Block Low-Rank multifrontal solvers: complexity, performance, and scalability

Théo Mary

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)*

Présentée et soutenue le *24 Novembre 2017* **par :**

THÉO MARY

Block Low-Rank multifrontal solvers: complexity, performance, and scalability

Solveurs multifrontaux exploitant des blocs de rang faible: complexité, performance et parallélisme

---

JURY

PATRICK AMESTOY	INP-IRIT	Directeur de thèse
CLEVE ASHCRAFT	LSTC	Examinateur
OLIVIER BOITEAU	EDF	Examinateur
ALFREDO BUTTARI	CNRS-IRIT	Codirecteur de thèse
IAIN DUFF	STFC-RAL	Examinateur
JEAN-YVES L'EXCELLENT	Inria-LIP	Invité
XIAOYE SHERRY LI	LBNL	Rapportrice
GUNNAR MARTINSSON	Univ. of Oxford	Rapporteur
PIERRE RAMET	Inria-LaBRI	Examinateur

École doctorale et spécialité :
MITT : Domaine STIC : Sureté de logiciel et calcul de haute performance

Unité de Recherche :
IRIT - UMR 5505

Directeur(s) de Thèse :
Patrick AMESTOY et *Alfredo BUTTARI*

Rapporteurs :
Xiaoye Sherry LI et *Gunnar MARTINSSON*

Remerciements
Acknowledgments

Tout d'abord, je voudrais remercier mes encadrants, Patrick Amestoy, Alfredo Buttari et Jean-Yves L'Excellent, qui ont tous les trois suivi mon travail de très près. Merci à vous trois d'avoir mis autant d'énergie à me former, à m'encourager et à me guider ces trois années durant. Cette aventure n'aurait pas été possible sans votre disponibilité, votre capacité d'écoute et de conseil, et votre intelligence tant scientifique qu'humaine ; j'espère que cette thèse le reflète. Travailler avec vous a été un honneur, une chance et surtout un grand plaisir ; j'espère que notre collaboration continuera encore longtemps. J'ai beaucoup appris de vous en trois ans, sur le plan scientifique bien sûr, mais aussi humain : votre sérieux, votre modestie, votre générosité et surtout votre sens fort de l'éthique m'ont profondément impacté. J'espère que je saurai à mon tour faire preuve des mêmes qualités dans mon avenir professionnel et personnel.

I am grateful to Sherry Li and Gunnar Martinsson for having accepted to report on my thesis, for their useful comments on the manuscript, and for travelling overseas to attend the defense. I also thank the other members of my jury, Cleve Ashcraft, Olivier Boiteau, Iain Duff, and Pierre Ramet.

I would like to thank all the researchers and scientists I interacted with since 2014. My first research experience took place in Tennessee, at the Innovative Computing Laboratory where I did my master thesis. I thank Jack Dongarra for hosting me in his team for five months. I especially thank Ichitaro Yamazaki for his patience and kindness. I also thank Jakub Kurzak, Piotr Luszczek, and Stanimire Tomov. I wish to thank Julie Anton, Cleve Ashcraft, Roger Grimes, Robert Lucas, François-Henry Rouet, Eugene Vecharynski, and Clément Weisbecker from the Livermore Software Technology Corporation for numerous fruitful scientific exchanges. I especially thank Cleve Ashcraft whose work in the field has been an endless source of inspiration, and Clément Weisbecker whose PhD thesis laid all the groundwork that made this thesis possible. I also thank Pieter Ghysels and Sherry Li for hosting me twice at the Lawrence Berkeley National Lab and for improving my understanding of HSS approximations. I would also like to thank Emmanuel Agullo, Marie Durand, Mathieu Faverge, Abdou Guermouche, Guillaume Joslin, Hatem Ltaief, Gilles Moreau, Grégoire Pichon, Pierre Ramet, and Wissam Sid-Lakhdar for many interesting discussions.

During this thesis, I had the great opportunity of interacting with scientists from the applications field. I also want to thank all of them: their eagerness to solve larger and larger problems is a constant source of motivation for us, linear algebraists. I am grateful to Stéphane Operto for hosting me at the Geoazur Institute in Sophia Antipolis for a week, and for taking the time to introducing me to the world of geophysics. I also thank Romain Brossier, Ludovic Métivier, Alain Miniussi, and Jean Virieux from SEISCOPE. Thanks to Paul Wellington, Okba Hamitou, and Borhan Tavakoli for a very fun SEG conference in New Orleans. Thanks to Stéphane Operto from SEISCOPE, Daniil Shantsev from EMGS, and Olivier Boiteau from EDF for providing the test matrices that were used in the experiments presented in this thesis. I also thank Benoit Lodej from ESI, Kostas Sikelis and Frédéric Vi from ALTAIR, and Eveline Rosseel from FFT for precious feedback on our BLR solver.

I thank Simon Delamare from LIP, Pierrette Barbaresco and Nicolas Renon from CALMIP, Boris Dintrans from CINES, and Cyril Mazauric from BULL for their reactivity, their amability, and their help in using the computers on which the numerical experiments presented in this thesis were run. Thanks to Nicolas Renon for various scientific exchanges.

I am grateful to all the people belonging to the Tolosean research community (IRIT, CERFACS, ISAE, ...) for making an active, dynamic, and always positive work environment. I especially thank Serge Gratton, Chiara Puglisi, and Daniel Ruiz for their kindness. Many thanks to Philippe Leleux who kindly took charge of the videoconference's technical support during my defense. I owe special thanks to my coworkers in F321, Florent Lopez, Clément Royer, and Charlie Vanaret, with whom I shared countless good times.

Je remercie mes amis de toujours, Loïc, Alison, Paco et Daniel, ainsi que ma famille pour leur soutien constant à travers toutes ces années, et pour le bonheur partagé avec eux.

Ringrazio Elisa per essere entrata nella mia vita nel momento in cui ne avevo più bisogno. Grazie per avermi supportato (e a volte sopportato !) durante la stesura di questa tesi e per farmi più felice ogni giorno.

Merci infiniment à ma maman, qui depuis mon plus jeune âge m'a toujours permis et encouragé à faire ce que j'aimais ; cette thèse en étant le meilleur exemple, je la lui dédie.

Résumé

Nous nous intéressons à l'utilisation d'approximations de rang faible pour réduire le coût des solveurs creux directs multifrontaux. Parmi les différents formats matriciels qui ont été proposés pour exploiter la propriété de rang faible dans les solveurs multifrontaux, nous nous concentrons sur le format Block Low-Rank (BLR) dont la simplicité et la flexibilité permettent de l'utiliser facilement dans un solveur multifrontal algébrique et généraliste. Nous présentons différentes variantes de la factorisation BLR, selon comment les mises à jour de rang faible sont effectuées, et comment le pivotage numérique est géré.

D'abord, nous étudions la complexité théorique du format BLR qui, contrairement à d'autres formats comme les formats hiérarchiques, était inconnue jusqu'à présent. Nous prouvons que la complexité théorique de la factorisation multifrontale BLR est asymptotiquement inférieure à celle du solveur de rang plein. Nous montrons ensuite comment les variantes BLR peuvent encore réduire cette complexité. Nous étayons nos bornes de complexité par une étude expérimentale.

Après avoir montré que les solveurs multifrontaux BLR peuvent atteindre une faible complexité, nous nous intéressons au problème de la convertir en gains de performance réels sur les architectures modernes. Nous présentons d'abord une factorisation BLR multithreadée, et analysons sa performance dans des environnements multicœurs à mémoire partagée. Nous montrons que les variantes BLR sont cruciales pour exploiter efficacement les machines multicœurs en améliorant l'intensité arithmétique et la scalabilité de la factorisation. Nous considérons ensuite à la factorisation BLR sur des architectures à mémoire distribuée.

Les algorithmes présentés dans cette thèse ont été implémentés dans le solveur MUMPS. Nous illustrons l'utilisation de notre approche dans trois applications industrielles provenant des géosciences et de la mécanique des structures. Nous comparons également notre solveur avec STRUMPACK, basé sur des approximations Hierarchically Semi-Separable. Nous concluons cette thèse en rapportant un résultat sur un problème de très grande taille (130 millions d'inconnues) qui illustre les futurs défis posés par le passage à l'échelle des solveurs multifrontaux BLR.

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

Abstract

We investigate the use of low-rank approximations to reduce the cost of sparse direct multifrontal solvers. Among the different matrix representations that have been proposed to exploit the low-rank property within multifrontal solvers, we focus on the Block Low-Rank (BLR) format whose simplicity and flexibility make it easy to use in a general purpose, algebraic multifrontal solver. We present different variants of the BLR factorization, depending on how the low-rank updates are performed and on the constraints to handle numerical pivoting.

We first investigate the theoretical complexity of the BLR format which, unlike other formats such as hierarchical ones, was previously unknown. We prove that the theoretical complexity of the BLR multifrontal factorization is asymptotically lower than that of the full-rank solver. We then show how the BLR variants can further reduce that complexity. We provide an experimental study with numerical results to support our complexity bounds.

After proving that BLR multifrontal solvers can achieve a low complexity, we turn to the problem of translating that low complexity in actual performance gains on modern architectures. We first present a multithreaded BLR factorization, and analyze its performance in shared-memory multicore environments on a large set of real-life problems. We put forward several algorithmic properties of the BLR variants necessary to efficiently exploit multicore systems by improving the arithmetic intensity and the scalability of the BLR factorization. We then move on to the distributed-memory BLR factorization, for which additional challenges are identified and addressed.

The algorithms presented throughout this thesis have been implemented within the MUMPS solver. We illustrate the use of our approach in three industrial applications coming from geosciences and structural mechanics. We also compare our solver with the STRUMPACK package, based on Hierarchically Semi-Separable approximations. We conclude this thesis by reporting results on a very large problem (130 millions of unknowns) which illustrates future challenges posed by BLR multifrontal solvers at scale.

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

Contents

List of most common abbreviations and symbols

BLAS	Basic Linear Algebra Subroutines
BLR	Block Low-Rank
CB	Contribution block
FR/LR	Full-rank/Low-rank
FSCU	Factor–Solve–Compress–Update (and similarly for other variants)
$\mathscr{H}$	Hierarchical
HBS	Hierarchically Block-Separable
HODLR	Hierarchically Off-Diagonal Low-Rank
HSS	Hierarchically Semi-Separable
LUAR	Low-rank updates accumulation and recompression
MF	Multifrontal
RL/LL	Right-looking/Left-looking
UFSMC	University of Florida Sparse Matrix Collection
$A_{i,j}$	$(i,j)-$th entry of matrix A
$A_{i,:}$	$i-$th row of matrix A
$A_{:,j}$	$j-$th column of matrix A
$[a,b]$	continuous interval from a to b
$[a;b]$	discrete interval from a to b

List of Algorithms

List of Tables

List of Figures

Introduction

We are interested in efficiently computing the solution of a large sparse system of linear equations:

$$Ax = b,$$

where A is a square sparse matrix of order n, and x and b are the unknown and right-hand side vectors. This work focuses on the solution of this problem by means of direct, factorization-based methods, and in particular based on the multifrontal approach (Duff and Reid, 1983).

Direct methods are widely appreciated for their numerical robustness, reliability, and ease of use. However, they are also characterized by their high computational complexity: for a three-dimensional problem, the total amount of computations and the memory consumption are proportional to $\mathcal{O}(n^2)$ and $\mathcal{O}(n^{4/3})$, respectively (George, 1973). This limits the scope of direct methods on very large problems (matrices with hundreds of millions of unknowns).

The goal of this work is to reduce the cost of sparse direct solvers without sacrificing their robustness, ease of use, and performance.

In numerous scientific applications, such as the solution of partial differential equations, the matrices resulting from the discretization of the physical problem have been shown to possess a low-rank property (Bebendorf, 2008): well-defined off-diagonal blocks B of their Schur complements can be approximated by low-rank products $\widetilde{B} = XY^T \approx B$. This property can be exploited in multifrontal solvers to provide a substantial reduction of their complexity.

Several matrix representations, so-called low-rank formats, have been proposed to exploit this property within multifrontal solvers. The $\mathcal{H}$-matrix format (Hackbusch, 1999), where $\mathcal{H}$ stands for hierarchical, has been widely studied in the literature, as well as its variants $\mathcal{H}^2$ (Börm, Grasedyck, and Hackbusch, 2003), HSS (Xia, Chandrasekaran, Gu, and Li, 2010), and HODLR (Aminfar, Ambikasaran, and Darve, 2016).

In the hierarchical framework, the matrix is hierarchically partitioned in order to maximize the low-rank compression rate; this can lead to a theoretical complexity of the multifrontal factorization as low as $\mathcal{O}(n)$, both in flops and memory consumption. However, because of the hierarchical structure of the matrix, it is not straightforward to use in a general purpose, algebraic "black-box" solver, and to achieve high performance.

Alternatively, a so-called Block Low-Rank (BLR) format has been put forward by Amestoy et al. (2015a). In the BLR framework, the matrix is partitioned by means of a flat, non-hierarchical blocking of the matrix. The simpler structure of the BLR format makes it easy to use in a parallel, algebraic solver. Amestoy et al. (2015a) also introduced the "standard" BLR factorization variant, which can easily handle numerical pivoting, a critical feature often lacking in other low-rank solvers.

Despite these advantages, the BLR format was long dismissed due to its unknown theoretical complexity; it was even conjectured it could asymptotically behave as the full-rank $\mathcal{O}(n^2)$ solver. One of the main contributions of this thesis is to investigate the complexity of the BLR format and to prove it is in fact asymptotically lower than $\mathcal{O}(n^2)$. We show that the theory for hierarchical matrices does not provide a satisfying result when applied to BLR matrices (thereby justifying the initial pessimistic conjecture). We extend the theory to compute the theoretical complexity of the BLR multifrontal factorization. We show that the standard BLR variant of Amestoy et al. (2015a) can lead to a complexity as low as $\mathcal{O}(n^{5/3})$ in flops and $\mathcal{O}(n \log n)$ in memory. Furthermore, we introduce new BLR variants that can further reduce the flop complexity, down to $\mathcal{O}(n^{4/3})$. We provide an experimental study with numerical results to support our complexity bounds.

The modifications introduced by the BLR variants can be summarized as follows:

- During the factorization, the sum of many low-rank matrices arises in the update of the trailing submatrix. We propose an algorithm, referred to as low-rank updates accumulation and recompression (LUAR), to accumulate and recompress these low-rank updates together, which improves both the performance and complexity of the factorization. We provide an in-depth analysis of the different recompression strategies that can be considered.

- The compression can be performed at different stages of the BLR factorization. In the standard variant, it is performed relatively late so that only part of the operations are accelerated. We propose novel variants that perform the compression earlier in order to further reduce the complexity of the factorization. In turn, we also show that special care has to be taken to maintain the ability to perform numerical pivoting.

Our complexity analysis, together with the improvements brought by the BLR variants, therefore shows that BLR multifrontal solvers can achieve a low theoretical complexity.

However, achieving low complexity is only half the work necessary to tackle increasingly large problems. In a context of rapidly evolving architectures, with an increasing number of computational resources, translating the complexity reduction into actual performance gains on modern architectures is a challenging problem. We first present a multithreaded BLR factorization, and analyze its performance in shared-memory multicore environments on a large set of problems coming from a variety of real-life applications. We put forward several algorithmic properties of the BLR variants necessary to efficiently exploit multicore systems by improving the efficiency and scalability of the BLR factorization. We then present and ana-

lyze the distributed-memory BLR factorization, for which additional challenges are identified; some solutions are proposed.

The algorithms presented throughout this thesis have been implemented within the MUMPS (Amestoy, Duff, Koster, and L'Excellent, 2001; Amestoy, Guermouche, L'Excellent, and Pralet, 2006; Amestoy et al., 2011) solver. We illustrate the use of our approach in three industrial applications coming from geosciences and structural mechanics. We also compare our solver with the STRUMPACK (Rouet, Li, Ghysels, and Napov, 2016; Ghysels, Li, Rouet, Williams, and Napov, 2016; Ghysels, Li, Gorman, and Rouet, 2017) solver, based on Hierarchically Semi-Separable approximations, to shed light on the differences between these formats. We compare their usage as accurate, high precision direct solvers and as more approximated, fast preconditioners coupled with an iterative solver. Finally, we conclude this thesis by discussing some future challenges that await BLR solvers for large-scale systems and applications. We propose some ways to tackle these challenges, and illustrate them by reporting results on very large problems (up to 130 million unknowns).

The remainder of this thesis is organized as follows. In order to make the thesis self-contained, we provide in Chapter 1 general background on numerical linear algebra, and more particularly sparse direct methods and low-rank approximations. Chapter 2 is central to this work. It articulates and describes in detail the Block Low-Rank (multifrontal) factorization in all its variants. Chapter 3 provides an in-depth analysis of the different strategies to perform the so-called LUAR algorithm. Chapter 4 deals with the theoretical aspects regarding the complexity of the BLR factorization, including the proof that it is asymptotically lower than that of the full-rank solver, together with its computation for all BLR variants and experimental validation. Chapters 5 and 6 focus on the performance of the BLR factorization on shared-memory (multicore) and distributed-memory architectures, respectively. The BLR variants are shown to improve the performance and scalability of the factorization. Then, two real-life applications which benefit from BLR approximations are studied in Chapter 7. Chapter 8 provides an experimental comparison with the HSS solver STRUMPACK. Chapter 9 tackles the solution of a very large problem, to show the remaining challenges of BLR solvers at scale. Finally, we conclude the manuscript by summarizing the main results, and mentioning some perspectives.

CHAPTER

1 General Background

We are interested in efficiently computing the solution of a large sparse system of linear equations

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

where A is a square 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 this chapter, we give an overview of existing methods to solve this problem and provide some background that will be useful throughout the thesis. To achieve this objective, we consider an illustrative example (2D Poisson's equation) and guide the reader through the steps of its numerical solution.

1.1 An illustrative example: PDE solution

We consider the solution of Poisson's equation in $\mathbb{R}^d$

$$\Delta u = f, \tag{1.2}$$

where Δ is the Laplace operator, also noted ∇^2, defined in two-dimensional Cartesian coordinates as

$$\Delta u(x,y) = \left(\frac{\partial^2}{\partial^2 x} + \frac{\partial^2}{\partial^2 y}\right) u(x,y). \tag{1.3}$$

Equation (1.2) is therefore a partial derivative equation (PDE).

While this particular equation has an analytical solution on simple domains, our purpose here is to use it as an illustrative example to describe the solution of general PDEs on more complicated domains. We are thus interested in computing an approximate solution restricted to a subset of discrete points. Two widely used approaches to compute such a solution are the finite-difference method and the finite-element method, which consist in discretizing the system using a subset of points (or polygons) forming a mesh. In the following, we consider as an example a finite-difference discretization resulting in a 5×5 regular equispaced mesh, as illustrated in Figure 1.1.

The function u is then approximated by a vector, also noted u, whose elements $u_{i,j}$ correspond to the value of u at the mesh points (i,j). f is similarly approxi-

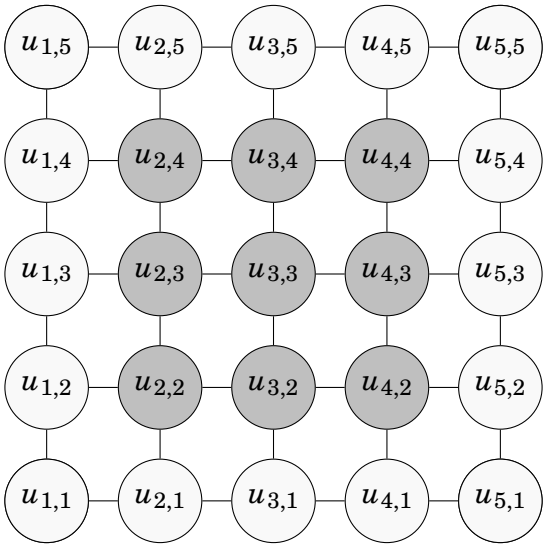


Figure 1.1 – Example of a 5-point stencil finite-difference mesh. The boundary nodes (light color) are used to compute the approximate derivatives at the interior nodes (dark color) but are not part of the linear system solved.

mated. The next step is to approximate the partial derivatives of u at a given point (x,y) of the mesh, which can be done (using the central step method) as follows:

$$\frac{\partial}{\partial x}u(x,y) \approx \frac{u(x+h,y)-u(x-h,y)}{2h}, \tag{1.4}$$

$$\frac{\partial}{\partial y}u(x,y) \approx \frac{u(x,y+h)-u(x,y-h)}{2h}, \tag{1.5}$$

where the grid size is h in both dimensions, and should be taken small enough for the approximation to be accurate. Furthermore, second order derivatives can also be approximated:

$$\frac{\partial^2}{\partial^2 x}u(x,y) \approx \frac{u(x+h,y)-2u(x,y)+u(x-h,y)}{h^2}, \tag{1.6}$$

$$\frac{\partial^2}{\partial^2 y}u(x,y) \approx \frac{u(x,y+h)-2u(x,y)+u(x,y-h)}{h^2}. \tag{1.7}$$

Therefore, the Laplacian operator in two dimensions can be approximated as

$$\Delta u(x,y) \approx \frac{1}{h^2}\left(u(x+h,y)+u(x-h,y)+u(x,y+h)+u(x,y-h)-4u(x,y)\right), \tag{1.8}$$

which is known as the five-point stencil finite-difference method.

