$$

The compression is then performed on the computed factors $\widehat{L}_{21}$ and $\widehat{U}_{12}$ and thus, based on assumption (4), we have:

$$\begin{aligned} &\widehat{U}_{12} = \tilde{U}_{12} + E_{12} \quad \text{where } ||E_{12}|| \quad \varepsilon_{12} \\ &\widehat{L}_{21} = \tilde{L}_{21} + E_{21} \quad \text{where } ||E_{12}|| \quad \varepsilon_{21} \end{aligned} \tag{2.7}$$

Thus, using (2.5), we obtain:

$$\begin{aligned} &\widehat{L}_{11}\tilde{U}_{12} = A_{12} + \quad \widehat{A}_{12} - \widehat{L}_{11}E_{12} = A_{12} + \quad A_{12} \\ &|| \quad A_{12}|| \quad c_2(r, n-r)u||\widehat{L}_{11}||||\widehat{U}_{12}|| + r\varepsilon_{12}||\widehat{L}_{11}|| \end{aligned} \tag{2.8}$$

and similarly with (2.6):

$$\begin{aligned} &\widehat{L}_{21}\tilde{U}_{11} = A_{21} + \quad \widehat{A}_{21} - E_{21}\widehat{U}_{11} = A_{21} + \quad A_{21} \\ &|| \quad A_{21}|| \quad c_2(r, n-r)u||\widehat{U}_{11}||||\widehat{L}_{21}|| + r\varepsilon_{21}||\widehat{U}_{11}|| \end{aligned} \tag{2.9}$$

We now want to update the Schur complement B using low-rank approximations. We know $B = A_{22} - L_{21}U_{12}$. We first compute $\widehat{C} = \tilde{L}_{21}\tilde{U}_{12} + \quad \widehat{C}$. Then, thanks to assumption (1), we obtain:

$$|| \quad \widehat{C}|| \quad c_1(n-r, r, n-r)u \quad ||\tilde{L}_{21}||||\tilde{U}_{12}||$$

Using (2.7), we have $\tilde{L}_{21}\tilde{U}_{12} = (\widehat{L}_{21}+E_{21})(\widehat{U}_{12}+E_{12}) = \widehat{L}_{21}\widehat{U}_{12}+\widehat{L}_{21}E_{12}+E_{21}\widehat{U}_{12}+E_{21}E_{12}$, which leads to:

$$\widehat{C} = \widehat{L}_{21}\widehat{U}_{12} + \widehat{L}_{21}E_{12} + E_{21}\widehat{U}_{12} + E_{21}E_{12} + \quad \widehat{C}$$

and:

$$\widehat{C} \quad c_1(n-r, r, n-r)u \quad ||\widehat{L}_{21}||||\widehat{U}_{12}|| + ||\widehat{L}_{21}||\varepsilon_{12} + ||\widehat{U}_{12}||\varepsilon_{21} + \varepsilon_{12}\varepsilon_{21}$$

We also have

$$\widehat{B} = A_{22} - \widehat{C} + F, \tag{2.10}$$

where $||F|| \quad u \quad ||A_{22}|| + ||\widehat{C}|| \quad$. It follows that $\widehat{B} = A_{22} - \tilde{L}_{21}\tilde{U}_{12} + \quad B$ where

$$\begin{aligned} || \quad B|| \quad & u \; ||A_{22}|| + ||\tilde{L}_{21}\tilde{U}_{12}|| \quad + || \quad \widehat{C}|| \\ & u \; ||A_{22}|| + r||\widehat{L}_{21}||||\widehat{U}_{12}|| + r||\widehat{L}_{21}||\varepsilon_{12} + r||\widehat{U}_{12}||\varepsilon_{21} + r\varepsilon_{12}\varepsilon_{21} \\ & +c_1(n-r, r, n-r) \; ||\widehat{L}_{21}||||\widehat{U}_{12}|| + ||\widehat{L}_{21}||\varepsilon_{12} + ||\widehat{U}_{12}||\varepsilon_{21} + \varepsilon_{12}\varepsilon_{21} \end{aligned}$$

Now we compute the LU factorization of $\widehat{B}$, leading to $\widehat{L}_{22}\widehat{U}_{22} = \widehat{B} + \quad \widehat{B}$. Thanks to the induction hypothesis (which holds for all n and thus for $n-r$ too), we have:

$$\begin{aligned} || \quad \widehat{B}|| \quad u \quad (n-r,r)||\widehat{B}|| + \quad (n-r,r) \quad ||\widehat{L}_{22}||||\widehat{U}_{22}|| + \max(\varepsilon_{L_{22}})||\widehat{U}_{22}|| \\ + \max(\varepsilon_{U_{22}})||\widehat{L}_{22}|| + \max(\varepsilon_{L_{22}}) \quad \max(\varepsilon_{U_{22}}) \\ + r \max \quad \max(\varepsilon_{U_{22}})||\widehat{L}||, \max(\varepsilon_{L_{22}})||\widehat{U}|| \end{aligned}$$

Then, since $A_{22} = \widehat{B} + \widehat{L}_{21}\widehat{U}_{12} - \quad B = \widehat{L}_{22}\widehat{U}_{22} + \widehat{L}_{21}\widehat{U}_{12} - \quad \widehat{B} - \quad B$, we obtain $\widehat{L}_{22}\widehat{U}_{22} + \widehat{L}_{21}\widehat{U}_{12} = A_{22} + \quad A_{22}$, where

$$\begin{aligned} || \quad A_{22}|| \quad & || \quad \widehat{B}|| + || \quad B|| \\ & \begin{array}{ll} (n-r,r)u||\widehat{B}|| & \\ + \quad (n-r,r)u \quad ||\widehat{L}_{22}||||\widehat{U}_{22}|| + \max(\varepsilon_{L_{22}})||\widehat{L}_{22}|| & \\ + \max(\varepsilon_{U_{22}})||\widehat{U}_{22}|| + \max(\varepsilon_{L_{22}})\max(\varepsilon_{U_{22}}) & || \quad \widehat{B}|| \\ + r\max \quad \max(\varepsilon_{U_{22}})||\widehat{L}||, \max(\varepsilon_{L_{22}})||\widehat{U}|| & \end{array} \\ & \begin{array}{ll} + u \quad ||A_{22}|| + r||\widehat{L}_{21}||||\widehat{U}_{12}|| + r||\widehat{L}_{21}||\varepsilon_{12} + r||\widehat{U}_{12}||\varepsilon_{21} + r\varepsilon_{12}\varepsilon_{21} & \\ & || \quad B|| \\ + c_1(n-r,r,n-r) \quad ||\widehat{L}_{21}||||\widehat{U}_{12}|| + ||\widehat{L}_{21}||\varepsilon_{12} + ||\widehat{U}_{12}||\varepsilon_{21} + \varepsilon_{12}\varepsilon_{21} & \end{array} \end{aligned}$$

$$\begin{aligned} || \quad A_{22}|| \quad & || \quad \widehat{B}|| + || \quad B|| \\ & (n-r,r)u||\widehat{B}|| \\ & + \quad (n-r,r)u \quad ||\widehat{L}_{22}||||\widehat{U}_{22}|| + \max(\varepsilon_{L_{22}})||\widehat{L}_{22}|| \\ & + \max(\varepsilon_{U_{22}})||\widehat{U}_{22}|| + \max(\varepsilon_{L_{22}})\max(\varepsilon_{U_{22}}) \\ & + r\max \quad \max(\varepsilon_{U_{22}})||\widehat{L}||, \max(\varepsilon_{L_{22}})||\widehat{U}|| \\ & + u \quad ||A_{22}|| + r||\widehat{L}_{21}||||\widehat{U}_{12}|| + r||\widehat{L}_{21}||\varepsilon_{12} + r||\widehat{U}_{12}||\varepsilon_{21} + r\varepsilon_{12}\varepsilon_{21} \\ & + c_1(n-r,r,n-r) \quad ||\widehat{L}_{21}||||\widehat{U}_{12}|| + ||\widehat{L}_{21}||\varepsilon_{12} + ||\widehat{U}_{12}||\varepsilon_{21} + \varepsilon_{12}\varepsilon_{21} \end{aligned}$$

Using (2.10) to expand $||\widehat{B}||$, we obtain (ignoring terms in u^2):

$$\begin{aligned} || \quad A_{22}|| \quad & u\,[1 + \quad (n-r,r)]\,||A_{22}|| \\ & + u \quad (n-r,r) \quad ||\widehat{L}_{22}||||\widehat{U}_{22}|| + \max(\varepsilon_{L_{22}})||\widehat{L}_{22}|| + \max(\varepsilon_{U_{22}})||\widehat{U}_{22}|| + \max(\varepsilon_{L_{22}})\max(\varepsilon_{U_{22}}) \\ & + u \quad r + c_1(n-r,r,n-r) + \quad (n-r,r) \quad ||\widehat{L}_{21}||||\widehat{U}_{12}|| + ||\widehat{L}_{21}||\varepsilon_{12} + ||\widehat{U}_{12}||\varepsilon_{21} + \varepsilon_{12}\varepsilon_{21} \\ & + r\max \quad \max(\varepsilon_{U_{22}})||\widehat{L}||, \max(\varepsilon_{L_{22}})||\widehat{U}|| \end{aligned} \tag{2.11}$$

Finally, knowing that $|| \quad A|| \quad \max(|| \quad A_{11}||, || \quad A_{21}||, || \quad A_{12}||, || \quad A_{22}||)$, collecting (2.4), (2.8), (2.9) and (2.11), and using the following bounds:

$$\begin{aligned} ||A_{22}|| \quad & ||A|| \\ i,j, ||\widehat{L}_{ij}|| \quad & ||\widehat{L}|| \\ i,j, ||\widehat{U}_{ij}|| \quad & ||\widehat{U}|| \end{aligned}$$

we obtain the desired result with:

$$(n, r) = 1 + \quad (n - r, r)$$
$$(n, r) = \max \quad (n - r, r) + r + c_1(n - r, r, n - r) + \quad (n - r, r), c_2(n - r, r), c_3(r)$$

□

This bound allows us to better define the low-rank threshold ε to use in order to guarantee a good accuracy. Equation (2.8) shows that to obtain a very fine truncation, the low-rank threshold $\varepsilon_{ij}(i < j)$ of the (i, j) block of U should be chosen with respect to $||\widehat{L_{ii}}||$. The same statement holds with $\varepsilon_{ij}(i > j)$ and $||\widehat{U_{jj}}||$ thanks to Equation (2.9). However, computing norms at each step of the factorization requires many operations and should be in practice avoided. We can relax these statement by looking directly at the bound, which indicates that ε_L could be chosen with respect to $||\widehat{U}||$ and ε_U with respect to $||\widehat{L}||$, which cannot be done because the low-rank thresholds need to be known before all the factors have been computed. In order to avoid all these norms computations, an absolute low-rank threshold is chosen. Then, it does not need to be different for L and U so we will assume Theorem 2.3 is simplified as follows:

$$\begin{aligned} || \quad A|| \quad u \quad (n, r)||A|| + \quad (n, r) \quad ||\widehat{L}||||\widehat{U}|| + \varepsilon||\widehat{U}|| + \varepsilon||\widehat{L}|| + \varepsilon^2 \\ + r\varepsilon \max \quad ||\widehat{L}||, ||\widehat{U}|| \quad + O(u^2), \end{aligned} \tag{2.12}$$

Note that in most cases, $\varepsilon < \quad \overline{u}$ so that $u\varepsilon^2 = O(u^2)$. Interestingly, the second term is not multiplied by u so it will be dominant in most cases. Moreover, all the terms in $u\varepsilon$ will also often be dominated by the terms in u and in ε, meaning that for values of ε not too far from u, the error added to the full-rank standard LU error is $r\varepsilon \max \quad ||\widehat{L}||, ||\widehat{U}|| \quad$. Given a stable full-rank factorization, $\max \quad ||\widehat{L}||, ||\widehat{U}|| \quad$ is contained (relatively to $||A||$) so that the Block Low-Rank factorization should also be stable.

2.4.2 Experiments and analysis

We present experimental results for the bound of Theorem 2.1 on different dense matrices. The constants are set up to 1 as this configuration has led to satisfying results. For more details about these constants, one can refer to Demmel and Higham [37]. As said before, the low-rank threshold is chosen absolute. We want to show that err $= \frac{A}{A}$ is sharply bounded by the bound of Theorem 2.1 and that it is close to the low-rank dropping parameter. The dense matrices studied in this section are obtained from sparse matrices: we compute the top level separator (with `METIS`) of the graph of these sparse matrices and eliminate all other variables so that a dense Schur complement is obtained. These Schur complements are our dense matrices. The sparse matrices we use are `Poisson32`, `Helmholtz32` and some other matrices from the University of Florida Sparse Matrix Collection (`bbmat`, `onetone1`, `wang3` and `wang4`). These latter four matrices were used in Li and Demmel [73] and are numerically difficult. The sizes of the corresponding Schur complements are presented in Table 2.2.

This forms a set of unsymmetric, numerically difficult, matrices from different types of applications on which the analysis is done with different block sizes and low-rank thresholds. For each matrix, we give in Table 2.3 the block size used, the error err $= \frac{A}{A}$ and the bound.

matrix name (UFL)	`Poisson32`	`Helmholtz32`	`bbmat`	`onetone1`	`wang3`	`wang4`
root size	1024	1024	1064	1032	961	957

Table 2.2: Size of the dense Schur complement associated with the top level separator of a nested dissection computed with `METIS`.

The bounds are sharp for all the matrices for any low-rank threshold. In most of the case, the bound is the same order of the observed error. However, the bound is sometimes too optimistic, but even in these cases it remains very close: for instance for `wang4`, $\varepsilon = 10^{-6}$ and block size 107, the error is 1.38E-06 and the bound is 9.98E-07.

Note that other types of low-rank thresholds (such as relative to the diagonal L or U) have been experimented in Buttari et al. [26], without any substantial improvement.

problem	ε	block size	err	bound
`Helmholtz32`	10^{-14}	121	2.075068E-14	1.182143E-14
		256	1.202998E-14	1.208495E-14
	10^{-10}	121	1.985522E-10	1.132319E-10
		256	1.142500E-10	1.155068E-10
	10^{-6}	121	2.059789E-06	1.132314E-06
		256	9.122070E-07	1.155063E-06
`Poisson32`	10^{-14}	121	3.477928E-14	9.142465E-15
		256	1.962034E-14	9.144515E-15
	10^{-10}	121	3.028155E-10	8.937881E-11
		256	2.204627E-10	8.939897E-11
	10^{-6}	121	1.588831E-06	8.937861E-07
		256	1.408218E-06	8.939876E-07
`bbmat`	10^{-14}	119	9.655596E-13	5.635941E-12
		266	1.237731E-12	1.662243E-12
	10^{-10}	119	2.293800E-11	1.055635E-09
		266	1.109681E-11	6.407317E-10
	10^{-6}	119	6.781275E-07	1.050105E-05
		266	5.127564E-08	6.391335E-06
`onetone1`	10^{-14}	115	6.913305E-11	1.800261E-09
		258	1.318127E-11	1.124705E-10
	10^{-10}	115	9.018629E-11	4.152371E-08
		258	1.832740E-11	9.562843E-09
	10^{-6}	115	1.732020E-06	3.972760E-04
		258	4.626845E-07	9.451328E-05
`wang3`	10^{-14}	107	1.374437E-14	2.071843E-14
		241	2.648347E-14	2.071843E-14
	10^{-10}	107	1.362081E-10	2.014647E-10
		241	1.117009E-10	2.014647E-10
	10^{-6}	107	4.192184E-06	2.014641E-06
		241	2.891060E-06	2.014641E-06
`wang4`	10^{-14}	107	7.289383E-15	1.063247E-14
		240	1.198408E-14	1.063247E-14
	10^{-10}	107	1.659250E-10	9.982301E-11
		240	8.704217E-11	9.982301E-11
	10^{-6}	107	1.388475E-06	9.982236E-07
		240	6.900322E-07	9.982236E-07

Table 2.3: Bounds and errors obtained for different problems, block sizes and low-rank thresholds.

Chapter 3

Block Low-Rank multifrontal method

In this chapter, we present how the standard multifrontal method can be adapted to benefit from the BLR format. We show how the BLR format can be used in the three main phases of a multifrontal process and propose different factorization algorithms (which offer different performance and features). We explain how standard symmetric and unsymmetric pivoting can be easily adapted and present the distributed memory parallelization scheme used with BLR.

3.1 General idea

As presented in Section 1.3, most of the linear algebra computations of the multifrontal process are performed within dense matrices called fronts, which can be viewed as Schur complements. The idea of a BLR multifrontal solver is to represent these dense fronts as BLR matrices and to perform the partial factorization of the front with BLR algorithms. The structure of a BLR front is given in Figure 3.1.

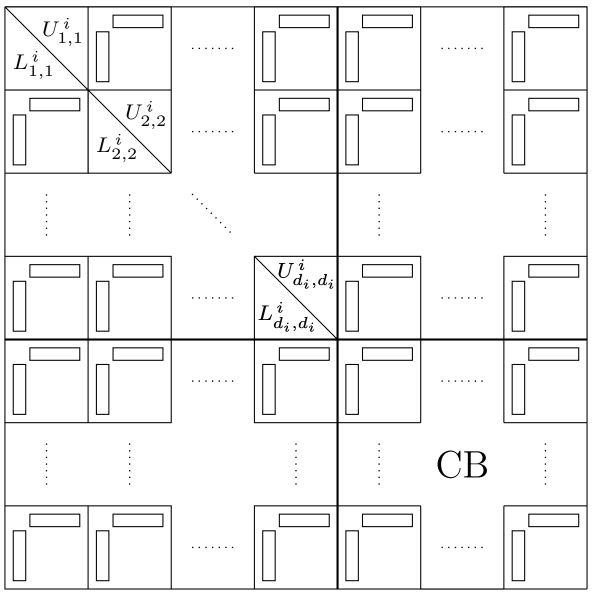


Figure 3.1: Structure of a BLR front in its compressed and factorized form.

The diagonal blocks of the fully-summed part of the front are kept full-rank, while all the others are approximated when worth it. The size of the blocks, although usually quite regular, may vary slightly within a front (see Section 3.3). Note that Figure 3.1 represents the front at the end of the partial factorization.

3.2 Assembly

Because of their nature, assembly operations are relatively inefficient [16] and their cost can significantly affect the speed of the overall multifrontal factorization. In the experimental code we developed, we decided not to perform the assembly operations in BLR format in order to avoid an excessive amount of indirect addressing which may seriously impact performance. An analogous choice was also made in related work on the usage of HSS matrices within multifrontal solvers, as explained in Section 1.5.5.3.

Frontal matrices are, thus, assembled in full-rank form and compressed progressively, by panels, as the partial front factorization proceeds, as shown in the next section. This, however, does not mean that contribution blocks cannot be compressed; despite the fact that compressing contribution blocks does not help reducing the overall number of floating point operations (unless the updates are performed directly on compressed blocks, which would have to be recompressed afterwards [18]), storing them in low-rank form has a twofold advantage:

1. It allows the peak of active memory (see Section 1.3.3) to be reduced. In a sequential, multifrontal method, at any moment, the active memory is defined as the sum of the size of the CB stack (described in Section 1.3) plus the size of the front currently being factorized. Stacking the contribution blocks in low-rank form reduces the active memory. Although this makes the relative weight of the current front in the active memory higher, it has to be noted that if the partial front factorization is done in a left-looking fashion, frontal matrices can be assembled panelwise which means that in the active memory only one panel at a time is stored in full-rank. We have not implemented this feature yet, and, as discussed in the next section, the front factorization is still performed in a right-looking fashion.

2. In a parallel environment, it reduces the communications volume (the total amount of data sent between processors during the execution). In a parallel solver, frontal matrices are generally mapped onto different MPI tasks which means that assembly operations involve transferring contribution blocks from one MPI task to another. By storing contribution blocks in low-rank form, the total volume of communications can be reduced.

Because we want to assemble fronts in full-rank form, if a contribution block is in low-rank form, it has to be decompressed before the associated assembly takes place. This only increases the overall number of floating point operations by a small amount because the cost of converting to and from the BLR format is contained, as shown in Figures 2.3 and 2.5. Because it is not necessary to systematically compress the contribution blocks, in the experimental results of Chapter 4 and Chapter 5 the gains provided by this operation and the associated cost are always reported separately (when reported at all).

3.3 Front variables clustering

The objective of clustering variables is to induce an efficient BLR representation of fronts. To achieve this goal, two clusterings are needed: one for the fully-summed variables and one for the non fully-summed variables. For the sake of clarity, we always consider the fully-summed variables of a frontal matrix to be associated with a separator. This analogy has been justified in Section 1.3.1, both when the assembly tree has been computed through a nested dissection and through any other method. In this latter case, a *separator tree* will refer to a tree matching the shape of the assembly tree where each node consists of the fully-summed variables of the corresponding node in the assembly tree. Similarly, the non fully-summed variables of a frontal matrix are associated with a *border* made of pieces of other separators (again, see Section 1.3.1). To ease the presentation of our algorithms and without loss of generality, we will assume that a nested dissection has been used to build the assembly tree. The root frontal matrix (the one at the top of the assembly tree) is then associated with the top level separator of the graph of the original sparse matrix A. The fully-summed variables of any other frontal matrix are associated with a separator located lower in the separator tree.

The BLR representation of a matrix is based upon the assumption that the frontal matrices are assembled in such a way that variables belonging to the same cluster appear contiguously along the diagonal. These clusters are defined by a clustering, whose efficiency relies on two tasks:

1. defining a suitable admissibility condition which basically describes what an optimal clustering is;

2. partitioning variables with a strategy which gives as many admissible blocks as possible, for each front.

Those two tasks are critical as the overall efficiency of the BLR multifrontal factorization relies mainly on them.

3.3.1 Algebraic admissibility condition for BLR

To obtain the BLR representation of a front, variables have to be grouped into clusters which satisfy as much as possible the admissibility condition presented in Section 1.5.4, ensuring that the interaction block of two given admissible clusters is low-rank and can thus be efficiently represented. However, in the context of a general purpose multifrontal solver, this condition is not satisfying as it requires geometric information of the underlying problem. For this reason, a black-box admissibility condition, which only needs the graph $\mathcal{G}$ of the original matrix (or subraphs of it, for instance $\mathcal{G}_{\mathcal{S}}$, the graph induced by the nodes of the separator $\mathcal{S}$ corresponding to front F), can be used (as explained in Section 1.5.4 and in Grasedyck et al. [59]). This condition, however, can be further simplified and complemented with other practical considerations in order to define a clustering strategy suited for the BLR format:

1. For efficiency reasons, the size of the clusters should be chosen between a minimum value that ensures a good efficiency of the BLAS operations performed later with X and Y matrices and a maximum value that allows the compressions (using SVDs or RRQRs) to be performed with sequential routines and that enables an easy and flexible distribution of blocks in a parallel environment.

2. The distance between any two clusters σ and τ has to be greater than zero ($dist_{\mathcal{G}}(\sigma, \tau) > 0$) which amounts to saying that all the clusters are disjoint. Note that this point alone is equivalent to the *w*eak admissibility condition proposed by Börm [20, Remark 3.17].

3. For a given size of clusters (and consequently a given number of clusters in $\mathcal{G}$) the diameter of each cluster should be as small as possible in order to group within a cluster only those variables that are likely to strongly interact with each other. For example, in the case where $\mathcal{G}$ is a flat surface, it is better to define clusters by partitioning $\mathcal{G}$ in a checkerboard fashion rather than cutting it into slices.

Once the size of the clusters is chosen, the objectives 2 and 3 can be easily achieved by feeding the graph $\mathcal{G}$ to any modern graph partitioning tool such as `METIS` or `SCOTCH`, as discussed in the next two sections which show how to compute the clustering of the fully assembled and non-fully assembled variables in a frontal matrix. Note that, in practice, partitioning tools take the number of partitions as input and not their size; however the size of the resulting clusters will differ only slightly because partitioning methods commonly try to balance the weight of the partitions. Because of the simple nature of the BLR format which does not require a relative order between clusters, simpler and cheaper partitioning or clustering techniques may be employed instead of complex tools such as `METIS` or `SCOTCH`. Experimental results in Chapter 4 show that the cost induced by the separators clustering using `METIS` is however acceptable.

To illustrate the BLR admissibility condition (and more particularly its point 3) and give some insight about what the clustering should look like, we consider the dense matrix `geo_root_Poisson81` of size 6561, i.e. the root F associated with the top level separator of the nested dissection of the 81 81 81 cubic domain. This geometric separator is thus a square surface. To compute the BLR representation of F (we only consider the compression of `geo_root_Poisson81` without factorizing it as we want to focus on how the admissibility condition influences the compression), we need to partition the separator to obtain the desired blocking of the matrix. Figure 3.2 illustrates two different ways of performing such a partitioning.

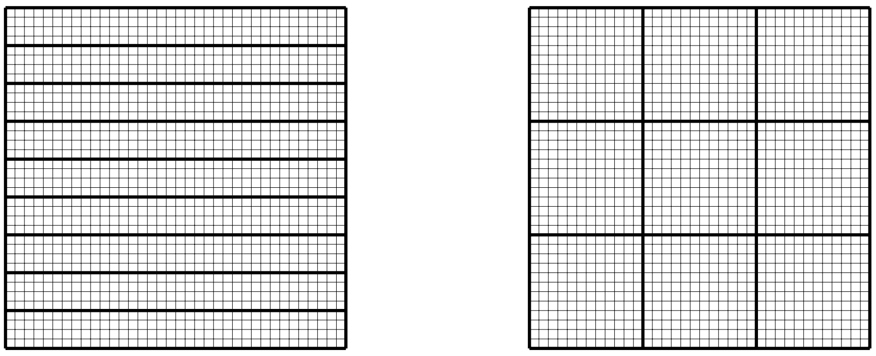

(a) slices clustering (b) checkerboard clustering

Figure 3.2: Two different clustering of the same surface.

The number of entries required to store this matrix in a BLR format based on the latter two clustering heuristics is given in Table 3.1. The low-rank threshold ε is set up to 10^{-14}.

Slices clustering does not respect the admissibility condition (since it leads to small distances between clusters, large cluster diameters) which translates into a bad memory compression rate of 83% while checkerboard clustering does respect the admissibility condition (large distances between clusters, small cluster diameters) and provides a compression rate of 57%. Although separators are in practice less regular than in this illustrative

Format	slices	checkerboard
FR	$43,046,721$ (100%)	$43,046,721$ (100%)
BLR	$35,728,778$ (83%)	$24,536,630$ (57%)

Table 3.1: Number of entries to store the dense matrix A in full-rank (FR), BLR format with slices clustering and BLR format with checkerboard clustering. Percentages are with respect to the full-rank number of entries.

example, this gives a good intuition on the kind of heuristic we are targeting for the partitioning.

3.3.2 Fully-summed variables/separator clustering

Given our admissibility condition, a method to actually perform the clustering on variables of a front has to be found. Because a front is made of two different (and more importantly, independent) sets of variables (the fully-summed variables and the non fully-summed variables, see Section 1.3), we need to compute two clusterings (note the clustering of each front is an independent problem, even though the same method is applied each time). We explain how the clustering can be computed first for the fully-summed variables (corresponding to variables of a separator) and then for the non fully-summed ones (corresponding to variables of a border). Although ideas such as partitioning the whole original graph or partitioning the subgraph induced by the separator's variables are very natural and simple, we will first show why they fail (Section 3.3.2.1) and then propose another method called the halo method (Section 3.3.2.2). In this section, we consider $\mathcal{G}_\mathcal{S}$, the graph induced by the nodes of the separator $\mathcal{S}$ corresponding to front F.

3.3.2.1 Why basic approaches fail

Let us consider the graph shown in Figure 3.3(a). The squared grid represents the graph of the original sparse matrix A. The dots represent the nodes associated with the fully-summed variables of a given front. As explained in Section 1.3.1, these nodes are considered to always form a separator (in this example, it is the top level separator). Assume we want to partition it to obtain the clustering of the front corresponding to this separator and thus its BLR representation. Figure 3.3(b) gives one suitable partitioning of the separator which is the optimal one on this academic example, for a given number of clusters (here, 3).

A graph partitioning tool such as `METIS` or `SCOTCH` with a target partition size will compute BLR admissible blocks, given that the graph is connected enough, which cannot be guaranteed for $\mathcal{G}_\mathcal{S}$. Figure 3.4 shows the result obtained with the routine `PartGraphKway` of `METIS`. This routine partitions a graph into k equal-size parts using the multilevel k-way partitioning algorithm [69].