Equation (1.2) can thus be approximated by the discrete Poisson equation

$$(\Delta u)_{i,j} = \frac{1}{h^2}(u_{i+1,j}+u_{i-1,j}+u_{i,j+1}+u_{i,j-1}-4u_{i,j}) = f_{i,j}, \tag{1.9}$$

where $i,j \in [2;N-1]$; N is the number of grid points. Note that the approximation of the Laplacian at node (i,j) requires the values of u at the neighbors in all directions;

therefore, in practice, the boundary nodes are prescribed and the equation is solved for the interior points only. In this case, equation (1.9) is equivalent to a linear system $Au = g$ where u contains the interior nodes (shaded nodes in Figure 1.1), and A is a block-diagonal matrix of order $(N-2)^2$ of the form

$$A = \begin{bmatrix} D & -I & 0 & \cdots & 0 \\ -I & D & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & D & -I \\ 0 & \cdots & 0 & -I & D \end{bmatrix}, \tag{1.10}$$

where I is the identity matrix of order $N-2$ and D, also of order $N-2$, is a tridiagonal matrix of the form

$$D = \begin{bmatrix} 4 & -1 & 0 & \cdots & 0 \\ -1 & 4 & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & 4 & -1 \\ 0 & \cdots & 0 & -1 & 4 \end{bmatrix}. \tag{1.11}$$

Finally, we have $g = -h^2 f + \beta$, where β contains the boundary nodes information.

For example, with the mesh of Figure 1.1, we obtain the following 9×9 linear system:

$$\begin{bmatrix} 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 \\ 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 \end{bmatrix} \begin{bmatrix} u_{2,2} \\ u_{3,2} \\ u_{4,2} \\ u_{2,3} \\ u_{3,3} \\ u_{4,3} \\ u_{2,4} \\ u_{3,4} \\ u_{4,4} \end{bmatrix} = \begin{bmatrix} -h^2 f_{2,2} + u_{1,2} + u_{2,1} \\ -h^2 f_{3,2} + u_{3,1} \\ -h^2 f_{4,2} + u_{5,2} + u_{4,1} \\ -h^2 f_{2,3} + u_{1,3} \\ -h^2 f_{3,3} \\ -h^2 f_{4,3} + u_{5,3} \\ -h^2 f_{2,4} + u_{1,4} + u_{2,5} \\ -h^2 f_{3,4} + u_{3,5} \\ -h^2 f_{4,4} + u_{5,4} + u_{4,5} \end{bmatrix}. \tag{1.12}$$

We now discuss how to solve such a linear system. There are two main classes of methods:

- *Iterative* methods build a sequence of iterates x_k which hopefully converges towards the solution. Although they are relatively cheap in terms of memory and computations, their effectiveness strongly depends on the ability to find a good preconditioner to ensure convergence.
- *Direct* methods build a factorization of matrix A (e.g. $A = LU$ or $A = QR$) to solve directly the system. While they are commonly appreciated for their numerical robustness, reliability, and ease of use, they are however also characterized by a large amount of memory consumption and computations.

In this thesis, we focus on direct methods based on Gaussian elimination, i.e. methods that factorize A as LU (general case), LDL^T (symmetric indefinite case),

or LL^T (symmetric positive case, also known as Cholesky factorization). We first give an overview of these methods in the dense case.

1.2 Dense LU or LDL^T factorization

1.2.1 Factorization phase

1.2.1.1 Point, block, and tile algorithms

The LU factorization of a dense matrix A, described in Algorithm 1.1, computes the decomposition $A = LU$, where L is unit lower triangular and U is upper triangular. At each step k of the factorization, a new column of L and a new row of U are computed; we call the diagonal entry $A_{k,k}$ a *pivot* and step k its *elimination*.

Algorithm 1.1 Dense LU (Right-looking) factorization (without pivoting)

1: /* **Input:** a matrix A of order n */
2: **for** $k = 1$ **to** $n-1$ **do**
3: $\quad A_{k+1:n,k} \leftarrow A_{k+1:n,k}/A_{k,k}$
4: $\quad A_{k+1:n,k+1:n} \leftarrow A_{k+1:n,k+1:n} - A_{k+1:n,k}A_{k,k+1:n}$
5: **end for**

Algorithm 1.1 is referred to as *in-place* because A is overwritten during the factorization: its lower triangular part is replaced by L and its upper triangular part by U (note that its diagonal contains the diagonal of U, as the diagonal of L is not explicitly stored since $L_{i,i} = 1$).

Algorithm 1.1 is also referred to as *point* because the operations are performed on single entries of the matrix. This means the factorization is mainly performed with BLAS-2 operations. Its performance can be substantially improved (typically by an order of magnitude) by using BLAS-3 operations instead, which increase data locality (i.e. cache reuse) (Dongarra, Du Croz, Hammarling, and Duff, 1990). For this purpose, the operations in Algorithm 1.1 can be reorganized to be performed on blocks of entries: the resulting algorithm is referred to as *block* LU factorization and is described in Algorithm 1.2.

Algorithm 1.2 Dense block LU (Right-looking) factorization (without pivoting)

1: /* **Input:** a $p \times p$ block matrix A of order n; $A = [A_{i,j}]_{i,j \in [1;p]}$ */
2: **for** $k = 1$ **to** p **do**
3: $\quad$ Factor: $A_{k,k} \leftarrow L_{k,k}U_{k,k}$
4: $\quad$ Solve (L): $A_{k+1:p,k} \leftarrow A_{k+1:p,k}U_{k,k}^{-1}$
5: $\quad$ Solve (U): $A_{k,k+1:p} \leftarrow L_{k,k}^{-1}A_{k,k+1:p}$
6: $\quad$ Update: $A_{k:p,k:p} \leftarrow A_{k:p,k:p} - A_{k:p,k}A_{k,k:p}$
7: **end for**

The block LU factorization consists of three main steps: Factor, Solve, and Update. The Factor step is performed by means of a point LU factorization (Algorithm 1.1). The Solve step takes the form of a triangular solve (so-called `trsm` kernel), while the Update step takes the form of a matrix-matrix multiplication

(so-called gemm kernel). Therefore, both the Solve and Update rely on BLAS-3 operations.

The LU factorization can be accelerated when several cores are available by using multiple threads. Both the point and block LU factorizations, as implemented for example in LAPACK (Anderson et al., 1995), solely rely on multithreaded BLAS kernels to multithread the factorization.

More advanced versions (Buttari, Langou, Kurzak, and Dongarra, 2009; Quintana-Ortí, Quintana-Ortí, Geijn, Zee, and Chan, 2009) of Algorithm 1.2 have been designed by decomposing the matrix into *tiles*, where each tile is stored contiguously in memory. This algorithm, described in Algorithm 1.3, is referred to as tile LU factorization, and is usually associated with a task-based multithreading which fully takes advantage of the independencies between computations. In this work, we will not consider task-based factorizations but will discuss tile factorizations because they are the starting point to the BLR factorization algorithm, as explained in Chapter 2.

Algorithm 1.3 Dense tile LU (Right-looking) factorization (without pivoting)

/* **Input:** a $p \times p$ tile matrix A of order n; $A = [A_{i,j}]_{i,j \in [1;p]}$ */
for $k = 1$ **to** p **do**
  Factor: $A_{k,k} \leftarrow L_{k,k}U_{k,k}$
  for $i = k+1$ **to** p **do**
    Solve (L): $A_{i,k} \leftarrow A_{i,k}U_{k,k}^{-1}$
    Solve (U): $A_{k,i} \leftarrow L_{k,k}^{-1}A_{k,i}$
  end for
  for $i = k+1$ **to** p **do**
    for $j = k+1$ **to** p **do**
      Update: $A_{i,j} \leftarrow A_{i,j} - A_{i,k}A_{k,j}$
    end for
  end for
end for

For the sake of conciseness, in the following, we will present the algorithms in their tile version. Unless otherwise specified, the discussion also applies to block algorithms.

The number of operations and memory required to perform the LU factorization is independent of the strategy used (point, block, or tile) and are equal to $\mathcal{O}(n^3)$ and $\mathcal{O}(n^2)$, respectively (Golub and Van Loan, 2012).

1.2.1.2 Right-looking vs Left-looking factorization

Algorithms 1.1, 1.2, and 1.3 are referred to as *right-looking*, in the sense that as soon as column k is eliminated, the entire trailing submatrix (columns to its "right") is updated. These algorithms can be rewritten in a *left-looking* form, where at each step k, column k is updated using all the columns already computed (those at its "left") and then eliminated.

The tile version of the Left-looking LU factorization is provided in Algorithm 1.4 (the point and block versions are omitted for the sake of conciseness).

Algorithm 1.4 Dense tile LU (Left-looking) factorization (without pivoting)

```
1: /* Input: a p × p block matrix A of order n; A = [A_{i,j}]_{i,j∈[1;p]} */
2: for k = 1 to p do
3:    for i = k to p do
4:       for j = 1 to k − 1 do
5:          Update (L): A_{i,k} ← A_{i,k} − A_{i,j}A_{j,k}
6:          if i ≠ k then
7:             Update (U): A_{k,i} ← A_{k,i} − A_{k,j}A_{j,i}
8:          end if
9:       end for
10:   end for
11:   Factor: A_{k,k} ← L_{k,k}U_{k,k}
12:   for i = k + 1 to p do
13:      Solve (L): A_{i,k} ← A_{i,k}U^{-1}_{k,k}
14:      Solve (U): A_{k,i} ← L^{-1}_{k,k}A_{k,i}
15:   end for
16: end for
```

1.2.1.3 Symmetric case

In the symmetric case, the matrix is decomposed in LDL^T, where L is a unit lower triangular matrix and D a diagonal (or block-diagonal, in the case of pivoting, as explained in Section 1.2.4.1) matrix.

When the matrix is positive definite, it can even be decomposed in LL^T (where L is not unit anymore), commonly referred to as Cholesky factorization. In this thesis, we will consider the more general symmetric indefinite case (LDL^T decomposition), but the discussion also applies to the Cholesky decomposition.

Algorithm 1.5 Dense tile LDL^T (Right-looking) factorization (without pivoting)

```
1: /* Input: a p × p tile matrix A of order n; A = [A_{i,j}]_{i,j∈[1;p]} */
2: for k = 1 to p do
3:    Factor: A_{k,k} ← L_{k,k}D_{k,k}L^T_{k,k}
4:    for i = k + 1 to p do
5:       Solve: A_{i,k} ← A_{i,k}L^{-T}_{k,k}D^{-1}_{k,k}
6:    end for
7:    for i = k + 1 to p do
8:       for j = k + 1 to i do
9:          Update: A_{i,j} ← A_{i,j} − A_{i,k}A^T_{j,k}
10:      end for
11:   end for
12: end for
```

In Algorithm 1.5, we present the tile LDL^T right-looking factorization. The other versions (block, left-looking) are omitted for the sake of conciseness. The algorithm is similar to the unsymmetric case, with the following differences:

- The Factor step is now a LDL^T factorization.

- The Solve step still takes the form of a triangular solve, but also involves a column scaling with the inverse of D.

- Finally, since the matrix is symmetric, the Update is only done on its lower triangular part. In particular, note that in Algorithm 1.5, for the sake of simplicity, the diagonal blocks $A_{i,i}$ are entirely updated (line 9) when they are in fact lower triangular, which would thus result in additional flops. One could instead update each column one by one, minimizing the flops, but this would be very inefficient as it consists of BLAS-1 operations. In practice, one can find a compromise between flops and efficiency by further refining the diagonal blocks, which leads to a "staircase" update, as illustrated in Figure 1.2.

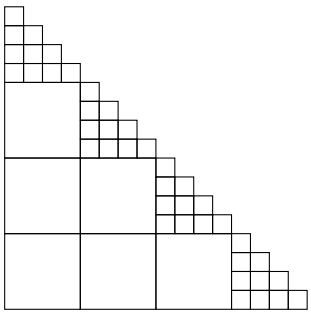


Figure 1.2 – Part of the matrix that is updated in the symmetric case.

1.2.2 Solution phase

Once matrix A has been decomposed under the form LU, computing the solution x of equation (1.1) is equivalent to solve two linear systems:

$$Ax = LUx = b \Leftrightarrow \begin{cases} Ly = b \\ Ux = y \end{cases}, \tag{1.13}$$

which can be achieved easily thanks to the special triangular form of L and U. The two solves $Ly = b$ and $Ux = y$ are respectively referred to as *forward elimination* and *backward substitution*. They are described, in their tile version, in Algorithm 1.6.

Note that the right- and left-looking distinction also applies for the solution phase. In Algorithm 1.6, the forward elimination is written in its right-looking version while the backward substitution is written in left-looking.

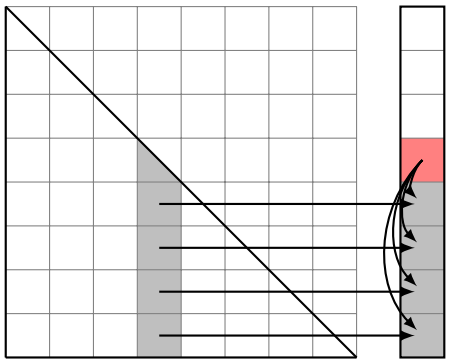
1.2.3 Numerical stability

Once the solution x is computed, we may want to evaluate its quality. Indeed, computations performed on computers are inexact: they are subject to *roundoff errors* due to the floating-point representation of numbers. Thus, the linear system that is solved is in fact

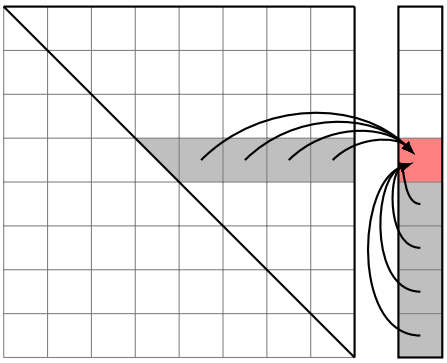
$$(A + \delta A)(x + \delta x) = b + \delta b, \tag{1.14}$$

Algorithm 1.6 Dense tile LU solution (without pivoting)

/* **Input:** a $p \times p$ block matrix A of order n; $A = LU = [A_{i,j}]_{i,j \in [1;p]}$; a vector b of size n. */
Forward elimination ($Ly = b$):
$y \leftarrow b$
for $j = 1$ **to** p **do**
  $y_j \leftarrow L_{j,j}^{-1} y_j$
  for $i = j + 1$ **to** p **do**
    $y_i \leftarrow L_{i,j} y_j$
  end for
end for
Backward substitution ($Ux = y$):
$x \leftarrow y$
for $i = p$ **to** 1 **by** -1 **do**
  for $j = i + 1$ **to** p **do**
    $x_i \leftarrow U_{i,j} x_j$
  end for
  $x_i \leftarrow U_{i,i}^{-1} x_i$
end for



(a) Forward elimination (right-looking).



(b) Backward substitution (left-looking).

Figure 1.3 – Dense triangular solution algorithms.

where δA, δx, and δb are called *perturbations*. An important aspect of any algorithm is to evaluate its *stability*. An algorithm is said to be stable if small perturbations lead to a small error. Of course, this depends on how the error is measured.

A first metric is to measure the quality of the computed solution $\tilde{x}$ with respect to the exact solution x, referred to as the *forward error* metric:

$$\frac{\|x - \tilde{x}\|}{\|x\|}. \tag{1.15}$$

The forward error can be large for two reasons: an unstable algorithm; or an ill-conditioned matrix (i.e., a matrix whose condition number is large), in which case even a stable algorithm can lead to a poor forward error.

To distinguish these two cases, Wilkinson (1963) introduces the *backward error* metric, which measures the smallest perturbation δA such that $\tilde{x}$ is the exact solution of the perturbed system $(A + \delta A)\tilde{x} = b$. The normwise backward error can be evaluated as (Rigal and Gaches, 1967):

$$\frac{\|A\tilde{x} - b\|}{\|A\|\|\tilde{x}\| + \|b\|}, \tag{1.16}$$

which does not depend on the conditioning of the system. Its componentwise version (Oettli and Prager, 1964) will also be of interest in the sparse case:

$$\max_i \frac{|A\tilde{x} - b|_i}{(|A||\tilde{x}| + |b|)_i}. \tag{1.17}$$

In this manuscript, we will use the (normwise and componentwise) backward error to measure the accuracy of our algorithms and we will refer to it as (normwise and componentwise) *scaled residual*.

It turns out Algorithm 1.1 is not backward stable. In particular, at step k, if $A_{k,k} = 0$, the algorithm will fail since line 3 would take the form of a division by zero. Furthermore, even if $A_{k,k}$ is not zero but is very small in amplitude, Algorithm 1.1 will lead to significant roundoff errors by creating too large off-diagonal entries in column and row k. This is measured by the *growth factor*.

One can partially overcome this issue by preprocessing the original matrix, e.g. by scaling the matrix so that its entries are of moderate amplitude (see Skeel (1979) and, for the sparse case, Duff and Pralet (2005) and Knight, Ruiz, and Uçar (2014)). However, in many cases, preprocessing strategies are not sufficient to ensure the numerical stability of the algorithm. In these cases, we need to perform *numerical pivoting*.

1.2.4 Numerical pivoting

The objective of numerical pivoting is to limit the growth factor by avoiding small pivots. The most conservative option is thus to select, at each step k, the entry $A_{i,j}$ in the trailing submatrix of maximal amplitude:

$$A_{i,j} = \max_{i',j' \in [k;n]} |A_{i',j'}|, \tag{1.18}$$

and to permute row k with row i and column k with column j. This is referred to as *complete pivoting* (Wilkinson, 1961). It is however rarely used since it implies a significant amount of searching and swapping which can degrade the performance of the factorization. A more popular strategy, known as row or column *partial pivoting*, consists in restricting the search of the pivot on row k or column k, respectively:

$$A_{k,j} = \max_{j' \in [k;n]} |A_{k,j'}| \text{ or } A_{i,k} = \max_{i' \in [k;n]} |A_{i',k}|, \tag{1.19}$$

and permute column k with column j, or row k with row i, respectively. In the literature, as well as in reference implementations such as LAPACK (Anderson et

al., 1995), column partial pivoting is often considered, and we will thus also consider it in the rest of this section, for the sake of clarity.

Partial pivoting can still lead to significant amounts of swapping and is therefore sometimes relaxed by choosing a threshold τ and accepting some pivot $A_{j,k}$ if

$$|A_{j,k}| \geq \tau \max_{i' \in [k;n]} |A_{i',k}|. \tag{1.20}$$

This strategy, referred to as *threshold partial pivoting* (Duff, Erisman, and Reid, 1986), is often enough to ensure the stability of the factorization, typically with a threshold of the order $\tau = 0.1$ or $\tau = 0.01$. It is of particular interest in the sparse case as explained in Section 1.3.