As some of the nodes of the subgraph are singletons (due to no connection with any other vertex of the separator), no neighborhood information is available for the partitioner so that disconnected variables (top right location in the subgraph) are basically assigned a partition number which has no geometric meaning. To avoid this problem, let us consider partitioning the whole graph of the domain and compute the partitioning of the separator from this one, as shown in Figure 3.5. Unlike the previous case, the partitioner has too much information, so that the induced partitioning of the separator has no guarantee to be consistent with respect to what we aim at.

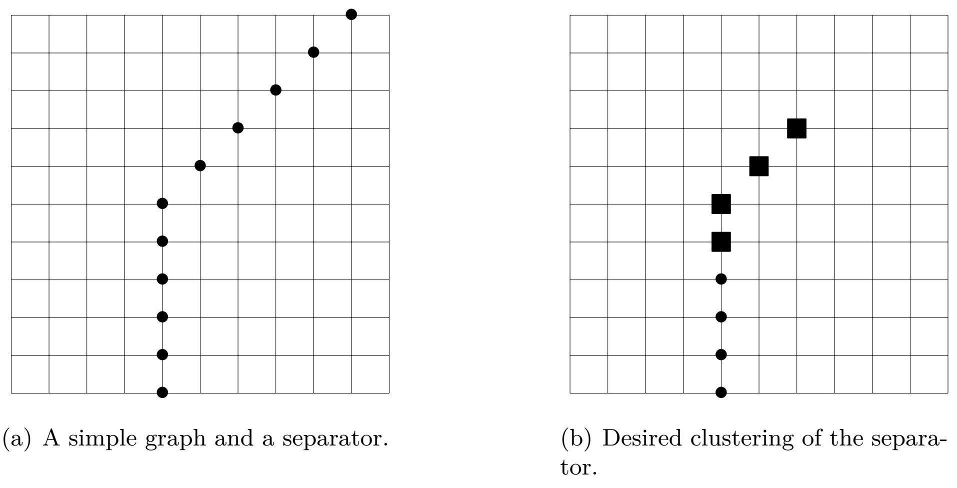

(a) A simple graph and a separator. (b) Desired clustering of the separator.

Figure 3.3: A simple graph, a separator and the desired partitioning of the separator.

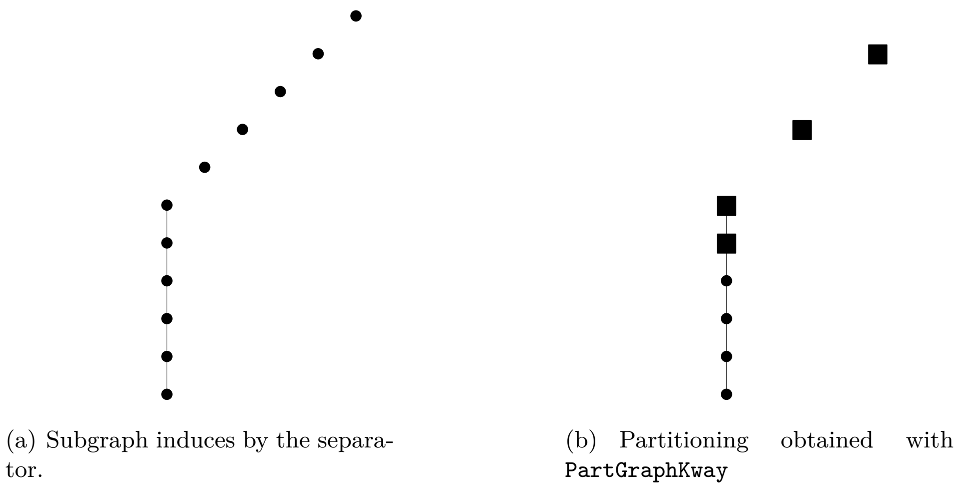

(a) Subgraph induces by the separator. (b) Partitioning obtained with `PartGraphKway`

Figure 3.4: Partitioning the variables of the separator using its subgraph.

Another method has thus to be designed to overcome these difficulties. Called the halo method, it can be viewed as a compromise between the latter two basic methods.

3.3.2.2 Halo method

We consider $\mathcal{G}_\mathcal{S}$, the graph induced by the nodes of the separator $\mathcal{S}$ corresponding to front F. To easily present the halo method, we consider a new example corresponding to the worst case of a disconnected separator (Figure 3.6(a)). We have seen in Section 3.3.2.1 that a partitioning tool cannot always compute an admissible clustering based on such a disconnected separator.

Although in reality such an extreme situation would rarely occur, it is quite commonly the case where a separator is formed by multiple connected components that are close to each other in the global graph $\mathcal{G}$. This may lead to a sub-optimal clustering because variables that strongly interact due to their adjacency will end up in different clusters.

This problem may be overcome by reconnecting $\mathcal{G}_\mathcal{S}$ in a way that takes into account the geometry and shape of the separator: variables close to each other in the original graph

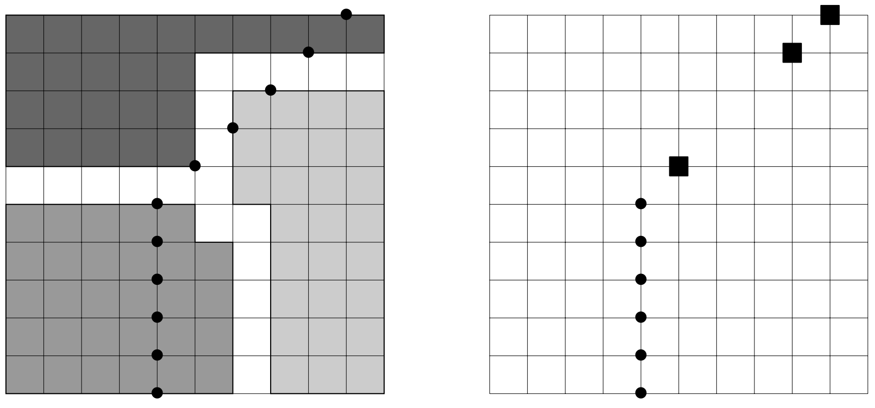

(a) *pmetis* partitioning (b) Induced separator partitioning

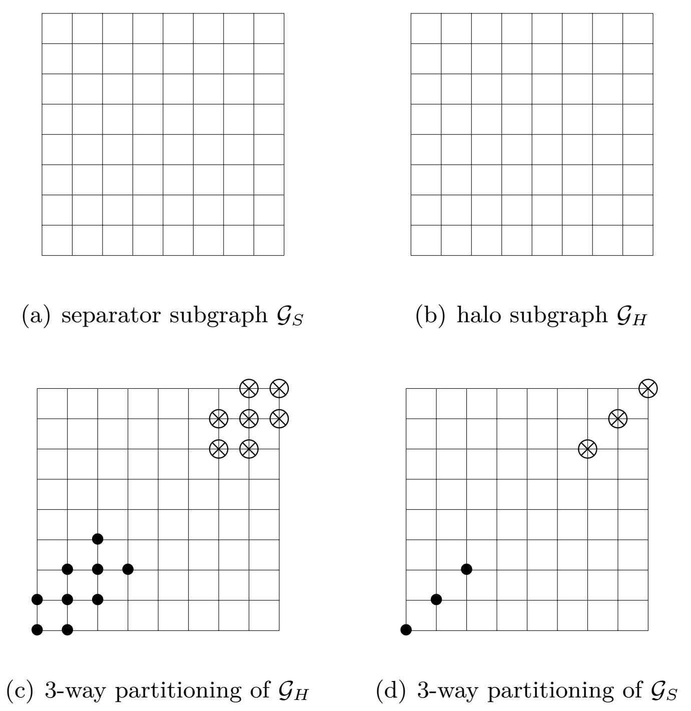
Figure 3.5: Separator partitioning using the whole original graph.

$\mathcal{G}$ have to be close to each other in the reconnected $\mathcal{G}_{\mathcal{S}}$ in order to satisfy the admissibility condition. We describe an approach that achieves this objective by extending the subgraph induced by each separator with a relatively small number of level sets (a vertex v belongs to the level set $\mathcal{L}_i$ of $\mathcal{S}$ if and only if there exist $s \in \mathcal{S}$ such that the distance between v and s is exactly i). The union on these level sets for $i = 1$ to p the depth of the halo (together with the original nodes of the separator) is called a halo. The graph of the halo will be referred to as $\mathcal{G}_H$. Note that $\mathcal{G}_{\mathcal{S}}$ is included in $\mathcal{G}_H$. The graph partitioning tool is then run on $\mathcal{G}_H$ and the resulting partitioning projected back on the original subgraph $\mathcal{G}_{\mathcal{S}}$. Figure 3.6 shows how this is done on the example above using just one level set. For a limited number of level sets, the extended graph preserves the shape of the separator, keeps the cost of computing the clustering limited and allows us to compute clusterings that better comply with the strategy presented in the previous section. In practice, we observed that two layers are enough to reconnect the separator and to obtain good performance (in 2D, one layer is sufficient). However, on very complex domains, the optimal value may have to be found experimentally.

To illustrate the halo method, we experimented it on top level separators computed with `SCOTCH` on a 2D 5-points Laplacian and a 3D 7-points Laplacian. These separators are highly disconnected due to the lack of diagonal edges in the graph. The results in Figures 3.7 and 3.8(b) show that the partitioning, although not exactly shaped as a checkerboard, does achieve our objective and respect the admissibility condition (clusters are far and their diameter is relatively small). Two vertices which are close in the original graph but disconnected in the separator graph are indeed kept in the same cluster (unless a new cluster starts). Figure 3.8(a) illustrates the natural clustering obtained without halo, i.e. obtained by partitioning the subgraph induced by the separator nodes itself.

3.3.3 Non fully-summed variables/border clustering

As explained before, two clusterings have to be found in order to obtain the BLR representation of a front. The halo method has been introduced and allows the clustering of the variables of the separator (i.e., the fully-summed variables) to be robust and efficient. Obviously, the halo method applies for any set of variables, regardless of our context. Thus, the first natural idea is to apply the halo method to the set of non fully-summed


(a) separator subgraph $\mathcal{G}_S$ (b) halo subgraph $\mathcal{G}_H$

(c) 3-way partitioning of $\mathcal{G}_H$ (d) 3-way partitioning of $\mathcal{G}_S$

Figure 3.6: Halo-based partitioning of $\mathcal{G}_S$ with depth 1. $\mathcal{G}_H$ is partitioned using METIS.

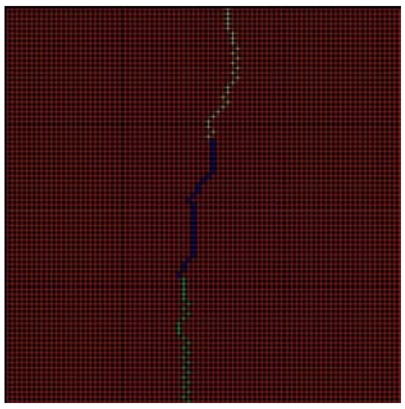
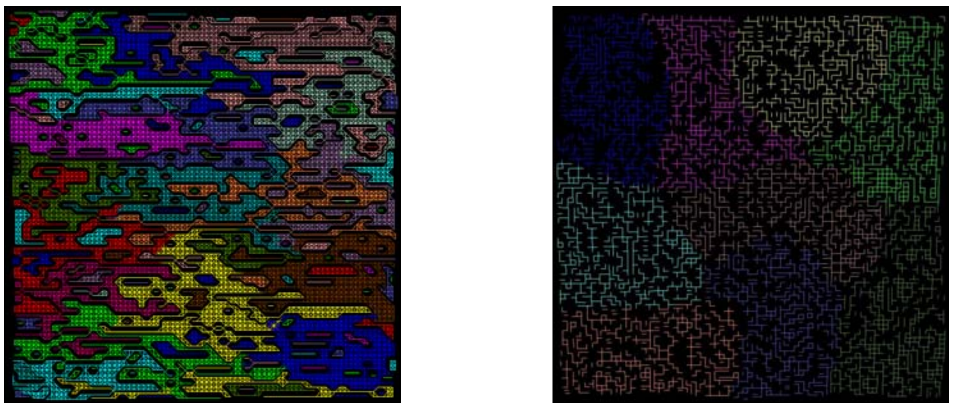


Figure 3.7: The halo method applied on the top level separator computed with SCOTCH on a 2D 5-points stencil Laplacian problems.

variables which yields a good quality clustering, as shows Section 3.3.3.1. However, we will show that this strategy is far from being optimal in terms of the amount of work to do to achieve this clustering. Consequently, we will propose in Section 3.3.3.2 a simplified strategy which benefits from the underlying tree structure in the multifrontal process and gives almost equally efficient BLR representations.

3.3.3.1 Explicit border clustering

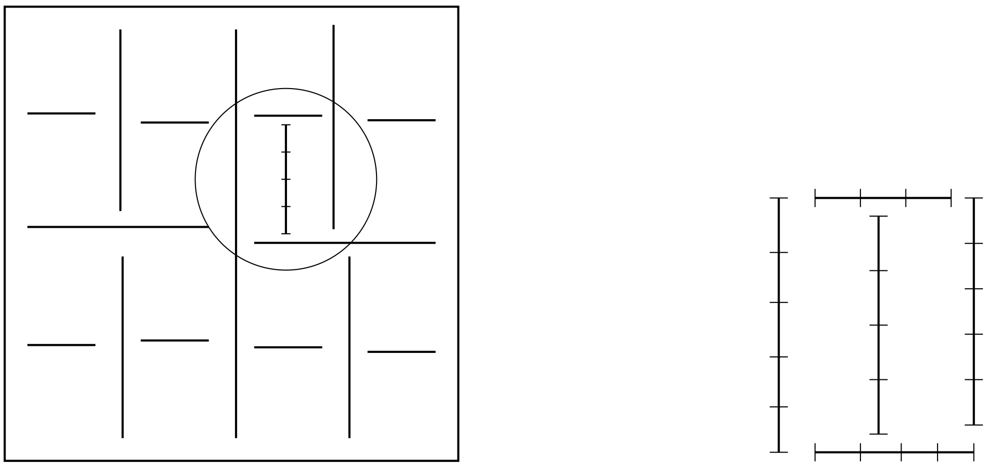
The previous section shows how the halo method can be employed to compute the clustering of a separator. This method can obviously be used for clustering any set of variables



(a) Without halo. (b) With halo.

Figure 3.8: The clustering of the top level separator computed with SCOTCH on a 3D 7-points stencil Laplacian problems, with and without the halo method.

and, therefore, also those in the non fully-assembled part of the front that form a border around the corresponding separator (see Section 1.3). This ensures that the clustering of the border will be admissible. Figure 3.9(b) illustrates the way the clusters are computed using the halo method for both the separator and the border. Let us consider a particular border associated with the separator circled in Figure 3.9(a). This separator has been clustered with the halo method. Then, the same method is applied on the border variables which gives a similar clustering, illustrated in Figure 3.9(b).



(a) A nested dissection of the domain where the variables of the current separator have been partitioned with the halo method. The clustering of the current border remains to be computed.

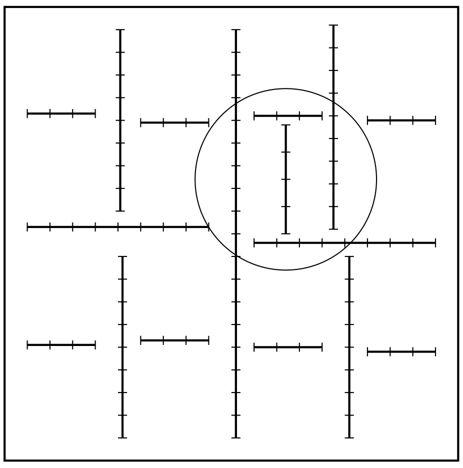
(b) Zoom on current separator and border. The clustering of the border of the current separator is obtained with the halo method, so that optimal regular clusters are found. Each segment is a cluster.

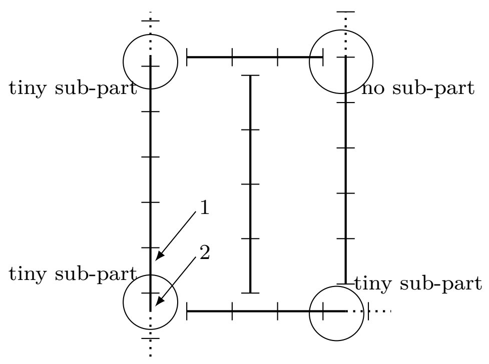
Figure 3.9: Explicit clustering where the halo method is applied both on the separator and on the border variables.

Although this approach leads to an efficient clustering of the non fully-summed variables, it may suffer performance issues because it actually does many times the same task. As illustrated in Figure 3.9(a), because of the structure of the assembly tree, one particular variable belongs to one separator which, in turn, may be (partially) included in multiple borders. Inversely, as Figure 3.9(a) and 3.11 illustrate, a border can be viewed as a union of parts of separators lying on the path that connects the related node to the root of the elimination tree.

3.3.3.2 Inherited border clustering

As a consequence of the previous comment, each clustering of a separator can be used partially for the clustering of several borders. Thus, clustering the variables of all the separators is sufficient to obtain, by induction, a clustering of the borders.


(a) A nested dissection of the domain where the variables of each separator have been clustered with the halo method, i.e. each segment is a group of variables.



(b) Zoom on current separator and border. The separator s clustering has been computed with the halo method. The border s clustering is inherited from several separator clusterings. At some corners, a cluster maybe not be entirely involved in current front so that tiny clusters may appear.

Figure 3.10: Inherited clustering where the separators are partitioned with the halo method and the borders inherit their clustering.

A top-down traversal of the separator tree is performed and the variables of each separator are clustered with the halo method proposed in Section 3.3.2.2. Then, the clustering of a given border is inherited from the clustering of all the separators it is made of, as Figure 3.10(b) shows. Figure 3.11 illustrates this idea from the point of view of a front.

In a given front (viewed here as a child front), variables which are non fully-summed will be eliminated in other fronts located higher in the tree. Among these variables, some will be eliminated in the *same* front (viewed here as a parent front), meaning they belong to the same separator. In the child front, one can thus obtain a clustering of these variables based on the clustering of the separator they belong to. This clustering is in fact included in the clustering of the separator (as *all* the fully-summed variables of the parent front may not be present in the non fully-summed variables of the child front). This being valid for all the non fully-summed variables of the child front, one can obtain a clustering of this front from the clustering of separators located upper in the tree.

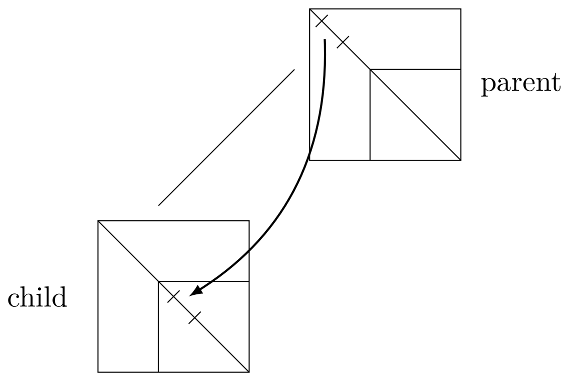
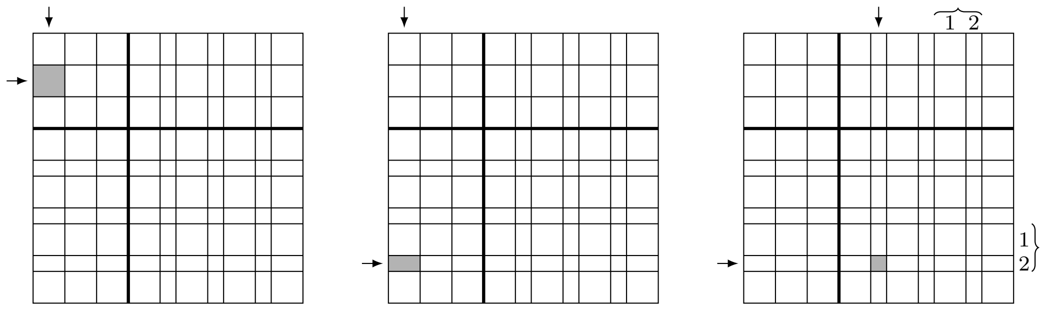


Figure 3.11: Inheriting the clustering from the front point of view: The clustering of the Schur complement in inherited from the clustering of the fully-summed variables of the parents.

Depending on where the separators intersect, small clusters, on which the compression cannot provide much memory and operation reduction, may be formed; as a result, the CB may include blocks which are too small to be effectively compressed and to achieve a good BLAS efficiency for the related operations. Note that this problem also affects the blocking of the L_{21} submatrix but to a lower extent because the effect is damped by the good clustering computed for the separator (see Figure 3.12).



(a) L_{11} blocking : optimal blocks because any block interconnects two optimal clusters from the corresponding original separator clustering

(b) L_{21} blocking : close to optimal block because any block interconnects at least one optimal cluster from the corresponding original separator clustering

(c) CB blocking : not always optimal because a block possibly interconnects two non-optimal small clusters from the inherited clustering of the border. Braces show a reclustering possibility.

Figure 3.12: Relation between inherited clustering and front blocking. The small clusters correspond to clusters located in corners in Figure 3.10(b). The large clusters correspond to optimal clusters which are integrally kept in the current front. Note that not all the large clusters of Figure 3.10(b) are represented here.

To recover BLAS efficiency, a reclustering step can be performed by merging neighbor clusters together in order to increase the block size, as Figure 3.12(c) shows. Recovering the low-rank compression is less straightforward. Indeed, it is not guaranteed that two neighbor blocks in the front correspond to two neighbor clusters in the graph. Constraining the clustering strategy to obtain this property considerably increases its complexity (by the addition of notions such as global cluster ordering, recursive ordering as in HSS for instance) with a small payoff since the proportion of small clusters in a front is usually very small.

It has to be noted that the inherited clustering also provides another convenient property: the blocking of a frontal matrix is compatible with the blocking of its parent front. This translates into the fact that one block of its Schur complement will be assembled into exactly one block of the parent front, as by construction a block of a Schur complement is always included in one single block in the fully-summed part of another frontal matrix. This considerably eases the assembly of frontal matrices especially in the parallel case where frontal matrices are distributed.

Experiments have shown that, as expected, the inherited clustering is much faster than the explicit clustering (see Table 3.2 with the example of the `Helmholtz128` problem), with very similar global performance. Moreover, the overhead due to the clustering is quite low compared to the analysis phase (which is itself, often, dominated by the factorization phase).

clustering		full rank analysis
inh	exp	
5 s.	42 s.	62 s.

Table 3.2: Time spent in the clustering phase in the explicit (exp) and inherited (inh) cases. The clustering being performed during the analysis phase, we also indicate the full-rank analysis time.

The inherited clustering strategy will thus be used throughout this study.

3.4 Factorization

Fronts are assembled following an order computed during the analysis and leading to contiguous BLR blocks. The objective is then to perform all the standard computations in a way that the low-rank kernels presented in Section 1.5.3 can be exploited to reduce the computational cost associated to the partial factorizations. This requires the design of new algorithms which are able to benefit from the low-rank operation efficiency without giving up all the other features needed in a general multifrontal solver such as pivoting and parallelism. We first define some notations related to the standard tasks performed in a full-rank partial factorization and use them afterwards to present several new BLR algorithms, which shows the algorithmic flexibility of our approach. We then explain how pivoting and parallelism can be achieved within this context.

3.4.1 Standard tasks

Once a front has been assembled, a partial factorization is performed in order to compute the corresponding part of the global factors. These computations have a large influence on the global performance of the software. Moreover, the storage of these factor parts becomes larger and larger as matrix size grows. It is thus critical to improve this phase. We assume that clusterings $\mathcal{C}_{FS}$ for the fully-summed (FS) variables and $\mathcal{C}_{NFS}$ for the non fully-summed (NFS) variables have been computed, which give a blocking of the front. In the full-rank standard case, four fundamental tasks must be performed in order to compute the blocked incomplete factorization of the front F:

1.	factor (F)	$L_{\rho,\rho}L_{\rho,\rho}^T = F_{\rho,\rho}$	for $\rho \in \mathcal{C}_{FS}$.
2.	solve (S)	$L_{\tau,\rho} = F_{\tau,\rho}L_{\rho,\rho}^{-T}$	for $\rho \in \mathcal{C}_{FS}$, $\tau \in \mathcal{C}_{FS} \cup \mathcal{C}_{NFS}$, where $\tau > \rho$ ($\tau > \rho$ if $F_{\tau,\tau}$ appears after $F_{\rho,\rho}$ on the diagonal).
3.	internal update (U)	$F_{\tau,\sigma} = F_{\tau,\sigma} - L_{\tau,\rho}L_{\sigma,\rho}^T$	for $\rho, \sigma \in \mathcal{C}_{FS}$, $\tau \in \mathcal{C}_{FS} \cup \mathcal{C}_{NFS}$, where $\tau \geq \sigma > \rho$.
4.	external update (U)	$F_{\tau,\sigma} = F_{\tau,\sigma} - L_{\tau,\rho}L_{\sigma,\rho}^T$	for $\rho \in \mathcal{C}_{FS}$, $\tau, \sigma \in \mathcal{C}_{NFS}$, where $\tau \geq \sigma$.

As the factorization is done panelwise, two "update" phases are defined. The internal update is the right-looking update of blocks whose rows or columns correspond to fully summed variables. The external update is the right-looking update of the contribution block, as illustrated in Figure 3.13.

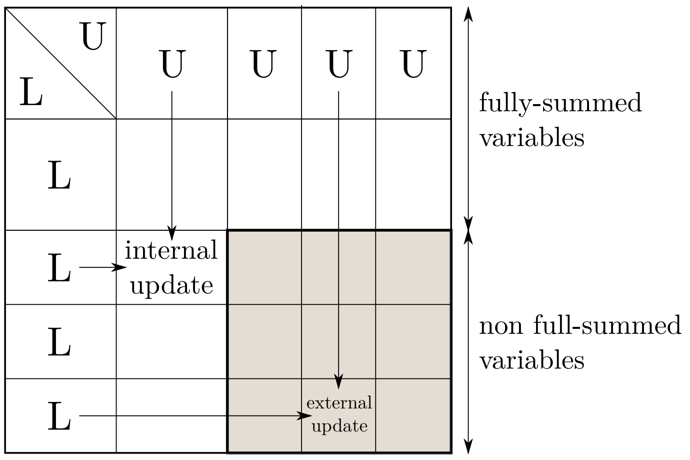


Figure 3.13: Difference between internal and external update. All shaded blocks are updated through external update are they belong to the contribution block. All white blocks are updated through internal update.

3.4.2 Low-rank algorithms

Looking back at Section 1.5.3, one can observe that three of the four standard tasks can be improved by using low-rank blocks: the "solve" and the two "update" phases.

To obtain BLR algorithms, it is necessary to compress the blocks at some point. We thus add a new task which corresponds to the compression of a block defined by the clusterings:

5.	compress (C)	$L_{\tau,\rho} \simeq X_{\tau,\alpha_1}Y_{\rho,\alpha_2}^T$	for $\rho \in \mathcal{C}_{FS}$, $\tau \in \mathcal{C}_{FS} \cup \mathcal{C}_{NFS}$, where $\lvert\alpha_1\rvert = \lvert\alpha_2\rvert =$ numerical rank of $L_{\tau,\rho}$.

Due to the flexibility of the BLR format, several versions of the BLR factorization of a front can be implemented based on these five tasks, depending on the position of the compression. Figure 3.14 presents five different algorithms for the factorization of a front. Note that for the sake of simplicity, the order relations between σ, ρ and τ are now ignored. Similarly, the subscripts $_1$ and $_2$ in α_1 and α_2 will be omitted. The first algorithm is the conventional full-rank partial factorization [57] and is called FSUU, which stands for *Factor, Solve, internal Update, external Update.* The other four are based on the BLR representation of the front and differ on when the compression is performed. As we move down the figure, from FSUUC to FCSUU, the computational cost decreases as we are performing the compression earlier and thus making more and more of an approximation to the factorization.

```
FSUU : no compression
    for $\rho \in \mathcal{C}_{FS}$ in ascending order
F       factor $F_{\rho,\rho} = L_{\rho,\rho}L^T_{\rho,\rho}$
S       solve $L_{\tau,\rho} = F_{\tau,\rho}L^{-T}_{\rho,\rho}$                          for $\tau \in \mathcal{C}_{FS} \cup \mathcal{C}_{NFS}$
U       internal update $F_{\tau,\sigma} = F_{\tau,\sigma} - L_{\tau,\rho}L^T_{\sigma,\rho}$   for $\sigma \in \mathcal{C}_{FS}$, $\tau \in \mathcal{C}_{FS} \cup \mathcal{C}_{NFS}$
U       external update $F_{\tau,\sigma} = F_{\tau,\sigma} - L_{\tau,\rho}L^T_{\sigma,\rho}$   for $\tau, \sigma \in \mathcal{C}_{NFS}$

FSUUC : compress after updates
    for $\rho \in \mathcal{C}_{FS}$ in ascending order
F       factor $F_{\rho,\rho} = L_{\rho,\rho}L^T_{\rho,\rho}$
S       solve $L_{\tau,\rho} = F_{\tau,\rho}L^{-T}_{\rho,\rho}$                          for $\tau \in \mathcal{C}_{FS} \cup \mathcal{C}_{NFS}$
U       internal update $F_{\tau,\sigma} = F_{\tau,\sigma} - L_{\tau,\rho}L^T_{\sigma,\rho}$   for $\sigma \in \mathcal{C}_{FS}$, $\tau \in \mathcal{C}_{FS} \cup \mathcal{C}_{NFS}$
U       external update $F_{\tau,\sigma} = F_{\tau,\sigma} - L_{\tau,\rho}L^T_{\sigma,\rho}$   for $\tau, \sigma \in \mathcal{C}_{NFS}$
C       compress $L_{\tau,\rho} \simeq X_{\tau,\alpha}Y^T_{\rho,\alpha}$                    for $\tau \in \mathcal{C}_{FS} \cup \mathcal{C}_{NFS}$

FSUCU : compress after internal and before external updates
    for $\rho \in \mathcal{C}_{FS}$ in ascending order
F       factor $F_{\rho,\rho} = L_{\rho,\rho}L^T_{\rho,\rho}$
S       solve $L_{\tau,\rho} = F_{\tau,\rho}L^{-T}_{\rho,\rho}$                          for $\tau \in \mathcal{C}_{FS} \cup \mathcal{C}_{NFS}$
U       internal update $F_{\tau,\sigma} = F_{\tau,\sigma} - L_{\tau,\rho}L^T_{\sigma,\rho}$   for $\sigma \in \mathcal{C}_{FS}$, $\tau \in \mathcal{C}_{FS} \cup \mathcal{C}_{NFS}$
C       compress $L_{\tau,\rho} \simeq X_{\tau,\alpha}Y^T_{\rho,\alpha}$                    for $\tau \in \mathcal{C}_{FS} \cup \mathcal{C}_{NFS}$
U       external update $F_{\tau,\sigma} = F_{\tau,\sigma} - X_{\tau,\alpha}\ \ Y^T_{\rho,\alpha}Y_{\rho,}\ \ X^T_{\sigma,}$   for $\tau, \sigma \in \mathcal{C}_{NFS}$

FSCUU : compress before updates
    for $\rho \in \mathcal{C}_{FS}$ in ascending order
F       factor $F_{\rho,\rho} = L_{\rho,\rho}L^T_{\rho,\rho}$
S       solve $L_{\tau,\rho} = F_{\tau,\rho}L^{-T}_{\rho,\rho}$                          for $\tau \in \mathcal{C}_{FS} \cup \mathcal{C}_{NFS}$
C       compress $L_{\tau,\rho} \simeq X_{\tau,\alpha}Y^T_{\rho,\alpha}$                    for $\tau \in \mathcal{C}_{FS} \cup \mathcal{C}_{NFS}$
U       internal update $F_{\tau,\sigma} = F_{\tau,\sigma} - X_{\tau,\alpha}\ \ Y^T_{\rho,\alpha}Y_{\rho,}\ \ X^T_{\sigma,}$   for $\sigma \in \mathcal{C}_{FS}$, $\tau \in \mathcal{C}_{FS} \cup \mathcal{C}_{NFS}$
U       external update $F_{\tau,\sigma} = F_{\tau,\sigma} - X_{\tau,\alpha}\ \ Y^T_{\rho,\alpha}Y_{\rho,}\ \ X^T_{\sigma,}$   for $\tau, \sigma \in \mathcal{C}_{NFS}$

FCSUU : compress before solve
    for $\rho \in \mathcal{C}_{FS}$ in ascending order
F       factor $F_{\rho,\rho} = L_{\rho,\rho}L^T_{\rho,\rho}$
C       compress $F_{\tau,\rho} \simeq X_{\tau,\alpha}Z^T_{\rho,\alpha}$                    for $\tau \in \mathcal{C}_{FS} \cup \mathcal{C}_{NFS}$
S       solve $L_{\tau,\rho} = X_{\tau,\alpha}(Z^T_{\rho,\alpha}L^{-T}_{\rho,\rho})$             for $\tau \in \mathcal{C}_{FS} \cup \mathcal{C}_{NFS}$
        i.e, $L_{\tau,\rho} = X_{\tau,\alpha}Y^T_{\rho,\alpha}$ with $Y^T_{\rho,\alpha} = Z^T_{\rho,\alpha}L^{-T}_{\rho,\rho}$
U       internal update $F_{\tau,\sigma} = F_{\tau,\sigma} - X_{\tau,\alpha}\ \ Y^T_{\rho,\alpha}Y_{\rho,}\ \ X^T_{\sigma,}$   for $\sigma \in \mathcal{C}_{FS}$, $\tau \in \mathcal{C}_{FS} \cup \mathcal{C}_{NFS}$
U       external update $F_{\tau,\sigma} = F_{\tau,\sigma} - X_{\tau,\alpha}\ \ Y^T_{\rho,\alpha}Y_{\rho,}\ \ X^T_{\sigma,}$   for $\tau, \sigma \in \mathcal{C}_{NFS}$
```

Figure 3.14: Standard and BLR factorizations of a symmetric front. The flexibility of the BLR format allows the definition of 4 different BLR factorization algorithms. When two different compressed blocks have to be used, variable   is used in addition to α.

The algorithms in Figure 3.14 target different objectives in the context of applicative solvers. FSUUC can be used when an accurate factorization is needed (there is no approximation during the factorization in this case because the compression is fully done off-line at the end of standard processing of each front) and when speeding up the solve phase is critical, for instance if many right-hand sides are involved. This may be particularly beneficial in Newton-type solvers where multiple solves have to be done based on the same factorization for example. On the other hand, FCSUU computes a more approximated factorization, useful in the typical context of preconditioning or when a very accurate solution is not needed due to, for instance, measurements limitations. This strategy limits the scope of BLR pivoting because no permutations within a BLR panel can be done after compression.

Note that a sixth task can be added which corresponds to the compression of the CB:

6. external compress (C_e) $\quad F_{\tau,\sigma} \simeq U_{\tau,\alpha} V_{\sigma,\alpha}^T \quad$ for $\tau, \sigma \in \mathcal{C}_{NFS}$.

This task can be performed at the end of any of the algorithms in Figure 3.14 in order to decrease the active memory size as well as the amount of communication necessary to assemble the parent front in a parallel context, as explained in Section 3.2. This external compression could also theoretically be done before the external update, in which case these right-looking updates have to be done in compressed format. This considerably increases the complexity of the operation as recompression phases are needed. For this reason, we did not consider this option.

We decided to implement and study Algorithm FSCUU as it gives the best compromise between savings (for both memory and operations) and robustness of the solver:

- The number of operations needed for the solve task is much lower than for the internal and external updates so that we can focus on these two latter tasks without using a low-rank solve operation.
- The solve task is done accurately which avoids a twofold approximation of the factors.
- Standard symmetric and unsymmetric pivoting strategies can be used which ensures the robustness of the solver (in which case the tasks F and S are, in pratice, done simultaneously). In the context of a general distributed multifrontal solver which aims at solving large problems, pivoting is indeed critical to tackle challenging problems, on which low-rank technologies are particularly efficient (see results in Chapter 4).

In the case of symmetric matrices, note that the Update operation can be further improved by exploiting the orthogonality of the U part of the low-rank form. If one compresses $L_{\sigma,\rho}^T$ instead of $L_{\sigma,\rho}$, the BLR update operation becomes

$$F_{\tau,\sigma} = F_{\tau,\sigma} - Y_{\tau,\alpha}^T \quad X_{\tau,\alpha}^T X_{\sigma,} \quad Y_{\sigma,}$$

When $\tau = \sigma$, we obtain $X_{\tau,\alpha} = X_{\sigma,}$ which yields $X_{\tau,\alpha}^T X_{\sigma,} \quad = I$ thanks to the orthogonality of X.

3.4.3 Implementation

To implement algorithm FSCUU efficiently, nested panels are introduced for the partial factorization of each front. The original panels, as explained in Section 1.6.2 (Figure 1.23), are kept unchanged and are from now on called the *inner* panels. The other panels are larger than the inner ones and are called *outer* panels. They can be viewed as panels of panels and correspond to the BLR blocks defined by the clustering of the fully-summed variables (for this reason, they will be sometimes also referred to as BLR panels, without distinction). This idea is illustrated in Figure 3.15.

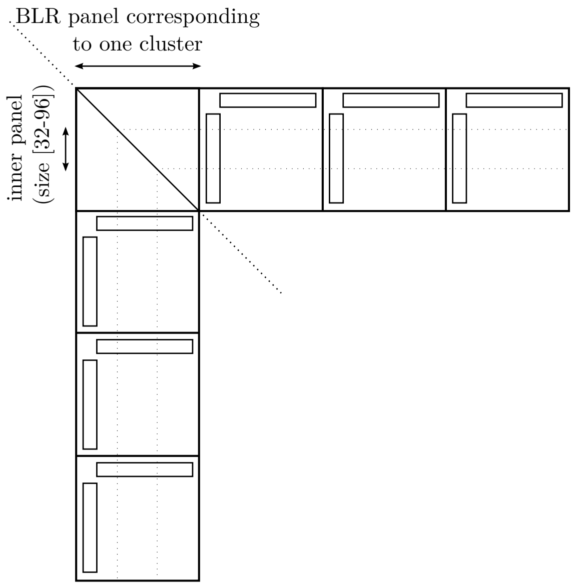

BLR panel corresponding to one cluster
inner panel (size [32-96])

Figure 3.15: An example of nested panels. Inner panels correspond to standard panels of size 32, 64 or 96, typically. Outer BLR panels are defined by the clustering. Once each inner panel has been processed, the BLR panel is ready to be demoted.

In the BLR factorization, a given inner panel behaves the same way as in the standard factorization, except that it no longer updates all the trailing submatrix with BLAS 3 operations; instead, it performs BLAS 3 right-looking updates only within the BLR panel. At the end of the processing of a BLR panel, the trailing submatrix can be updated through BLAS 3 operations taking advantage of the compressed form of the blocks. In practice, this yields a procedure where the Factor and Solve phases (see Section 3.4.2) are performed within each BLR panels by means of inner panels computations. Then the compression is performed on current BLR panel and the BLR right-looking update can be done. If a full-rank factorization is performed, this nested panel scheme will also be used with a default outer panel size of 100. This allows for a slightly faster full-rank factorization due to faster cache accesses.

Note that this nested panels scheme can also be adapted to any of the algorithms presented in Section 3.4.2, by slightly rearranging and modifying the way operations within inner panels are performed. For instance, if one is interested in implementing algorithm FCSUU, operations on inner panels would have to be limited to the diagonal block of the BLR panel. When this diagonal block is fully updated, the compression can be performed and then the Solve phase can be performed on already compressed block at the outer panel scale.

3.4.4 Pivoting

Thanks to the flexibility of the BLR format, standard pivoting can be applied in the BLR factorization algorithm in order to ensure a better numerical stability in case of hard problems. Pivoting can be performed for any BLR factorization algorithm besides FCSUU, as explained in Section 3.4.2. Pivot postponing happens at both the inner and the outer panels. Pivots are first postponed (if they could not be eliminated), as in the full-rank case, between inner panels in the current BLR panel. When all the variables of current BLR panel have been processed, if uneliminated variables remain, then the second level of postponing occurs: these variables are postponed to the next BLR panel (and, thus, to its first inner panel) which is consequently enlarged, as illustrated in Figure 3.16. At the end of the last outer panel, if uneliminated variables remain, they are then delayed to a parent front, just as in the standard algorithm presented in Section 1.6.2.

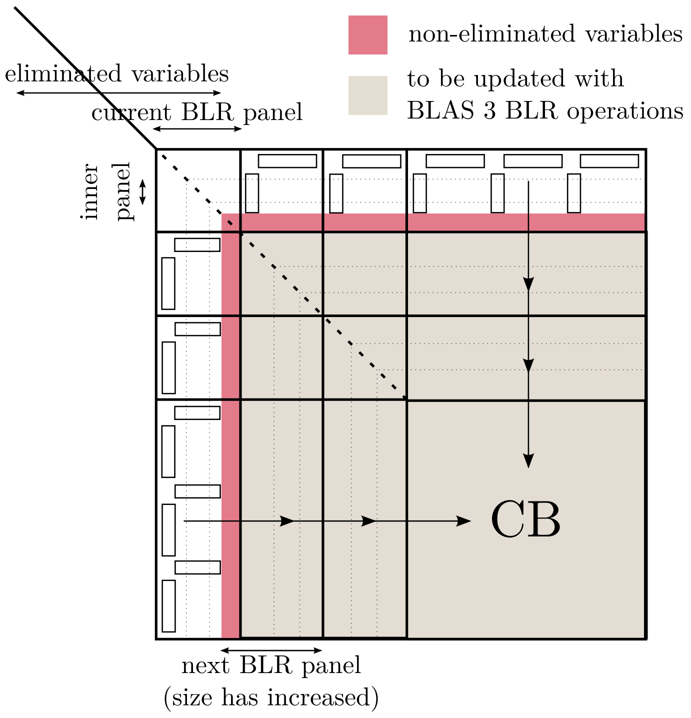

non-eliminated variables
to be updated with BLAS 3 BLR operations
eliminated variables
current BLR panel
inner panel
CB
next BLR panel (size has increased)

Figure 3.16: Sequential BLR factorization scheme with BLR pivoting. Inner panels are of size 32, 64 or 96, typically. Outer panels are defined by the clustering. The internal right-looking updates, performed within each outer panel using inner panels are not represented. Arrows represent the right-looking updates performed with BLR blocks, both in the fully-summed part and the non fully-summed part.

This standard, robust pivoting strategy can be easily coupled with the BLR format thanks to its flat structure. It is indeed possible to dynamically modify the size of a cluster without perturbing the format and without increasing the complexity of the computations. The only issue in this operation is whether or not the admissibility of the clustering will be maintained in such cases. In situations where many postponed (or delayed) pivots occur, the compression may indeed be heavily degraded. Fortunately, as a relatively small number of fronts is selected for BLR (see Section 3.6), the number of postponed (or delayed) pivots within BLR fronts is likely to be contained. The influence of pivoting over the compression rates will be investigated in Section 4.2.5.

Note that this nested panel implementation can also be used when a full-rank factorization is performed. For locality reasons (cache access), this implementation has showed slightly better performance in the full-rank case than with the single panel standard implementation.

3.4.5 Distributed-memory parallelism

The standard communication pattern presented in Section 1.6.2 also benefits from the nested panels and the compression of the BLR blocks within outer panels. For the sake of simplicity, the symmetric case is not described.

The master sends rows at the end of the processing of each outer panel, so that compressed blocks of U can be sent, decreasing the total volume of messages involved during the factorization of type 2 nodes. Each slave is thus in charge of compressing its own BLR blocks of L and to use them as well as the received BLR blocks to perform the BLR right-looking update of its rows. Note that the master also perform compressions in the blocks of L corresponding to the fully-summed variables block, as illustrated in Figure 3.17 in the unsymmetric case.

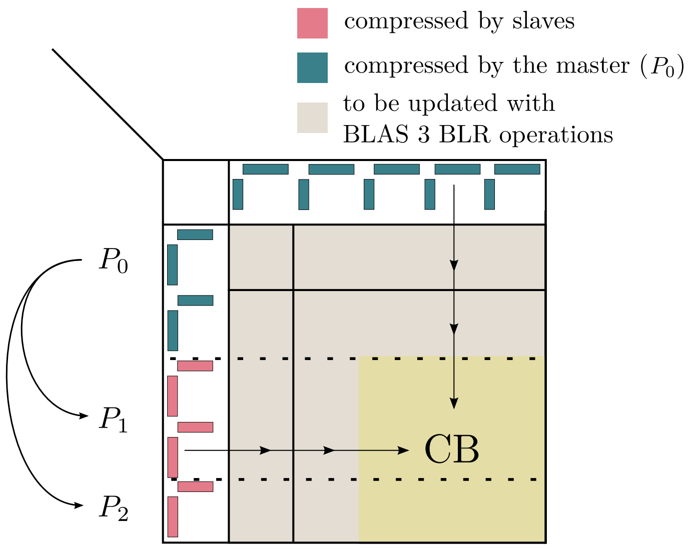

compressed by slaves
compressed by the master (P_0)
to be updated with BLAS 3 BLR operations
P_0
P_1
P_2
CB

Figure 3.17: Communication pattern used within type 2 nodes. P_0 is the master process, holding all the fully-summed rows. In the unsymmetric case, it is the only process sending messages. In the symmetric case, slaves also send messages to other slaves. Compression is done both by the master and by the slaves. Compressed blocks are sent so that the volume of communication decreases.

The data access pattern of the discussed parallel factorization algorithm raises the question of the distribution of the rows among available slave processes. In an ideal case, the block partitioning induced by the clustering would conform with the 1D row-wise distribution among slave processes, meaning a Block Low-Rank would not be split between two slaves. However, this is in practice not possible since the clustering is performed statically during the analysis while the distribution pattern is dynamically adapted during the execution (based on a complex strategy which aims at fulfilling some critical memory and workload constraints [11]). Thus, one must adjust the clustering to the distribution pattern on-fly to ensure that good parallel performance is maintained. Fortunately, one of the arguments for choosing a flat, flexible low-rank format (see Section 2.2) is that the clustering can easily be dynamically adapted to other constraints, such as parallel distribution, without greatly perturbing the structure. In this context, at most two blocks per slave will be affected (the first and the last one of each slave) and because several blocks will be given to each process, this does not affect much the overall compression, as shown in Section 4.2.6. The dynamic clustering adaptation is illustrated in Figure 3.18 in a typical configuration, where 4 blocks out of 17 have slightly changed. The figure represents only slave processes as the master process always respects the original clustering.

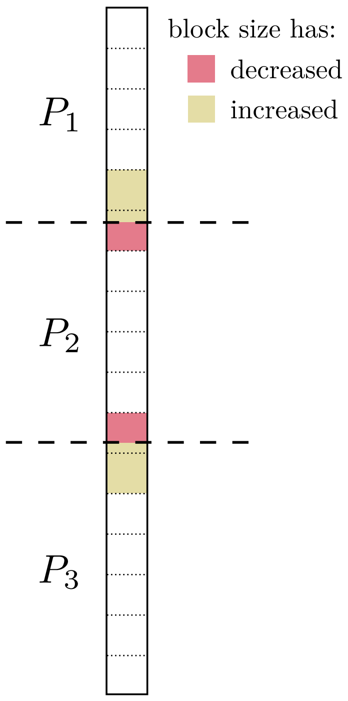


Figure 3.18: Dynamic adaptation of the clustering to adapt to the 1D row-wise distribution constraints, represented with dashed lines. Dotted lines represent the original clustering. The actual clustering used for low-rank compressions remains unchanged except for four blocks: two are enlarged, two are shortened.

The BLR multifrontal factorization is thus an improved factorization where both classical pivoting and distributed-memory parallelism methods can still be used with only minor modifications. The BLR factorization algorithms being based on compressing the blocks of the factors, the solution phase (which exploits the factors) has to be adapted, too. Interestingly, low-rank kernels together with compressed factors can also be exploited in this phase to decrease the number of operations involved.

3.5 Block Low-Rank solution

We first present the BLR solution algorithm and then theoretically demonstrate how the memory compression of the factor and the operation compression of the solution are related. Note that the BLR solution algorithm has not been implemented yet, so that the factors have to be decompressed before performing the forward elimination and the backward substitution phases.

Let F^i be the partial factor computed within front i. The lower triangular part of F^i is L^i while its upper triangular part is U^i. Assume F^i is stored as a BLR lower triangular matrix with d_i clusters in the fully-summed part, and h_i clusters on the non fully-summed part. The structure of F^i is illustrated in Figure 3.1. Each diagonal block is full-rank. Each off-diagonal block L^i_{α} or U^i_{α} has a rank of k^i_{α}. The clusters are all assumed of same size b. This assumption is not a strong constraint as the clustering of the front is usually well balanced, and will be useful for Section 3.5.2.

As the same computations are performed on all the fronts during the multifrontal solution phase, we will focus throughout this section on the factor F^i associated with a given front.

3.5.1 Algorithm

As showed in Section 1.3, the multifrontal solution consists in two phases, namely the forward elimination and the backward substitution. Each of these two phases can be viewed as a tree traversal. At each node of the tree, a partial solve operation is performed to compute the entries of the unknowns corresponding to the fully summed variables of the corresponding front. Some entries of the right-hand side are also updated. If the factor is compressed, the standard algorithm in Section 1.3 has to be slightly modified to benefit from the BLR forms, resulting in Algorithm 3.1 and 3.2.

The BLR forward elimination is given in Algorithm 3.1. This procedure is applied to all the fronts of the assembly tree, in a bottom to up traversal.

Algorithm 3.1 BLR forward elimination in front F_i.

```
1: **Solution of** $L^i y^i = b^i$
2:
3: $y^i \quad b^i$
4: **for** $c = 1$ **to** $d_i$ **do**
5:     $y_c \quad (L^i_{c,c})^{-1} \quad y_c$
6:     **for** $r = c + 1$ **to** $d_i + h_i$ **do**
7:         $y_r \quad y_r - X^i_{rc} \quad (Y^i_{rc})^T \quad y_c$        ▶ low-rank product
8:     **end for**
9: **end for**
```

Figure 3.19 gives a pictorial explanation of Algorithm 3.1.

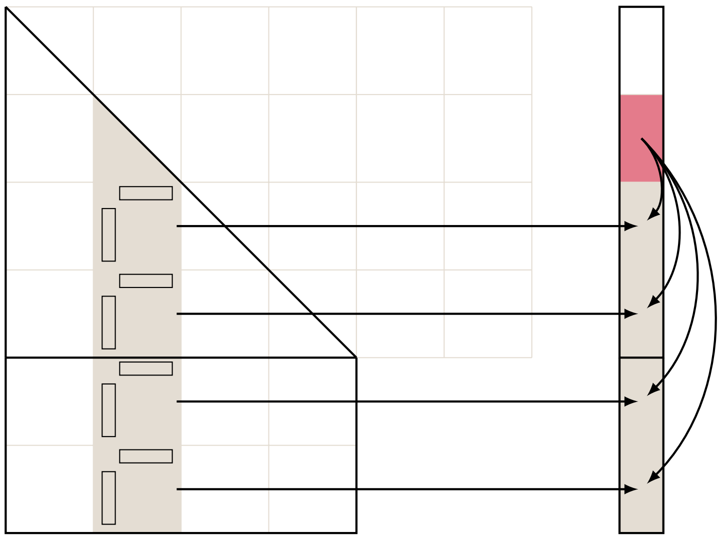


Figure 3.19: Illustration of the BLR forward elimination in front F_i.

Note that the forward elimination is done in a right-looking way. It is followed by a backward substitution phase, which is done in a left-looking fashion. This phase is applied to each front following a top-down traversal of the tree and the algorithm is given in Algorithm 3.2.

Algorithm 3.2 BLR backward substitution F_i.

```
1: Solution of $U^i x^i = y^i$
2:
3: $x^i \quad y^i$
4: for $r = d_i$ to 1 by $-1$ do
5:     for $c = r + 1$ to $d_i + h_i$ do
6:         $x^i_r \quad x^i_r - X_{rc} \quad ((Y_{rc})^T \quad x^i_c)$        ▶ low-rank product
7:     end for
8:     $x^i_r \quad (U^i_{r,r})^{-1} \quad x^i_r$
9: end for
```

As for the forward elimination, we give in Figure 3.20 a pictorial explanation of the algorithm.

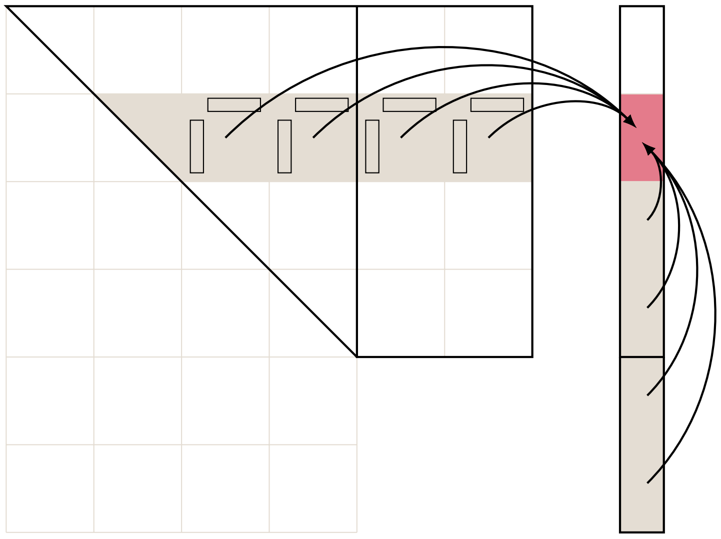


Figure 3.20: Illustration of the BLR forward elimination in front F_i.

The low-rank blocks which appear in the factors L and U can thus be used to reduce the amount of operations needed for both the forward elimination and the backward substitution phases of the multifrontal solution phase. Interestingly, this operation reduction is directly related to the memory reduction achieved in the factors. We show in the next section the actual relationship between them.

3.5.2 Operation count

If using the low-rank blocks during the solution phase is algorithmically quite straightforward thanks to the flat non-hierarchical BLR approach, the exact relationship between the overall factor compression of the factor and the operation compression in the solve phase is less obvious.

We first focus on a given front i (i.e., on a partial solution), as illustrated in Figure 3.1 and will use the notations introduced in the introduction of Section 3.5. Then, the result is extended to the overall solution process and finally to the overall solve process.

We remind the reader that the number of operations needed to compute x from $LUx = b$ (where L and U are $N \quad N$ lower and upper triangular matrices, respectively, b is a vector of size N) is $2N^2$ [57] (using forward elimination and backward substitution).

A compression rate $T \in \mathbb{R}, 0 \quad T \quad 1$ between an original data D_o and a final data D_f is defined as follows:

$$D_f = T \quad D_o$$

For instance, the BLR factor compression rate T_F is defined as:

$$|L_{BLR}| = T_F \quad |L_{FR}|$$

where $|L_{FR}|$ and $|L_{BLR}|$ are the number of entries in the full-rank factors and in the BLR factors, respectively.

We first want to compute the number of operations needed to perform the partial solution on a given front i. The result is given in Lemma 3.1.

Lemma 3.1 - Partial solution operation count for a BLR front.
Let L^i and U^i be the factors associated with front i, having d_i clusters of size b in the fully-summed part and h_i clusters of size b in the non fully-summed part, the operation count for the partial solution with a single right-hand side is

$$S_i(d_i, h_i) = 2d_ib^2 + 4b \underset{=1}{\overset{d_i}{}} \underset{\alpha= \quad +1}{\overset{d_i+h_i}{}} (k^i_{\alpha} + k^i{}_{\alpha}),$$

where k^i_{α} is the rank of the block cluster F^i_{α} .

Proof. On each diagonal block two triangular solutions (a lower and an upper) are performed. Each of these needs b^2 operations so that $2b^2$ are needed for each diagonal block. There are d_i diagonal blocks, which gives the first term in $S_i(d_i, h_i)$.

Then, each off-diagonal block steps in a low-rank update of a part of the right-hand side. The cost of this operation for F^i_{α} is $4bk^i_{\alpha}$. We sum up this term over all the off-diagonal blocks in L^i and U^i and we obtain the second term in $S_i(d_i, h_i)$. □

This absolute operation count has to be related to the factor compression so that it is possible to estimate easily the operation count. The result is given in Lemma 3.2.