1.2.4.1 Symmetric case

In the symmetric case, special care must be taken to avoid losing the symmetry of A by considering symmetric permutations only. However, in general, it is not always possible to find a safe pivot with symmetric permutations only. For example, consider the following symmetric matrix:

$$A = \begin{pmatrix} 0 & 2 & 1 \\ 2 & 0 & 2 \\ 1 & 2 & 0 \end{pmatrix}.$$

At step 1, $A_{1,1} = 0$ is not an acceptable pivot. Thus row 1 must be exchanged with row 2 or 3. In both cases, the symmetry of A is lost. For example, if rows 1 and 2 are exchanged, the resulting matrix is

$$\begin{pmatrix} 2 & 0 & 2 \\ 0 & 2 & 1 \\ 1 & 2 & 0 \end{pmatrix},$$

which is not symmetric anymore, since $A_{1,3} \neq A_{3,1}$ (and $A_{2,3} \neq A_{3,2}$).

To ensure the stability of the factorization while maintaining the symmetry, *two-by-two* pivots (noted 2×2 pivots hereinafter) must be used (Bunch and Parlett, 1971). For example, the previous matrix can be factored as

$$A = LDL^T = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & \frac{1}{2} & 1 \end{pmatrix} \begin{pmatrix} 0 & 2 & 0 \\ 2 & 0 & 0 \\ 0 & 0 & -2 \end{pmatrix} \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & \frac{1}{2} \\ 0 & 0 & 1 \end{pmatrix},$$

where D is made of one 2×2 pivot, $\begin{pmatrix} 0 & 2 \\ 2 & 0 \end{pmatrix}$, and one 1×1 pivot, -2. Thus D is not diagonal anymore but block-diagonal.

Many strategies have been proposed in the literature to search and choose 2×2 pivots. The Bunch-Parlett algorithm (Bunch and Parlett, 1971) uses a complete pivoting strategy to scan the entire trailing submatrix at each step to search the best possible 2×2 pivot; it is a backward stable algorithm and also leads to bounded entries in L, but it is also expensive and requires a point factorization to be used. On

the contrary, the Bunch-Kaufman (Bunch and Kaufman, 1977) algorithm is based on partial pivoting, scanning at most two columns/rows at each step, which makes it cheaper and allows the use of block or tile factorizations; it is a backward stable algorithm but may lead to unbounded entries in L. For this reason, Ashcraft, Grimes, and Lewis (1998) propose two variants in between the previous two algorithms, the bounded Bunch-Kaufman and the fast Bunch-Parlett algorithms, which trade off some of Bunch-Kaufman's speed to make the entries in L bounded. The extension of these algorithms to the sparse case is discussed in Section 1.3.2.6.

Note that all previously mentioned strategies seek to decompose A as LDL^T. An alternative LTL^T decomposition, where T is a tridiagonal matrix, has been proposed by Parlett and Reid (1970) and Aasen (1971). We do not discuss it in this work.

1.2.4.2 LU and LDL^T factorization algorithm with partial pivoting

To perform numerical pivoting during the LU or LDL^T factorization, the algorithms presented above (e.g. Algorithms 1.3 and 1.5) must be modified. Indeed, in the block or tile versions, both the Solve and the Update step are performed on blocks/tiles to use BLAS-3 operations. However, this requires the Solve step to be performed after an entire block/tile has been factored, and therefore numerical pivoting can only be performed inside the diagonal block/tile. This strategy is referred to as restricted pivoting, and is discussed in the sparse context in Section 1.3.2.6.

To perform standard, non-restricted partial pivoting, each column of the current panel must be updated each time a new pivot is eliminated. Therefore, the Factor and Solve steps must be merged together in a Factor+Solve step. This step is described in Algorithm 1.8, and the resulting algorithm in Algorithm 1.7.

Algorithm 1.7 Dense tile LU factorization (with pivoting)

```
1: /* Input: a p × p tile matrix A of order n; A = [A_{i,j}]_{i,j∈[1;p]} */
2: for k = 1 to p do
3:    Factor+Solve: A_{k:p,k} ← L_{k:p,k} U_{k,k}
4:    for i = k + 1 to p do
5:       Solve (U): A_{k,i} ← L_{k,k}^{-1} A_{k,i}
6:    end for
7:    for i = k + 1 to p do
8:       for j = k + 1 to p do
9:          Update: A_{i,j} ← A_{i,j} − A_{i,k} A_{k,j}
10:      end for
11:   end for
12: end for
```

As a consequence, while the Update step remains in BLAS-3, the Solve step is now based on BLAS-2 operations instead. This leads to two contradictory objectives: on one hand, the panel size should be small so as to reduce the part of BLAS-2 computations; on the other hand, it should be big enough so that the Update operation is of high granularity. To find a good compromise between these two objectives, a possible strategy is to use double panels (i.e. two levels of blocking): the (small)

Algorithm 1.8 Factor+Solve step (unsymmetric case)

1: /* **Input:** a panel A with n_r rows and n_c columns */
2: **for** $k = 1$ **to** n_c **do**
3: $\quad i \leftarrow \arg\max_{i' \in [k;n_r]} |A_{i',k}|$
4: $\quad$ Swap rows k and i
5: $\quad A_{k+1:n_r,k} \leftarrow A_{k+1:n_r,k} / A_{k,k}$
6: $\quad A_{k+1:n_r,k+1:n_c} \leftarrow A_{k+1:n_r,k+1:n_c} - A_{k+1:n_r,k} A_{k,k+1:n_c}$
7: **end for**

inner panels are factored in BLAS-2; once an inner panel is fully factored, the corresponding update is applied inside the current outer panel (which corresponds to the original block/tile panel); once the (big) outer panel is factored, the entire sub-trailing matrix can be updated with a high granularity operation. This strategy can be generalized to a greater number of levels (Gustavson, 1997). This strategy is not presented in Algorithm 1.8 for the sake of simplicity.

1.2.4.3 Swapping strategy: LINPACK vs LAPACK style

There are two ways to perform the row swaps, commonly referred to as LINPACK and LAPACK styles of pivoting. At step k, assume $A_{i,k}$ has been selected as pivot: row k must then be swapped with row i.

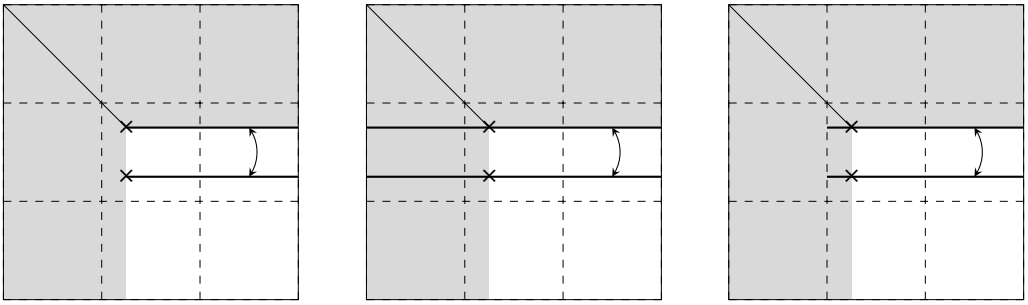
- In LINPACK (Dongarra, Bunch, Moler, and Stewart, 1979), only $A_{k,k:n}$ and $A_{i,k:n}$ are swapped, i.e. the subpart of the rows that is yet to be factored. This results in a series of Gauss transformations interlaced with matrix permutations that must be applied in the same order during the solution phase.

- In LAPACK (Anderson et al., 1995), the entire rows $A_{k,1:n}$ and $A_{i,1:n}$, including the already computed factors, are swapped. This results in a series of transformations that can be expressed as $PA = LU$.

- Finally, the two styles can be combined into a hybrid style in the case of a blocked factorization, where the LAPACK style is used inside the block while the LINPACK style is used outside. This option is of particular interest when only the current panel is accessible (e.g., in the context of an out-of-core execution (Agullo, Guermouche, and L'Excellent, 2010), or, as we will discuss in Sections 2.2.3 and 9.2, in the context of the BLR solution phase).

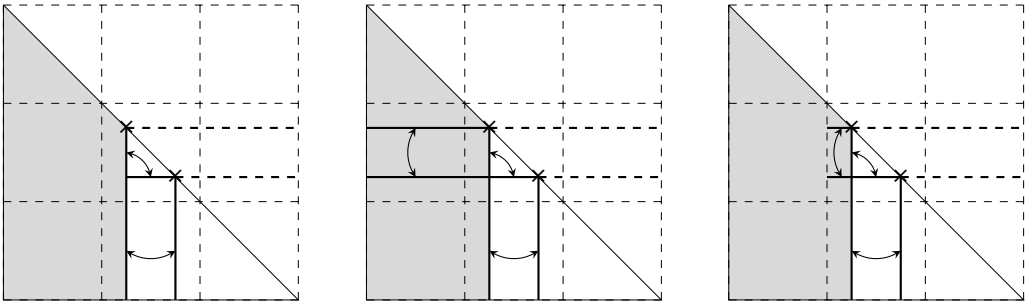
These three different styles are illustrated in Figures 1.4 and 1.5 in the unsymmetric and symmetric cases, respectively.

In the dense case, the same number of permutations is performed in any style. The LINPACK style does not allow the solution phase to be performed using BLAS-3 kernels and is thus generally avoided. The LAPACK style and the hybrid style (with a big enough block size) can both exploit BLAS-3 kernels and can be expected to perform comparably. This does not remain true in the sparse case, as we will explain in Section 1.3.2.7.



(a) LINPACK style (b) LAPACK style (c) Hybrid style

Figure 1.4 – LINPACK and LAPACK pivoting styles (unsymmetric case).



(a) LINPACK style (b) LAPACK style (c) Hybrid style

Figure 1.5 – LINPACK and LAPACK pivoting styles (symmetric case).

1.3 Exploiting structural sparsity: the multifrontal method

In this section, we provide background on sparse direct methods and in particular on the multifrontal method.

Let us first go back to our illustrative example: in the linear system of equation (1.12), the matrix A is *sparse*, i.e. has many zero entries. These zeros come from the independence between grid points not directly connected in the mesh (Figure 1.1). For example, $u_{2,2}$ and $u_{4,2}$ are not directly connected and therefore $A_{1,3}$ and $A_{3,1}$ are zero entries. This property is referred to as *structural sparsity*, as opposed to data sparsity which will be the object of Section 1.4.

If the dense algorithms presented in the previous section are used to compute the factorization of a sparse matrix, its structural sparsity is not taken into account to reduce the amount of computations and memory to store the factors. However, this is not immediate to achieve because the sparsity pattern of the original matrix differs from that of the factors. Specifically, the sparsity pattern of the original matrix is included in that of the factors (assuming numerical cancellations are not taken into account), i.e., some new entries in the factors become nonzeros. Indeed,

consider the update operation

$$A_{i,j} \leftarrow A_{i,j} - A_{i,k}A_{k,j}.$$

If the original entry $A_{i,j}$ is zero but both $A_{i,k}$ and $A_{k,j}$ are nonzeros, then $A_{i,j}$ will also be nonzero in the factors. This property is referred to as *fill-in* and $A_{i,j}$ is said to be a *filled* entry.

Therefore, a new phase, referred to as *analysis phase* is necessary to analyze the matrix to predict the sparsity pattern of the factors (by means of a symbolic factorization) and perform other important preprocessing operations such as reordering the unknowns.

1.3.1 The analysis phase

1.3.1.1 Adjacency graph

Graph formalism is introduced to analyze the properties of a sparse matrix A. The sparsity pattern of any sparse matrix A can be modeled by a so-called *adjacency graph* $\mathcal{G}(A)$.

Definition 1.1 (Adjacency graph)**.** *The adjacency graph $\mathcal{G}(A)$ of a matrix A of order n is a graph (V,E) such that:*

- *V is a set of n vertices, where vertex i is associated with variable i.*
- *There is an edge $(i,j) \in E$ iff $A_{i,j} \neq 0$ and $i \neq j$.*

If A is structurally symmetric (i.e. if $A_{i,j} \neq 0$ iff $A_{j,i} \neq 0$), then $\mathcal{G}(A)$ is an undirected graph. We assume for the moment that the matrix is structurally symmetric and thus its adjacency graph undirected. We discuss the generalization to structurally unsymmetric matrices in Section 1.3.1.5.

In our illustrative example, the adjacency graph corresponds to the mesh formed by the interior points in Figure 1.1, i.e. the graph in Figure 1.6a.

1.3.1.2 Symbolic factorization

The *symbolic factorization* consists in simulating the elimination of the variables that takes place during the numerical factorization to predict the fill-in that will occur. When variable k is eliminated, the update operation

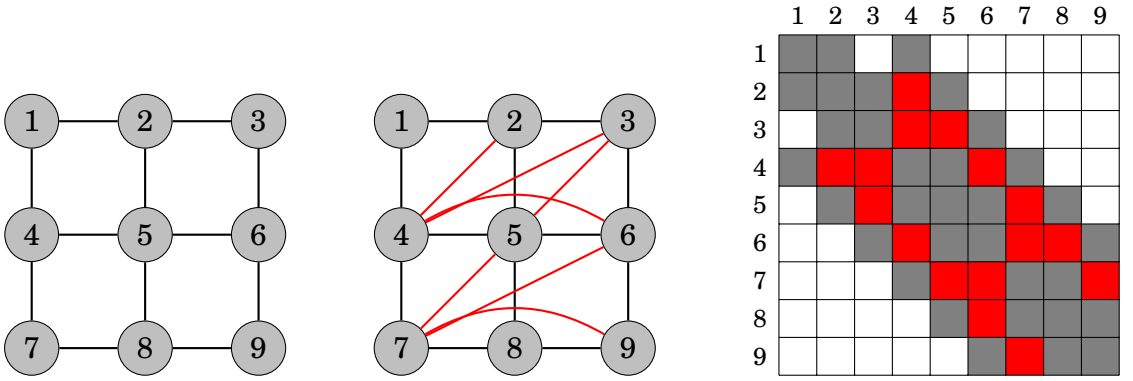
$$A_{i,j} \leftarrow A_{i,j} - A_{i,k}A_{k,j}$$

will fill any entry $A_{i,j}$ such that both $A_{i,k}$ and $A_{k,j}$ are nonzeros, as said before. In terms of graph, this means that when vertex k is eliminated, all its neighbors become interconnected, i.e. a *clique* is formed. For example, in Figure 1.6a, if vertex 1 is eliminated, vertices 2 and 4 are interconnected, i.e. an edge (2,4) must be added (as done in red in Figure 1.6b).

After the elimination of variable k, the next step in the numerical factorization is to factorize the trailing submatrix $A_{k+1:n,k+1:n}$, and therefore, in the symbolic

factorization, vertex k is removed from the graph as well as all its edges. The process is then applied again to the resulting graph until no vertices are left.

We define the *filled graph* $\mathcal{G}(F)$ as the adjacency graph where all edges that were created during the symbolic factorization have been added. On our illustrative example, this is illustrated in Figure 1.6b. Since the new edges correspond to filled entries, the filled graph is the adjacency graph of the factors $F = L + U$ (or $F = L + L^T$). The factor sparsity pattern on our example is reported in Figure 1.6c.



(a) Adjacency graph.

(b) Filled graph.

(c) Sparsity pattern of the factors (original nonzeros are in gray, fill-in in red, zeros in white).

Figure 1.6 – Symbolic factorization: predicting the sparsity pattern of the factors.

1.3.1.3 Influence of the ordering on the fill-in

We now show that the order in which the variables are eliminated, referred to as *ordering*, can significantly influence the fill-in. Obviously, we want to minimize the amount of fill-in, since it increases the computational cost to factorize and store the matrix; thus, finding a good ordering is a crucial issue to make sparse direct methods effective.

However, there is no general rule on how to compute a good ordering. Finding the ordering that minimizes the fill-in is an NP-complete problem (Yannakakis, 1981). Several heuristic strategies exist, whose effectiveness is matrix-dependent. We can distinguish:

- Local heuristics, that successively eliminate vertices in an order depending on some local criterion: for example, the vertex of minimum degree (such as AMD (Amestoy, Davis, and Duff, 1996) or MMD (Liu, 1985)), or the vertex that produces the minimum fill (such as AMF (Ng and Raghavan, 1999) or MMF (Rothberg and Eisenstat, 1998)).

- Global heuristics, that recursively partition the graph into subgraphs, such as nested dissection (ND) (George, 1973).

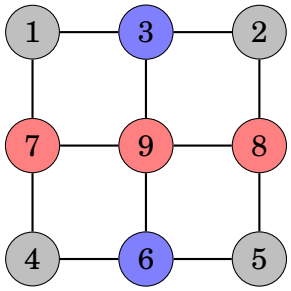
- Hybrid heuristics, which first use a global heuristic to partition the graph, and then apply local heuristics to each subgraph. This is the strategy imple-

mented in several partitioning libraries, such as METIS (Karypis and Kumar, 1998) and SCOTCH (Pellegrini, 2007).

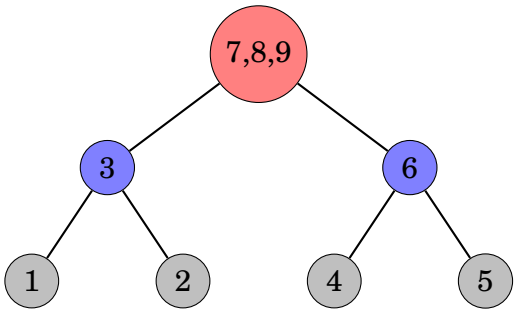
Let us briefly review the nested dissection ordering, which is one of the most widely used and that we will use in this work (building it geometrically or, more commonly, algebraically, with one of the previously cited partitioning libraries).

Nested dissection divides the adjacency graph into a given number s of *domain* subgraphs separated by a *separator* subgraph. The vertices of a given domain are only connected to other vertices in the same domain or in the separator, but not to other domains (Rose, Tarjan, and Lueker, 1976). This way, the elimination of a vertex will not create any fill-in in the other domains. In general, we want to choose the separator so that the size of the domains is as balanced as possible and the size of the separator is as small as possible. The process is then recursively applied to the s domain subgraphs until the domain subgraphs become too small to be subdivided again; this generates a separator tree.

We illustrate this process in Figure 1.7a on our illustrative example with a nested dissection ordering with $s = 2$, i.e. recursive bisection. The top level separator is in red, while the second level separators are in blue. The remaining nodes are the third level subdomains. This generates an associated separator tree reported in Figure 1.7b.



(a) Ordering computed by the nested dissection on the adjacency graph.

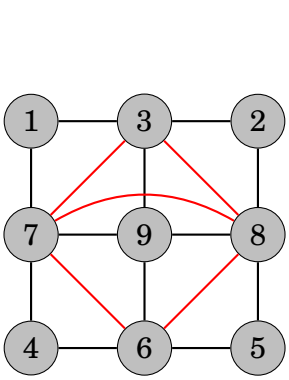


(b) Associated separator tree.

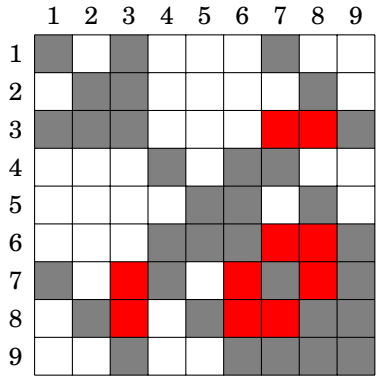
Figure 1.7 – Nested dissection on example mesh.

We can then reorder the vertices using a topological order on the separator tree (i.e., the nodes in any subtree are numbered consecutively). Vertices belonging to the same separator are ordered arbitrarily. In Figure 1.8a and 1.8b, we report the corresponding filled graph and factor sparsity pattern obtained by performing the symbolic factorization on the reordered matrix. Compared to the natural ordering used in Figure 1.6, the fill-in has decreased from 16 to 10 filled entries, which is a considerable improvement compared to the number of nonzeros in the original matrix.

As said before, the elimination of vertex in a given domain does not affect vertices in other domains. A key consequence of this is that vertices in different branches of the separator tree could be eliminated independently, and thus, concurrently. The object of the next section is to generalize this concept to general orderings.



(a) Filled graph.



(b) Sparsity pattern of the factors (original nonzeros are in gray, fill-in in red, zeros in white).

Figure 1.8 – Symbolic factorization with nested dissection reordering.

1.3.1.4 Elimination tree

First, we formalize the concept of *dependency* between vertices. We want to express the fact that j depends on i iff the elimination of i modifies column j.

Definition 1.2 (Vertex dependency)**.** *Let i, j be two vertices such that $i < j$. Vertex j depends on i (noted $i \rightarrow j$) iff $\exists k \in [i+1; n]$ such that $L_{k,i}U_{i,j} \neq 0$.*

Indeed, in that case, the update $A_{k,j} \leftarrow A_{k,j} - L_{k,i}U_{i,j}$ will fill the entry $A_{k,j}$ and thus modify column j. Furthermore, in the structurally symmetric case, it is straightforward to prove that this definition simplifies as follows.

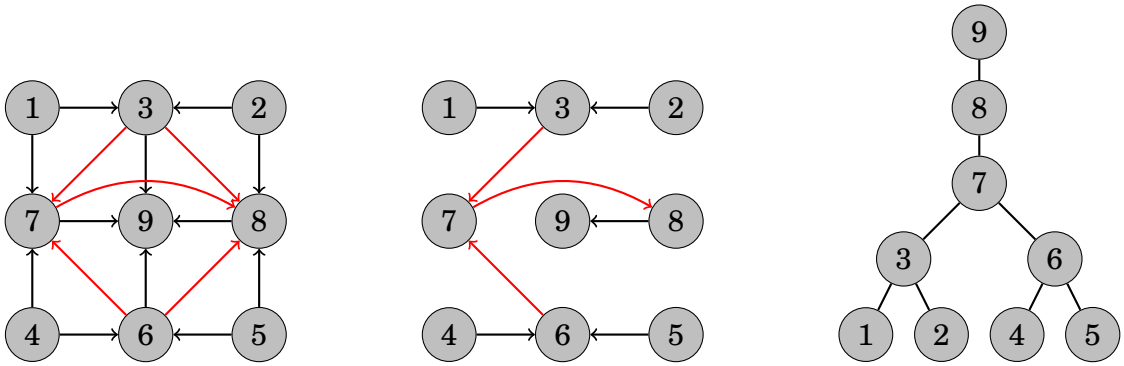
Definition 1.3 (Vertex dependency for structurally symmetric matrices)**.** *Let i, j be two vertices such that $i < j$. Vertex j depends on i (noted $i \rightarrow j$) iff $L_{j,i} \neq 0$ (or equivalently $U_{i,j} \neq 0$).*

Therefore, the vertex dependencies are characterized by the sparsity pattern of the factors, and thus by the filled graph $\mathcal{G}(F)$. However, the dependency is not a symmetric relation: a vertex j can only depend on vertices eliminated before it, i.e. $i \rightarrow j$ implies $i < j$. Therefore, we introduce the directed version of the filled graph.

Definition 1.4 (Directed filled graph)**.** *Let the undirected filled graph be $\mathcal{G}(F) = (V, E)$. Then, the directed filled graph $\overrightarrow{\mathcal{G}}(F)$ is the graph $(\overrightarrow{V}, \overrightarrow{E})$ such that:*

- *$\overrightarrow{V} = V$, i.e., both graphs have the same vertices;*
- *For all edges $(i, j) \in E$, if $i < j$ then $(i, j) \in \overrightarrow{E}$ else $(j, i) \in \overrightarrow{E}$, i.e., the edges of E have been directed following the elimination order.*

The directed filled graph thus characterizes the dependencies between vertices. By definition, $(i, j) \in \overrightarrow{E} \Rightarrow i < j$ and therefore it cannot have any cycle. It is therefore a directed acyclic graph (DAG) (Aho, Hopcroft, and Ullman, 1983). The directed filled graph on our example is given on Figure 1.9a.



(a) Directed filled graph. (b) Transitive reduction. (c) Elimination tree.

Figure 1.9 – Construction of the elimination tree.

Furthermore, it contains many redundant dependencies: in our example, $1 \to 7$ can be obtained from $1 \to 3$ and $3 \to 7$. We can thus obtain a more compact representation of the vertex dependencies by removing the redundant dependencies, i.e., by computing the transitive reduction $T(\vec{\mathcal{G}}(F))$ (which is unique for DAGs (Aho, Garey, and Ullman, 1972)) of $\vec{\mathcal{G}}(F)$. This is illustrated on Figure 1.9b.

$T(\vec{\mathcal{G}}(F))$ is obviously still a DAG. The key observation at the foundation of sparse direct methods is that its undirected version $T(\mathcal{G}(F))$ is still acyclic, i.e. it is actually a spanning tree of the directed filled graph, as illustrated in Figure 1.9c. This tree is referred to as *elimination tree* (Schreiber, 1982) and we note it $\mathcal{E}$.

We briefly sketch the proof that $\mathcal{E} = T(\mathcal{G}(F))$ is a tree. We first prove the following lemma.

Lemma 1.1. *Let $(i,j,k) \in \vec{V}^3$ be three vertices of the directed filled graph $\vec{\mathcal{G}}(F)$ of a matrix A. If $i < j < k$, $i \to j$, and $i \to k$, then $j \to k$.*

Proof. The proof comes from the fill-in phenomenon and the structural symmetry of the matrix. By definition of $i \to j$, it holds $L_{j,i} \neq 0$, which by symmetry is equivalent to $U_{i,j} \neq 0$; and since $i \to k$, it also holds $L_{k,i} \neq 0$. Therefore, $L_{k,i}U_{i,j} \neq 0$ which fills $L_{k,j}$ and thus $j \to k$. □

We can now prove that $\mathcal{E}$ is a tree.

Theorem 1.1 (Existence of the elimination tree)**.** *The undirected transitive reduction $\mathcal{E} = T(\mathcal{G}(F))$ of the directed filled graph $T(\vec{\mathcal{G}}(F))$ is a tree (if the matrix is irreducible, otherwise it is a forest).*

Proof. We have to prove there is no cycle (directed or undirected) in $\mathcal{E}$. As said before, there are no directed cycles since it is a DAG. By reductio ad absurdum: let (i,j,k) be three vertices forming a cycle with $i < j < k$, i.e. $i \to j$ and $i \to k$. Then Lemma 1.1 implies $j \to k$. Therefore $i \to k$ is reduced to $(i \to j, j \to k)$ and cannot be part of $\mathcal{E}$. This is a contradiction. □

The elimination tree is a key structure in sparse direct methods because it is a compact representation of the dependencies, containing the order in which the variables must be eliminated, as well as the variables that can be eliminated independently and thus concurrently.

Before describing how it is used to schedule the computations done during the numerical factorization, let us first discuss the case of structurally unsymmetric matrices.

1.3.1.5 Generalization to structurally unsymmetric matrices

So far, we have been assuming the matrix is structurally symmetric, which is in particular a key element of the proof of Lemma 1.1 and thus Theorem 1.1. In fact, for unsymmetric matrices, the transitive reduction of the directed filled graph is not a tree but a DAG. A possible way to overcome this issue is to consider the symmetrized structure of the matrix instead, i.e. work on $A + A^T$. This approach was suggested by Duff and Reid (1984); it is used for example in MUMPS or SuperLU_Dist (Li and Demmel, 2003), and is the one we will consider in this thesis.

Note there are other ways to deal with unsymmetric matrices, such as a generalization of the elimination tree structure that was formalized by Gilbert and Liu (1993) and Eisenstat and Liu (2005), and used for instance in UMFPACK (Davis and Duff, 1997), SuperLU (Demmel, Eisenstat, Gilbert, Li, and Liu, 1999), WSMP (Gupta, 2002), or the unsymmetrized MA41 (Amestoy and Puglisi, 2002).

In the following, we will assume we are working on the symmetrized structure of the matrix $A + A^T$.

1.3.2 The multifrontal method

We now describe how the elimination tree structure is used to drive the computations performed during the numerical factorization, and focus in particular on the multifrontal method.

1.3.2.1 Right-looking, left-looking and multifrontal factorization

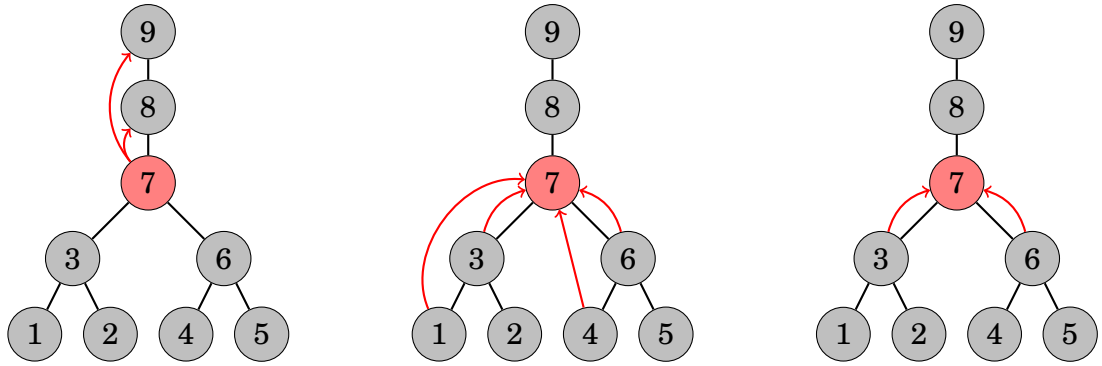
The factorization phase consists in performing two operations on each node of the tree: the elimination of the node variable i, and the computation of the *contributions* this elimination yields, which are used to update all ancestor nodes j such that $i \to j$. In our example, when node 1 is eliminated, nodes 3 and 7 must be updated; when node 3 is eliminated, nodes 7, 8, and 9 are updated.

As said before, the tree parallelism offers some flexibility to schedule the node eliminations. Furthermore, because the updates may concern ancestors much higher up in the tree than the node being eliminated, we have some more freedom to schedule when the updates are actually performed. Just as in the dense case, we can distinguish two approaches:

- In the right-looking approach, the updates are performed as soon as possible: after the elimination of i, all ancestors j such that $i \to j$ are updated.

- In the left-looking approach, the updates are performed as late as possible: all the contributions coming from descendants i such that $i \to j$ are applied just before the elimination of j.

This is illustrated in Figures 1.10a and 1.10b. In parallel contexts, these right- and left-looking approaches are also referred to as *fan-out* and *fan-in*, respectively.



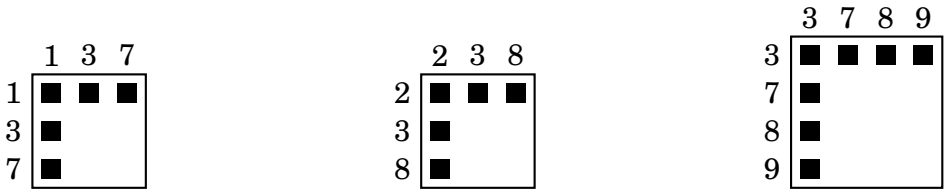
(a) Right-looking approach. (b) Left-looking approach. (c) Multifrontal approach.

Figure 1.10 – Comparison on the right-looking, left-looking, and multifrontal approaches. The red arrows indicate the contributions sent at node 7, either before (left-looking and multifrontal) or after (right-looking) its elimination.

Therefore, the updates in both the right- and left-looking approaches are based on the variable dependencies and thus on the directed filled graph. Instead, the *multifrontal* method (Duff and Reid, 1983; Liu, 1992) is directly based on the more compact dependencies represented by the elimination tree, thanks to the following observation: if $i \to j$, then node j is an ancestor of node i in the elimination tree. Therefore, after the elimination of node i, its contribution to j is immediately computed but does not need to be applied directly; instead, it is carried along the tree, from parent to parent, until node j is reached and is ready to be eliminated. This is illustrated in Figure 1.10c, where contributions of nodes 3 and 6 to nodes 8 and 9 are carried through node 7. We describe the involved algorithms using our illustrative example.

We associate with each node of the tree a dense matrix called *frontal matrix* or just *front*, as illustrated on Figure 1.11. This front is formed of the node variable to be eliminated as well as all variables this elimination contributes to. For example, the front associated with node 1, noted F_1, is formed of variables 1, 3, and 7, because $1 \to 3$ and $1 \to 7$. Similarly, the fronts F_2 and F_3, associated with node 2 and 3 are formed of variables 2, 3, and 8, and variables 3, 7, 8, and 9, respectively,

The elimination of variable 1 consists in a partial factorization of F_1. It yields a 2×2 Schur complement containing the contributions to variables 3 and 7. It is for this reason called *contribution block* (CB) and noted CB_1. A similar contribution block CB_2, containing contributions to variables 3 and 8, is computed during the elimination of variable 2. Once both fronts F_1 and F_2 have been factorized, their contribution blocks CB_1 and CB_2 are passed to the parent front F_3. One more



(a) Front F_1 associated with variable 1.

(b) Front F_2 associated with variable 2.

(c) Front F_3 associated with variable 3.

Figure 1.11 – Three of the frontal matrices involved in the factorization of matrix A.

operation, referred to as *assembly*, must be performed before the factorization of F_3 can begin.

The assembly of F_3 consists in summing together all updates related to its variables. Specifically:

- Variable 3 is updated with contributions from both CBs. These are summed together with the corresponding values of the original matrix. Then, all contributions related to variable 3 have been summed, and for this reason variable 3 is said to be *fully-summed*.

- Similarly, the other variables of F_3, 7, 8 and 9, are computed: 7 and 8 are equal to their corresponding contribution in CB_1 and CB_2, respectively, while variable 9 is simply initialized to zero. These variables are called *non fully-summed*, because there are still some contributions left to sum. For instance, variable 7 also receives a contribution from the elimination of variables 3, 4, and 6.

The assembly of F_3 can thus be written as follows:

$$F_3 = A_3 \dashv\vdash CB_1 \dashv\vdash CB_2, \tag{1.21}$$

where A_3 contains the entries of the original matrix corresponding to variable 3, and $\dashv\vdash$ denotes the *extend-add* operation.

The process then continues until all the fronts have been treated.

1.3.2.2 Supernodes and assembly tree

In practice, the nodes of the elimination tree are grouped together when their associated variables have the same sparsity structure in the reduced matrix: nodes i and j can be grouped if $(i \to k) \Leftrightarrow (j \to k)$ for all $k > i$ (assuming $i < j$). This is referred to as *amalgamation* and the resulting nodes are called *supernodes*. The resulting tree is then known as the *assembly tree*. This allows the fronts to have more than one fully-summed variable and thus their elimination takes the form of a *partial dense factorization*. This allows the computations to be performed efficiently by being based on the dense linear algebra kernels described in Section 1.2.

For example, in Figure 1.9c, nodes 7, 8, and 9 have the same sparsity structure after the elimination of variables 1 through 6 and can thus be amalgamated into a single supernode; the resulting assembly tree is shown in Figure 1.12.

Note that even if the amalgamated nodes are called supernodes, *supernodal* methods usually refer to the right-looking and left-looking approaches described earlier, rather than multifrontal methods.

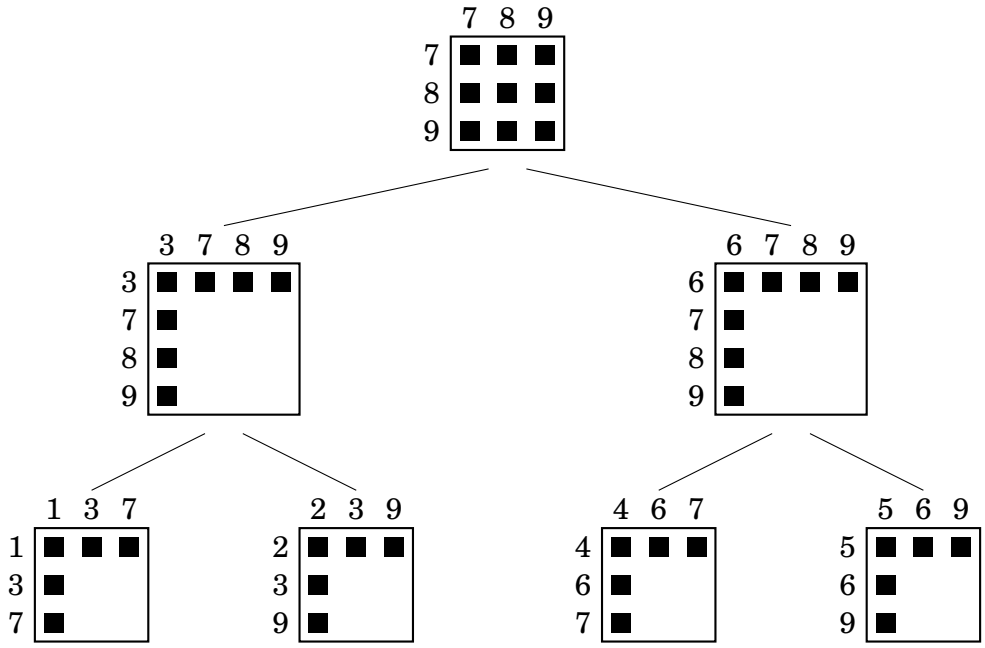


Figure 1.12 – Assembly tree resulting from the amalgamation of nodes 7,8,9.

It can sometimes also be interesting to amalgamate variables that have different but similar sparsity structure. In this case, the amalgamation increases the computational cost of the factorization but also improves its efficiency.

1.3.2.3 Theoretical complexity

In the context of a nested dissection ordering on a regular grid, the theoretical complexity of the multifrontal method can be easily computed as it is directly derived from the dense complexities (George, 1973). For the sake of readability, and to simplify the following computations, we assume that at each level the domains are divided in 2^d, where d is the dimension number (i.e. cross-shaped separators in 2D). The asymptotical theoretical complexity would be the same with a recursive bisection ordering.

We consider a sparse matrix A of order n arising from a discretized mesh of N^d points, where d denotes the dimension; we only consider the 2D and 3D cases (i.e. $d = 2$ and $d = 3$). The number of operations for the multifrontal factorization of A can be computed as

$$\mathscr{C}_{MF}(N) = \sum_{\ell=0}^{L} \mathscr{C}_\ell(N) = \sum_{\ell=0}^{L} (2^d)^\ell \mathscr{C}((\frac{N}{2^\ell})^{d-1}), \tag{1.22}$$

where $\mathscr{C}_\ell(N)$ is the cost of factorizing all the fronts on the ℓ-th level. At level ℓ, there are $(2^d)^\ell$ fronts of order $m_\ell = (\frac{N}{2^\ell})^{d-1}$, and thus $\mathscr{C}_\ell(N) = (2^d)^\ell \mathscr{C}(m_\ell)$. Since the number of operations to factorize a dense matrix of order m is $\mathscr{O}(m^3)$, we obtain

$$\mathscr{C}_{MF}(N) = \mathscr{O}(N^{3(d-1)} \sum_{\ell=0}^{L} 2^{(3-2d)\ell}) = \mathscr{O}(N^{3(d-1)}), \tag{1.23}$$

because the sum term is a geometric series of common ratio $2^{3-2d} < 1$.

The factor size complexity (i.e. the memory required to store the factors) can similarly be computed.

We provide the flops and factor size complexities in the 2D and 3D cases in Table 1.1.

d	$\mathscr{C}_{MF}(n)$	$\mathscr{M}_{MF}(n)$
2D	$\mathscr{O}(n^{3/2})$	$\mathscr{O}(n \log n)$
3D	$\mathscr{O}(n^2)$	$\mathscr{O}(n^{4/3})$

Table 1.1 – Theoretical complexity for the multifrontal factorization of a sparse matrix of order n. $\mathscr{C}_{MF}(n)$: flop complexity; $\mathscr{M}_{MF}(n)$: factor size complexity.

1.3.2.4 Parallelism in the multifrontal factorization

We briefly discuss possible ways to parallelize the multifrontal factorization. Two sources of parallelism can be distinguished:

- As said before, different branches of the elimination/assembly tree are independent; they can thus be traversed concurrently. This is referred to as *tree parallelism*.
- For large enough fronts, their partial factorization may also be performed on several threads/processes. This is referred to as *node parallelism*.

Parallelism issues will be central in Chapters 5 (shared-memory OpenMP parallelism) and 6 (distributed-memory MPI parallelism); we defer algorithmic details about parallelism to these chapters.