Lemma 3.2 - Operation compression rate for the BLR partial solution.
Given L^i and U^i the BLR factors associated with front i and the corresponding factor memory compression T_{F_i}, the operation compression rate for the BLR partial solution with a single right-hand side is

$$T_{S_i} = T_{F_i}$$

Proof. We write the number of elements in the compressed L^i and U^i in two different ways. The first way is to sum up the entries blockwise. Each diagonal block has b^2 entries while each off-diagonal block F^i_{α} has $bk^i_{\alpha} + bk^i_{\alpha}$, which gives a total number of entries of:

$$d_ib^2 + 2b \underset{=1}{\overset{d_i}{}} \underset{\alpha= \quad +1}{\overset{d_i+h_i}{}} (k^i_{\alpha} + k^i{}_{\alpha}) \tag{3.1}$$

The second way is to use the definition of T_{F_i}, which multiplies the full-rank number of entries to obtain the BLR number of entries in F^i:

$$T_{F_i}(d_i^2b^2 + 2d_ih_ib^2) \tag{3.2}$$

By definition, we have (3.1) = (3.2) so that:

$$\frac{1}{2}\ \ 2d_ib^2 + 4b \overset{d_i}{\underset{=1}{}}\ \overset{d_i+h_i}{\underset{\alpha=\ \ +1}{}} (k^i_\alpha \ \ + k^i{}_{\ \alpha}) \ = T_{F_i}\ \ d_i^2b^2 + 2h_id_ib^2$$

We substitute the left-hand side term using Lemma 3.1 to obtain:

$$S_i(d_i, h_i) = 2T_{F_i}\ \ d_i^2b^2 + 2h_id_ib^2 \tag{3.3}$$

$FR_i = 2h_id_ib^2 + b^2d_i^2 = (bd_i)^2 + 2(h_ib)(d_ib)$ is the number of operations needed to perform a full-rank partial solution on factor L^i. Then:

$$S_i(d_i, h_i) = T_{F_i}\ \ FR_i$$

Then we obtain the desired result by definition of the compression rate. □

In Theorem 3.1, we extend this result to the whole multifrontal solution, assuming all the fronts have been processed with BLR techniques. In practice, the smallest fronts are kept full-rank but their associated volume of memory usage and computation is negligible so that our assumption is consistent.

Theorem 3.1 - Operation compression rate for the BLR multifrontal solution.
Given a BLR multifrontal factorization of a matrix of size N with NZ entries in the factors L and U and a global factor memory compression rate of T_F, then the global operation compression rate for the BLR multifrontal solution is:

$$T_S = T_F$$

Proof. The number of fronts in the assembly tree will be referred to as N_F. The proof basically follows the same methods as for Lemma 3.1 and Lemma 3.2.

We first want to calculate $S = \ {}^{N_F}_{i=1} S_i(h_i, d_i)$. Equation (3.3) gives:

$$S = 2 \overset{N_F}{\underset{i=1}{}} T_{F_i}(d_i^2b^2 + 2h_id_ib^2) \tag{3.4}$$

By definition of T_S, we also have:

$$T_F = \frac{{}^{N_F}_{i=1} T_{F_i}(d_i^2b^2 + 2h_id_ib^2)}{{}^{N_F}_{i=1} d_i^2b^2 + 2h_id_ib^2)} \tag{3.5}$$

Substituting (3.5) in (3.4), we obtain:

$$S = 2T_F \overset{N_F}{\underset{i=1}{}} (b^2d_i^2 + h_id_ib^2)$$

$FR = \ {}^{N_F}_{i=1} 2h_id_ib^2 + b^2d_i^2 = \ {}^{N_F}_{i=1}(bd_i)^2 + 2(h_ib)(d_ib)$ is the number of operations needed to perform the full-rank solution. Then:

$$S = T_F\ \ FR$$

which gives the desired result by definition of a compression rate. □

3.6 Switch level

It is not necessary, nor efficient, to use low-rank approximations on all the frontal matrices. Indeed, to obtain good compression rates which overcome the compression costs, it is better to consider only larger fronts. How much work and memory is done and consumed in fronts larger than a given size can be easily assessed on a regular 9-point stencil by reusing some work and notations by Rouet [86], itself based on George [51] paper on nested dissection; the results of this analysis are shown in Figure 3.21. The elimination tree directly based on a complete nested dissection is used (i.e. a nested dissection where the smallest separators have size 1). However, in practice, the nested dissection is stopped before the separators reach size 1 and the corresponding elimination tree is post-processed with different techniques, such as *amalgamation*, which aim at merging fronts in order to increase efficiency. The resulting tree is called the *assembly* tree and contains basically less fronts but larger ones. For this reason, in a practical context, the graphs shown in Figure 3.21(a) would be translated upwards. This remark should be taken into account when reading the analysis below.

Figures 3.21(a) and 3.21(b) show that most of the factor entries are computed and almost all of the floating point operations are done within fronts of relatively large size. This is particularly interesting considering that these larger fronts only account for a very small fraction of the total number of fronts in the assembly tree, as shown in Table 3.3.

	mesh size					
	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}
$N = 200$	1 61‰	1 68‰	1 71‰	1 73‰	1 74‰	1 74‰
$N = 400$	0 37‰	0 40‰	0 42‰	0 42‰	0 43‰	0 43‰
$N = 600$	0 17‰	0 20‰	0 21‰	0 22‰	0 23‰	0 23‰
$N = 800$	0 07‰	0 09‰	0 10‰	0 11‰	0 11‰	0 11‰
$N = 1000$	0 05‰	0 06‰	0 06‰	0 06‰	0 06‰	0 06‰

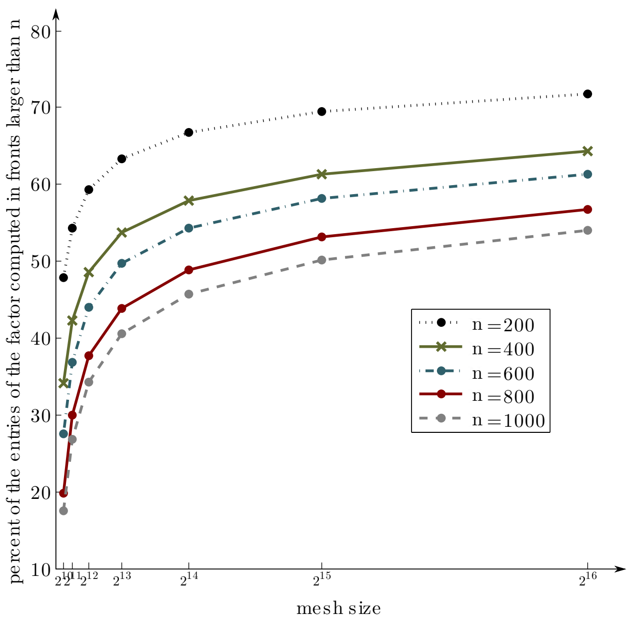
Table 3.3: Proportion of fronts larger than N with respect to the total number of fronts in the assembly tree, for different mesh sizes.

This shows that there is no need to compress small fronts for two reasons:

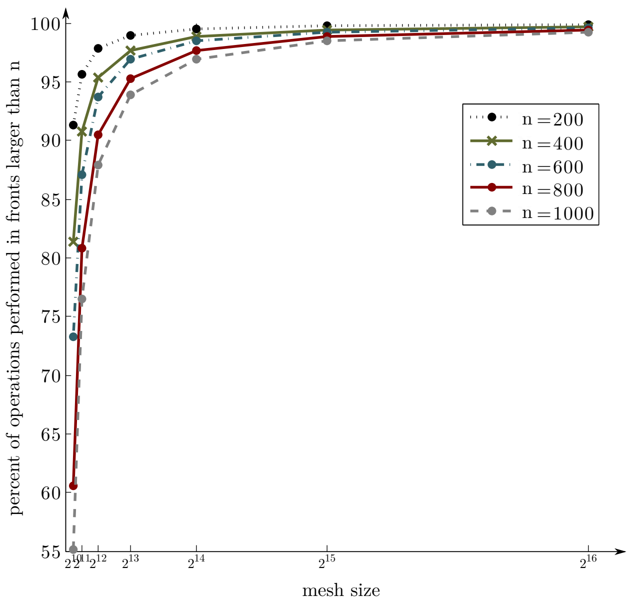
1. It would be too much work for a small gain (in small fronts we can only target a few entries and a few computations that cannot justify the additional cost of compression) ;

2. It is more critical to focus on the top of the tree because the corresponding large fronts represent most of the work to be done in the multifrontal process, in terms of computations and memory.

In practice, this notion of switch level translates into a condition on the minimal separator size (or *NASS min*, which is also the number of fully-summed variables of a given front) to select a front for BLR. Even if it corresponds to an actual level only when the global ordering of the matrix is based on a nested dissection, we will use this expression with any global ordering. How to select properly the switch level for an efficient



(a) Proportion of entries of the factors computed in the frontal matrices larger than n.



(b) Proportion of the total number of operations performed for the partial factorization of the frontal matrices larger than n .

Figure 3.21: Proportions of entries of the factors and of number of operations in the top levels of the assembly tree. The problem studied is a 2D PoissonN. The corresponding matrix size is thus the square of the mesh size N. Note that the largest fronts we look at in these plots represent very few fronts compared to the total number of fronts, as shows Table 3.3.

BLR factorization will be investigated in Section 4.2.2, where we will show that now fine tuning is needed, which is critical in the context of general multifrontal solver.

Chapter 4

Evaluation of the potential of the method

In this Chapter, we present experimental results which demonstrate the efficiency of the BLR multifrontal method on the two equations presented in Section 1.6.4.1: Poisson and Helmholtz. We investigate in sequential the influence of many parameters of the BLR method (cluster size, switch level, low-rank threshold) and of the problem (ordering). We present results obtained with distributed-memory parallelism. We also investigate numerically difficult matrices that need pivoting and allow simultaneously good compressions. Then, we compare our approach with the HSS partially-structured approach through a memory and operation complexity study (done against the Hsolver package). We also compare our code with the multifrontal HSS partially-structured solver StruMF and analyze the differences between the two techniques.

Except for Section 4.3 where the influence of the mesh size is analyzed, the experiments are run with `Poisson128` and `Helmholtz128`. Details about these matrices are given in Table 4.1. The ordering is `SCOTCH`, except when the influence of the ordering is studied. The `Poisson` and the `Helmholtz` problems are studied with a low-rank threshold of $\varepsilon = 10^{-14}$ (double precision real) and $\varepsilon = 10^{-8}$ (double precision complex), respectively.

Matrix	N	NZ	ops	memory peak	factor size
`Poisson128`	$2,097,152$	$8,339,456$	$2\ 7E+13$	7GB	19GB
`Helmholtz128`	$2,097,152$	$55,742,968$	$6\ 6E+13$	21GB	98GB

Table 4.1: Full-rank results obtained with MUMPS on matrices coming from the Poisson and the Helmholtz equations discretized on a 128 128 128 mesh.

Note that though the two problems have the same size in terms of the matrix dimension, the volume of computations required for `Helmholtz128` is much higher than for `Poisson128`. Moreover, `Helmholtz128` is a double precision complex problem so that each computation is performed at a higher cost.

To perform the compression efficiently, we modified `LAPACK`'s `*geqp3` to make this Householder QR routine a rank-revealing QR.

4.1 Metrics

The effectiveness of the BLR multifrontal solver will be measured with different metrics mostly related to the reduction of memory and operations. Relative and absolute metrics are used. The relative ones are always given as percentages of the equivalent FR quantity, computed with MUMPS. They are thus quantities that we want to be as low as possible. The metrics we will typically use to evaluate the efficiency of a BLR multifrontal solver are the following:

1. **Memory:** two memory compression metrics have to be distinguished:

   factor compression: written $|L|$, it is defined as the ratio of the number of entries needed to store the factor computed with the BLR approach over the number of entries needed to store the regular factor. We will sometimes refer to this metric simply as the "memory compression rate".

   peak of CB stack compression: written $|CB|$, it is the ratio of the maximum size of the CB stack with the BLR approach over the maximum size of the CB stack with a regular full rank approach. It will also be referred to as the *maximum size of CB stack compression.* For a definition of the *CB stack*, please refer to Section 1.3.3. Note that to obtain this compression, an external compress task (C_e, see Section 3.4) must be performed. As it is optional, and usually low due to efficient compression, it will never be indicated. This metric together with the factor compression give a good insight of the global memory reduction that can be obtained thanks to the BLR approach.

2. **Operations:** For each task (or tasks combination) defined in Section 3.4.2, we will indicate either the corresponding absolute low-rank **operation count**, or the corresponding **operation compression** as a percentage of the full-rank factorization (FR factorization) operation count.

   For instance, a column called "ops F+S" shows the operation count (in which case no unity is indicated) or the operation compression (in which case a % is always added) related to the factor and solve tasks of the algorithm. Note that even when a task has no equivalent in full-rank (e.g., the compression task C), it will be express anyway relatively to the full-rank factorization operation count, illustrating the overhead. Also note that when a U is indicated, it denotes both the internal and external updates.

   We will often use "operation compression rate" to indicate the reduction in operations obtained on the overall process (i.e., including all the tasks and overheads).

3. **Timings:** We will indicate in seconds the **time** spent in a task (or task combination) of the BLR algorithm defined defined in Section 3.4.2.

   For instance, a column called "time F+S+U" indicates the time spent in the factor, solve and update tasks of the algorithm. This allows us, for instance, to distinguish the compression time from the other more standard tasks of the process.

4.2 General results with the BLR multifrontal solver

The objective of this section is to establish whether the BLR multifrontal method demands a fine tuning of the parameters previously described (block size, switch level etc.). As the results presented in the following subsections show, this is not the case, which is clearly a very desirable property as it yields a twofold advantage:

1. it is possible to define default settings that perform well in most practical cases. This considerably improves the ease of use of the BLR multifrontal solver which can be used as a black-box tool by the end user.

2. it guarantees that all the algorithms proved efficient in the full-rank case (such as the pivoting scheme or the distribution among processors in a distributed-memory context, for instance), will remain efficient as the Block Low-Rank technologies will not add strong constraint on them.

We show in this section that the Block Low-Rank method has all the needed flexibility to fulfill the two latter goals.

4.2.1 Cluster size

The first parameter we want to study is the cluster size, which defines the size of the blocks which will be compressed within the frontal matrices. This parameter is very important as the efficiency of the BLAS 3 operations and the compression cost rely on it. This dependency was expressed in the Block Low-Rank admissibility condition in Section 3.3.1. Throughout this dissertation, the cluster size is always fixed for all the frontal matrices which are processed with BLR techniques. However, it is likely that choosing the cluster size depending on the separator size may lead to better gain but we have not tested it yet. Figures 4.1 and 4.2 show how the memory compression rates are influenced by the block size.

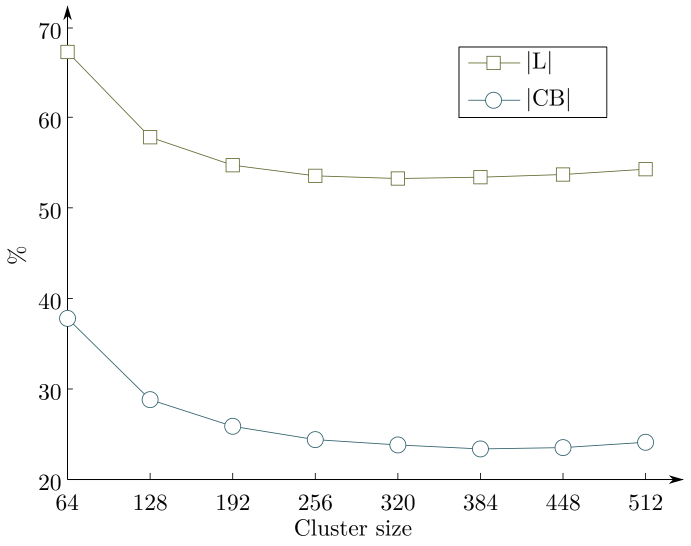


Figure 4.1: Influence of the cluster size on the memory compression rates obtained thanks to BLR approximations on the `Poisson128` problem. The low-rank threshold is set up to $\varepsilon = 10^{-14}$.

When the cluster size is very small (64 and 128), the compression rates are quite poor for both problems in terms of memory: blocks are too small to be compressed. Then, as the block size increases, the memory compression rates improve, but not significantly. Finally, when the block size becomes larger than 320 for `Poisson128` and 448 for `Helmholtz128`, it comes a point when the memory compression rates start to increase again.

Figures 4.3 and 4.4 show similar results for the operation compression rates and the speedups. The compression cost and the operation gain obtained for the factorization are plotted separately.

With smaller cluster sizes, compression costs are very low for both problems. However, the corresponding operation compression rates are quite high, which is consistent with the

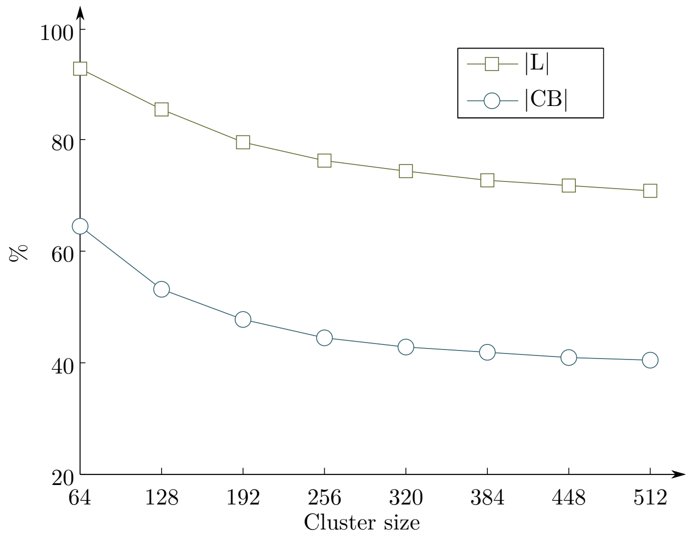


Figure 4.2: Influence of the cluster size on the memory compression rates obtained thanks to BLR approximations on the `Helmholtz128` problem. The low-rank threshold is set up to $\varepsilon = 10^{-8}$.

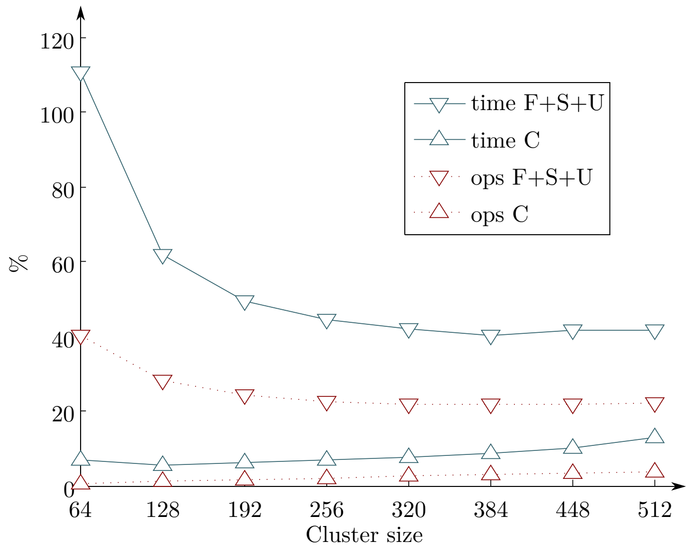


Figure 4.3: Influence of the cluster size on the operation compression rates obtained thanks to BLR approximations on the `Poisson128` problem. The execution times for compression and factorization are also indicated. The low-rank threshold is set up to $\varepsilon = 10^{-14}$.

poor memory compression observed with these cluster sizes. Simultaneously, and this is even more critical, the time reduction rates are even worse due to low BLAS 3 efficiency on such small blocks, which are further reduced because of the low-rank compression. When the cluster size increases, the compression costs increase too, but remain at a low level. The factorization compression rates decrease substantially until they become stable. As far as the timings are concerned, because the BLAS 3 efficiency increases with the cluster size, the difference between the operation compression rates and the time reduction rates decreases to an acceptable amount for `Poisson128` (15% for a cluster size of 320) and to almost

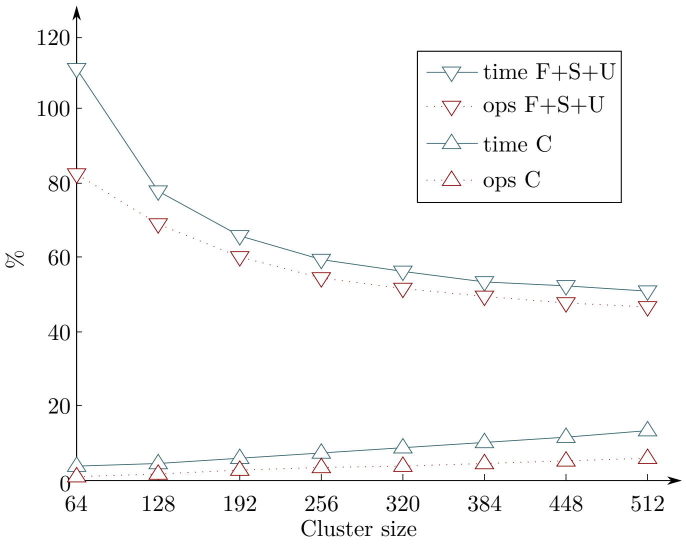


Figure 4.4: Influence of the cluster size on the operation compression rates obtained thanks to BLR approximations on the `Helmholtz128` problem. The execution times for compression and factorization are also indicated. The low-rank threshold is set up to $\varepsilon = 10^{-8}$.

zero for `Helmholtz128`. Finally, when the cluster sizes become too large, the compression costs become too high to be compensated by the gain obtained for the factorization, and the BLAS 3 efficiency does not improve anymore. Note that for `Helmholtz128`, larger cluster sizes have also be experimented without further improvements.

An important point to make is that, excluding smaller block sizes (64 and 128) the difference between the best block size and the worst one is roughly 10%, which means that the cluster size has a limited influence on the overall efficiency of the method, provided that the block size is not unreasonably small. For all the rest of the experiments in this section, the cluster size will thus be set up to 320 for `Poisson128` and 448 for `Helmholtz128`, consistently with the above analysis. The direct algorithmic consequence is that the pivoting scheme proposed in Figure 3.16 should not perturb too much the compression rates because of dynamic change of the block size, which is confirmed by results presented in Section 4.2.5.

4.2.2 Switch level

As explained in Section 3.6, not all the frontal matrices of the assembly tree are candidate for Block Low-Rank processing because excessively small frontal matrices do not provide enough room for gain as most of the memory and operations are consumed in the largest frontal matrices. Figures 4.5 and 4.6 show the influence of the minimum number of fully-summed variables (NASS) required for a frontal matrix to be selected for Block Low-Rank processing.

Selecting too many frontal matrices does not decrease the overall memory compression rates but does increase the total cost of compressing by all the operations performed in RRQR on blocks which at the end are not compressed (in this case, the compression cost is not compensated by operations performed on the compressed form). However, as can be observed in Figures 4.5 and 4.6, this overhead is highly negligible. This can be explained by the fact that when NASS becomes smaller than the block size, the fully-summed block of

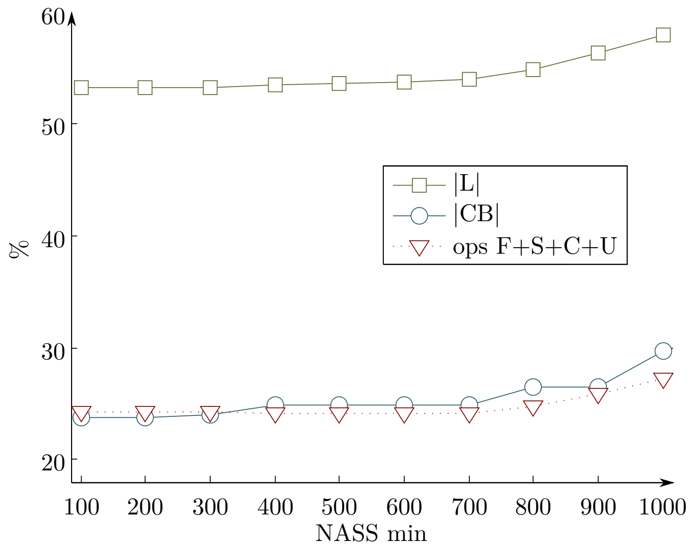


Figure 4.5: Influence of the minimal size of the separator (NASS min) required for a frontal matrix to be selected for BLR processing on the `Poisson128` problem. The low-rank threshold is set up to $\varepsilon = 10^{-14}$ and the cluster size is 320.

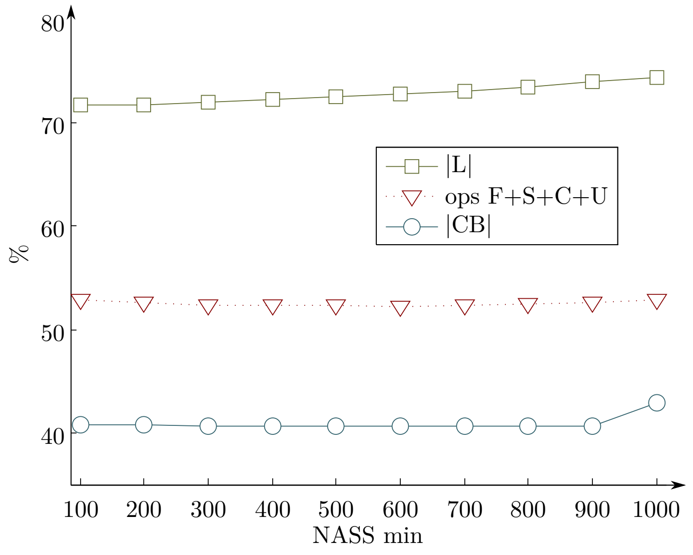


Figure 4.6: Influence of the minimal size of the separator (NASS min) required for a frontal matrix to be selected for BLR processing on the `Helmholtz128` problem. The low-rank threshold is set up to $\varepsilon = 10^{-8}$ and the cluster size is 448.

the frontal matrix (the $(1,1)$ block) is viewed as a diagonal block and thus no compression is attempted. Then, on small fronts, the number of non fully-summed variables can also be smaller than the block size so that the $(2,1)$ (and the $(1,2)$ block if unsymmetric) block is viewed as one single block to compress, which leads to small compression cost. As far as memory is concerned, selecting too many frontal matrices has on the other hand no influence on the global factor memory compression as only useless operations are added to the process.

Selecting too few frontal matrices globally degrades the performance, both in terms of memory and operations count, because more frontal matrices could have been successfully compressed.

Finally, the peak of |CB| stack remains constant as long as the nodes that are in the active memory at the moment when the peak is achieved are compressed with the BLR format.

Tables 4.2 and 4.3 give more details about results presented in Figures 4.5 and 4.6, in terms of number of selected fronts (# fronts), of the fraction of the global factor which is computed in these fronts ($|L\%|$; for instance, if all the fronts are selected, then $|L\%| = 100\%$ because 100% of the factors entries are computed in these selected fronts) and the fraction of the global operation count which is actually performed within them. Note that unlike $|L|$, $|L\%|$ is not a compression rate.

NASS min	# fronts	$\|L\%\|$	|FAC%|
100	1461	92.4	99.8
200	606	87.8	99.3
300	321	83.6	98.6
400	205	79.8	97.6
500	157	77.6	97.0
600	131	75.6	96.3
700	112	73.9	95.6
800	91	70.9	94.1
900	69	66.8	91.7
1000	49	62.4	88.9

Table 4.2: `Poisson128` problem, $\varepsilon = 10^{-14}$, cluster size= 320. $|L\%|$ is the fraction of the global factor which is actually computed within BLR frontal matrices. |FAC%| is the fraction of the global operation count which is actually performed within BLR frontal matrices.

NASS min	# fronts	$\|L\%\|$	|FAC%|
100	3108	91.3	99.7
200	1044	84.7	99.1
300	307	76.2	97.6
400	231	74.2	97.0
500	169	71.3	96.0
600	135	68.9	95.0
700	123	67.9	94.6
800	101	65.3	93.1
900	81	62.5	91.5
1000	66	59.8	89.7

Table 4.3: `Helmholtz128` problem, $\varepsilon = 10^{-8}$, cluster size= 448. $|L\%|$ is the fraction of the global factor which is actually computed within BLR frontal matrices. |FAC%| is the fraction of the global operation count which is actually performed within BLR frontal matrices.