1.3.2.5 Memory management

As said before, in the multifrontal method, the contributions from one supernode to another are not applied directly but rather stored in a temporary workspace as a contribution block. As a consequence, two types of memories must be managed during the multifrontal factorization:

- The *factors memory*, which stores the LU factors as they are being computed;
- The *active memory*, which stores the contribution blocks waiting to be assembled (the *CB memory*), as well as the current fronts being assembled/factorized (the so-called *active fronts*).

The factors memory is simple to analyze since it increases monotonically as the factorization progresses. However, the active memory has a more complex behavior and strongly depends on the order in which the fronts are factorized. Therefore, the total memory consumption (the sum of the two types of memory) is also not monotonic and depends on the ordering; its maximal value over time is referred to as *memory peak* and represents the total amount of storage required to perform the numerical factorization.

In a sequential context, if the assembly tree is traversed in a postorder, then the CB memory behaves like a stack. This property is lost in parallel since the tree is not traversed in a postorder anymore.

1.3.2.6 Numerical pivoting in the multifrontal method

We now discuss how to adapt the pivoting strategies discussed in Section 1.2.4 to the multifrontal case.

Postponed and delayed pivots Due to the special structure of the frontal matrices, the Factor+Solve algorithm described in Algorithm 1.8 must be adapted, as explained in the following. Consider the factorization of a frontal matrix F of order $n = n_{fs} + n_{nfs}$, where n_{fs} and n_{nfs} denote its number of fully-summed and non fully-summed variables, respectively. At step k of the partial factorization of F, the non fully-summed variables $A_{n_{fs}+1:n,k}$ cannot be selected as pivots since they are not ready to be eliminated. Thus, the search for a pivot candidate should be restricted to the fully-summed variables $A_{k:n_{fs},k}$. However, the quality of the pivot candidates should still be assessed with respect to the entire column $A_{k:n,k}$, including the non fully-summed variables.

Therefore, contrarily to the dense case, it is possible that none of the pivot candidates on the current column are acceptable (due to a large non fully-summed variable). In that case, the elimination of pivot k must be *postponed* until a large enough fully-summed variable can be found, and we move on to try to eliminate pivot $k+1$. At the end of the partial factorization of F, there may still be some pivots that cannot be eliminated; they are referred to as *delayed pivots* and are passed to the parent front, where we will again try to eliminate them, until reaching the root front where all variables are fully-summed and can thus be eliminated.

Note that in the literature, the terms "postponed" and "delayed" are often interchangeable. In this manuscript, for the sake of clarity, we will exclusively refer to pivots whose elimination is deferred to later in the same front as "postponed", and those whose elimination is deferred to later in an ancestor front as "delayed".

Modified algorithm The Factor+Solve algorithm adapted to unsymmetric frontal matrices is presented in Algorithm 1.9. At step k, we try to eliminate a new pivot. We loop until an acceptable pivot j is found, which satisfies the threshold partial pivoting condition $|A_{i_{fs},j}| \geq \tau |A_{i,j}|$. The algorithm terminates when all n_c pivots have been eliminated, or when no more pivot candidates are left, in which case the remaining $n_c - k + 1$ pivots are postponed.

Algorithm 1.9 Factor+Solve step adapted to frontal matrices (unsymmetric case)

1: /* **Input:** a panel A with $n_r = n_{fs} + n_{nfs}$ rows and n_c columns. */
2: **for** $k = 1$ **to** n_c **do**
3: $j \leftarrow k - 1$
4: /* Loop until an acceptable pivot is found, or until no pivot candidates are left. */
5: **repeat**
6: $j \leftarrow j + 1$
7: **if** $j > n_c$ **then**
8: /* No pivot candidates left, exit. */
9: **go to** 19
10: **end if**
11: $i \leftarrow \arg\max_{i' \in [k;n_r]} |A_{i',j}|$
12: $i_{fs} \leftarrow \arg\max_{i' \in [k;n_{fs}]} |A_{i',j}|$
13: **until** $|A_{i_{fs},j}| \geq \tau |A_{i,j}|$
14: /* Pivot candidate has been chosen in column j: swap it with current column k. */
15: Swap columns k and j and rows k and i_{fs}
16: $A_{k+1:n_r,k} \leftarrow A_{k+1:n_r,k} / A_{k,k}$
17: $A_{k+1:n_r,k+1:n_c} \leftarrow A_{k+1:n_r,k+1:n_c} - A_{k+1:n_r,k} A_{k,k+1:n_c}$
18: **end for**
19: /* $n_{pp} = n_c - k + 1$ is the number of postponed pivots. */

The symmetric case In the symmetric case, the algorithms discussed in Section 1.2.4.1 must be adapted to the sparse case. Liu (1987) proposed a sparse variant of the Bunch-Kaufman algorithm , which suffers from the same problem as its dense counterpart: the entries in L are unbounded. Its bounded version, proposed in Ashcraft et al. (1998), is also extended to the sparse case in the same paper. It is closely related to the Duff-Reid algorithm from Duff and Reid (1983) (for a comparison between the two, see Section 3.4 of Ashcraft et al. (1998)).

In the Duff-Reid algorithm, if no 1×1 pivot has been found at step k, $P = \begin{pmatrix} A_{k,k} & A_{k,\ell} \\ A_{\ell,k} & A_{\ell,\ell} \end{pmatrix}$ is considered, where $A_{\ell,k} = \max_{\ell' \in [k+1;n]} A_{\ell',k}$. P is accepted as a 2×2 pivot if it verifies the following condition:

$$\left|P^{-1}\right| \begin{pmatrix} \max_{i \in [k+2;n]} |A_{i,k}| \\ \max_{i \in [k+2;n]} |A_{i,k+1}| \end{pmatrix} \leq \begin{pmatrix} 1/\tau \\ 1/\tau \end{pmatrix}. \tag{1.24}$$

Note that in the original Duff-Reid algorithm (Duff and Reid, 1983), an unnecessarily stricter criterion was used, which was then relaxed as above in Duff and Reid (1996). If P is rejected, pivot k is postponed and we repeat the process on pivot $k+1$ until either a 1×1 or 2×2 pivot is found.

Note that, in the block (or tile) version of the factorization, if $A_{\ell\ell}$ lies outside of the current panel, it must first be updated before P can be assessed; if P is rejected, that update will have been useless. To avoid that, it is common to limit the search of off-diagonal pivots to the diagonal block of the current panel (inside which all pivots are up to date).

Restricted pivoting Threshold partial pivoting requires to check all the entries in the current pivot column to assess the quality of the pivot candidates, and this can be expensive. To reduce the cost of pivoting, one possible strategy is to limit the search space to a subset R of rows. This is known as *restricted pivoting*. The hope is that the pivot search can be accelerated while not leading to the acceptance of pivot candidates that would otherwise have been rejected. Typical values for R are the set of fully-summed rows or the rows belonging to the diagonal block of the current panel.

To assess the quality of a given pivot candidate $A_{i,k}$ (with $i \in R$), several strategies have been proposed to take into account the rest of the pivot rows (i.e. $A_{j,k}$ with $j \in [k;n] \backslash R$).

A first basic idea, presented for example in Section 6.1 of Duff and Pralet (2007), is to simply ignore the pivots outside of the restricted search space R. Obviously, this can be dangerous as $A_{j,k}$ can be arbitrarily larger than $A_{i,k}$.

In Section 6.2 of Duff and Pralet (2007), a strategy based on the estimation of the maximum norm of the rows outside R is presented. At the beginning of the factorization, the element of maximum amplitude outside R on each column is computed:

$$m_k = \max_{j \in [k;n] \backslash R} |A_{j,k}|,$$

and subsequently used as an estimation of the maximum norm of column $A_{[k;n]\backslash R,k}$. Note that in the cited paper, this strategy was developed for the parallel context and therefore m_k is actually also estimated as the maximum of each local maximum on each processor (cf. equation (6.3) in Duff and Pralet (2007)). This strategy is implemented and used as default in MUMPS.

Another strategy implemented in MUMPS (since version 4.5) consists in updating the m_k values each time a pivot $\ell < k$ is eliminated. In other words, the row vector $m = (m_k)_{k \in [1;n]}$ formed of all the column maximums can be seen as an additional row of the matrix that is also factorized.

This latter strategy has been further pushed in Hogg and Scott (2014), where more than one row vector m is computed and updated. This allows for computing a richer estimation of the maximum norm of the row space outside R.

Static pivoting The threshold partial pivoting strategy is referred to as *dynamic* in the sense that at each step, it performs a swap which dynamically modifies the data structure. In the context of the multifrontal factorization, modifying the structure of the factors can increase the fill-in and thus the computational cost of the factorization.

To avoid this behavior, it is sometimes preferred to use *static* pivoting, which replaces too small diagonal pivots by an artificial value: for example, in Li and Demmel (1998), the diagonal entries smaller in absolute value than $\sqrt{\varepsilon_{mach}}\|A\|$ are replaced by $\sqrt{\varepsilon_{mach}}\|A\|$, where ε_{mach} is the machine precision.

Static pivoting allows operations to be performed in a more efficient and scalable way, but usually requires some steps of Iterative Refinement (described in Algorithm 1.10) to achieve a similar accuracy as dynamic pivoting.

Finally, note that an algorithm based on a mixture of dynamic and static pivoting was also proposed in Section 4.1 of Duff and Pralet (2007): its core idea is to

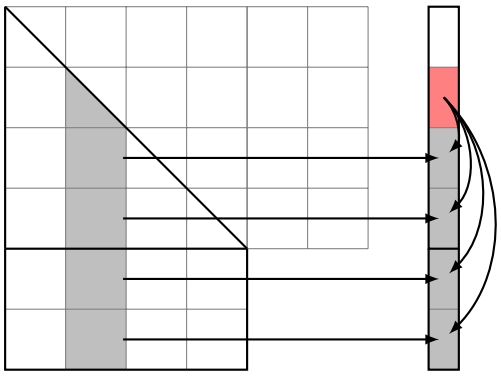
Algorithm 1.10 Iterative refinement

1: /* **Input:** a matrix A, right-hand side b, and initial solution x */
2: **for** $k = 1$ **to** n_{steps} **do**
3: $r \leftarrow b - Ax$
4: Solve $A\Delta x = r$
5: $x \leftarrow x + \Delta x$
6: **end for**

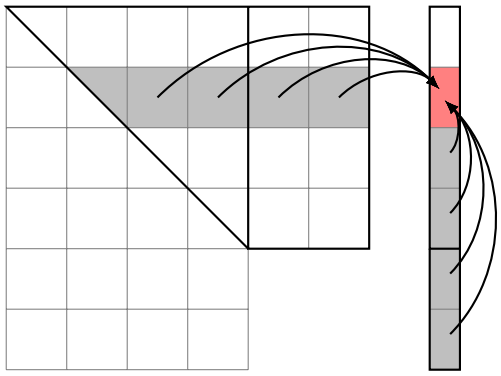
perform dynamic pivoting inside the fronts, and then use a static pivoting technique to eliminate the remaining columns rather than delaying them.

1.3.2.7 Multifrontal solution phase

Once all the frontal matrices of the assembly tree have been partially factorized, the linear system can be solved through forward elimination and backward substitution, just as in the dense case (equation (1.13)). A key point is that the fronts are not square, as illustrated in Figure 1.13. Therefore, Algorithm 1.6 must be modified so that, for each front F, a part of the solution (y in the forward phase or x in the backward phase) is computed (corresponding to the fully-summed variables of F) and another part of it is updated (corresponding to the non fully-summed variables of F). This results into the FrontalForward and FrontalBackward algorithms described in Algorithm 1.11.



(a) Frontal forward elimination (right-looking).



(b) Frontal backward substitution (left-looking).

Figure 1.13 – Frontal solution phase.

The overall multifrontal solution phase is also described in Algorithm 1.11. First, the forward elimination is achieved by means of a bottom-up traversal of the assembly tree; then, a top-down traversal of the assembly tree is performed for the backward substitution phase. 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. For the sake of simplicity, we ignore in Algorithm 1.11 issues concerning pivoting or parallelism.

Algorithm 1.11 Multifrontal solution phase (without pivoting).

1: /* **Input:** the right-hand side vector b. */
2: $y \leftarrow b$
3: Initialize pool with leaf nodes.
4: **repeat**
5: Extract a node F from pool.
6: $y \leftarrow \text{FrontalForward}(F, y)$
7: **if** all siblings of F have been processed **then**
8: Insert parent of F in pool.
9: **end if**
10: **until** pool is empty (all fronts have been processed)
11: $x \leftarrow y$
12: Initialize pool with root nodes.
13: **repeat**
14: Extract a node F from pool
15: $x \leftarrow \text{FrontalBackward}(F, x)$
16: Insert children of F in pool.
17: **until** pool is empty (all fronts have been processed)
18:
19: FrontalForward(F, y)
20: Gather rows of y corresponding to variables of F in w.
21: **for** $j = 1$ **to** p_{fs} **do**
22: $w_j \leftarrow L_{j,j}^{-1} w_j$
23: **for** $i = j + 1$ **to** p_{nfs} **do**
24: $w_i \leftarrow L_{i,j} w_j$
25: **end for**
26: **end for**
27: Scatter w in the rows of y corresponding to variables F.
28:
29: FrontalBackward(F, x)
30: Gather rows of x corresponding to variables of F in w.
31: **for** $i = p_{fs}$ **to** 1 **by** -1 **do**
32: **for** $j = i + 1$ **to** p_{nfs} **do**
33: $w_i \leftarrow U_{i,j} w_j$
34: **end for**
35: $w_i \leftarrow U_{i,i}^{-1} w_i$
36: **end for**
37: Scatter w in the rows of x corresponding to variables F.

When a front F is processed, the corresponding rows of x (or y) are gathered in a temporary workspace w, since in general they are not contiguous in x (or y). After w has been computed, the result is scattered back in x (or y). A direct consequence of this observation is that when w is gathered, it can be directly built in the order specified by the permutations performed at the factorization (for numerical pivoting). This makes the LINPACK and LAPACK styles very different; indeed, with LAPACK, no permutations need to be performed after w is gathered, while

with LINPACK, all permutations except the first must still be performed on w (and similarly, with the hybrid style, only the permutations corresponding to the first panel can be avoided). Therefore, in the sparse case, the LAPACK style is usually preferred, especially in the case of multiple right-hand sides.

Note that rather than scattering the entries of w back to x and y each time a front is processed, we could instead store them in a temporary workspace (similar to the contribution blocks used in the factorization phase) until they are gathered directly from w when needed. This strategy is not considered in Algorithm 1.11.

1.4 Exploiting data sparsity: low-rank formats

Let us come back once more to our 2D Poisson example. We have discussed how to take advantage of the structural sparsity. If we take for example the root node corresponding to the root separator in the nested dissection ordering, it is fully dense. However, $A_{7,9}$ and $A_{9,7}$ are filled entries, since nodes 7 and 9 in the mesh are not directly connected. This fill-in can thus be physically interpreted as the interaction between nodes 7 and 9; because these nodes are at distance $2h$ from each other (compared to h for neighbor nodes), we may expect its absolute value to be lower than that of other nonzero entries. Thus, one may consider computing an approximate factorization by dropping these fill-in entries corresponding to distant interactions.

This is exactly the principle behind incomplete LU (ILU) factorization, where filled entries below a given threshold are dropped (i.e. replaced by zero). In ILU(0), in particular, all filled entries are dropped.

Low-rank approximations can be seen as a generalization of dropping: instead of considering the interaction between single entries, we consider the interactions between subdomains made of several entries. This is formalized in Section 1.4.2.1. First, let us introduce some concepts related to low-rank matrices.

1.4.1 Low-rank matrices

1.4.1.1 Definitions and properties

In this section, we consider a matrix B of size $m \times n$. We first define the *rank* of a matrix.

Definition 1.5 (Rank)**.** *The rank k of B is defined as the smallest integer such that there exist matrices X and Y of size $m \times k$ and $n \times k$ such that*

$$B = XY^T.$$

Definition 1.6 (Rank-k approximation at accuracy ε)**.** *We call a rank-k approximation of B at accuracy ε any matrix $\widetilde{B}$ of rank k such that*

$$\|B - \widetilde{B}\| \leq \varepsilon.$$

In particular, when the norm used in the definition above is the 2-norm, we know from Eckard and Young (1936) that the optimal rank-k approximation of a matrix B can be computed from its singular value decomposition (SVD).

Theorem 1.2. *Let $U\Sigma V^T$ be the SVD decomposition of B and let us note $\sigma_i = \Sigma_{i,i}$ its singular values. Then $\widetilde{B} = U_{1:m,1:k}\Sigma_{1:k,1:k}V^T_{1:n,1:k}$ is the optimal rank-k approximation of B and*

$$\|B - \widetilde{B}\|_2 = \sigma_{k+1}.$$

We finally define the *numerical rank* of a matrix at a given accuracy ε.

Definition 1.7 (Numerical rank)**.** *The numerical rank $Rk_\varepsilon(B)$ of B at accuracy ε is defined as the smallest integer k_ε such that there exists a matrix $\widetilde{B}$ of rank k_ε such that*

$$\|B - \widetilde{B}\| \leq \varepsilon.$$

In particular, using the two-norm, we can also compute the numerical rank of B from its SVD.

Theorem 1.3. *Let $U\Sigma V^T$ be the SVD decomposition of B and let us note $\sigma_i = \Sigma_{i,i}$ its singular values. Then the numerical rank of B at accuracy ε is given by*

$$k_\varepsilon = \min_{1 \leq k \leq \min(m,n)} \sigma_{k+1} \leq \varepsilon.$$

Proof. Let $X_k = U_{1:m,1:k}\Sigma_{1:k,1:k}$ and $Y_k = V_{1:n,1:k}$ for some $1 \leq k \leq \min(m,n)$. Then $\|B - X_{k_\varepsilon}Y^T_{k_\varepsilon}\|_2 = \sigma_{k_\varepsilon+1} \leq \varepsilon$ and $\|B - X_{k_\varepsilon-1}Y^T_{k_\varepsilon-1}\|_2 = \sigma_{k_\varepsilon} > \varepsilon$. Therefore $\mathrm{Rk}_\varepsilon(B) = k_\varepsilon$. □

If the numerical rank of B is equal to $\min(m,n)$ then B is said to be *full-rank*. If $\mathrm{Rk}_\varepsilon(B) < \min(m,n)$, then B is said to be *rank-deficient*. A class of rank-deficient matrices of particular interest are *low-rank* matrices, defined as follows.

Definition 1.8 (Low-rank matrix)**.** *B is said to be low-rank (for a given accuracy ε) if its numerical rank k_ε verifies*

$$k_\varepsilon(m+n) \leq mn.$$

The above condition means that matrix B is considered low-rank iff it requires less storage to store its rank-k_ε approximation $\widetilde{B} = XY^T$ than the full-rank matrix B itself. In that case, $\widetilde{B}$ is said to be a *low-rank approximation* of B and ε is called the *low-rank threshold*.

In the following, for the sake of simplicity, we refer to the numerical rank of a matrix at accuracy ε simply as its "rank".

1.4.1.2 Compression kernels

Computing $\widetilde{B}$ from B is called *compressing* B. There are several ways to compress matrix B. As a consequence of Theorem 1.2, the optimal low-rank approxi-

mation of B can be computed with its SVD. However, computing the SVD of a $m \times n$ matrix has complexity $\mathcal{O}(mn^2)$ (assuming $m > n$) and is therefore expensive.

Alternatively, a widely used compression kernel is the truncated version of the QR factorization with column pivoting (Businger and Golub, 1965), which consists in stopping the factorization when the diagonal coefficient of R falls below the prescribed threshold ε. It is slightly less accurate than the SVD but much cheaper to compute, with complexity $\mathcal{O}(mnk_\varepsilon)$.

More recently, randomized algorithms have been gaining attention and in particular random sampling (Liberty, Woolfe, Martinsson, Rokhlin, and Tygert, 2007). We briefly summarize its main idea. For a detailed study, we recommend the survey by Halko, Martinsson, and Tropp (2011). For example, a truncated QR factorization of B can be computed via random sampling in three main steps:

1. Sample a subspace $S = B\Omega$ which approximates the range of B, where Ω is some random matrix, such as a Gaussian or randomized Fast Fourier transform matrix.

2. Compress the sample matrix S via the classical truncated QR factorization with column pivoting: $SP = \widehat{Q}\begin{pmatrix}\widehat{R}_{1:k} & \widehat{R}_{k+1:n}\end{pmatrix}$.