The number of selected frontal matrices is in any case very small (`Poisson128` and `Helmholtz128` have 76403 and 47127 frontal matrices, respectively), which means that most of the fronts are still processed in the classical, full-rank way. Within this small amount of frontal matrices, almost all the operations of the multifrontal process are performed (from 88 9% to 99 8% for `Poisson128` and from 89 7% to 99 7% for `Helmholtz128`) and a very large fraction of the global factor is computed (from 62 4% to 92 4% for `Poisson128` and from 59 8% to 91 3% for `Helmholtz128`). These results confirm the theoretical study of Section 3.6 and justify the choice of compressing only the larger fronts. An illustration of this idea is given in Figure 4.7 where the assembly tree of the `Geoazur32` problem is represented. The tree results from applying to the input matrix a nested dissection ordering computed with `SCOTCH`. From blue to red (see the colormap in the picture), the factor compression (local to each frontal matrix) improves.

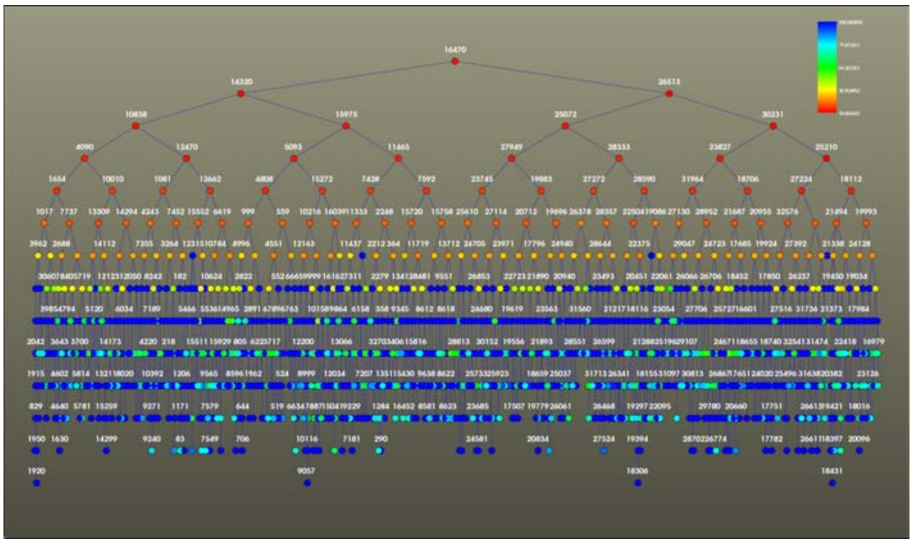


Figure 4.7: `SCOTCH` tree of `Geoazur32` problem, $\varepsilon = 10^{-4}$, double precision, obtained with MUMPS. Colormap: red = high compression.

The tree of Figure 4.7 illustrates well the fact that not much can be gained when compressing small frontal matrices as most of them turn out to be not low-rank. Same result holds for the operation counts. Note that the figure has been generated with a smaller problem and with a larger low-rank threshold for the sake of clarity and to better emphasize the behavior.

These results have shown that the critical aspect in setting up the minimal NASS size for a front to be candidate for Block Low-Rank processing is to ensure that enough fronts are selected. As selecting too many fronts has no important impact both in terms of memory and operation counts (because the compression attempts which failed are done on small frontal matrices, so that the computational cost is low), the efficiency of the solver can be easily ensured by setting up a fairly small minimal NASS. In the rest of this Chapter, this size will be 300 for both `Poisson128` and `Helmholtz128` problems.

4.2.3 Orderings

The global ordering of the matrix plays a critical role in multifrontal methods, as the amount of fill-in, the efficiency of the tree parallelism and the size of the frontal matrices (and thus the node parallelism) heavily rely on them [63, 71]. It is thus critical to have the flexibility to use several orderings during the preprocessing phase of the solver together with the BLR approach. Even though our description was often based on the nested dissection, we show that any of the common ordering algorithms is suitable for our purpose. Tables 4.4 and 4.5 present statistics about full-rank and low-rank runs with five different ordering methods, for `Poisson128` and `Helmholtz128` problems. Like for Sections 4.2.1 and 4.2.2, our aim is to show that good Block Low-Rank performance is compatible with a wide range of matrix orderings, in order to guarantee that the most efficient reordering strategy can be kept together with the Block Low-Rank feature. The orderings experimented in this section have been previously mentioned in Section 1.6.2. Note that `METIS` and `SCOTCH` are the only nested dissection based orderings used in this section.

ordering	FR			top separator			global		
	mry	ops	peak	size	|L|	ops F+S+C+U	|L|	ops F+S+C+U	|CB|
`AMD`	36GB	$1\,1E+14$	23GB	43669	22 0%	9 6%	35 3%	13 2%	16 8%
`AMF`	27GB	$6\,4E+13$	15GB	17405	35 6%	17 4%	40 3%	15 8%	22 6%
`PORD`	15GB	$1\,8E+13$	5GB	7	100 0%	100 0%	55 6%	24 3%	37 5%
`METIS`	18GB	$2\,6E+13$	7GB	16397	32 3%	16 0%	52 6%	24 6%	65 5%
`SCOTCH`	19GB	$2\,7E+13$	8GB	16384	32 8%	16 3%	53 3%	24 2%	24 0%

Table 4.4: Low-rank compressions for the top separator and the global process with different orderings on the `Poisson128` problem with $\varepsilon = 10^{-14}$. Top level separator of `PORD` is too small to be selected for low-rank. “FR peak” is the peak of active memory (excluding factors).

ordering	FR			top separator			global		
	mry	ops	peak	size	|L|	ops F+S+C+U	|L|	ops F+S+C+U	|CB|
`AMD`	218GB	$3\,9E+14$	92GB	53895	54 8%	42 2%	60 3%	42 8%	34 5%
`AMF`	162GB	$2\,8E+14$	80GB	23788	50 3%	37 6%	61 8%	42 0%	47 5%
`PORD`	110GB	$1\,0E+14$	28GB	6146	84 6%	97 3%	63 9%	40 0%	43 0%
`METIS`	98GB	$6\,5E+13$	22GB	17731	41 9%	31 5%	72 3%	53 0%	40 9%
`SCOTCH`	98GB	$6\,6E+13$	21GB	17949	41 8%	31 0%	72 0%	52 3%	40 7%

Table 4.5: Low-rank compressions for the top separator and the global process with different orderings on the `Geoazur128` problem with $\varepsilon = 10^{-8}$. “FR peak” is the peak of active memory (excluding factors).

Unsurprisingly, `METIS` and `SCOTCH` perform the best in full-rank on these types or problem. Because these tools are based on the nested dissection method the computed topmost separators are also very well compressed because they are large, smooth and very suitable for efficient admissible clustering, as Figure 4.9 illustrates. This is not the case of the top level separator from `AMD` presented in Figure 4.8 since it highly irregular.

However, one can observe that the overall compression rates (both in operations and memory) is worse for `SCOTCH` and `METIS` than for any other ordering strategy. However, because the corresponding reductions are applied to a lower original absolute memory consumption and operation count (because `SCOTCH` and `METIS` perform the best in full-rank), the overall performance remains the best for these two methods. Given that separators in these methods are close to minimal in size at each level of the nested dissection tree, the low-rank compression is simultaneously very efficient at the top of tree and poorly

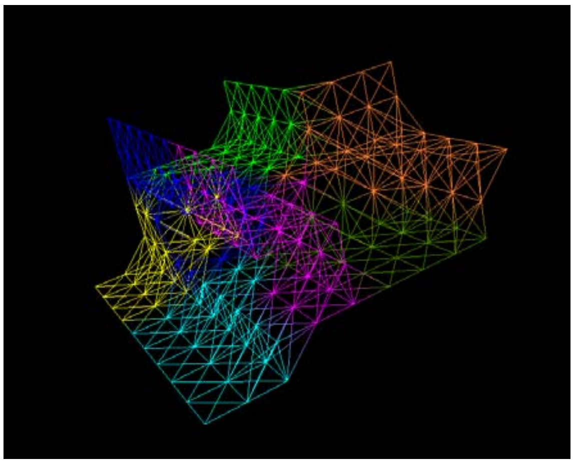
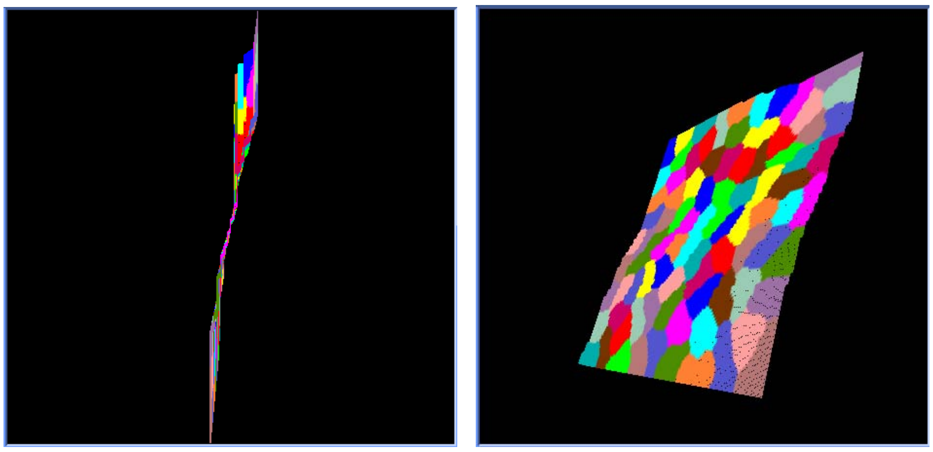


Figure 4.8: AMD top level separator associated with Helmholtz10 matrix.



(a) Side face (b) Perspective

Figure 4.9: SCOTCH top level separator

efficient below a still fairly high level where separators are already becoming small (see again Figure 4.7). For instance, it is not the case of AMF where large frontal matrices appear in lower levels of the tree, as illustrated in Figure 4.10.

Note that the minimal NASS to select a frontal matrix for BLR as well as the block size is the same for all 5 ordering methods. Because we have shown that the compression rates are stable for these two parameters, we decided to keep them for all other 4 heuristics, which eases the comparison. Also note that, even if the compression rates are slightly worse for METIS and SCOTCH, the overall operation count remains much lower than with the other ordering methods thanks to a much better reduction of the fill-in. SCOTCH will thus be used for the other experiments in this section.

4.2.4 Low-rank threshold

The low-rank threshold has an important role to play in Block Low-Rank multifrontal solvers, as it controls both the efficiency and the accuracy of the solution. Depending on the underlying equation, one will have to set up a value that is suited to the wanted

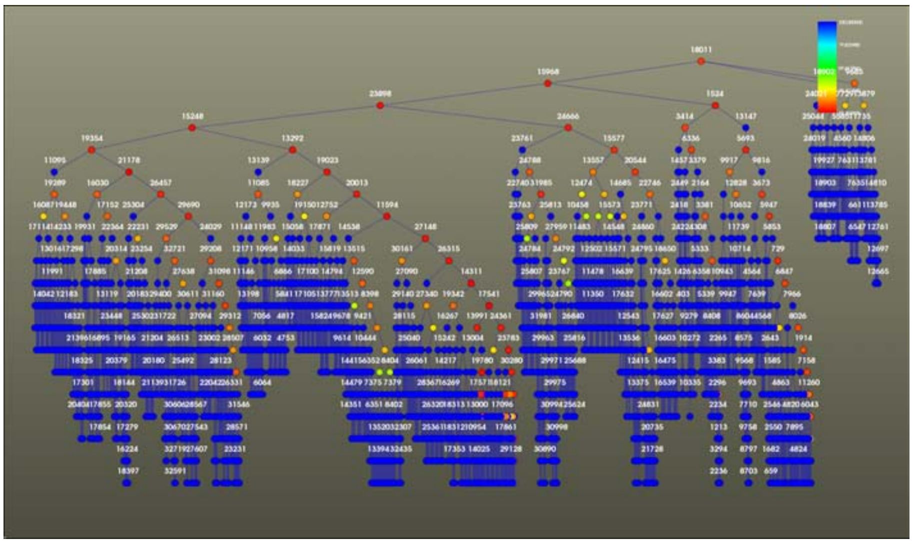


Figure 4.10: `AMF` tree of `Geoazur32` problem, $\varepsilon = 10^{-4}$, double precision, obtained with MUMPS. Colormap: red = high compression.

compression and accuracy. Figures 4.11 and 4.12 show how the memory compression rates improve when the low-rank threshold increases.

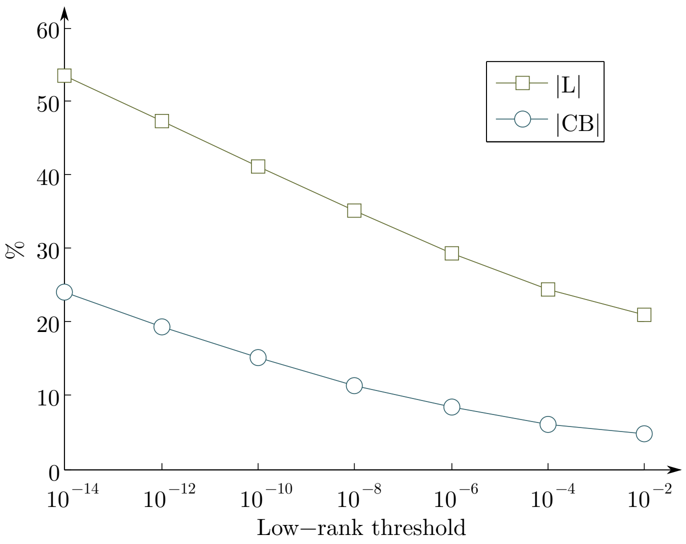


Figure 4.11: Influence of the low-rank threshold ε on the memory compression rates on the `Poisson128` problem. The cluster size is set up to 320 and the minimal NASS to select a frontal matrix for BLR is 300.

Unsurprisingly, large values of ε lead to high memory compression rates for both problems. The main difference between the two behaviors is that for the `Poisson128` problem, the memory compression rates are already good at full accuracy, while for the `Helmholtz128` problem, the low-rank threshold has to be slightly larger to obtain interesting memory compression rates. Note that the contribution blocks are better compressed than the factor, as they usually correspond to farther interactions and larger blocks. Fig-

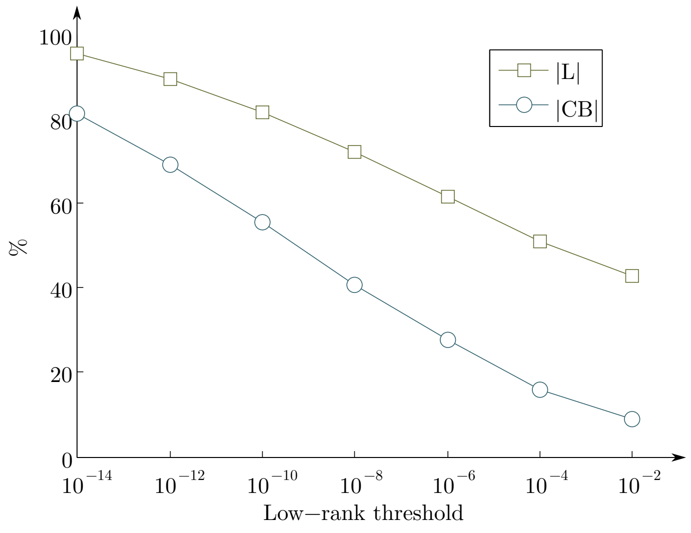


Figure 4.12: Influence of the low-rank threshold ε on the memory compression rates on the `Helmholtz128` problem. The cluster size is set up to 448 and the minimal NASS to select a frontal matrix for BLR is 300.

ures 4.13 and 4.14 present the corresponding operations and time reduction rates.

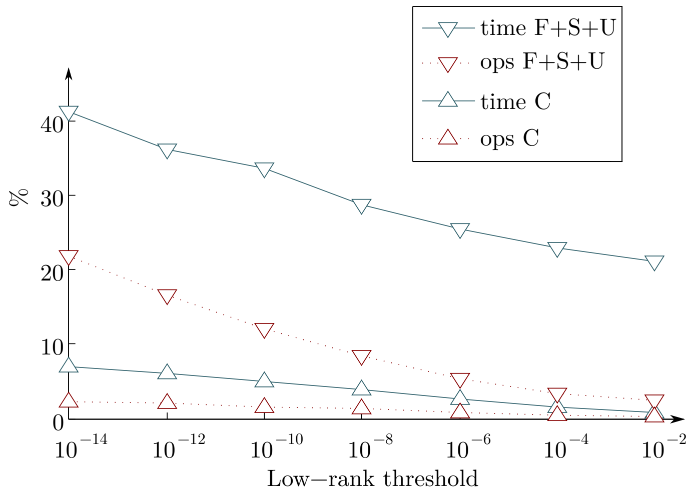


Figure 4.13: Influence of the low-rank threshold ε on the operations and time reduction rates on the `Poisson128` problem. The cluster size is set up to 320 and the minimal NASS to select a frontal matrix for BLR is 300.

The efficiency of the factorization increases when the low-rank threshold increases. Consistently with the memory compression rates, the `Helmholtz128` problem requires a larger low-rank threshold in order to obtain good efficiency. The computational speed is good: for `Poisson128`, the time reduction is 20% more than the operation compression, which is acceptable. This overhead corresponds to the time spent outside the BLR fronts, which is by definition not compressed and thus remains constant. For `Helmholtz128`, the time reduction is almost the same as the operation compression. Because of the very large

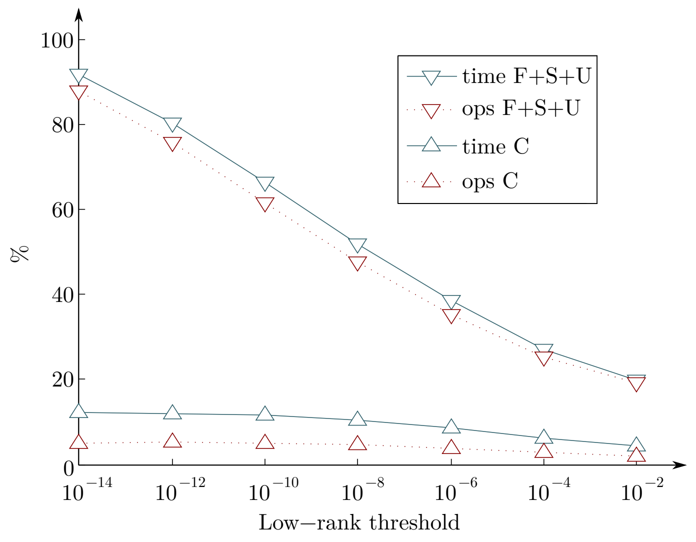


Figure 4.14: Influence of the low-rank threshold ε on the operations and time reduction rates on the `Helmholtz128` problem. The cluster size is set up to 448 and the minimal NASS to select a frontal matrix for BLR is 300.

computational volume of the `Helmholtz128` problem, the time spent outside BLR fronts is here negligible. Note that to obtain the BLR overall time, the two solid curves and the two dashed curves should be added, which means that the total time reduction rate is always higher (only slightly higher for `Helmholtz128`) that the total operation compression rate.

These compression rates have to be related to the accuracy obtained for the solution as too bad accuracy is usually not workable. As explained in Section 2.4, the threshold we use is absolute as matrices are scaled. Figures 4.15 and 4.16 show how the Scaled Residual (SR) and the Componentwise Scaled Residual (CSR) introduced in Section 1.2 behave with respect to the low-rank threshold ε. These residuals are also reported with Iterative Refinement (IR, see Section 1.6.2).

Without iterative refinement, the residuals are close to the order of magnitude of the low-rank threshold, which means no error propagation is observed. Moreover, to recover full-accuracy, iterative refinement is almost always very efficient (it does not work only for very large value of ε i.e., beyond 10^{-4} for `Poisson128` and 10^{-2} for `Helmholtz128`) with a small number of steps. Table 4.6 reports the number of steps of iterative refinement which were performed for each case.

ε	10^{-14}	10^{-12}	10^{-10}	10^{-8}	10^{-6}	10^{-4}	10^{-2}
`Poisson128`	1	1	1	2	4	5	5
`Helmholtz128`	1	1	1	2	3	6	10

Table 4.6: Number of iterative refinement steps performed to obtain the accuracy reported in Figures 4.15 and 4.16.

4.2.5 Pivoting

In this section, we focus on the pivoting within Block-Low Rank factorizations and show how it can be critical in order to obtain good low-rank compressions on numerically difficult

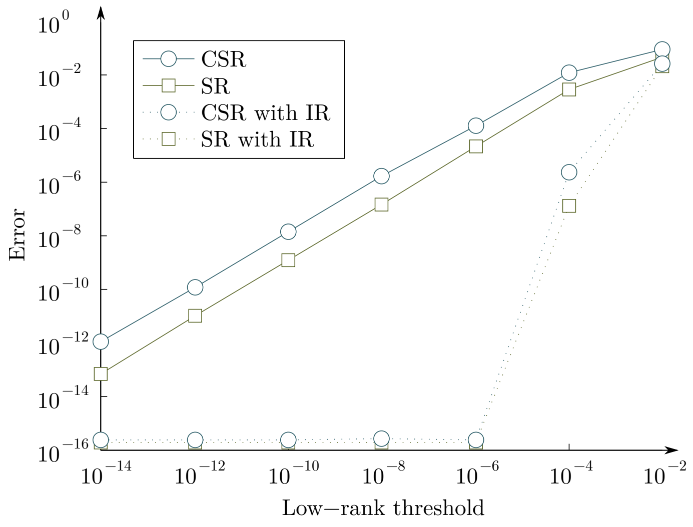


Figure 4.15: Influence of the low-rank threshold ε on the accuracy of the solution on the `Poisson128` problem. The cluster size is set up to 320 and the minimal NASS to select a frontal matrix for BLR is 300. Both Scaled Residual (SR) and Componentwise Scaled Residual (CSR) are reported, with and without Iterative Refinement (IR).

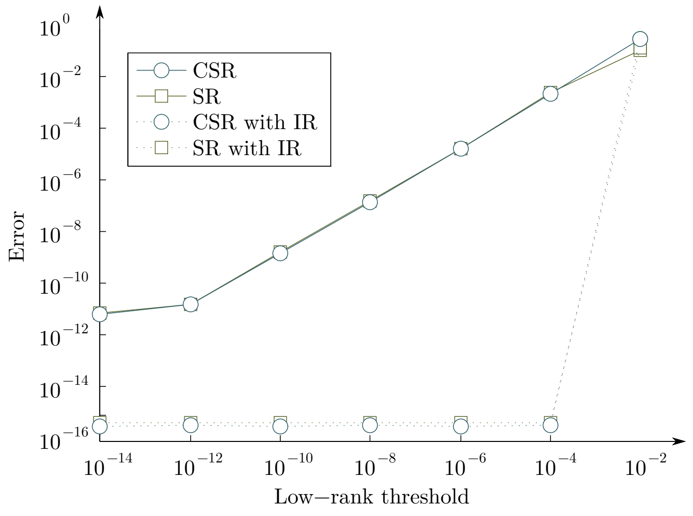


Figure 4.16: Influence of the low-rank threshold ε on the accuracy of the solution on the `Helmholtz128` problem. The cluster size is set up to 448 and the minimal NASS to select a frontal matrix for BLR is 300. Both Scaled Residual (SR) and Componentwise Scaled Residual (CSR) are reported, with and without Iterative Refinement (IR).

problems. Because `Helmholtz128` and `Poisson128` do not require any pivoting, we use for this section three different matrices `cont-300`, `kkt_power` and `d-plan-inco`, presented in Section 1.6.4.2. Note that the two first matrices come from the University of Florida Sparse Matrix Collection. These matrices have 68,243 (38% of matrix order), 56,646 (3%) and 67,890 (6%) delayed pivots, respectively. Even if the percentages are low the last two matrices, it still is a fairly high amount of delayed pivots. Table 4.7 summarizes the results obtained with these matrices. We test our strategy with threshold partial pivoting and

	`cont-300`	`kkt_power`	`d-plan-inco`
MUMPS FR partial	5.68E-8	3.05E-10	1.90E-12
MUMPS FR static	5.71E-3	4.66E-4	8.88E-7
BLR (10^{-14}) static	5.91E-3	7.90E-5	1.04E-6
Compressions ($\|L\|$ - $\|CB\|$ - f)	98 - 90 - 116	75 - 62 - 64	46 - 6 - 8
BLR (10^{-14}) partial	2.06E-8	2.65E-9	2.37E-12
Compressions ($\|L\|$ - $\|CB\|$ - f)	92 - 72 - 82	75 - 61 - 62	45 - 5 - 7
BLR (ε) partial	1.59E-3(10^{-9})	9.77E-4(10^{-8})	6.28E-7(10^{-7})
Compressions ($\|L\|$ - $\|CB\|$ - f)	86 - 54 - 67	55 - 46 - 30	42 - 5 - 5
Time MUMPS FR partial	4 s.	4043 s.	263 s.
Time BLR partial (ε)	2 s. (10^{-9})	557 s. (10^{-8})	71 s. (10^{-7})

Table 4.7: Results about partial and static pivoting with BLR and FR factorizations. The residuals reported are Componentwise Scaled Residuals. The low-rank threshold is either indicated in the first column (when identical for all other columns), either indicated directly in the corresponding columns. f is the operations compression rate for F+S+U+C.

without, in which case static pivoting is activated and explicitly indicated in the table. We show that partial pivoting allows for more compression and that a high amount of pivoting does not degrade the compression rates even if the BLR panels are dynamically changed (see Section 3.4.4). Note that the amount of delayed pivots remains stable for all these runs.

Table 4.7 consists of five blockrows of two rows, besides the header. The first one indicates the Componentwise Scaled Residual (CSR = $\max_i \frac{Ax-b_{\ i}}{(\ A\ \ x+b\ )_i}$, see Section 1.2.1) in full-rank for both static and threshold partial pivoting. It shows that partial pivoting substantially increases the stability of the factorization on these matrices. The second blockrow indicates the CSR and the compression rates obtained with a BLR factorization with static pivoting. The low-rank threshold ε is chosen so that the CSR is as close as possible to the CSR obtained in full-rank with static pivoting. The third blockrow reports the CSR and the compression rates obtained with a BLR factorization with partial pivoting. The low-rank threshold ε is chosen so the CSR is as close as possible to the CSR obtained in full-rank with partial pivoting. The fourth blockrow follows the same idea, except that the low-rank threshold ε is chosen so that the CSR is as close as possible to the CSR obtained in full-rank with static pivoting. Finally, the last blockrow reports the total execution time corresponding to the fourth blockrow.

Let us compare first the second and third blockrow. The compression rates are similar, which shows that the perturbation induced by delayed pivot has no influence on the global efficiency. Moreover, because partial pivoting does not artificially change values during the factorization (whereas static pivoting does), the compression rates are even sometimes better (see problem `cont-300`). Note if some delayed pivots are delayed to the same front, and if there are enough of them, they are considered as an original BLR panel so the compression can be efficient. All in all, this shows that for an equivalent (or better) compression (both in memory and operations), the accuracy of the solution is much better thanks to partial pivoting, even with the BLR factorization. The fact that the accuracy is better with BLR and partial pivoting than in BLR and static pivoting is not quite surprising, as the BLR factorization benefits from the same numerical stability as for the full-rank case. But, this is an interesting result only because the compression rates are maintained or improved, since this makes a substantial difference in terms of efficiency.

Now compare blockrows two and four. The CSR is fixed as it is chosen to be as close as possible to the CSR obtained with a full-rank factorization with static pivoting. To obtain this accuracy, a small low-rank threshold is needed if the BLR factorization is done with static pivoting ($\varepsilon = 10^{-14}$). However, in the context of a BLR factorization with partial pivoting, the low-rank threshold can be chosen much larger (from 10^{-9} to 10^{-7} in our example) which improves the compression rates: it requires almost twice fewer operations to perform the BLR factorization with partial pivoting than the one with static pivoting, for the same accuracy. The memory footprint is also slightly improved. Note that these improvements are worse for problem `d-plan-inco`.

Finally, the corresponding timings show that the operation efficiency remains very good for all of these three matrices. Quite surprisingly, for problem `kkt_power`, the time reduction (14%, not indicated in the Table) is higher than the operation compression rate (30%). This may be due to the fact that in the full-rank case, there are more outer panels (called BLR panels when BLR is applied) because they are smaller than in the BLR case. This means that the delayed pivots are potentially tested more times in full-rank than in BLR (because then are tested once at each panel), which slow down the factorization.

These results show the numerical robustness of our method, and show that on some classes of problems, pivoting may be critical either to keep a satisfying accuracy, either to obtain acceptable low-rank compressions. Thanks to the flexibility of the BLR format, both objectives can be achieved.

4.2.6 Distributed-memory parallelism

We show that the distribution scheme presented in Section 3.4.5 allows for good efficiency and maintains the compressions rates when the number of processors increases. We present in Table 4.8 the factor compression rates obtained for `Poisson128` and `Helmholtz128` with different numbers of processors.

#	`Poisson128`	`Helmholtz128`
procs	|L|	
1	53.3%	72.0%
4	54.5%	n/a
8	54.5%	n/a
16	54.6%	73.6%
32	54.9%	73.6%
64	55.3%	73.7%
128	56.0%	74.1%

Table 4.8: Factor compression rates in parallel for `Poisson128` and `Helmholtz128` problems. Low processor numbers do not provide enough memory to run the problems.

A monotonic deterioration of the factor compression rates can be observed. Because it is very low (even with 128 processors, which is quite high for this size of problems), we are willing to pay this price to obtain a better parallel efficiency. This shows that the strategy of dynamically adapting some of the BLR blocks (see Figure 3.18) is a good strategy in terms of factor compression. Operation compression and timings results are given in Table 4.9 or the `Helmholtz128` problem only, because it is unsymmetric. In the symmetric case, the slave to slave communications (see Figure 1.24(b) in the full-rank case) must be done in BLR (in order to avoid compressing blocks multiple times locally on each slave) to completely update the contribution blocks using low-rank products. Although there is no theoretical or structural obstacle to do it, this has not been implemented yet, so that the operation compressions rates and the timings are degraded when increasing the number of processors on symmetric matrices. For this reason, we do not present the results for `Poisson128` in Table 4.9. Also, the peak of CB stack compression rate is never indicated as it is difficult and expensive (because it requires communications) to compute in parallel. Moreover, the factor compression rates and the unsymmetric operation compression rates are sufficient to justify that constraining the BLR panels to the distribution among processors is a strategy that pays off.

#	ops	execution time	
procs	F+S+U+C	BLR	MUMPS
16	56.4%	3133s.	4905s.
32	56.5%	2125s.	2648s.
64	56.7%	1409s.	1537s.
128	57.3%	1014s.	979s.

Table 4.9: Operation compression and timings in parallel for `Helmholtz128`.

First notice that the scalability for the full-rank factorization is good. Results are less good for the BLR factorization. This can be explained by the fact that because the amount of computations is lower in the BLR case (half of the full-rank number of operations), the number of processors used for the same problem size should be lower in order to maintain a good work load per processor. Moreover, because the compression rates are experimentally usually higher in the $(2, 1)$ blocks, which are distributed among the slaves, the workload of the master becomes relatively higher compared to the full-rank case, meaning than adding more slaves does not improve the execution times because the computations performed by the master are dominant. Without any adaptation to the dynamic scheduler to the BLR factorization, these results show a very good potential and we are confident that they can be considerably improved. Work in this direction is in progress.

4.3 Comparison with HSS techniques in a multifrontal context

We focus on the experimental complexities which can be achieved using both HSS and BLR representations but will also investigate the accuracy of the solution which can be obtained using HSS-embedded and BLR solvers. We compare our approach with the HSS partially-structured approach implemented in Hsolver in Sections 4.3.1 and 4.3.2, by exploiting complexity results from Wang et al. [102]. Note that in these sections, the low-rank threshold used for the BLR experiments provides the same accuracy as in the HSS case, so that results can be compared. Also the low-rank threshold is constant for any size of mesh. In Section 4.3.3, we will analyse the compression rates which can be obtained with respect to the accuracy of the solution, using new experiments performed with StruMF.

This work has been done in collaboration with the Lawrence Berkeley National Laboratory (Xiaoye Sherry Li, Artem Napov and François-Henry Rouet).

The efficiency of a low-rank format can be assessed by taking into consideration two types of metrics:

1. the memory and operation complexity, which can be written as αn .
2. the flop rate which can be achieved s.

4.3.1 Experimental multifrontal memory complexity

For the 3D `Helmholtz` equations, we could derive experimental memory (i.e., the memory to store the factors) complexities from Wang et al. [102] (unfortunately, no results on `Poisson` are presented in this paper). Results are summarized in Table 4.10. The corresponding theoretical complexities are given in Table 1.3.

Problem	Hsolver
3D `HelmholtzN`	$O(n^{1\ 1})$

Table 4.10: Experimental memory complexities obtained with the partially-structured HSS multifrontal solver Hsolver. n is the size of the matrix.

The memory experimental complexities obtained with Hsolver are very consistent with the theory and are satisfying.

In order to obtain similar results, we performed the BLR factorization of `PoissonN` and `HelmholtzN` for different sizes of mesh N and derived the experimental memory complexity of the multifrontal BLR method. The results are presented in Figures 4.17 and 4.18, for mesh sizes ranging from 32 to 256 (224 for `HelmholtzN`).

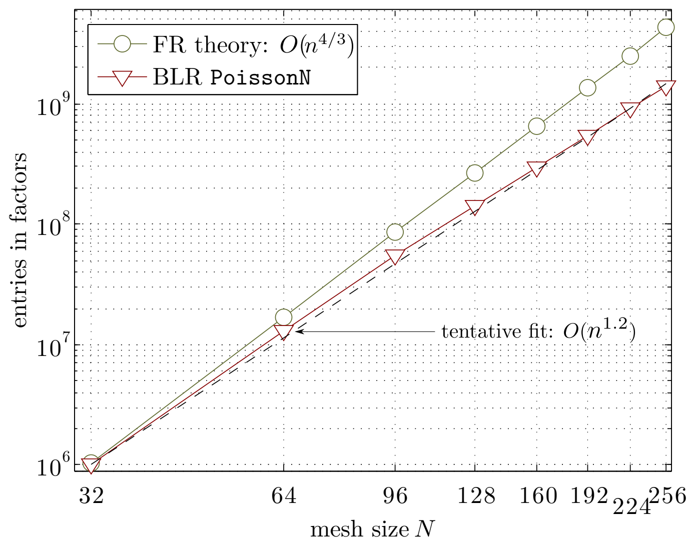


Figure 4.17: Experimental memory complexity for the multifrontal BLR factorization of matrix `PoissonN` on a $N \quad N \quad N$ mesh. $n = N^3$.

This shows that the experimental memory complexity of the BLR multifrontal method is slightly higher than for the HSS case ($O(n^{1\ 2})$ versus $O(n^{1\ 1})$) but are still satisfying for both problems.

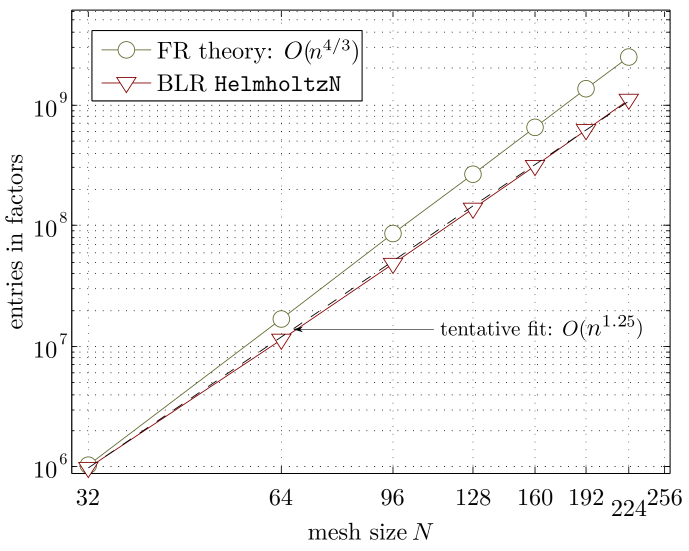


Figure 4.18: Experimental memory complexity for the multifrontal BLR factorization of matrix `HelmholtzN` on a $N \quad N \quad N$ mesh. $n = N^3$.

These latter results correspond to the   values defined in the introduction of Section 4.3. Table 4.11 summarize the results for both BLR and HSS formats.

Problem	Hsolver	MUMPS BLR
`Helmholtz`	2800 $n^{1\ 1}$	375 $n^{1\ 25}$

Table 4.11: Summary of the memory experimental complexities obtained with the partially-structured HSS multifrontal solver Hsolver and the BLR solver based on MUMPS. n is the size of the matrix. The full-rank multifrontal operation complexity is 180 $n^{4\ 3}$.

Interestingly, HSS has a better complexity but its constant is much higher. This means that BLR will be better than HSS on relatively small problems. Using these results, we can prove that the cutoff matrix size is $n = 700,000$ (which corresponds to a cubic mesh size of $N = 88$), after which Hsolver requires less memory to store the factors. Similarly, the BLR multifrontal method will perform better (in terms of memory for the factors) than the full-rank multifrontal method if $n > 6859$ (i.e., if $N > 19$).

4.3.2 Experimental multifrontal operation complexity

To study the multifrontal operation complexity, we will first investigate the value of   , then the value of α and finally the speed s.

As for the experimental memory complexities, we used results presented in Wang et al. [102], on the same problems. They are summarized in Table 4.12. The corresponding theoretical complexities were given in Table 1.3.

Unlike the experimental memory complexities, the experimental operations complexities are worse than the theoretical ones, which is mainly due to the fact that the experimental code is a partially-structured approach while the theory [103] focuses on a fully-structured approach. However, the complexity which can be achieved is still satisfying.

Problem	Hsolver (ops)
3D `HelmholtzN`	$O(n^{1\ 6})$

Table 4.12: Experimental operation complexities obtained with the partially-structured HSS multifrontal solver Hsolver. *ops* stands for number of operations for the factorization. n is the size of the matrix.

As far as the BLR factorization is concerned, we did the same experiments as for the experimental memory complexity of Section 4.3.1. The results are presented in Figures 4.19 and 4.20, for mesh sizes ranging from 32 to 256 (224 for `HelmholtzN`).

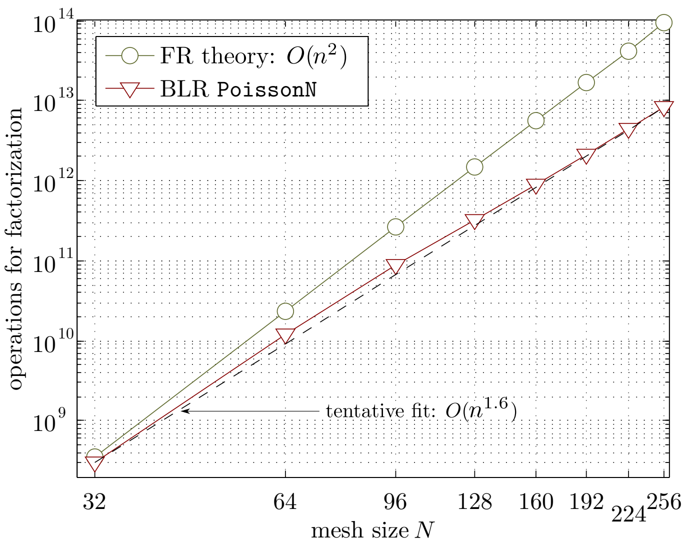


Figure 4.19: Experimental operation complexity for the multifrontal BLR factorization of matrix `PoissonN` on a N N N mesh. $n = N^3$.

These results show that BLR has a good experimental complexity for both problems. For `HelmholtzN`, it is slightly higher than the one achieved through HSS partially-structured factorizations. The original full-rank multifrontal solver has thus been well improved. Note that for the largest mesh sizes, the block size could be increased in order to obtain better compressions, although it was not done in these experiments to ease the comparison.

These results give the   value and in order to better compare the two methods, we have experimentally computed the constants (α), which yields more accurate complexities. They are summarized in Table 4.13 where both   and α are indicated for both BLR and HSS factorizations.

Problem	Hsolver	MUMPS BLR
`Helmholtz`	2350 $n^{1\ 6}$	150 $n^{1\ 77}$

Table 4.13: Summary of the operation experimental complexities obtained with the partially-structured HSS multifrontal solver Hsolver and the BLR solver based on MUMPS. n is the size of the matrix. The full-rank multifrontal operation complexity is 55 n^2.

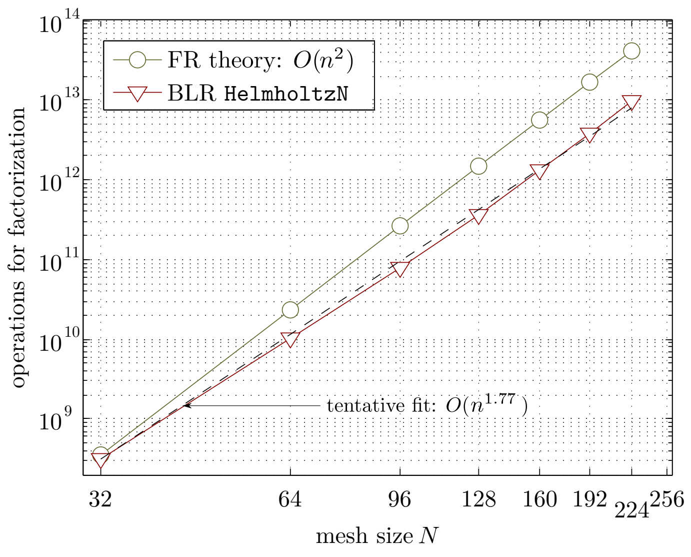


Figure 4.20: Experimental operation complexity for the multifrontal BLR factorization of matrix `HelmholtzN` on a N N N mesh. $n = N^3$.

The same behavior as for the memory case can be observed: HSS has a better complexity but its constant is much higher. Again, this means that BLR will be better than HSS (in terms of operations) on relatively small problems. Using these results, we can easily find this cutoff matrix size which is $n = 11,000,000$ (which corresponds to a cubic mesh size of $N = 220$), after which Hsolver requires less operations to factorize the matrix.

These operation results have to be correlated with the speed that is achieved during the factorization. Based on experiments performed with Hsolver (and also visible with StruMF) on `HelmholtzN`, we obtained that HSS achieves 35% of the full-rank speed, while BLR achieves 80% of it. If we integrate the flop rate in our reasoning, it changes substantially the cutoff after which Hsolver is faster than BLR MUMPS, which becomes $n = 1,700,000,000$, corresponding to a mesh size of $N = 1200$, which makes a big difference. Similarly, the BLR multifrontal method will perform better (in terms of memory for the factors) than the full-rank multifrontal method if $n > 64$ (i.e., if $N > 4$).

The fact that HSS is slower than the full-rank equivalent might be due to the fact that, by construction, the partially-structured HSS approach handles small blocks in order to represent the matrices (the generators) which slows down the operation efficiency of the compressions and the factorization. It is less the case of BLR, which handles relatively large blocks that all have a comparable size.

4.3.3 Accuracy and compression

To have a fair and complete comparison, the accuracy of the solution should also be taken into account. We experimented with the BLR solver and StruMF on a 3D `Helmholtz96` problem only with different low-rank thresholds ε. Results are presented in Table 4.14.

At high accuracy, StruMF was too slow so that results could not be obtained in an acceptable time (with $\varepsilon = 1e-6$, the HSS-embedded factorization in StruMF is still three time slower than the full-rank factorization). However, for these accuracies, BLR is able to compress very well, which means that both accuracy and compression can be simultaneously targeted. For the HSS solver, when the factorization is very relaxed ($\varepsilon = 1$ 1 for instance), the compression rates become close to the ones of BLR but then the solution is

ε	$1e-14$		$1e-10$		$1e-6$		$1e-2$		1 1	
BLR (|L| ops)	63%	35%	51%	24%	37%	15%	28%	9%	25%	9%
HSS (|L| ops)	/	/	/	/	66%	85%	45%	28%	35%	9%

Table 4.14: Low-rank threshold with respect to the factor compression rate (|L|) and the operation compression rate (ops=ops F+S+C+U) obtained with BLR MUMPS and StruMF on a 3D `Helmholtz96` problem.

not acceptable in the context of a direct solver. However, it can be used as a preconditioner.

For this kind of model problem on regular meshes, low-rank technologies are appealing but the full-rank multifrontal method should be preferred for small grid sizes. For medium-size grids, BLR seems to be a better candidate than HSS (in terms of operation count and performance) due the smaller pre-factor in the complexity and the better performance of the underlying dense kernels. However, for very large problems (10M+ degrees of freedom), the better asymptotic behavior of HSS might make it a more suitable approach. Also, in memory-constrained environments, the HSS approach should be used in priority even for medium-size grids since it seems to provide a better compression of the LU factors. We conclude that there is no clear winner even for these kinds of simple model problems; furthermore, the behavior of these different low-rank techniques remains to be investigated for non-PDE problems or PDE problems on irregular unstructured meshes, but this preliminary study, along with the advantages listed in Chapter 2, demonstrate that BLR techniques are worth investigating and can be of interest for many applications. We plan to extend this study to different classes of problems, usages of the solver (as a preconditioner, as a direct solver...), and larger applications by using distributed-memory implementations of each technique, and new low-rank compression algorithms. In particular, recently investigated fully-structured HSS techniques relying on randomized sampling [104] seem to be a promising direction for improving HSS-based sparse solvers. This is the object of an on-going collaboration with Xiaoye S. Li., Artem Napov and François-Henry Rouet.

Chapter 5

Application to real-life industrial problems

In the previous chapter, we evaluated the potential of our method on two widely used operators. We have shown the flexibility and the efficiency of our solver. In this chapter, we present experiments on the BLR multifrontal solver used in various applicative contexts. We interfaced our code with EDF's finite element industrial code Code_Aster (see 1.6.3) and used it as a direct solver and as a preconditioner for the conjugate gradient. We show how double precision Block Low-Rank with a low-rank threshold $\varepsilon = 10^{-8}$ can be a good alternative to full-rank single precision in order to efficiently precondition hard problems. Then, we present a geophysics study in a 3D seismic modeling context for front-wave inversion (in collaboration with the Seiscope project). Finally, we show some results on a set of matrices available in the University of Florida Sparse Matrix Collection and coming from various applications, aiming at widening the scope of our study. All these problems have been introduced in Section 1.6.4.

5.1 Structural mechanics in EDF Code_Aster

We experiment our approach in the context of EDF structural mechanics package Code_Aster. Our BLR solver has been coupled to Code_Aster so that results could be obtained for different reference test-cases, corresponding to large and archetypal applications. In a first section, we study the BLR multifrontal solver on three matrices presented in Section 1.6.4.2. In a second section, we analyse how loosely approximated factors can be used to precondition the conjugate gradient.

5.1.1 Direct solver

Three matrices coming from different application have been experimented in a direct solver context. `pompe` is quite small while the two others, `amer12` and `dthr7`, are very large (from 8 to 134 millions of unknowns each, see Section 1.6.4.2). These experiments have been performed in sequential as distributed-memory results have already been presented in Chapter 4. We first present in Table 5.1 results about the overhead due to clustering for all three problems.

The extra-time spent in the clustering appears to be very low compared to the original analysis time, which means that the price to pay to build BLR frontal matrices will be negligible compared to the factorization, which often dominates the analysis.

matrix	FR analysis time	clustering time	total BLR analysis time
`pompe`	22 s.	8 s.	30 s.
`amer12`	2650 s.	140 s.	2790 s.
`dthr7`	517 s.	15 s.	532 s.

Table 5.1: Timings related to the clustering phase. The original full-rank analysis time is indicated together with the time overhead due to clustering. The total BLR analysis time is thus the sum of the two first timings.

The results related to the factorization of the three problems are given in Table 5.2. Many metrics are proposed: the factor compression |L|, the peak of CB stack compression |CB|, the operation compression for the overall factorization (F+S+C+U) together with the corresponding elapsed time, the componentwise scaled residual (CSR) and the scaled residual (SR), without and with iterative refinement (IR), in which case the extra-time is indicated (time IR). For metrics given as percentages, the corresponding full-rank (FR) quantity is also given, so that one can understand the savings obtained in terms of gigabytes or operations. The BLR solver is experimented with low-rank thresholds ε ranging from 10^{-14} to 10^{-2}.

The `pompe` matrix being small, the improvements which can be achieved due to low-rank approximations are low, since both the volume of computations and the memory footprint are already small. With $\varepsilon = 10^{-14}$, the factorization takes 10 seconds less time than in full-rank, which represents more than 10% of time savings. With $\varepsilon = 10^{-8}$, the time saved goes up to 20 seconds and a quarter of the factor memory has also been saved, which becomes more interesting. The scaled residual remains good while the componentwise scaled residual is quite heavily affected. However, a few steps of iterative refinement can be done to recover full accuracy, but the price to pay is 20 seconds, meaning that it can be done if the memory constraint is stronger than the time constraint. Note that this problem is also numerically difficult as $3,375$ delayed pivots and $5,960$ two-by-two pivots have been detected. With a condition number of $3\ 1e+07$, the factorization may be unstable and this can explain the behavior of the componentwise scaled residual. Nevertheless, these are promising results and experimenting refined meshes would increases the potential of compression with a better discretization.

The other two matrices, `amer12` and `dthr7`, are numerically easier (almost no delayed pivot) and present very satisfying performance. With $\varepsilon = 10^{-14}$, both of them require half of the full-rank factor storage and 5 to 10 times less operations. The peak of CB stack is also well reduced, to 10% to 20% of the full-rank peak, which gives a good insight of the global memory reduction (note that the time spent in the external compression phase, which is needed to compress the CB stack and which is not indicated in the table, is always lower than 20% of the overall BLR factorization time indicated in the table). Moreover, even if the time compression is worse than the operation compression, it remains very good as both problems could be solved in just 40% of the time for the full-rank method, at a comparable accuracy. Because the operation compression rates are very low, most of the operations performed during the factorization are done on small blocks, which decreases a lot the BLAS 3 efficiency. Simultaneously, because the number of remaining operations is very low, the decrease in BLAS 3 efficiency is counterbalanced and all in all the overall time decreases.

		|L|	|CB|	ops	time	SR	CSR	SR IR	CSR IR	time IR
				F+S+C+U	F+S+C+U					
pompe	FR	2.6GB	100 %	4.7e+11	89 s.	2.9e-17	2.3e-15	2.1e-17	7.5e-16	20 s.
	10^{-14}	86.3 %	59.3 %	67.3 %	81 s.	2.8e-17	2.2e-10	2.8e-17	6.7e-16	20 s.
	10^{-12}	82.8 %	52.6 %	60.0 %	79 s.	2.8e-17	2.5e-08	2.5e-17	6.0e-16	20 s.
	10^{-10}	78.4 %	44.9 %	52.0 %	76 s.	1.5e-15	2.4e-06	2.3e-17	6.1e-16	22 s.
	10^{-8}	72.9 %	36.0 %	43.0 %	69 s.	2.9e-13	4.5e-04	2.3e-17	8.9e-15	17 s.
	10^{-6}	65.8 %	25.9 %	33.3 %	67 s.	3.5e-11	6.9e-03	2.3e-17	4.3e-13	28 s.
	10^{-4}	56.2 %	15.7 %	23.0 %	61 s.	4.2e-09	1.0e-01	2.4e-09	2.0e-01	22 s.
	10^{-2}	43.5 %	9.1 %	12.5 %	52 s.	7.0e-08	2.0e-01	6.8e-08	5.0e-01	22 s.
amer12	FR	200GB	100 %	1.5e+14	16747 s.	1.8e-14	7.6e-14	1.8e-16	5.6e-16	1300 s.
	10^{-14}	54.9 %	7.7 %	9.2 %	6602 s.	3.8e-15	4.3e-14	1.7e-16	6.1e-16	1489 s.
	10^{-12}	52.8 %	6.6 %	7.9 %	6317 s.	3.7e-13	4.0e-12	1.9e-16	5.7e-16	1332 s.
	10^{-10}	50.4 %	5.4 %	6.7 %	6798 s.	5.7e-11	3.2e-10	1.7e-16	6.3e-16	1489 s.
	10^{-8}	47.7 %	4.2 %	5.4 %	6566 s.	1.6e-08	7.5e-08	1.9e-16	6.1e-16	2293 s.
	10^{-6}	44.6 %	2.7 %	4.1 %	6312 s.	3.1e-06	9.9e-06	1.0e-06	1.7e-05	1728 s.
	10^{-4}	41.1 %	1.7 %	2.8 %	6020 s.	1.4e-04	3.0e-03	1.6e-04	1.0e-03	2951 s.
	10^{-2}	37.5 %	1.5 %	2.0 %	5786 s.	2.6e-03	2.6e-02	1.2e-03	6.7e-03	1758 s.
dthr7	FR	128GB	100 %	2.7e+14	25815 s.	5.9e-17	4.6e-14	7.7e-19	5.5e-16	796 s.
	10^{-14}	49.1 %	22.7 %	18.5 %	10716 s.	2.2e-16	7.3e-13	7.8e-19	5.4e-16	867 s.
	10^{-12}	42.8 %	17.4 %	13.6 %	9188 s.	2.2e-14	1.1e-10	7.6e-19	5.3e-16	566 s.
	10^{-10}	36.4 %	12.6 %	9.4 %	8263 s.	3.3e-12	1.1e-08	7.9e-19	5.8e-16	765 s.
	10^{-8}	30.2 %	8.5 %	6.2 %	8122 s.	4.4e-10	2.5e-06	8.2e-19	5.3e-16	861 s.
	10^{-6}	23.7 %	5.2 %	3.6 %	7338 s.	1.0e-07	1.4e-04	8.3e-19	5.8e-16	1502 s.
	10^{-4}	17.4 %	2.8 %	1.9 %	6528 s.	6.6e-06	2.3e-02	2.8e-06	6.8e-03	905 s.
	10^{-2}	12.5 %	1.6 %	1.1 %	6423 s.	3.7e-05	1.0e-01	1.9e-05	2.9e-02	755 s.

Table 5.2: Results on 3 matrices from EDF applications, given in terms of factor size |L|, peak of CB stack size |CB|, number of operations for the factorization (ops F+S+C+U) and total elapsed time for factorization time (time F+S+C+U). The componentwise scaled residual (CSR) and scaled residual (SR) are given with and without iterative refinement (IR). The time spent in the iterative refinement steps is also indicated (time IR). A percentage in a given row is given relatively to the first value of this row (i.e., value in column FR).

Again, if the memory constraint is stronger than the time constraint, one can relax the low-rank threshold to, say, $\varepsilon = 10^{-8}$ and perform iterative refinement to recover full accuracy. This allows for more factor compression (a factor of 2 5 to 5) and to divide the peak of CB stack by a factor of more than 10. Moreover, even with the extra time spent in the iterative refinement (around 1000 s. for `dthr7` and around 2000 s. for `amer12`), the overall time would stay twice lower as in the full-rank case.

Note that the fact that the accuracy suddenly fails when the low-rank threshold is too large is consistent with what was observed in Figures 4.15 and 4.16 so that $\varepsilon = 10^{-8}$ works well for all problems in this study (so far).

These results pave the way for a new class of numerically approximated solvers with a good potential for iterative refinement, because the low-rank truncation has a numerical meaning. This idea naturally leads to experimenting BLR factorizations as preconditioner in a pure iterative context.

5.1.2 BLR preconditioned conjugate gradient

Block Low-Rank techniques can also be efficiently used to design preconditioners, using a larger threshold ε than in a direct solver context. This is particularly critical for EDF applications when direct solvers are too memory and time consuming, in which case a conjugate gradient preconditioned with a single precision factorization of the original matrix is used. The conjugate gradient iterations are always performed in double precision. We experimented with our BLR solver on two problems which present different behaviors. These problems were also presented in Section 1.6.4.3. `piston` is a problem where the conjugate gradient works well in full-rank. `perf` is ill-conditioned and the convergence is slow. The algorithm used to solve these problems within EDF's Code_Aster and full-rank single precision preconditioner is reported in Algorithm 5.1 for the unsymmetric case.