3. Compute a QR factorization of $BP_{1:k}$: $BP_{1:k} = Q\overline{R}$, which yields $BP \approx QR$ with $R = \overline{R}\begin{pmatrix}I_k & \widehat{R}_{1:k}^{-1}\widehat{R}_{k+1:n}\end{pmatrix}$.

The compression (second step) is performed on a lower-dimensional sample and is therefore usually dominated by the sampling (first step), which can be performed efficiently. This makes random sampling more efficient and scalable than the previous compression kernels and is therefore particularly useful on large matrices in parallel environments. Similar random sampling algorithms can be devised to compute a truncated SVD decomposition.

Many other compression kernels have been proposed, among which we can cite adaptive cross-approximation (Bebendorf, 2000), interpolative decomposition (Cheng, Gimbutas, Martinsson, and Rokhlin, 2005), CUR decomposition (Mahoney and Drineas, 2009), and boundary domain low-rank approximation (Aminfar et al., 2016).

1.4.1.3 Algebra on low-rank matrices

Classical linear algebra operations involving low-rank matrices can be accelerated by taking advantage of their low-rank property.

The triangular solve operation

$$A \leftarrow AL^{-1},$$

where A is a matrix of size $m \times n$ and L is a triangular matrix of order n, requires mn^2 operations if A is full-rank. However, if $A \approx \widetilde{A} = XY^T$ is low-rank of rank k, the operation

$$\widetilde{A} \leftarrow \widetilde{A}L^{-1} = X(Y^TL^{-1})$$

can be computed by

$$Y \leftarrow L^{-T}Y,$$

and thus only requires n^2k operations. Therefore, exploiting the low-rank form of a matrix to perform a triangular solve is always beneficial, regardless of its rank.

The matrix-matrix product operation

$$C \leftarrow AB,$$

where A,B,C are matrices of size $m \times n$, $n \times p$, and $m \times p$, respectively, requires $2mnp$ operations if both A and B are full-rank. If A is low-rank of rank k, the LR-FR product

$$C \leftarrow \widetilde{A}B = X(Y^TB)$$

consists of two operations: the *inner product* $Z = Y^TB$, which requires $2npk$ operations, and the *outer product* $C \leftarrow XZ$, which requires $2mpk$ operations, for a total of $2(m+n)pk$. Thus, using the low-rank form of A is beneficial if $k < mn/(m+n)$, just as for the storage requirements. The analysis is similar if B is low-rank instead. Finally, if both A and B are low-rank, of ranks k_A and k_B, then the LR-LR product

$$C \leftarrow \widetilde{A}\widetilde{B} = X_A(Y_A^TX_B)Y_B^T$$

also consists of inner and outer products. The inner product first computes $Z \leftarrow Y_A^TX_B$, at the cost of $2nk_Ak_B$ operations. The rest of the computations depend on which side Z is multiplied:

- If we multiply Z to the right ($W \leftarrow ZY_B^T$), which requires $2pk_Ak_B$ operations, then the outer product computes $C \leftarrow X_AW$, at the cost of $2mpk_A$, which leads to a total of $2(n+p)k_Ak_B + 2mpk_A$.

- If we multiply Z to the left ($W \leftarrow X_AZ$), which requires $2mk_Ak_B$ operations, then the outer product computes $C \leftarrow WY_B^T$, at the cost of $2mpk_B$, which leads to a total of $2(n+m)k_Ak_B + 2mpk_B$.

The side that minimizes the computations thus depends on the sign of

$$mp(k_A - k_B) + (p-m)k_Ak_B.$$

Note that if C is a square matrix ($m = p$), this simplifies to the sign of $k_A - k_B$. Also note that if the full-rank form of C is not needed, the outer product can be skipped, since the result of the inner product yields a low-rank form of C.

Finally, let us consider the matrix sum operation

$$C \leftarrow A + B,$$

where A, B, and C are all of size $m \times n$. The full-rank sum requires mn operations. If A is low-rank but not B, then A must be decompressed back to full-rank (by means of an outer product of cost $2mnk$) before performing the sum in full-rank. Similarly, if only B is low-rank, it must be decompressed. However, if both A and B are low-rank, a low-rank form of C can be obtained at no cost:

$$\widetilde{C} = \widetilde{A} + \widetilde{B} \Leftrightarrow X_CY_C^T = (X_A\ X_B)(Y_A\ Y_B)^T.$$

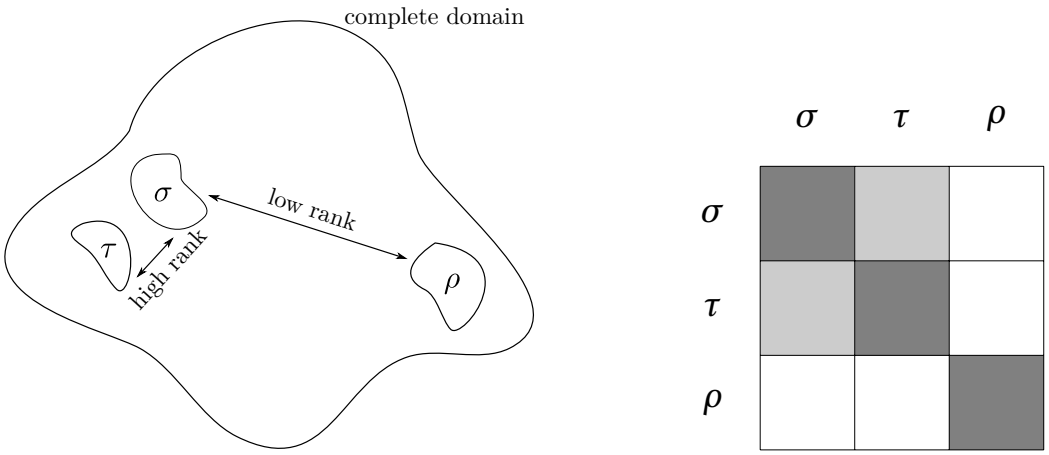
Note that in this case, we obtain a rank of C equal to $k_C = k_A + k_B$, which can be significantly larger than the actual rank. It is thus a good idea to recompress C. This will be further discussed in the context of the BLR factorization in Section 2.6.

1.4.2 Low-rank matrix formats

In this section, we present the different low-rank matrix representations that exist to take advantage of the low-rank property arising in many applications. These so-called *low-rank formats* can be distinguished and classified based on three criteria:

- The block-admissibility condition (Section 1.4.2.1): strong or weak?
- The block-clustering (Section 1.4.2.2): flat or hierarchical?
- The low-rank basis (Section 1.4.2.3): nested or not?

1.4.2.1 Block-admissibility condition



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

(b) Corresponding block-clusters in the matrix.

Figure 1.14 – Illustration of the block-admissibility condition. The dark grey blocks represent self-interactions and are full-rank; the light grey blocks represent near interactions and are high-rank; the white blocks represent distant interactions and are low-rank.

In the context of the solution of linear systems $Ax = b$, A is not usually low-rank itself. However, in many applications, some of its off-diagonal blocks are. Indeed, in the context of the solution of some discretized PDE, a block B corresponds to an interaction between two subdomains σ and τ, where σ contains the row indices of B while τ contains its column indices, as illustrated by Figure 1.14. σ and τ are referred to as *clusters*, while $B = \sigma \times \tau$ is referred to as *block-cluster*.

The rank of a given block-cluster B depends on the interaction it represents, as illustrated by Figure 1.14. Indeed, if B is a diagonal block, it represents a *self-interaction* $\sigma \times \sigma$ and is thus full-rank. However, if B is an off-diagonal block, it may

be either full-rank or low-rank depending on the interaction $\sigma \times \tau$ it represents: the weaker the interaction, the lower the rank.

This is formalized with a key concept called the *admissibility* condition. The block-admissibility condition determines whether a block $\sigma \times \tau$ is admissible for low-rank compression. The standard block-admissibility condition, also called *strong* block-admissibility, takes the following form:

$$\sigma \times \tau \text{ is admissible } \Leftrightarrow \max(\operatorname{diam}(\sigma), \operatorname{diam}(\tau)) \leq \eta \operatorname{dist}(\sigma, \tau), \tag{1.25}$$

where $\eta > 0$ is a fixed parameter. Condition (1.25) formalizes the intuition that the rank of a block $\sigma \times \tau$ is correlated to the distance between σ and τ: the greater the distance, the weaker the interaction, the smaller the rank; that distance is to be evaluated relatively to the subdomain diameters.

The η parameter controls how strict we are in considering a block admissible. The smaller the η, the fewer admissible blocks. On the contrary, if we choose

$$\eta_{max} = \max_{\substack{\sigma,\tau \in \mathfrak{S}(\mathscr{I}) \\ \operatorname{dist}(\sigma,\tau)>0}} \frac{\max(\operatorname{diam}(\sigma), \operatorname{diam}(\tau))}{\operatorname{dist}(\sigma, \tau)}, \tag{1.26}$$

where $\mathfrak{S}(\mathscr{I})$ is a clustering defining the subdomains (see next section), then condition (1.25) can be simplified into the following condition, that we call *least-restrictive* strong block-admissibility:

$$\sigma \times \tau \text{ is admissible } \Leftrightarrow \operatorname{dist}(\sigma, \tau) > 0. \tag{1.27}$$

Finally, there is an even less restrictive admissibility condition, called *weak* block-admissibility:

$$\sigma \times \tau \text{ is admissible } \Leftrightarrow \sigma \neq \tau. \tag{1.28}$$

With the weak admissibility, even blocks that correspond to neighbors (subdomains at distance zero) are admissible, as long as they are not self-interactions (i.e., the diagonal blocks).

1.4.2.2 Flat and hierarchical block-clusterings

Before deciding which block is admissible and which is not, an essential step is to compute a *block-clustering* that defines these blocks, i.e., that determines which variable goes into which block-cluster.

The clustering of the unknowns (noted $\mathscr{I}$ hereinafter) is formalized as a partition $\mathfrak{S}(\mathscr{I})$ of $\mathscr{I}$. In general, the row and column indices can be clustered differently. In the following, for the sake of simplicity, and without loss of generality, we assume that the row and column indices are identically clustered. Then, the block-clustering is formalized as a partition $\mathfrak{S}(\mathscr{I} \times \mathscr{I})$ of $\mathscr{I} \times \mathscr{I}$.

Flat block-clustering A special type of block-clustering, referred to as *flat*, can be defined solely based on the partition $\mathfrak{S}(\mathscr{I})$.

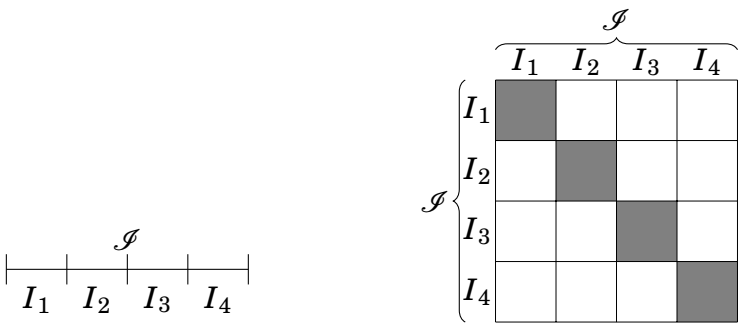
Definition 1.9 (Flat block-clustering)**.** *Let $\mathfrak{S}(\mathscr{I})$ be a partition of $\mathscr{I}$. Then we define the flat block-clustering $\mathfrak{S}(\mathscr{I}\times\mathscr{I})$ associated with $\mathfrak{S}(\mathscr{I})$ as*

$$\mathfrak{S}(\mathscr{I}\times\mathscr{I}) = \mathfrak{S}(\mathscr{I})\times\mathfrak{S}(\mathscr{I}).$$

The flat block-clustering associated with $\mathfrak{S}(\mathscr{I})$ verifies the following property:

$$(\sigma,\tau)\in\mathfrak{S}(\mathscr{I})^2 \text{ iff } \sigma\times\tau\in\mathfrak{S}(\mathscr{I}\times\mathscr{I}).$$

An example of clustering $\mathfrak{S}(\mathscr{I}) = \{I_1, I_2, I_3, I_4\}$ and its associated flat block-clustering are given in Figure 1.15.



(a) Clustering $\mathfrak{S}(\mathscr{I})$. (b) Flat block-clustering.

Figure 1.15 – An example of clustering and its associated flat block-clustering.

The closely related Block Separable (BS) and Block Low-Rank (BLR) formats are two examples of low-rank representations based on a flat, non-hierarchical block-clustering. The BS format was introduced in Cheng et al. (2005) (Section 5.2) and described in Gillman, Young, and Martinsson (2012) and Gillman (2011). A Block Separable matrix $\widetilde{A}$ can be written as $D+UBV^T$, where D is a block-diagonal matrix and UBV^T is a low-rank term. Thus, the off-diagonal blocks of a BS matrix are all assumed to be low-rank, i.e. the BS format is weakly admissible.

The BLR format, introduced Amestoy et al. (2015a), is more general in the sense that it allows some of the off-diagonal blocks to be full-rank (i.e. it is strongly admissible).

Definition 1.10 (BLR matrix)**.** *Assuming the unknowns have been partitioned into p clusters, and that a permutation P has been defined so that permuted variables of a given cluster are contiguous, a BLR representation $\widetilde{A}$ of a dense matrix A is of the form*

$$\widetilde{A} = \begin{bmatrix} A_{1,1} & \widetilde{A}_{1,2} & \cdots & \widetilde{A}_{1,p} \\ \widetilde{A}_{2,1} & \cdots & \cdots & \vdots \\ \vdots & \cdots & \cdots & \vdots \\ \widetilde{A}_{p,1} & \cdots & \cdots & A_{p,p} \end{bmatrix}.$$

Subblocks $A_{i,j} = (PAP^T)_{i,j}$, of size $m_{i,j}\times n_{i,j}$ and numerical rank $k^{\varepsilon}_{i,j}$, are approximated by a low-rank product $\widetilde{A}_{i,j} = X_{i,j}Y^T_{i,j}$ at accuracy ε, where $X_{i,j}$ is a $m_{i,j}\times k^{\varepsilon}_{i,j}$

matrix and $Y_{i,j}$ *is a* $n_{i,j} \times k^{\epsilon}_{i,j}$ *matrix.*

For the sake of simplicity, and without loss of generality, we will assume in the following that, for a given matrix $\widetilde{A}$, the property $\forall i,j \;\; m_{i,j} = n_{i,j} = b$ holds, where b, the block size, can depend on the order of the matrix.

The BLR format is at the heart of the work presented in this thesis. Chapter 2 is dedicated to the discussion on how to perform the LU or LDL^T factorization of a BLR matrix.

Hierarchical block-clusterings Much more general block-clusterings $\mathfrak{S}(\mathscr{I} \times \mathscr{I})$ can be considered. In the literature, so-called *cluster trees* have been introduced as a convenient way to describe them.

Definition 1.11 (Cluster tree)**.** *Let* $\mathscr{I}$ *be a set of unknowns and* $T_{\mathscr{I}}$ *a tree whose nodes* v *are associated with subsets* σ_v *of* $\mathscr{I}$*.* $T_{\mathscr{I}}$ *is said to be a cluster tree iff:*

- *The root of* $T_{\mathscr{I}}$*, noted* r*, is associated with* $\sigma_r = \mathscr{I}$*;*
- *For each node* $v \in T_{\mathscr{I}}$*,* $\sigma_v \subset \mathscr{I}$ *is contiguous;*
- *For each non-leaf node* $v \in T_{\mathscr{I}}$*, with children noted* $C_{T_{\mathscr{I}}}(v)$*, the subsets associated with the children form a partition of* t_v*, i.e.*

$$\bigcup_{c \in C_{T_{\mathscr{I}}}(v)} \sigma_c = \sigma_v.$$

An example of cluster tree is provided in Figure 1.16a. As illustrated, cluster trees establish a *hierarchy* between clusters. In fact, for a given cluster tree, one can uniquely define an associated weakly admissible hierarchical block-clustering. This particular class of block-clusterings, referred to as Hierarchically Off-Diagonal Low-Rank (HODLR), has been studied in particular by Aminfar et al. (2016).

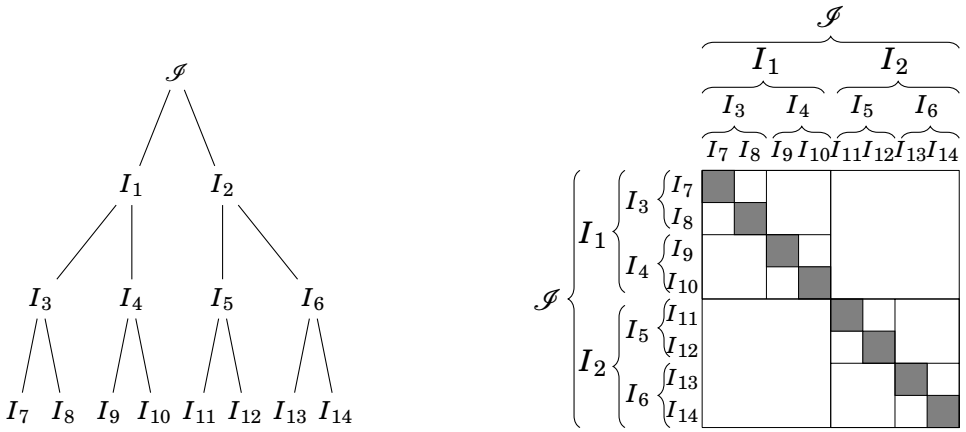
Definition 1.12 (HODLR matrix)**.** *The HODLR block-clustering* $\mathfrak{S}(\mathscr{I} \times \mathscr{I})$ *associated with a cluster tree* $T_{\mathscr{I}}$ *is defined as*

$$\mathfrak{S}(\mathscr{I} \times \mathscr{I}) = \left\{ \sigma_{v_1} \times \sigma_{v_2};\ v_1 \text{ and } v_2 \text{ are siblings in } T_{\mathscr{I}} \text{ or } v_1 = v_2 \right\}.$$

The HODLR matrix associated with the previous example cluster tree is provided in Figure 1.16b.

Thus, the off-diagonal blocks of an HODLR matrix are not refined but instead are directly approximated as low-rank matrices. To define more general block-clustering $\mathfrak{S}(\mathscr{I} \times \mathscr{I})$, that are based on the strong block-admissibility condition, a new tree structure, so-called *block-cluster tree*, must be introduced.

Definition 1.13 (Block-cluster tree)**.** *Let* $\mathscr{I}$ *be a set of unknowns,* $T_{\mathscr{I}}$ *a cluster tree, and* $T_{\mathscr{I}\times\mathscr{I}}$ *a tree whose nodes are of the form* $v \times w$*, for* $v, w \in T_{\mathscr{I}}$*.* $v \times w$ *is thus associated with a subset* $\sigma_v \times \sigma_w$ *of* $\mathscr{I} \times \mathscr{I}$*.* $T_{\mathscr{I}\times\mathscr{I}}$ *is said to be a block-cluster tree (associated with* $T_{\mathscr{I}}$*) iff:*



(a) Cluster tree.

(b) HODLR block-clustering

Figure 1.16 – An example of cluster tree and its associated HODLR block-clustering.