Algorithm 5.1 Full-rank conjugate gradient preconditioned with a single precision factorization of A, in the unsymmetric case. The subscript shows the precision used for storing the corresponding data (s=single, where not specified, double is assumed).

```
1: Compute the preconditioner: $A_s = A$; $L_s U_s = A_s$
2: Compute $r = b - Ax$ for some initial guess $x$
3: for $it = 1$ to ... do
4:     $r_s = r$
5:     $z_s = U_s \ \ L_s \ \ r_s$
6:     $z = z_s$
7:     $\rho =< r, z >$
8:     if $it = 1$ then
9:         $p = z$
10:    else
11:        $\quad = \rho \ \ \rho o$
12:        $p = z + \quad p$
13:    end if
14:    $q = Ap$
15:    $\alpha = \rho \quad < p, q >$
16:    $x = x + \alpha p$
17:    $r = r + \alpha q$
18:    $\rho o = \rho$
19:    check convergence; continue if necessary
20: end for
```

Algorithm 5.1 will be referred to as FR SP (where SP stands for Single Precision) in this section. Then, the BLR based variants which we experiment for these problems are:

1. BLR SP: L_s and U_s are replaced with $\tilde{L}_s$ and $\tilde{U}_s$; the tilde is added to denote factors computed with a BLR factorization.
2. BLR DP (where DP stands for Double Precision): L_s and U_s are replaced with $\tilde{L}$ and $\tilde{U}$ and steps 4 and 6 removed.

Note that FR DP does not make sense as it requires to compute the full-rank double precision factors (then, the solution should be obtained directly by means of forward elimination and backward substitution).

We give in Table 5.3 results in terms of number of iterations to convergence and time to convergence (i.e. the time for computing the preconditioner and perform the conjugate gradient iterations). These reference results have been obtained with a single precision preconditioner (computed with MUMPS) and correspond to a regular usage within Code_Aster at EDF. As explained before, `piston` has a quick convergence whereas `perf` is much slower.

	# it.	Time
`piston`	3	373 s.
`perf`	69	815 s.

Table 5.3: Number of iterations and time to convergence with FR SP.

We used our BLR solver to compute an alternative preconditioner. We test both single and double precision with different low-rank threshold ε. In single precision, ε ranges from 10^{-8} to 10^{-3} or 10^{-4}, depending on the convergence. In double precision, ε ranges from 10^{-14} to 10^{-3} or 10^{-4}, depending on the convergence. Recall that the conjugate gradient iterations are always performed in double precision. We present in Tables 5.4 and 5.5 the number of iterations obtained with a BLR preconditioner, both in single and double precision. We also give the corresponding memory and operations compression rates due to low-rank compressions. The run times do not appear in these two tables since they will be discussed separately through Figures 5.1 and 5.2.

	BLR SP				BLR DP			
ε	#it	|L|	|CB|	ops F+S+C+U	#it	|L|	|CB|	ops F+S+C+U
10^{-14}	not applicable in single precision				1	79.0%	35.4%	48.4%
10^{-13}					1	77.2%	32.8%	45.3%
10^{-12}					1	75.3%	30.2%	42.3%
10^{-11}					1	73.2%	27.5%	37.1%
10^{-10}					1	70.9%	24.9%	36.0%
10^{-9}					2	68.4%	22.1%	32.7%
10^{-8}	4	66.4%	22.8%	31.0%	2	65.6%	19.2%	29.5%
10^{-7}	3	63.4%	20.0%	27.5%	2	62.5%	16.3%	26.2%
10^{-6}	3	60.1%	17.4%	24.0%	3	59.1%	13.4%	22.9%
10^{-5}	4	56.3%	14.9%	20.5%	4	55.1%	10.9%	19.6%
10^{-4}	6	51.7%	12.8%	17.0%	7	50.4%	8.9%	16.3%
10^{-3}	66	45.9%	11.0%	13.2%	24	44.5%	7.3%	12.8%
10^{-2}	no convergence in 1000s.							

Table 5.4: Compression and convergence results for `piston`, for both BLR SP and BLR DP variants.

In single precision, both Tables 5.4 and 5.5 show that when the low-rank threshold is not too aggressive (e.g. $\varepsilon < 10^{-3}$ for `piston` and $\varepsilon < 10^{-4}$ for `perf`), the increase in number of iterations due to the approximations is fairly contained. With more aggressive thresholds, both problems start converging much slower until the convergence cannot be obtained within 1000 seconds because too much information has been lost.

	BLR SP				BLR DP			
ε	#it	|L|	|CB|	ops F+S+C+U	#it	|L|	|CB|	ops F+S+C+U
10^{-14}	not applicable in single precision				1	72.7%	27.9%	41.9%
10^{-13}					1	70.6%	25.7%	39.2%
10^{-12}					1	68.3%	23.0%	36.2%
10^{-11}					1	66.1%	20.5%	33.4%
10^{-10}					2	64.2%	18.3%	31.0%
10^{-9}					2	62.2%	16.1%	28.8%
10^{-8}	67	59.4%	18.9%	26.7%	3	58.7%	13.7%	25.5%
10^{-7}	68	56.9%	16.9%	24.3%	4	56.4%	11.4%	23.4%
10^{-6}	66	52.4%	15.2%	20.2%	8	51.9%	9.6 %	19.7%
10^{-5}	67	49.1%	14.1%	17.1%	19	48.6%	8.6 %	17.0%
10^{-4}	81	45.1%	13.5%	14.1%	68	44.4%	8.0 %	14.1%
10^{-3}	no convergence in 1000s.							
10^{-2}								

Table 5.5: Compression and convergence results for `perf`, for both BLR SP and BLR DP variants.

In double precision, unsurprisingly, the convergence is very fast at high accuracy, as the preconditioner becomes closer to the exact inverse. At lower accuracy, the number of iterations to convergence increases but remains below the corresponding single precision number of iteration in most cases, as single precision can be viewed as a coarse, basic truncation of double precision while low-rank truncation only dropped values with no numerical influence. For `piston`, the number of iterations cannot be really improved by using a double precision preconditioner as it is already low in single precision. However, for `perf`, the number of iterations can be substantially decreased since the convergence is slow in single precision.

In terms of compression rates due to low-rank approximations, the behavior when the threshold increases is the same as already observed in Chapter 4: the compression rates become better. A factor of at least 2 and 3 can be obtained in memory and operations, respectively, even at full single or double precision.

The iteration counts have to be related to the overall process timing (the spent in computing the preconditioner *and* performing all the required conjugate gradient iterations) in order to find the best strategy. Figures 5.1 and 5.2 give the time spent in the overall process (i.e., computing the preconditioner and performing the conjugate gradient iterations) for both problems.

Interestingly, the speed-up obtained during the preconditioner computation can exceed the extra cost due to the increase in the number of iterations. This is the case for both problems, in single or double precision: the optimal time is not obtained for the lowest number of iteration. For `piston`, $\varepsilon = 10^{-4}$ is the optimal low-rank threshold in terms of execution time, both for single and double precisions. The overall optimal time is obtained in single precision and is 44% of the full-rank case. The corresponding operation compression is 17%, so that, as already observed for many problems in this dissertation, the time reduction is not equivalent to the operations reduction due to the worse efficiency

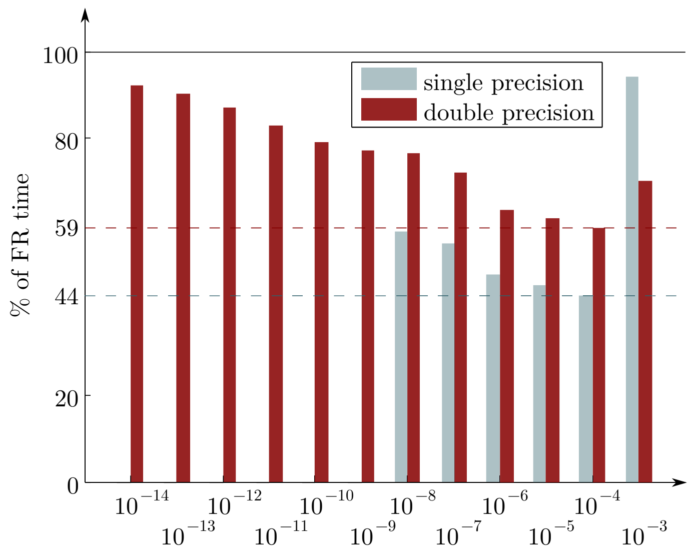


Figure 5.1: Timing results for `piston`. The corresponding total full-rank execution time is 373 seconds.

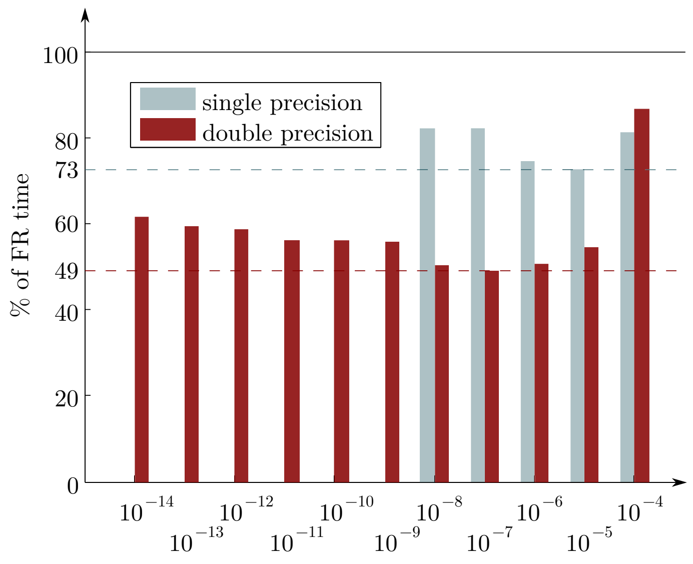


Figure 5.2: Timing results for `perf`. The corresponding total full-rank execution time is 815 seconds.

of BLAS operations in the BLR case. However, the overall time is still cut by more than half because so many operations have been saved that it pays off even if these operations are slower. For `perf`, $\varepsilon = 10^{-5}$ and $\varepsilon = 10^{-7}$ are the optimal low-rank thresholds for single and double precision, respectively. It is very interesting to note that on this problem, the overall optimal time is obtained in double precision, being 49% of the full-rank time. Similarly as for `piston`, the operation efficiency has decreased too. Although double precision operations are slower, this behavior is possible because a very large amount of iterations have been saved. Block-Low Rank double precision preconditioner thus may make sense when the number of iterations in full-rank single precision is large, because a considerable reduction of the number of iterations to convergence can be achieved thanks to the numerically meaningful low-rank truncation.

In terms of memory, because the compression rates are similar in single and double precision, double precision preconditioners are still twice as memory consuming as the single precision preconditioner.

The choice of which strategy to adopt is thus unclear and depends on the context. On one hand, if one needs to recompute the preconditioner many times, then it is worth increasing the number of iterations slightly and obtaining a very cheap preconditioner. On the other hand, if a preconditioner can be reused many times (because of small changes in the matrix for instance, like in a Newton process) it can be worth spending more time computing it but sparing afterwards many iterations.

Note that, as showed in Section 3.5, the solution phase is also improved by the Block Low-Rank techniques. Although not implemented yet, an interesting time reduction can be expected for these problems because the reduction in the factor storage ($|L|$) is important and thus the time spent in the conjugate gradient could be further divided by a factor of almost 2.

5.2 3D frequency-domain seismic modeling (Seiscope project)

5.2.1 Context

Seismic modeling and full-waveform inversion (FWI) can be performed either in the time domain or in the frequency domain [98]. One distinct advantage of the frequency domain is to allow for a straightforward implementation of attenuation in seismic modeling [95]. Second, it provides a suitable framework to implement multi-scale FWI by frequency hopping, which is useful to mitigate the nonlinearity of the inversion [84]. Frequency domain seismic modeling consists of solving an elliptic boundary-value problem, which can be recast in matrix form where the solution (i.e., the monochromatic wavefield) is related to the right-hand side (i.e., the seismic source) through a sparse impedance matrix, whose coefficients depend on frequency and subsurface properties [78]. The resulting linear system can be solved with a sparse direct solver based on the multifrontal method to efficiently compute solutions for multiple sources by forward/backward substitutions, once the impedance matrix is LU factorized. However, the LU factorization of the impedance matrix generates fill-in, which makes the direct-solver approach memory demanding. Dedicated finite-difference stencils of local support [82] and fill-reducing matrix ordering based on nested dissection are thus simultaneously used to minimize this fill-in. A second limitation is that the volume of communications limits the scalability of the LU factorization on a large number of processors. Despite these two limitations, Operto et al. [82] and Brossier et al. [22] showed that few discrete frequencies in the low part of the seismic bandwidth [2 - 7 Hz] can be efficiently modeled for a large number of sources in 3D realistic visco-acoustic subsurface models using small-scale computational platforms equipped with large-memory nodes. This makes this technique attractive for velocity model building from wide-azimuth data by FWI [19]. A first application to real Ocean Bottom Cable data including a comparison with time-domain modeling was recently presented by Brossier et al. [23]. To reduce the memory demand and the operation counts, Wang et al. [99] proposed computing approximate solutions of the linear system by exploiting the low-rank properties of elliptic partial differential operators, based on a HSS solver. Their approach exploits the regular pattern of the impedance matrix built with finite-difference stencils on uniform grid. Our algebraic approach is amenable to matrices with a non regular pattern such as those generated with finite element methods on unstructured meshes. We test our solver on the 3D SEG/EAGE overthrust model and give quantitative insights on the memory and operation count savings provided by the BLR solver.

5.2.2 Numerical example and analysis

We perform acoustic finite-difference frequency-domain seismic modeling in the 3D SEG EAGE overthrust model [14] of dimension 20km 20km 4 65km (km=kilometers) with the 27-point mixed-grid finite-difference stencil [82] for the 2-Hz, 4-Hz and 8-Hz frequencies (Figures 5.3 and 5.4).

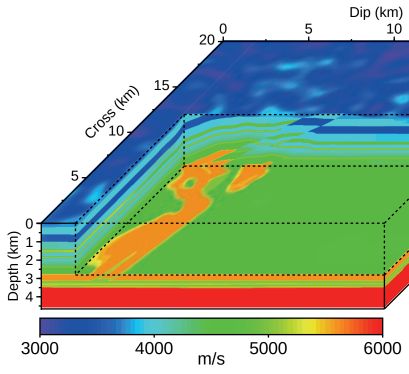


Figure 5.3: SEG/EAGE overthrust velocity model.

We perform the simulation on a single 64-core AMD Opteron node equipped with 384GB of shared memory. No timing could be obtained because no exclusive access to the machine was provided, so that the timings we obtained where not stable enough to be exploitable; we were not allowed to install the encompassing code in any other machine neither. The compression rates could be obtained as they do not require any exclusive access. We use single precision arithmetic to perform the full-rank (FR) and the BLR factorizations. The point source is located at x (dip) = 2 km, y (cross) = 2 km and z = 0.5 km. A discretization rule of 4-grid points per minimum wavelength leads to a grid interval of 250 m, 135 m and 68 m and a finite-difference grid of 0.3, 1.4 and 8 millions of nodes for the 2-Hz, 4-Hz and 8-Hz frequency, respectively, considering 8 grid points in the perfectly-matched layers surrounding the computational domain. Results for the full-rank simulations obtained with MUMPS are given in Table 5.6.

Frequency	Operation count LU	Mem LU	Memory peak
2 Hz	8.957E+11	3 GB	4 GB
4 Hz	1.639E+13	22 GB	25 GB
8 Hz	5.769E+14	247 GB	283 GB

Table 5.6: Results for the full-rank simulations obtained with MUMPS. *Operation count LU*: number of operations during LU factorization. *Mem LU*: Memory for LU factors in GigaBytes. *Memory peak*: Maximum size of stack during LU factorization in GigaBytes.

The BLR solutions are computed for 3 values of the threshold ε (10^{-3}, 10^{-4} and 10^{-5}) and are validated against the FR solutions (Figure 5.4(a-c)). The accuracy of the BLR solutions can be qualitatively assessed in Figure 5.4(d-l) and the reduction of the memory demand and operation complexity resulting from the BLR approximation are outlined in Tables 5.7 and 5.8. We use the metrics presented in Section 4.1.

In a first attempt to assess the footprint of the BLR approximation on seismic imaging, we show 2-Hz monochromatic reverse-time migrated images (RTM) computed from the FR and the BLR solutions in Figure 5.5. RTM is computed in the true velocity model for an array of shots and receivers near the surface with a shot and receiver spacings of 500 m and 250 m, respectively.

BLR solutions for $\varepsilon = 10^{-3}$ are of insufficient quality for FWI applications (Fig. 5.4(d-f) and 5.5d), while those obtained with $\varepsilon = 10^{-5}$ closely match the FR solutions (Fig. 5.4(j-l) and 5.5b). BLR solutions for $\varepsilon = 10^{-4}$ show some slight differences with the FR solutions (Fig. 5.4(g-i)), but might be considered for FWI applications (Fig. 5.5c). For the 8-Hz frequency, the operation count performed during the BLR factorization represents 14.8% ($= 10^{-4}$) and 21.3% ($= 10^{-5}$) of the one performed during the FR factorization. It is worth noting that the operation count reduction increases with frequency (i.e., with the size of the computational grid) as the maximum distance between variables in the grid increases, a key feature in view of larger-scale simulations. For example, for $\varepsilon = 10^{-4}$, the operation count decreases from 28 4% to 14 8% when the frequency increases from 2 Hz to 8 Hz (see Table 5.7). The overhead associated with the decompression of the contribution block is negligible, a distinct advantage compared to the HSS approach of Wang et al. [99], and hence should not significantly impact the performance of the BLR factorization. The memory saving achieved during the BLR factorization follows the same trend as the operation count, since it increases with frequency. For the 8-Hz frequency and $\varepsilon = 10^{-5}$, the memory for LU-factor storage and the peak of CB stack during BLR factorization represent 41 6% and 23 9% of those required by the FR factorization, respectively, as shows Table 5.8. The reduction of the storage of the contribution blocks (CB) is even higher. Both (LU and CB compression) will contribute to reducing the volume of communication by a substantial factor and improving the parallel efficiency of the solver.

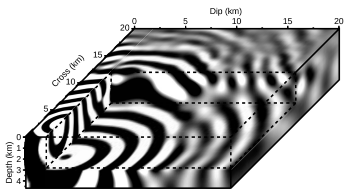
Frequency	ops F+S+U(%)			ops C (%)		
	$= 10^{-3}$	$= 10^{-4}$	$= 10^{-5}$	$= 10^{-3}$	$= 10^{-4}$	$= 10^{-5}$
2 Hz	21.1	28.4	36.6	3.5	4.5	5.2
4 Hz	12.7	18.6	25.6	1.1	1.4	1.8
8 Hz	9.5	14.8	21.3	0.3	0.4	0.5

Table 5.7: Operation statistics of the BLR simulations.

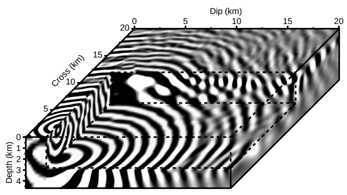
Frequency	|L|			|CB|		
	$= 10^{-3}$	$= 10^{-4}$	$= 10^{-5}$	$= 10^{-3}$	$= 10^{-4}$	$= 10^{-5}$
2 Hz	44.7	53.4	61.8	16.8	23.9	32.3
4 Hz	34.5	42.2	50.0	19.0	21.7	24.4
8 Hz	21.3	28.9	41.6	15.9	19.4	23.9

Table 5.8: Memory statistics of the BLR simulations.

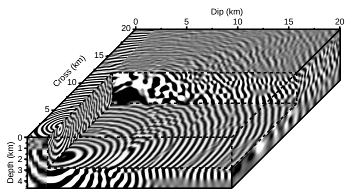
The computational time and memory savings achieved during BLR factorization increase with the size of the computational grid (i.e., frequency), suggesting than one order of magnitude of saving for these two metrics can be viewed for large-scale factorization



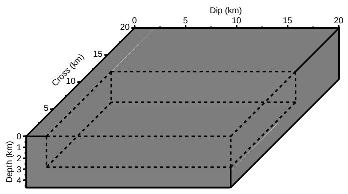
(a) FR solution at 2Hz



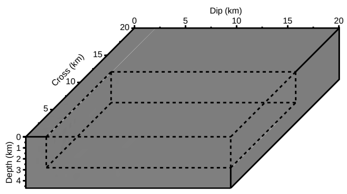
(b) FR solution at 4Hz



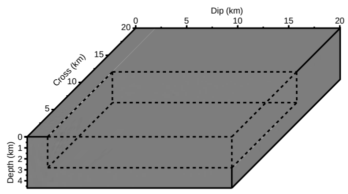
(c) FR solution at 8Hz



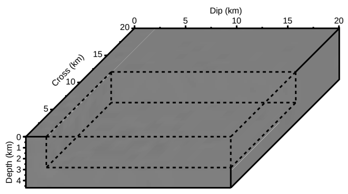
(d) Di erence between FR and BLR solution at $= 10^{-5}$ at 2Hz.



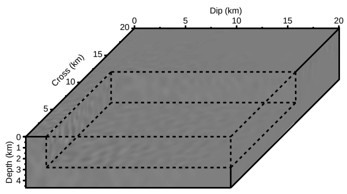
(e) Di erence between FR and BLR solution at $= 10^{-5}$ at 4Hz.



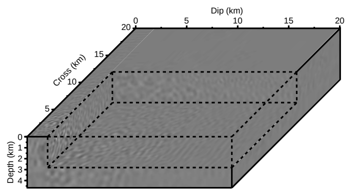
(f) Di erence between FR and BLR solution at $= 10^{-5}$ at 8Hz.



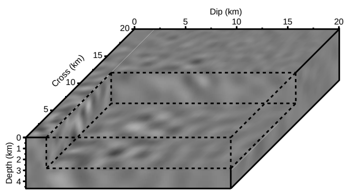
(g) Di erence between FR and BLR solution at $= 10^{-4}$ at 2Hz.



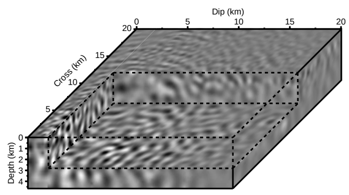
(h) Di erence between FR and BLR solution at $= 10^{-4}$ at 4Hz.



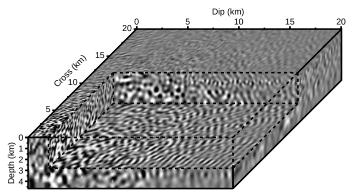
(i) Di erence between FR and BLR solution at $= 10^{-4}$ at 8Hz.



(j) Di erence between FR and BLR solution at $= 10^{-3}$ at 2Hz.



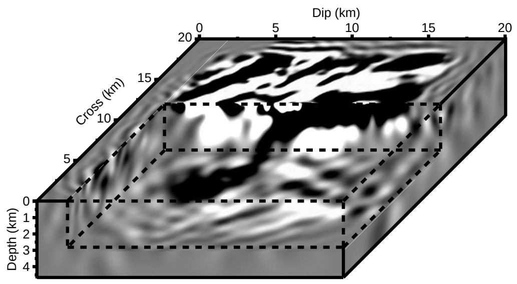
(k) Di erence between FR and BLR solution at $= 10^{-3}$ at 4Hz.



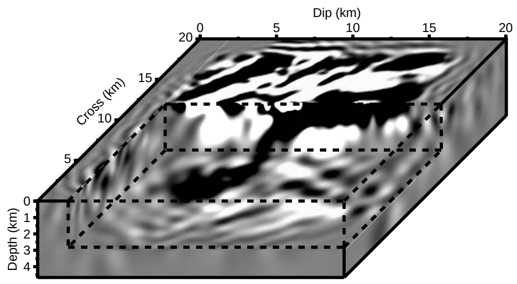
(l) Di erence between FR and BLR solution at $= 10^{-3}$ at 8Hz.

Figure 5.4: Study on the quality of the real part of the solution with different accuracies and frequencies. Amplitudes are clipped to half the mean amplitude of the full-rank wavefield on each panel.

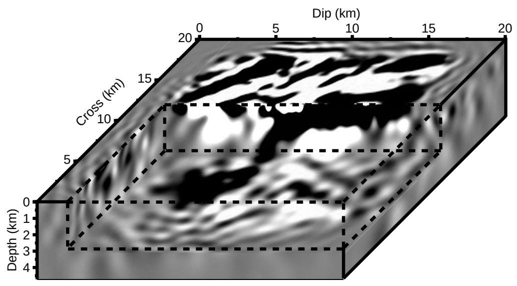
involving several tens of millions of unknowns. In this special applicative context, future work involves a sensitivity analysis of FWI to the BLR approximation before application of visco-acoustic FWI on wide-azimuth data recorded with fixed-spread acquisition geometries [23]. Note that, as already stated throughout this dissertation, the computational efficiency of the BLR solver might be improved by iterative refinement of the solutions (although this needs to be performed for each right-hand side, which can represent many computations in this particular application) or by performing the BLR factorization in double precision. Moreover, the computational savings provided by the BLR solver might allow frequency-domain seismic modeling in realistic 3D visco-elastic anisotropic media [100].



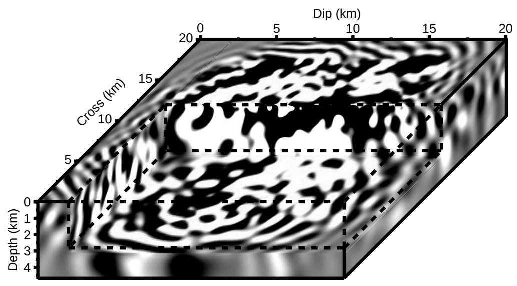
(a) FR



(b) $= 10^{-5}$



(c) $= 10^{-4}$



(d) $= 10^{-3}$

Figure 5.5: 2-Hz RTM image for full-rank and low-rank factorizations.

5.3 Problems from the University of Florida Sparse Matrix Collection

In addition to all the results already presented, we extend our study on matrices coming from different types of applications in order to show that BLR can be efficient on a wide range of problems. These matrices, also, appear in Xia [103] and give thus another insight of how BLR compares to HSS. Even though they are small, they give an interesting idea of different fields where low-rank technologies could be used. In Table 5.9, we indicate the complete results on this set of 6 matrices, including compression rates, timing and accuracy.

First note that on these small matrices, the full-rank execution time is so low that it is difficult to improve it significantly, which explains why the full-rank and low-rank execution times are so close, even if the operation compression rates are good (note that the time spent in the iterative refinement is not taken into consideration). As far as memory is concerned, the factors are reduced by less than a factor of 2 but the maximum peak of CB stack $|\text{CB}|$ is divided by 4 to 2, which is significant. To obtain these compression rates, the low-rank threshold had to be set up to $\varepsilon = 10^{-8}$, which leads to a CSR of the same order. However, with only 2 steps of iterative refinement, the full accuracy is recovered, meaning that on problems theoretically not suitable for low-rank approximations, it can nevertheless be worth using these technologies with loosely approximated factorizations and iterative refinement.

Finally, these results compare well with what could be obtained with an HSS solver in Xia [103]. In most matrices, the compression of the factor is worse with BLR but the operation compression is usually better, at a same accuracy. Timing could not be compared as they were not indicated in the reference paper.

		apache2	ecology2	G3_circuit	parabo-lic_fem	thermo-mech_dM	tmt_sym
|L|	MUMPS	1.68E8	4.32E7	1.13E8	2.98E7	7.53E6	3.62E7
	BLR	55%	70%	70%	79%	81%	75%
|CB|	BLR	28%	25%	23%	37%	46%	28%
ops	MUMPS	2.36E11	1.85E10	6.68E10	8.46E9	8.16E8	1.32E10
	BLR	21%	28%	36%	44%	52%	34%
time	MUMPS	28 s.	7 s.	15 s.	3 s.	1 s.	4 s.
	BLR	28 s.	5 s.	16 s.	2 s.	2 s.	4 s.
CSR	MUMPS	8.79E-15	2.60E-15	5.20E-15	2.65E-15	6.38E-16	4.39E-16
	BLR	4.44E-9	1.45E-8	4.58E-9	7.32E-9	4.31E-9	1.96E-9
	BLR (2 IR)	1.74E-16	2.00E-16	2.47E-16	2.60E-16	3.49E-16	6.63E-17

Table 5.9: Results on matrices coming from different application fields. CSR is the componentwise scaled residual.

Chapter 6

Conclusion

6.1 General results

In this work, we have proposed an efficient and flexible low-rank format suitable for the implementation of algebraic general purpose multifrontal solvers. We have first presented an extensive survey of the most popular already existing hierarchical low-rank formats and discussed the interest of a flat format referred to as Block Low Rank (BLR) in the context of algebraic distributed-memory multifrontal solvers. In the dense matrices, we have compared BLR, $\mathcal{H}$ and HSS formats based on the cost of compressing, the memory needed to store the low-rank form and the number of operations required to compute the factorization. We have shows that BLR is a good candidate; although it is slightly less efficient in terms of memory and operation compression, it allows for a good BLAS 3 efficiency and critical algorithms on which rely, for instance, the numerical pivoting and the distribution among processors, can be adapted without conceding their efficiency. To define the blocking on which the BLR format relies, we have designed an efficient algebraic clustering which exploits the assembly tree of the multifrontal process. The overhead of this clustering step, which is performed during the analysis phase, has been shown to be low. We have proposed different factorization algorithms which differently exploit the BLR approximations: from a loosely approximated factorization (for applications where one solution phase is performed per factorization) to an exact factorization with off-line approximated factors (for applications where many solutions phases are performed per factorization). We have focused in this thesis on an intermediate variant which gives a good compromise between numerical robustness and reduction of factorization and solution complexities. We have proposed an implementation scheme of this algorithm that can embed threshold partial pivoting and distributed-memory parallelism. A flexible recursive two levels of panels algorithm has been designed to accommodate both BLR format and numerically robust blocked factorization. We have shown that this scheme allows for efficient use of matrix-matrix operations (BLAS 3 kernels) within BLR based factorization.

To validate the proposed approach, we have firstly experimented with our method on two standard PDEs applications and have shown that substantial memory and time reductions can be obtained, even with a low-rank threshold close to machine precision. Moreover, we have observed that the efficiency of the BLR factorization does not need a fine tuning of the block size and does not rely on a particular underlying global ordering of the matrix. We have exhibited that partial pivoting is critical in some cases in order to preserve accuracy and low-rank compression rates. We have demonstrated that in the context of distributed-memory parallelism the compression rates are also preserved. BLR approach has then been experimented on a wide set of symmetric and unsymmetric large industrial problems. We have shown that quite interesting memory and time reductions can be obtained while preserving/controlling numerical accuracy in the solution.

Finally, we have tested the BLR factorization with larger low-rank threshold to use it as a preconditioner for the conjugate gradient method. We have showed that it can be a good alternative against general single precision full-rank preconditioners, thanks to a numerical truncation instead of the coarse sword cutting resulting from the use of single precision arithmetic.

6.2 Software improvements

All proposed algorithms have been implemented in the context of the MUMPS (MUltifrontal Massively Parallel Solver) solver. The full-rank factorization had to be completely redesigned in order to implement the two levels of panels even in full-rank, so that the same numerical kernels can be used in low-rank. We observed that the two levels of panel scheme allows for more efficiency in the full-rank case too, due to a better cache locality. We managed to maintain, together with the BLR factorization, most of the underlying algorithms of MUMPS, so that critical features such as out-of-core, pivoting, dynamic scheduling are still available together with BLR factorization.

This development version of MUMPS has been coupled with Code_Aster and experimented in the industrial context of EDF, validating that both full-rank and low-rank versions of the solver can be used. It has also been successfully used in the context of 3D frequency-domain seismic modeling (Seiscope project).

We emphasize that the technologies and ideas presented throughout this dissertation do not rely on a particular implementation of the multifrontal method, and can thus be applied or adapted to any other solver besides MUMPS.

6.3 Perspectives and future work

We describe possible directions for future work. We start with some implementation issues. As recalled in previous section, many variants of the BLR factorization have been defined and although the approach that offers the best compromise between numerical robustness and efficiency has been implemented, one could consider implement the remaining variants. Among them the variant that provides an exact factorization with off-line approximate factors could be interesting and reasonably easy to implement. This would be a natural and quite straightforward way to improve the flexibility of BLR-based solvers, and to adapt to applicative constraints. In the same context, we have shown that the solution phase of the multifrontal method can be substantially improved by taking advantage of the compression of the factors. Implementing the BLR solution phase will further improve the overall process, especially when many solution phases are performed for each factorization, such as in the conjugate gradient method or for the front-wave inversion problem (see Section 5.2). This requires the factors to be stored in BLR format and thus to define new data structure to store factors. This data structure would also allow for a better exploitation of the memory compression. To further reduce the memory footprint, a panelwise assembly could be performed: once the panel associated to the current BLR block is assembled and factorized, a compression step would be immediately performed, decreasing the global memory needed to store a front. Hence there is (almost) no large dense matrix; this is reminiscent of the fully-structured HSS approach. The same data structure could also be used to store the contribution blocks in compressed form. This would enable further communication volume reduction and decrease in the size of the memory stack.

In a distributed-memory context, slave to slave communications (i.e., within a node of the assembly tree) are also requested for symmetric matrices and should be done in BLR format so that the corresponding contribution blocks can be updated using low-rank products. The BLR factorization also raises many scheduling issues, as the amount of storage and computations decreases. Even though promising results could be obtained without any modification of the dynamic scheduling, many improvements could be obtained by taking into account compression rates. The main difficulty is that because of low-rank compression it is impossible to predict the memory and computational requirements for a given front. Moreover, because the (2,1) block of a BLR front is usually better compressed than the (1,1) block, the computations performed in the (1,1) block may become dominant creating an unbalanced between processes. The distribution of the rows among processors may thus be impacted. A first idea would be to further split nodes (see [71]) of the elimination tree to reduce the relative weight of the (1,1) block. in order to decrease the size of the dominant block. Similarly, because the relative computational weight of the fronts will dynamically and substantially change, the global mapping of the tasks on the processes will have to be redesigned and even if dynamic scheduling should naturally handle part of the difficulties our scheduling algorithms will have to be revisited in this context. For example, some studies have been carried out on static memory-aware mapping [86], showing that pivoting can perturb quite heavily the memory usage and thus that dynamic memory-aware mapping is needed. BLR factorizations enhance this need because of dynamic compressions.

In a more general context, BLR technologies pave the way to a broader usage of direct solvers (multifrontal and even supernodal) for industrial applications, as they allow some of the usual difficulties encountered by direct solvers users to be overcome. Finally, we believe that the proposed approaches can also be successfully and naturally exploited to improve the efficiency of hybrid and iterative solvers.

Appendix A

Publications

A.1 Submitted publications

[A] P. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L Excellent, and C. Weisbecker, *Improving multifrontal solvers by means of block low-rank representations.* Submitted to SIAM Journal on Scientific Computing, 2012 (under revision since June 2013).

A.2 Proceedings

[B] C. Weisbecker, P. Amestoy, O. Boiteau, R. Brossier, A. Buttari, J.-Y. L Excellent, S. Operto and J. Virieux, *3D frequency-domain seismic modeling with a Block Low-Rank algebraic multifrontal direct solver.* International Conference 'Society of Exploration Geophysicits (SEG) Annual Meeting', Houston, USA, September 2013.

[C] E. Agullo, P. Amestoy, A. Buttari, A. Guermouche, G. Joslin, J.-Y. L Excellent, X. S. Li, A. Napov, F.-H. Rouet, M. Sid-Lakhdar, S. Wang, C. Weisbecker and I. Yamazaki, *Recent advances in sparse direct solvers.* International Conference on Structural Mechanics in Reactor Technology (SMIRT-22), San Francisco, USA, August 2013.

[D] P. Amestoy, A. Buttari, G. Joslin, J.-Y. L Excellent, M. Sid-Lakhdar, C. Weisbecker, M. Forzan, C. Pozza, R. Perrin and V. Pellissier, *Shared memory parallelism and low-rank approximation techniques applied to direct solvers in FEM simulation.* International Conference on the Computation of Electromagnetic Fields (COMPUMAG), Budapest, Hungary, July 2013.

A.3 Conferences

[E] P. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L Excellent, and C. Weisbecker, *Block Low-Rank (BLR) approximations to improve multifrontal sparse solvers.* Sparse Days 13, Toulouse, France, June 2013.

[F] P. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L Excellent, and C. Weisbecker, *Improving multifrontal methods by means of low-rank approximations techniques.* SIAM Conference on Applied Linear Algebra (LA12), Valencia, Spain, June 2012.

[G] P. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L Excellent, and C. Weisbecker, *Grouping variables in Frontal Matrices to improve Low-Rank Approximations in a Multifrontal Solver.* International Conference On Preconditioning Techniques For Scientific And Industrial Applications (Preconditioning 2011), Bordeaux, France, May 2011.

A.4 Reports

[H] A. Buttari, S. Gratton, X.S. Li, M. Ngom, F.-H. Rouet, D. Titley-Peloquin, and C. Weisbecker, *Error Analysis of the Block Low-Rank LU factorization of dense matrices.* IRIT-CERFACS technical report RT-APO-13-7, in collaboration with LBNL, 2013.

[I] P. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L Excellent, and C. Weisbecker, *Improving multifrontal solvers by means of block low-rank representations.* INPT-IRIT technical report RT-APO-12-6, also appeared as INRIA technical report RR-8199, December 2012.

[J] P. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L Excellent, and C. Weisbecker, *Version V1 de MUMPS nalisée, version V2 préliminaire de MUMPS, nouveaux résultats, première version incomplète du manuscript de thèse, manuel d utilisation du prototype.* Rapport de contrat EDF RT7, RT-APO-13-5.

[K] P. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L Excellent, and C. Weisbecker, *Version V1 de MUMPS, nouveaux résultats, plan de thèse.* Rapport de contrat EDF RT6, RT-APO-13-4.

[L] P. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L Excellent, and C. Weisbecker, *Rapport de spéci cation décrivant le cadre d implantation de l approche low-rank dans MUMPS - Retour complet sur le jeu de problèmes M2.* Rapport de contrat EDF RT5, RT-APO-12-3.

[M] P. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L Excellent, and C. Weisbecker, *Version V0 de MUMPS : bilan, performance, limitations, perspectives d amélioration.* Rapport de contrat EDF RT4, RT-APO-12-1.

[N] P. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L Excellent, and C. Weisbecker, *Bilan du potentiel des méthodes low-rank sur la gamme de problèmes réels M1.* Rapport de contrat EDF RT3, RT-APO-11-3.

[O] P. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L Excellent, and C. Weisbecker, *Premier retour sur le potentiel des méthodes low-rank sur la gamme de problèmes réels M2.* Rapport de contrat EDF RT2, RT-APO-11-7.

[P] P. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L Excellent, and C. Weisbecker, *Bilan du potentiel des méthodes low-rank sur une gamme de problèmes synthétiques.* Rapport de contrat EDF RT1, RT-APO-10-11.

[Q] C. Weisbecker, *Frontal matrices factorization. Low-rank forms.* Rapport de Master, RT-APO-10-10, Institut National Polytechnique de Toulouse, September 2010.

Bibliography

[1] http://www.code-aster.org.

[2] http://www.innovation.edf.com.

[3] P. R. Amestoy. *Factorization of large sparse matrices based on a multifrontal approach in a multiprocessor environment.* PhD thesis, Institut National Polytechnique de Toulouse and CERFACS, TH/PA/91/2, 1991.

[4] P. R. Amestoy and I. S. Duff. Vectorization of a multiprocessor multifrontal code. *International Journal of High Performance Computing Applications*, 3(3):41 59, 1989.

[5] P. R. Amestoy and I. S. Duff. Memory management issues in sparse multifrontal methods on multiprocessors. *International Journal of Supercomputer Applications.*, 7:64 82, 1993.

[6] P. R. Amestoy and C. Puglisi. An unsymmetrized multifrontal LU factorization. *SIAM Journal on Matrix Analysis and Applications*, 24:553 569, 2002.

[7] P. R. Amestoy, T. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4):886 905, 1996.

[8] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Computer Methods in Applied Mechanics and Engineering*, 184(2-4):501 520, 2000.

[9] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15 41, 2001.

[10] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and X. S. Li. Analysis and comparison of two general sparse solvers for distributed memory computers. *ACM Transactions on Mathematical Software*, 27(4):388 421, 2001.

[11] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel computing*, 32(2):136 156, 2006.

[12] P. R. Amestoy, I. S. Duff, D. Ruiz, and B. Uçar. A parallel matrix scaling algorithm. In *High Performance Computing for Computational Science, VECPAR 08*, number 5336 in Lecture Notes in Computer Science, pages 309 321. Springer-Verlag, 2008.

[13] P. R. Amestoy, A. Buttari, I. S. Duff, A. Guermouche, J.-Y. L'Excellent, and B. Uçar. `MUMPS` (MUltifrontal Massively Parallel Solver). In *Encyclopedia of Parallel Computing.* Springer, 2010.

[14] F. Aminzadeh, J. Brac, and T. Kunz. *3-D Salt and Overthrust models*. SEG/EAGE 3-D Modeling Series No.1, 1997.

[15] M. Arioli, J. W. Demmel, and I. S. Duff. Solving sparse linear systems with sparse backward error. *SIAM Journal on Matrix Analysis and Applications*, 10:165 190, 1989.

[16] C. Ashcraft and R. Grimes. The influence of relaxed supernode partitions on the multifrontal method. *ACM Transactions on Mathematical Software*, 15:291 309, 1989.

[17] W. W. Barrett. A theorem on inverse of tridiagonal matrices. *Linear Algebra and Its Applications*, 27:211 217, 1979.

[18] M. Bebendorf. *Hierarchical Matrices: A Means to E ciently Solve Elliptic Boundary Value Problems (Lecture Notes in Computational Science and Engineering)*. Springer, 1st edition, 2008.

[19] H. Ben-Hadj-Ali, S. Operto, and J. Virieux. Velocity model building by 3D frequency-domain, full-waveform inversion of wide-aperture seismic data. *Geophysics*, 73:101 117, 2008.

[20] S. Börm. *E cient Numerical Methods for Non-Local Operators*. European Mathematical Society, 2010.

[21] S. Börm, L. Grasedyck, and W. Hackbusch. Introduction to hierarchical matrices with applications. *Engineering Analysis with Boundary Elements*, 27(5):405 422, 2003.

[22] R. Brossier, V. Etienne, S. Operto, and J. Virieux. Frequency-domain numerical modelling of visco-acoustic waves based on finite-difference and finite-element discontinuous galerkin methods. In *Acoustic Waves*, pages 125 158. SCIYO, 2010.

[23] R. Brossier, V. Etienne, G. Hu, S. Operto, and J. Virieux. Performances of 3D frequency-domain full-waveform inversion based on frequency-domain direct-solver and time-domain modeling: application to 3D OBC data from the Valhall field. *International Petroleum Technology Conference, Beijing, China*, 2013.

[24] J. R. Bunch and L. Kaufman. Some stable methods for calculating inertia and solving symmetric linear systems. *Mathematics of Computation*, 31:162 179, 1977.

[25] J. R. Bunch and B. N. Parlett. Direct methods for solving symmetric indefinite systems of linear systems. *SIAM Journal on Numerical Analysis*, 8:639 655, 1971.

[26] A. Buttari, S. Gratton, X. S. Li, M. Ngom, F.-H. Rouet, D. Titley-Peloquin, and C. Weisbecker. Error analysis of the block low-rank LU factorization of dense matrices. Technical report RT-APO-13-7, IRIT-CERFACS, in collaboration with LBNL, 2013.

[27] S. Chandrasekaran, P. Dewilde, M. Gu, T. Pals, and A.-J. van der Veen. *Fast stable solver for sequentially semi-separable linear systems of equations*. Springer, 2002.

[28] S. Chandrasekaran, M. Gu, and T. Pals. A fast ULV decomposition solver for hierarchically semiseparable representations. *SIAM Journal on Matrix Analysis and Applications*, 28(3):603 622, 2006.

[29] S. Chandrasekaran, P. Dewilde, M. Gu, and N. Somasunderam. On the numerical rank of the off-diagonal blocks of Schur complements of discretized elliptic PDEs. *SIAM Journal on Matrix Analysis and Applications*, 31(5):2261 2290, 2010.

[30] C. Chevalier and F. Pellegrini. `PT-SCOTCH`: A tool for efficient parallel graph ordering. *Parallel Computing*, 34(6):318 331, 2008.

[31] J. Choi, J. W. Demmel, I. Dhillon, J. J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. W. Walker, and R. C. Whaley. `SCALAPACK`: A portable linear algebra library for distributed memory computers - design issues and performance. Technical Report LAPACK Working Note 95, CS-95-283, University of Tennessee, 1995.

[32] T. A. Davis. Algorithm 832: `UMFPACK` v4.3 an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software*, 30(2):196 199, 2004.

[33] T. A. Davis. A column pre-ordering strategy for the unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software*, 30(2):165 195, 2004.

[34] T. A. Davis and I. S. Duff. An unsymmetric-pattern multifrontal method for sparse LU factorization. *SIAM Journal on Matrix Analysis and Applications*, 18:140 158, 1997.

[35] T. A. Davis and I. S. Duff. A combined unifrontal/multifrontal method for unsymmetric sparse matrices. *ACM Transactions on Mathematical Software*, 25(1):1 19, 1999.

[36] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1 25, 2011.

[37] J. W. Demmel and N. J. Higham. Stability of block algorithms with fast level-3 blas. *ACM Transactions on Mathematical Software*, 18(3):274 291, 1992.

[38] J. W. Demmel, N. J. Higham, and R. S. Schreiber. Stability of block LU factorization. *Numerical linear algebra with applications*, 2(2):173 190, 1995.

[39] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20(3):720 755, 1999.

[40] J. W. Demmel, J. R. Gilbert, and X. S. Li. An asynchronous parallel supernodal algorithm for sparse Gaussian elimination. *SIAM Journal on Matrix Analysis and Applications*, 20(4):915 952, 1999.

[41] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1): 1 17, 1990.

[42] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Transactions on Mathematical Software*, 9:302 325, 1983.

[43] I. S. Duff and J. K. Reid. The multifrontal solution of unsymmetric sets of linear systems. *SIAM Journal on Scienti c and Statistical Computing*, 5:633 641, 1984.

[44] I. S. Duff and J. K. Reid. Exploiting zeros on the diagonal in the direct solution of indefinite sparse symmetric linear systems. *ACM Transactions on Mathematical Software*, 22(2):227 257, 1996.

[45] S. C. Eisenstat and J. W. H. Liu. The theory of elimination trees for sparse unsymmetric matrices. *SIAM Journal on Matrix Analysis and Applications*, 26(3):686 705, 2005.

[46] S. C. Eisenstat and J. W. H. Liu. Algorithmic aspects of elimination trees for sparse unsymmetric matrices. *SIAM Journal on Matrix Analysis and Applications*, 29(4): 1363 1381, 2008.

[47] B. Engquist and L. Ying. Sweeping preconditioner for the Helmholtz equation: hierarchical matrix representation. *Communications on pure and applied mathematics*, 64(5):697 735, 2011.

[48] W. Fong and E. Darve. The black-box fast multipole method. *Journal of Computational Physics*, 228(23):8712 8725, 2009.

[49] A. G. Recent advances in direct methods for solving unsymmetric sparse systems of linear equations. *ACM Transactions on Mathematical Software*, 28(3):301 324, 2002.

[50] F. R. Gantmakher and M. G. Krein. *Oscillation matrices and kernels and small vibrations of mechanical systems.* Number 345. American Mathematical Society, 2002.

[51] A. George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, pages 345 363, 1973.

[52] J. R. Gilbert and J. W. H. Liu. Elimination structures for unsymmetric sparse LU factors. *SIAM Journal on Matrix Analysis and Applications*, 14:334 352, 1993.

[53] J. R. Gilbert and E. G. Ng. Predicting structure in nonsymmetric sparse matrix factorizations. In *Graph Theory and Sparse Matrix Computations*, pages 107 140. Springer-Verlag NY, 1993.

[54] A. Gillman. *Fast direct solvers for elliptic partial di erential equations.* PhD thesis, University of Colorado, 2011.

[55] A. Gillman, P. Young, and P.-G. Martinsson. A direct solver with $\mathcal{O}(\mathrm{N})$ complexity for integral equations on one-dimensional domains. *Frontiers of Mathematics in China*, 7:217 247, 2012.

[56] I. Gohberg, T. Kailath, and I. Koltracht. Linear complexity algorithms for semiseparable matrices. *Integral Equations and Operator Theory*, 8(6):780 804, 1985.

[57] G. Golub and C. Van Loan. *Matrix computations.* Johns Hopkins University Press, 3rd edition, 1996.

[58] L. Grasedyck and W. Hackbusch. Construction and arithmetics of $\mathcal{H}$-matrices. *Computing*, 70(4):295 334, 2003.

[59] L. Grasedyck, R. Kriemann, and S. Le Borne. Parallel black box $\mathscr{H}$-LU preconditioning for elliptic boundary value problems. *Computing and Visualization in Science*, 11(4):273 291, 2008.

[60] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73(2):325 348, 1987.

[61] L. Grigori, J. W. Demmel, and X. S. Li. Parallel symbolic factorization for sparse LU with static pivoting. *SIAM Journal on Scienti c Computing*, 29(3):1289 1314, 2007.

[62] M. Gu, X. S. Li, and P. S. Vassilevski. Direction-preserving and schur-monotonic semiseparable approximations of symmetric positive definite matrices. *SIAM Journal on Matrix Analysis and Applications*, 31(5):2650 2664, 2010.

[63] A. Guermouche, J.-Y. L'Excellent, and G. Utard. Impact of reordering on the memory of a multifrontal solver. *Parallel Computing*, 29(9):1191 1218, 2003.

[64] W. Hackbusch. A sparse matrix arithmetic based on $\mathcal{H}$-matrices. Part I: Introduction to $\mathcal{H}$-matrices. *Computing*, 62(2):89 108, 1999.

[65] M. T. Heath, E. G. Ng, and B. W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33:420 460, 1991.

[66] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear system. *Journal of Research of the National Bureau of Standards*, 49:409 436, 1952.

[67] N. J. Higham. Exploiting fast matrix multiplication within the level 3 blas. *ACM Transactions on Mathematical Software*, 16(4):352 368, 1990.

[68] G. Karypis and V. Kumar. *METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing ll-reducing orderings of sparse matrices version 4.0.* University of Minnesota, Army HPC Research Center, Minneapolis, 1998.

[69] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing*, 48(1):96 129, 1998.

[70] G. Karypis and V. Kumar. *PARMETIS, parallel graph partitioning and sparse matrix ordering library.* University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, Minneapolis, MN 55455, U.S.A., Aug. 2003.

[71] J.-Y. L'Excellent. *Multifrontal Methods: Parallelism, Memory Usage and Numerical Aspects.* Habilitation à diriger des recherches, École normale supérieure de Lyon, Sept. 2012.

[72] J.-Y. L'Excellent and M. W. Sid-Lakhdar. Introduction of shared-memory parallelism in a distributed-memory multifrontal solver, 2013. Submitted to SIAM Journal of Scientific Computing.

[73] X. S. Li and J. W. Demmel. Making sparse gaussian elimination scalable by static pivoting. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1 17. IEEE Computer Society, 1998.

[74] X. S. Li and J. W. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Transactions on Mathematical Software*, 29(2), 2003.

[75] R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. *SIAM Journal on Numerical Analysis*, 16(2):346 358, 1979.

[76] J. W. H. Liu. On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Transactions on Mathematical Software*, 12:127 148, 1986.

[77] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11(1):134 172, 1990.

[78] K. J. Marfurt. Accuracy of finite-difference anf finite-element modeling of the scalar and elastic wave equations. *Geophysics*, 49:533 549, 1984.

[79] P.-G. Martinsson. A fast direct solver for a class of elliptic partial differential equations. *Journal of Scienti c Computing*, 38(3):316 330, 2009.

[80] G. L. Miller, S.-H. Teng, W. Thurston, and S. A. Vavasis. Geometric separators for finite-element meshes. *SIAM Journal on Scienti c Computing*, 19(2):364 386, 1998.

[81] A. Napov, X. S. Li, and M. Gu. An algebraic multifrontal preconditioner that exploits a low-rank property. Technical report LBNL-xxxxx, Lawrence Berkeley National Laboratory, 2013.

[82] S. Operto, J. Virieux, P. R. Amestoy, J.-Y. L'Excellent, L. Giraud, and H. Ben Hadj Ali. 3D finite-difference frequency-domain modeling of visco-acoustic wave propagation using a massively parallel direct solver: A feasibility study. *Geophysics*, 72(5):195 211, 2007.

[83] F. Pellegrini and J. Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *High-Performance Computing and Networking*, pages 493 498. Springer, 1996.

[84] R. G. Pratt. Seismic waveform inversion in the frequency domain, part I : theory and verification in a physic scale model. *Geophysics*, 64:888 901, 1999.

[85] J. L. Rigal and J. Gaches. On the compatibility of a given solution with the data of a linear system. *Journal of the ACM*, 14(3):543 548, 1967.

[86] F.-H. Rouet. *Memory and performance issues in parallel multifrontal factorizations and triangular solutions with sparse right-hand sides.* PhD thesis, Institut National Polytechnique de Toulouse, 2012.

[87] D. Ruiz. A scaling algorithm to equilibrate both row and column norms in matrices. Technical Report RAL-TR-2001-034, 2001. Also appeared as ENSEEIHT-IRIT report RT/APO/01/4.

[88] Y. Saad. *Iterative methods for sparse linear systems.* SIAM, 2003.

[89] Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scienti c and Statistical Computing*, 7(3):856 869, 1986.

[90] R. S. Schreiber. A new implementation of sparse Gaussian elimination. *ACM Transactions on Mathematical Software*, 8(3):256 276, 1982.

[91] J. Schulze. Towards a tighter coupling of bottom-up and top-down sparse matrix ordering methods. *BIT*, 41(4):800 841, 2001.

[92] R. D. Skeel. Scaling for numerical stability in Gaussian elimination. *Journal of the ACM*, 26(3):494 526, 1979.

[93] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. J. Dongarra. *MPI: The Complete Reference.* The MIT Press, Cambridge, Massachusetts, 1996.

[94] G. W. Stewart. *Matrix Algorithms: Volume 1, Basic Decompositions.* Society for Industrial Mathematics, 1998.

[95] M. N. Toksöz and D. H. Johnston. *Geophysics reprint series, No. 2: Seismic wave attenuation.* Society of exploration geophysicists, Tulsa, OK, 1981.

[96] R. Vandebril, M. Van Barel, G. Golub, and N. Mastronardi. A bibliography on semiseparable matrices. *Calcolo*, 42(3-4):249 270, 2005.

[97] R. Vandebril, M. Van Barel, and N. Mastronardi. *Matrix computations and semiseparable matrices: linear systems*, volume 1. JHU Press, 2007.

[98] J. Virieux and S. Operto. An overview of full waveform inversion in exploration geophysics. *Geophysics*, 74(6):1 26, 2009.

[99] S. Wang, M. V. de Hoop, and J. Xia. On 3D modeling of seismic wave propagation via a structured parallel multifrontal direct Helmholtz solver. *Geophysical Prospecting*, 59(5):857 873, 2011.

[100] S. Wang, M. V. de Hoop, J. Xia, and X. S. Li. Massively parallel structured multifrontal solver for time-harmonic elastic waves in 3D anisotropic media. *Geophysical Journal International*, 191:346 366, 2012.

[101] S. Wang, X. S. Li, J. Xia, Y. Situ, and M. V. de Hoop. Efficient scalable algorithms for hierarchically semiseparable matrices. *Submitted to SIAM Journal on Scienti c Computing*, 2012.

[102] S. Wang, X. S. Li, F.-H. Rouet, J. Xia, and M. V. V. de Hoop. A parallel geometric multifrontal solver using hierarchically semiseparable structure. *Submitted to ACM Transactions on Mathematical Software*, 2013.

[103] J. Xia. Efficient structured multifrontal factorization for general large sparse matrices. *SIAM Journal on Scienti c Computing*, 35:A832 A860, 2013.

[104] J. Xia. Randomized sparse direct solvers. *SIAM Journal on Matrix Analysis and Applications*, 34(1):197 227, 2013.

[105] J. Xia, S. Chandrasekaran, M. Gu, and X. S. Li. Superfast multifrontal method for large structured linear systems of equations. *SIAM Journal on Matrix Analysis and Applications*, 31(3):1382 1411, 2009.

[106] J. Xia, S. Chandrasekaran, M. Gu, and X. S. Li. Fast algorithms for hierarchically semiseparable matrices. *Numerical Linear Algebra with Applications*, 17(6):953 976, 2010.