- *The root of $T_{\mathcal{I}\times\mathcal{I}}$ is associated with $\mathcal{I}\times\mathcal{I}$;*
- *For each node $v\times w \in T_{\mathcal{I}\times\mathcal{I}}$, σ_v and σ_w are contiguous or $v = w$;*
- *For each non-leaf node $v\times w \in T_{\mathcal{I}\times\mathcal{I}}$, the children of $v\times w$, noted $C_{T_{\mathcal{I}\times\mathcal{I}}}(v\times w)$ are associated with subsets of the form*

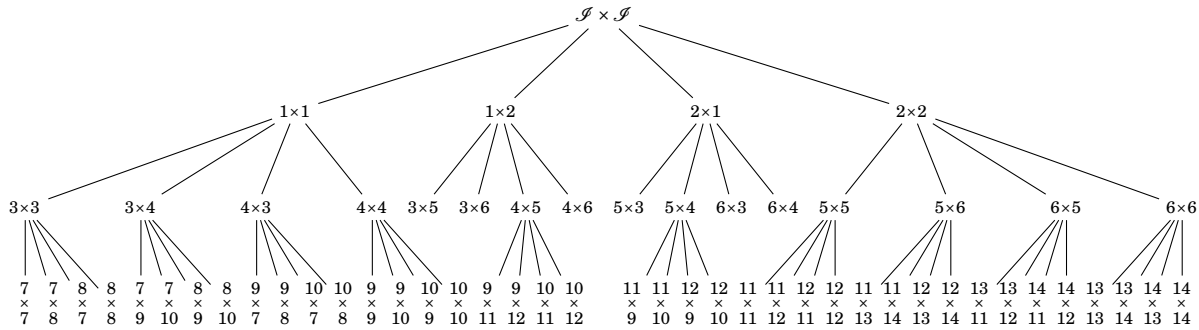
$$\begin{cases} \sigma_c \times \sigma_w, & \text{with } c \in C_{T_\mathcal{I}}(v) & \text{if } C_{T_\mathcal{I}}(v) \neq \emptyset, C_{T_\mathcal{I}}(w) = \emptyset; \\ \sigma_v \times \sigma_c, & \text{with } c \in C_{T_\mathcal{I}}(w) & \text{if } C_{T_\mathcal{I}}(v) = \emptyset, C_{T_\mathcal{I}}(w) \neq \emptyset; \\ \sigma_{c_1} \times \sigma_{c_2}, & \text{with } (c_1, c_2) \in C_{T_\mathcal{I}}(v) \times C_{T_\mathcal{I}}(w) & \text{otherwise.} \end{cases}$$

Note that if $C_{T_\mathcal{I}}(v) = C_{T_\mathcal{I}}(w) = \emptyset$, then $v\times w$ is necessarily a leaf of $T_{\mathcal{I}\times\mathcal{I}}$.

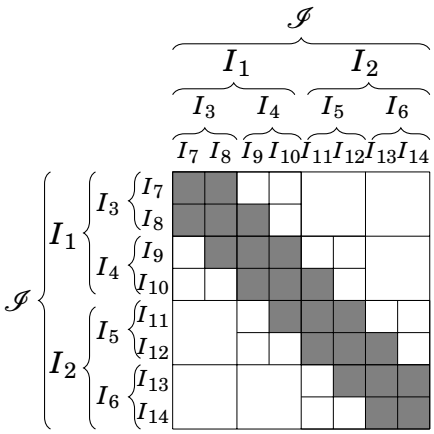
For a given node $v\times w \in T_{\mathcal{I}\times\mathcal{I}}$, note that, while it must be a leaf in $T_{\mathcal{I}\times\mathcal{I}}$ if both v and w are leaves in $T_\mathcal{I}$, it does not necessarily have children otherwise. Thus, there are many possible block-cluster trees associated with a given cluster tree, depending on which nodes are *refined*. However, we are only interested in a subset of them, referred to as *admissible block-cluster trees*.

Definition 1.14 (Admissible block-cluster tree)**.** *Let $T_{\mathcal{I}\times\mathcal{I}}$ be a block-cluster tree associated with a cluster tree $T_\mathcal{I}$. $T_{\mathcal{I}\times\mathcal{I}}$ is said to be admissible iff all leaves $v\times w$ of $T_{\mathcal{I}\times\mathcal{I}}$ are associated with a subset $\sigma_v\times\sigma_w$ that is either block-admissible (in the sense of one of the block-admissibility conditions defined in Section 1.4.2.1) or small enough (i.e. $\max(\#\sigma_v, \#\sigma_w) \leq c_{min}$).*

The block-clusterings associated with an admissible block-cluster tree are referred to as $\mathcal{H}$ block-clusterings (where $\mathcal{H}$ stands for hierarchical). The matrices they represent are known as $\mathcal{H}$-matrices. An example of an admissible block-cluster tree and its associated $\mathcal{H}$-matrix is provided in Figure 1.17. Note that, because the block-cluster tree of Figure 1.17a is balanced, all non-leaf nodes are of the form $\sigma_{c_1}\times\sigma_{c_2}$ (i.e. the first two forms in Definition 1.13 do not occur).



(a) Block-cluster tree ($I_i \times I_j$ has been abbreviated as $i \times j$).



(b) Associated block-clustering.

Figure 1.17 – An example of block-cluster tree and its associated block-clustering. The tree is admissible (and is then associated with an $\mathcal{H}$ block-clustering) if the grey blocks are smaller than c_{min} and the white ones are admissible.

.

The $\mathcal{H}$-matrix format, introduced by Hackbusch and his collaborators (Hackbusch, 1999; Grasedyck and Hackbusch, 2003; Börm et al., 2003; Bebendorf, 2008), is a strongly admissible format. It is the most general low-rank format (without nested basis, see next section). Indeed, if the weak admissibility condition is used in Definition 1.14, the resulting admissible block-cluster tree defines an HODLR matrix. Furthermore, a full block-cluster tree (i.e. if $v \times w$ is a leaf in $T_{\mathcal{I}\times\mathcal{I}}$, then v and w are leaves in $T_{\mathcal{I}}$) defines a BLR matrix. Therefore, both HODLR and BLR matrices are a particular kind of $\mathcal{H}$-matrices.

Both the $\mathcal{H}$ format and its weakly admissible counterpart HODLR can factorize a dense matrix of order n with $\mathcal{O}(r^2 n \log^2 n)$ (Grasedyck and Hackbusch, 2003) operations, where r is the maximal rank of the blocks. This complexity can be reduced to $\mathcal{O}(r^2 n)$ by exploiting a so-called *nested basis* property.

1.4.2.3 Low-rank formats with nested basis

The $\mathcal{H}^2$ format (Börm et al., 2003) is a general, strongly admissible, nested format. The particular case where a weak admissibility condition is used has been

the object of numerous studies. In this case, the format is referred to as Hierarchically Semi-Separable (HSS) (Xia et al., 2010; Chandrasekaran, Gu, and Pals, 2006). Because the presentation of nested low-rank formats is simpler with a weak admissibility condition, and because Chapter 8 is dedicated to comparing the BLR and HSS formats, we focus on the HSS format in this section.

A Hierarchically Block-Separable (HBS) format has been proposed by Gillman et al. (2012). It is closely related to HSS and we therefore do not discuss it further.

The HSS format

Definition 1.15 (HSS tree and HSS matrix)**.** *Let A be a dense matrix associated with a set of unknowns $\mathcal{I}$ and let $T_{\mathcal{I}}$ be a cluster tree defining a clustering of $\mathcal{I}$. $T_{\mathcal{I}}$ is said to be an* HSS tree *iff:*

- *It is a complete binary tree*[1]*;*
- *Each node $v \subset \mathcal{I}$ of $T_{\mathcal{I}}$ is associated with matrices D_v, U_v^{big}, U_v, V_v^{big}, V_v, and B_v, called* HSS generators*, which satisfy the following recursions if v is a non-leaf node with children c_1 and c_2:*

$$D_v = \begin{pmatrix} D_{c_1} & U_{c_1} B_{c_1} (V_{c_2}^{big})^T \\ U_{c_2}^{big} B_{c_2} (V_{c_1}^{big})^T & D_{c_2} \end{pmatrix}, \tag{1.29}$$

$$U_v^{big} = \begin{pmatrix} U_{c_1}^{big} & 0 \\ 0 & U_{c_2}^{big} \end{pmatrix} U_v, \tag{1.30}$$

$$\text{and } V_v^{big} = \begin{pmatrix} V_{c_1}^{big} & 0 \\ 0 & V_{c_2}^{big} \end{pmatrix} V_v. \tag{1.31}$$

In that case, A can be represented by an HSS matrix *defined as $\widetilde{A} = D_r$, where r is the root node.*

Of course, D_r is not explicitly stored but is instead implicitly represented by the recursive relation (1.29). For example, for the 3-level HSS tree given in Figure 1.18, $\widetilde{A}$ is of the form

$$\widetilde{A} = \left[\begin{array}{cc|cc} D_1 & U_1^{big} B_1 (V_2^{big})^T & & \\ U_2^{big} B_2 (V_1^{big})^T & D_2 & U_3^{big} B_3 (V_6^{big})^T & \\ \hline & U_6^{big} B_6 (V_3^{big})^T & D_4 & U_4^{big} B_4 (V_5^{big})^T \\ & & U_5^{big} B_5 (V_4^{big})^T & D_5 \end{array}\right]. \tag{1.32}$$

Thus, D_v is explicitly stored only when node v is a leaf. However, this property alone is not enough to differentiate the HSS format from the HODLR one, which also relies on a recursive approximation of the diagonal blocks. With an HSS matrix, a key property is that the low-rank basis U_v^{big} and V_v^{big} are also implicitly generated with

[1] In the literature, more general HSS tree are described (e.g. in Xia et al. (2010)); we focus on complete binary trees for the sake of simplicity.

the U_v and V_v generators, as described by the recursive relations (1.30) and (1.31) (hence their *nested* property). Thus, the HSS matrix $\widetilde{A}$ can finally be written as

$$\widetilde{A} = \left[\begin{array}{cc|c} D_1 & U_1 B_1 V_2^T & \begin{bmatrix} U_1 & 0 \\ 0 & U_2 \end{bmatrix} U_3 B_3 V_6^T \begin{bmatrix} V_4^T & 0 \\ 0 & V_5^T \end{bmatrix} \\ U_2 B_2 V_1^T & D_2 & \\ \hline \begin{bmatrix} U_4 & 0 \\ 0 & U_5 \end{bmatrix} U_6 B_6 V_3^T \begin{bmatrix} V_1^T & 0 \\ 0 & V_2^T \end{bmatrix} & & \begin{array}{cc} D_4 & U_4 B_4 V_5^T \\ U_5 B_5 V_4^T & D_5 \end{array} \end{array}\right]. \tag{1.33}$$

Thus, U_v^{big} and V_v^{big} are also explicitly stored only when node v is a leaf, in which case $U_v = U_v^{big}$ and $V_v = V_v^{big}$. This allows for some savings in storage and operations, which drops the logarithmic factor in the complexity.

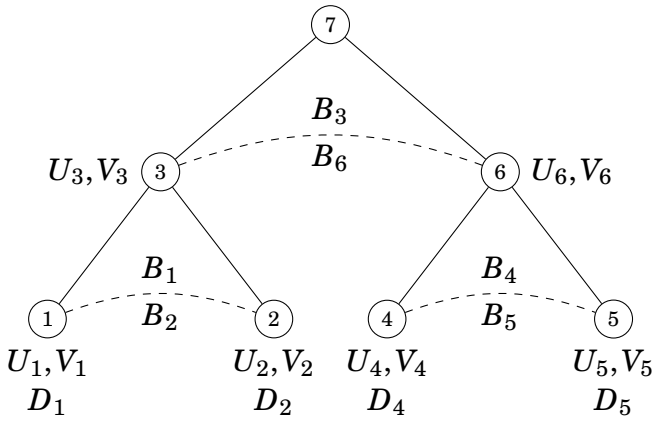


Figure 1.18 – HSS tree associated with the HSS matrix defined by equation (1.33).

For algorithms to build an HSS representation of a dense matrix and to use such a representation to accelerate classical linear algebra operations, such as matrix-vector product and LU factorization, we refer the reader to Xia et al. (2010).

1.4.2.4 Taxonomy of low-rank formats

Based on the three criteria (partitioning type, admissibility condition, and nested basis) described in the previous sections, the existing low-rank formats can be classified. This taxonomy is reported in Table 1.2.

admissibility condition	partitioning type: flat	hierarchical: non nested	hierarchical: nested
weak	BS	HODLR	HSS/HBS
strong	BLR	$\mathscr{H}$	$\mathscr{H}^2$

Table 1.2 – Taxonomy of existing low-rank formats.

1.4.3 Using low-rank formats within sparse direct solvers

Data sparsity can be efficiently exploited within sparse direct solvers to provide a substantial reduction of their complexity. Because the multifrontal method relies on dense factorizations, low-rank formats can be incorporated into the multifrontal factorization by representing the frontal matrices with one of the previously described low-rank formats.

1.4.3.1 Block-clustering of the fronts

One key aspect is the computation of the block-clustering of the frontal matrices. In the following discussion, we assume that the assembly tree is built by means of a nested dissection (George, 1973); this better suits the context of our work and allows for an easier understanding of how low-rank approximation techniques can be used within sparse multifrontal solvers.

Each frontal matrix is associated with a separator in the tree. The fully-summed variables of a frontal matrix match the variables of the separator. The non fully-summed variables of a front form a border of the separator's subtree and correspond to pieces of ancestor separators found higher in the separator tree. Thus, on each front, the two types of variables must be clustered. Two strategies have been proposed in Weisbecker (2013):

- *Explicit clustering*: each front is clustered independently; this leads to very balanced cluster sizes inside a given front but can be very expensive since each variable is clustered several times (as many times as fronts it belongs to).

- *Inherited clustering*: each separator is clustered independently; thus, each variable is only clustered once, leading to a much cheaper cost. For each front, the clustering of the fully-summed variables is directly derived from the clustering of the associated separator, while that of the non fully-summed variables is *inherited* from the separator they belong to (which corresponds to a front higher in the tree). Thus, the cluster sizes can be more unbalanced inside a given front (since each separator can be clustered with a different target size).

In Chapter 4, we will motivate the necessity to have a target cluster size that varies depending on the clustered separator size (usually the cluster size should be of the order of the square root of the separator size) and thus inherited clustering can potentially lead to fronts with relatively unbalanced cluster sizes. However, explicit clustering has been shown in Weisbecker (2013) to be unreasonably expensive and thus we will not consider it.

Thus, the block-clustering of the fronts can be derived from the block-clustering of the separators. We now discuss how to compute the latter. The computed clusters should respect the admissibility condition. However, this requires geometric information to compute the diameter and distances. To remain in a purely algebraic context, the adjacency graph of the matrix A can be used instead. The admissibility condition must then be adapted: Grasedyck, Kriemann, and Le Borne (2008)

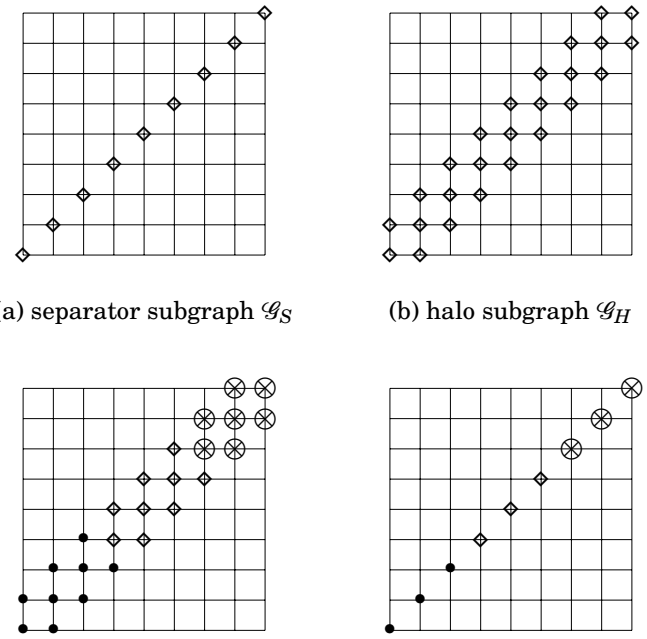
propose to substitute the geometric diameter and distance operations by algebraic graph operations, i.e.

$$\sigma \times \tau \text{ is admissible } \Leftrightarrow \max(\text{diam}_{\mathcal{G}(A)}(\sigma), \text{diam}_{\mathcal{G}(A)}(\tau)) \leq \eta \, \text{dist}_{\mathcal{G}(A)}(\sigma, \tau) \tag{1.34}$$

where σ and τ are two sets of graph points and $\text{diam}_{\mathcal{G}(A)}$ and $\text{dist}_{\mathcal{G}(A)}$ are the graph diameter and distance operations.

Furthermore, Weisbecker (2013) suggests simplifying this condition: indeed, other practical considerations must be considered, such as having a cluster size large enough to capture BLAS efficiency but small enough to be able to easily distribute the clusters in parallel. This relaxed clustering can be efficiently computed with a k-way partitioning of each separator subgraph.

A important issue is that the separator subgraph $\mathcal{G}_S$ of a given separator S is not necessarily connected, especially in a context where S has been computed algebraically. In that case, the clustering computed on a disconnected subgraph may be of poor quality (Weisbecker, 2013). A possible strategy to overcome this issue, proposed in Weisbecker (2013), and used in solvers such as MUMPS or STRUMPACK, is to cluster the separator subgraph together with its *halo subgraph* $\mathcal{G}_H$, defined as the set of neighbor points at a distance d_h or less; d_h is called the *halo depth* parameter. The clustering of $\mathcal{G}_S$ can then be derived from that of $\mathcal{G}_H$, as shown in Figure 1.19.



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

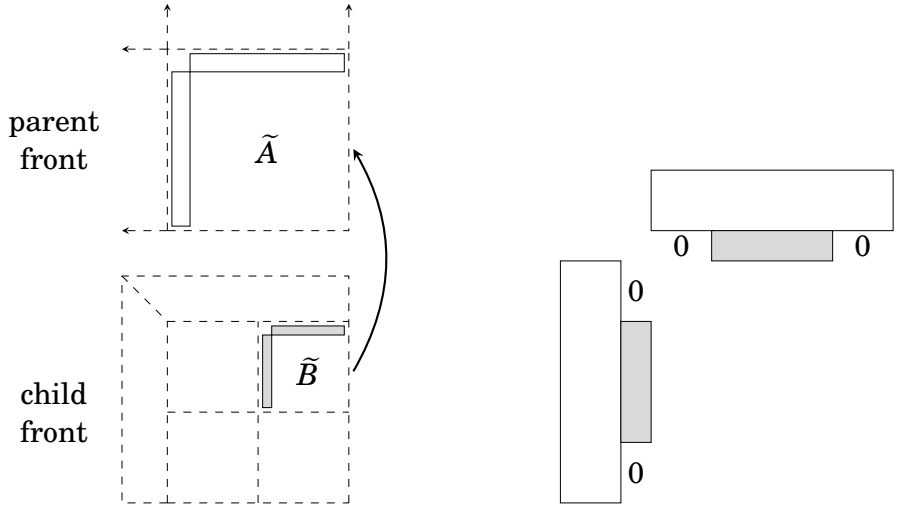
(c) 3-way partitioning of $\mathcal{G}_H$ (d) 3-way partitioning of $\mathcal{G}_S$

Figure 1.19 – Halo-based partitioning of $\mathcal{G}_S$ with depth 1.

1.4.3.2 Full-rank or low-rank assembly?

In addition to the front factorization, one may also exploit the low-rank property during the assembly. In this case, the fronts are directly assembled in low-rank format and are thus never stored in full-rank: for this reason, the factorization is referred to as *fully-structured* (Xia, 2013a).

The fully-structured factorization requires relatively complex low-rank extend-add operations. Indeed, let $\widetilde{A}$ and $\widetilde{B}$ be two low-rank blocks belonging to a parent front and a child front, respectively, and assume A and B share some variables in common, as shown in Figure 1.20a. In that case, we have to compute $\widetilde{S} = \widetilde{A} \dashv\!\vdash \widetilde{B}$. Doing that without decompressing $\widetilde{A}$ or $\widetilde{B}$ is not possible in general without further assumption.



(a) Assembly of $\widetilde{B}$ in $\widetilde{A}$. (b) Padding variables of $\widetilde{S}$.

Figure 1.20 – Low-rank extend-add operation.

However, if we use the inherited clustering strategy, we can guarantee that all the variables of $\widetilde{B}$ belong to $\widetilde{A}$ (but the inverse is not true in general). Thus, to compute $\widetilde{S}$ we can simply pad $\widetilde{S}$ with zeros corresponding to the variables of $\widetilde{A}$ that do not belong to $\widetilde{B}$, as illustrated in Figure 1.20b. Note that the variables of $\widetilde{B}$ can be assumed to be contiguous in $\widetilde{A}$ by simply sorting the variables of $\widetilde{A}$ in an adequate order.

Note that, in the hierarchical case, Martinsson (2011) proposed a randomized algorithm to perform a fully-structured HSS factorization. This algorithm somewhat simplifies the low-rank assembly operations (Martinsson, 2011; Xia, 2013b; Martinsson, 2016). It however cannot be directly applied to BLR matrices, and it is unclear whether it could be extended.

1.5 Experimental setting

1.5.1 The MUMPS solver

The MUltifrontal Massively Parallel Solver, or MUMPS (Amestoy et al., 2001; Amestoy et al., 2006; Amestoy et al., 2011), is a sparse direct solver based on

the multifrontal method and developed mainly in Toulouse, Lyon, and Bordeaux (France). The MUMPS project started in 1996 in the context of the European project PARASOL.

1.5.1.1 Main features

MUMPS provides a large range of features that make it a very robust, general purpose code:

- **Input**: MUMPS provides Fortran, C, and Matlab interfaces. The input matrix can be symmetric or unsymmetric; centralized or distributed; assembled or in elemental format. Real and complex, single and double precision arithmetics are supported.
- **Analysis phase**: several preprocessing strategies such as scaling and reordering can be used.
- **Factorization phase**: The LU, LDL^T, and Cholesky factorizations can all be performed and can benefit from both an MPI and/or OpenMP parallelization. Numerical robustness is guaranteed thanks to the use of threshold partial pivoting. Finally, the out-of-core (OOC) factorization is provided by writing asynchronously the factors to disk.
- **Solution phase**: The solution phase can be parallelized with both MPI and OpenMP; dense and sparse right-hand sides are supported; the solution can be centralized or distributed. Postprocessing features such as iterative refinement or the computation of the backward error are also available.
- **Miscellaneous**: Several other features are provided, such as the computation of the determinant, the partial factorization (computation of the Schur complement), and the detection of null-pivots (Rank Revealing factorization).

The BLR factorization and all its variants, which will be presented in Chapter 2, have been developed and integrated into the MUMPS solver, which was used to run all experiments and constitutes our reference Full-Rank solver.

1.5.1.2 Implementation details

Threshold partial pivoting is used. The threshold set to $\tau = 0.01$ for all experiments. Contrarily to the algorithms in most of the reference codes (e.g. LAPACK) and the literature, including this manuscript, row-major pivoting is performed in LU, to fit the row-major storage of the fronts. In parallel, the master process does not have access to the non fully-summed rows of the front and therefore pivoting is restricted to the fully-summed rows. The maximum norm of the non-fully summed part of each column is estimated following the restricted pivoting strategies described in Section 1.3.2.6 and based on Duff and Pralet (2007).

As explained in Section 1.2.4.2, the Factor+Solve operation can be efficiently implemented by using several levels of blocking. In MUMPS, we use two levels of blocking. The internal block size is set to 32 for all experiments. In FR, the

(external) panel size is constant and set to 128 for small and medium matrices and 256 for large ones. In BLR, it is chosen to be equal to the BLR block size b and is automatically set according to the theoretical result reported in Chapter 4, which states the block size should increase with the size of the fronts.

Both the nested-dissection matrix reordering and the BLR clustering of the unknowns are computed with METIS in a purely algebraic way (i.e., without any knowledge of the geometry of the problem domain). The BLR clustering is computed following the inherited clustering approach (cf. Section 1.4.3.1) with a halo depth of 1.

To compute the low-rank form of the blocks, we perform a truncated QR factorization with column pivoting (i.e., a truncated version of LAPACK's (Anderson et al., 1995) _geqp3 routine). We use an absolute tolerance (i.e. we stop the factorization after $|r_{k,k}| < \varepsilon$). To avoid the compression rate depending (too much) on the matrix norm, the original matrix is scaled, using an algorithm based on those discussed in Ruiz (2001) and Knight et al. (2014).

Using a low-rank representation for the smaller fronts does not pay off; therefore, we only compress fronts of order larger than 1000 and with 128 fully-summed variables or more.

Because of our purely algebraic context, we do not know which blocks are admissible and so we assume all off-diagonal blocks are admissible (which is equivalent to using a weak block-admissibility condition). Thus, in our experiments, we try to compress every off-diagonal block. If the prescribed accuracy ε is not achieved after a given number of steps k_{max}, we stop the compression and consider the block to be full-rank. In our complexity experiments (Chapter 4), we have set $k_{max} = b/2$, the rank after which the low-rank representation of a block is not beneficial anymore (in terms of both flops and memory) with respect to the full-rank one. In the rest of the experiments, where performance is more important than flops, we have set $k_{max} = b/4$.

1.5.2 Test problems

In the experiments presented throughout this thesis, we have used a variety of real-life problems coming from physics applications, as well as PDE generators for our complexity tests.

1.5.2.1 PDE generators

For our complexity experiments (mainly in Chapter 4, but also in Chapter 8), we have used the standard Poisson and Helmholtz operators.

The Poisson problem generates the symmetric positive definite matrix A from a 7-point finite-difference discretization of equation

$$\Delta u = f.$$

on a cubic domain with Dirichlet boundary conditions. We perform the computations in double-precision arithmetic.

The Helmholtz problem builds the matrix A as the complex-valued unsymmetric impedance matrix resulting from the finite-difference discretization of the heterogeneous Helmholtz equation, that is the second-order visco-acoustic time-harmonic wave equation

$$\left(-\Delta - \frac{\omega^2}{v(x)^2}\right) u(x,\omega) = s(x,\omega),$$

where ω is the angular frequency, $v(x)$ is the seismic velocity field, and $u(x,\omega)$ is the time-harmonic wavefield solution to the forcing term $s(x,\omega)$. The matrix A is built for an infinite medium. This implies that the input grid is augmented with PML absorbing layers. Frequency is fixed and equal to 4 Hz. The grid interval h is computed such that it corresponds to 4 grid point per wavelength. Computations are done in single-precision arithmetic.

1.5.2.2 Main set of real-life problems

Throughout this thesis, we use the following main set of real-life problems to assess the performance of our solver. These matrices come from three applications (seismic modeling, electromagnetic modeling, and structural mechanics). We complete the set with additional problems from the University of Florida Sparse Matrix Collection (UFSMC) (Davis and Hu, 2011).

Our first application is 3D seismic modeling and its study is the object of Section 7.1. The main computational bulk of frequency-domain Full Waveform Inversion (FWI) (Tarantola, 1984) is the solution of the forward problem, which takes the form of a large, single complex, sparse linear system. Each matrix corresponds to the finite-difference discretization of the Helmholtz equation at a given frequency (5, 7, and 10 Hz). In collaboration with the SEISCOPE consortium, we have shown how the use of BLR can reduce the computational cost of 3D FWI for seismic imaging on a real-life case-study from the North sea. We found that the biggest low-rank threshold ε for which the quality of the solution was still exploitable by the application was 10^{-3} and this is therefore the value we chose for the experiments on these matrices.

Our second application, studied in depth in Section 7.2, is 3D electromagnetic modeling applied to marine Controlled-Source Electromagnetic (CSEM) surveying, a widely used method for detecting hydrocarbon reservoirs and other resistive structures embedded in conductive formations (Constable, 2010). The matrices, arising from a finite-difference discretization of frequency-domain Maxwell equations, were used to carry out simulations over large-scale 3D resistivity models representing typical scenarios for the marine CSEM surveying. In particular, the S-matrices (S3, S21) correspond to the SEG SEAM model, a complex 3D earth model representative of the geology of the Gulf of Mexico. For this application, the biggest acceptable low-rank threshold is $\varepsilon = 10^{-7}$.

Our third application is 3D structural mechanics, in the context of the industrial applications from Électricité De France (EDF). 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, which are often based on numerical simulations. These

simulations are carried out using Code_Aster[2] and require the solution of sparse linear systems. A previous study (Weisbecker, 2013) showed that using BLR with $\varepsilon = 10^{-9}$ leads to an accurate enough solution for this class of problems.

To demonstrate the generality and robustness of our solver, we complete our set of problems with matrices coming from the UFSMC arising in several fields: computational fluid dynamics, structural mechanics and optimization.

The complete set of matrices used throughout this thesis and their description is provided in Table 1.3. Each application is separated by a solid line while each problem subclass is separated by a dashed line. We assign an ID to each matrix, that we will use to refer to them in subsequent chapters (mainly, Chapters 5 and 6). Note that the flops associated with a given matrix are provided in its corresponding arithmetic, i.e. we count complex flops if the matrix is complex.

application	matrix	ID	arith.	fact. type	n	nnz	flops	factor size
seismic modeling (SEISCOPE)	5Hz	1	c	LU	2.9M	70M	69.5 TF	61.4 GB
	7Hz	2	c	LU	7.2M	177M	471.1 TF	219.6 GB
	10Hz	3	c	LU	17.2M	446M	2.7 PF	728.1 GB
electromagnetic modeling (EMGS)	H3	4	z	LDL^T	2.9M	37M	57.9 TF	77.5 GB
	H17	5	z	LDL^T	17.4M	226M	2.2 PF	891.1 GB
	S3	6	z	LDL^T	3.3M	43M	78.0 TF	94.6 GB
	S21	7	z	LDL^T	20.6M	266M	3.2 PF	1.1 TB
structural mechanics (EDF Code_Aster)	perf008d	8	d	LDL^T	1.9M	81M	101.0 TF	52.6 GB
	perf008ar	9	d	LDL^T	3.9M	159M	377.5 TF	129.8 GB
	perf008cr	10	d	LDL^T	7.9M	321M	1.6 PF	341.1 GB
	perf009ar	11	d	LDL^T	5.4M	209M	23.6 TF	40.5 GB
computational fluid dynamics (UFSMC)	StocF-1465	12	d	LDL^T	1.5M	11M	4.7 TF	9.6 GB
	atmosmodd	13	d	LU	1.3M	9M	13.8 TF	16.7 GB
	HV15R	14	d	LU	2.0M	283M	1.9 PF	414.1 GB
structural problems (UFSMC)	Serena	15	d	LDL^T	1.4M	33M	31.6 TF	23.1 GB
	Geo_1438	16	d	LU	1.4M	32M	39.3 TF	41.6 GB
	Cube_Coup_dt0	17	d	LDL^T	2.2M	65M	98.9 TF	55.0 GB
	Queen_4147	18	d	LDL^T	4.1M	167M	261.1 TF	114.5 GB
DNA electrophoresis (UFSMC)	cage13	19	d	LU	0.4M	7M	80.1 TF	35.9 GB
	cage14	20	d	LU	1.5M	27M	4.1 PF	442.7 GB
optimization (UFSMC)	nlpkkt80	21	d	LDL^T	1.1M	15M	15.1 TF	14.4 GB
	nlpkkt120	22	d	LDL^T	3.5M	50M	248.4 TF	86.5 GB

Table 1.3 – Main set of matrices and their Full-Rank statistics.

1.5.2.3 Complementary test problems

In Table 1.4, we provide the description of some complementary test problems that are used in various chapters of this thesis. The matrices come from structural and optimization problems from the UFSMC, a magnetohydrodynamics application from the LBNL, and a reservoir simulation problem from the SPE10 bench-

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

mark. These problems will be used in Chapter 8 for the comparison with the STRUMPACK solver, as well as in Section 9.1 for a study on the BLR clustering.

application	matrix	arith.	fact. type	n	nnz	flops	factor size
magnetohydrodynamics (LBNL)	A30	d	LU	0.6M	123M	30.6 TF	28.3 GB
	A22	d	LU	0.6M	127M	33.8 TF	29.4 GB
structural problems (UFSMC)	PFlow_742	d	LDL^T	0.7M	37M	2.8 TF	8.5 GB
	audi_kw1	d	LDL^T	0.9M	39M	5.9 TF	10.5 GB
	Hook_1498	d	LDL^T	1.5M	31M	7.9 TF	12.2 GB
	ML_Geer	d	LU	1.5M	111M	4.3 TF	16.4 GB
	Transport	d	LU	1.6M	23M	10.7 TF	21.0 GB
reservoir simulation (SPE10)	spe10-aniso	d	LU	1.2M	31M	11.6 TF	18.9 GB
optimization (UFSMC)	kkt_power	d	LU	2.1M	8M	0.8 TF	2.3 GB
seismic modeling (SEISCOPE)	15Hz	c	LU	58.0M	1523M	29.6 PF	3.7 TB
	20Hz	c	LU	129.9M	3432M	150.0 PF	11.0 TB
electromagnetic modeling (EMGS)	D5	z	LDL^T	90.1M	1168M	29.2 PF	6.1 TB
structural mechanics (EDF)	perf008ar2	d	LDL^T	31.1M	1267M	24.1 PF	2.1 TB

Table 1.4 – Complementary set of matrices and their Full-Rank statistics.

In Section 2.3.2.3, we also consider a set of matrices from the UFSMC that require numerical pivoting (cf. Table 2.2).

Finally, we provide in Table 1.4 the description of a set of very large matrices from our three applications described above: 15Hz and 20Hz (SEISCOPE), D5 (EMGS), and perf008ar2 (EDF). They will be used in the experiments presented in Section 9.4.

1.5.3 Computational systems

The following computational systems were used to perform the experiments presented in this thesis:

- **brunch**, a shared-memory machine equipped with 1.5 TB of memory and four Intel 24-cores Broadwell processors running at a frequency varying between 2.2 and 3.4 GHz, due to the turbo technology. **brunch** was used both for the complexity experiments of Chapters 4 and 8 and the multicore experiments of Chapter 5.
- **grunch**, a shared-memory machine equipped with 768 GB of memory and two Intel 14-cores Haswell processors running at 2.3 GHz. **grunch** was used for some experiments of Chapter 5.
- **eos**, the supercomputer of the Calcul en Midi-Pyrénées (CALMIP) center. Each of its 612 nodes is equipped with 64 GB of memory and two Intel 10-cores Ivy Bridge processors running at 2.8 GHz. The nodes are interconnected

with an Infiniband FDR network with bandwidth 6.89 GB/s. **eos** was used for the distributed-memory experiments of Chapters 6 and 9, as well as some experiments in Section 7.2.

- **licallo**, the supercomputer of the SIGAMM mesocenter at l'Observatoire de la Côte d'Azur (OCA). Each of its 102 nodes is equipped with 64 GB of memory and two Intel 10-cores Ivy Bridge processors running at 2.5 GHz. The nodes are interconnected with Infiniband FDR. **licallo** was used for the experiments of Section 7.1.

- **farad**, a shared-memory machine equipped with 264 GB of memory and two Intel 16-cores Sandy Bridge processors running at 2.9 GHz. **farad** was used for some of the experiments of Section 7.2.

- **cori**, the supercomputer of the National Energy Research Scientific Computing (NERSC) center. Each of its 2388 nodes is equipped with 128 GB of memory and two Intel 16-cores Haswell processors running at 2.3 GHz. The nodes are interconnected with a Cray Aries network with bandwidth 8 GB/s. **cori** was used for the performance experiments of Chapter 8, and for one experiment in Chapter 6.

- **eosmesca**, a cache coherent NUMA (ccNUMA) node, part of the **eos** supercomputer (CALMIP center). It is a shared-memory machine equipped with 2 TB of memory and eight Intel 16-cores Haswell processors running at a 2.2 GHz. **eosmesca** was used to run the very large problem D5 (90M unknowns) in Chapter 9.

- **occigen**, the supercomputer of the Centre Informatique National de l'Enseignement Supérieur (CINES). We used 200 nodes equipped with 128 GB of memory and two Intel 12-cores Haswell processors running at 2.6 GHz. The nodes are interconnected with an Infiniband FDR network with bandwidth 6.89 GB/s. **occigen** was used to run the very large problem 20Hz (130M unknowns) in Chapter 9.

CHAPTER

2 The Block Low-Rank Factorization

In this chapter, we present the Block Low-Rank factorization algorithm. In order to perform the LU or LDL^T factorization of a dense BLR matrix, the standard tile LU or LDL^T factorization has to be modified so that the low-rank sub-blocks can be exploited to reduce the number of operations. Because the low-rank sub-blocks are by nature stored separately, the most natural starting point is the full-rank tile factorization rather than its block version. In fact, the BLR factorization has also been referred to as Tile Low-Rank (TLR) in the literature (Akbudak, Ltaief, Mikhalev, and Keyes, 2017).

To compute the low-rank sub-blocks, a so-called Compress step must be added in Algorithm 1.3 or 1.5. Several variants can be easily defined depending on when the Compress step is performed. These variants are listed in Table 2.1. Furthermore, they can be combined with the accumulation and recompression of the low-rank updates (so-called LUAR algorithm), that improve the cost of the Update step. We introduce it at the end of this chapter, in Section 2.6.

The BLR factorization variants will be studied under different angles throughout this manuscript: asymptotic complexity in Chapter 4, parallel performance in Chapters 5 and 6, and numerical accuracy in this chapter.

RL name	LL name	description	Section
FSUC	UFS;C	Offline BLR compression	2.1
FSCU	UFSC	Standard BLR from Weisbecker (2013)	2.2
FCSU	UFCS	Compress before Solve with restricted pivoting	2.3.1
CFSU	UCFS	Compress before Solve with BLR pivoting	2.3.2
C;FSU	CUFS	Compress as soon as possible	2.4

Table 2.1 – The Block Low-Rank factorization variants. All these variants (except FSUC) can be combined with the LUAR algorithm described in Section 2.6. The semicolon notation is explained in Section 2.4.

We present each algorithm in their LDL^T version. The extension to the unsymmetric case (LU factorization) is mostly straightforward.

The algorithms in this chapter are described in the context of the factorization of frontal matrices. Their discussion however applies to broader contexts, such as

dense BLR solvers, but also supernodal (right- or left-looking) solvers.

2.1 Offline compression: the FSUC variant

We first briefly discuss the simplest of the variants, so-called FSUC, where the compression is performed after the factorization. We refer to this strategy as offline compression. Since the number of operations for the factorization is not reduced, it is not of interest in the case of the factorization of a dense matrix. However, in a multifrontal context, the frontal matrices can be compressed after their partial factorization, which allows for storing the factors in low-rank and thus reduce the total memory consumption.

Because the factorization is performed with machine-precision accuracy, and the approximated low-rank factors are only used during the solution phase, this variant can be of interest in the following contexts:

- Resolution of systems with many right-hand sides where the solution phase is the bottleneck, such as some of those studied in Chapter 7.
- Use of BLR factorization to build a preconditioner, where many iterations need to be performed.
- Resolution of numerically difficult problems where a very accurate factorization is needed but where the potential for compression can be exploited to reduce the memory consumption.

2.2 The standard BLR factorization: the FSCU and UFSC variants

2.2.1 The standard FSCU variant

We describe the standard BLR factorization, for dense matrices, introduced in Amestoy et al. (2015a) as the so-called FSCU variant, in Algorithm 2.1. In Algorithm 2.1 and all subsequent algorithms in this chapter, p_{fs} and p_{nfs} denote the number of fully-summed and non fully-summed block-rows (or block-columns), respectively.

This algorithm is referred to as FSCU (standing for Factor, Solve, Compress, and Update), to indicate the order in which the steps are performed. In particular, in Algorithm 2.1, the compression is performed after the so-called Solve step and thus the low-rank blocks are used to reduce the number of operations of the Update step only. In Section 2.3, we present variants where the compression is performed before the Solve step and thus the number of operations for the Solve is also reduced.

The Factor and Solve steps are merged together and performed via the full-rank Factor+Solve algorithm already presented in Section 1.3.2.6 (Algorithm 1.9). The Compress step is performed by means of one of the compression kernels presented in Section 1.4.1.2. Finally, the Update step is described in Algorithm 2.2 and is based on the operations described in Section 1.4.1.3. First the Inner Product step

Algorithm 2.1 Frontal BLR LDL^T (Right-looking) factorization: standard FSCU variant.

1: /* **Input:** a $p \times p$ block frontal matrix F; $F = [F_{i,j}]_{i=1:p,j=1:p}$; $p = p_{fs} + p_{nfs}$ */
2: **for** $k = 1$ **to** p_{fs} **do**
3: Factor+Solve: $F_{k:p,k} \leftarrow L_{k:p,k} D_{k,k} L_{k,k}^T$
4: **for** $i = k+1$ **to** p **do**
5: Compress: $F_{i,k} \approx \widetilde{F}_{i,k} = X_{i,k} Y_{i,k}^T$
6: **end for**
7: **for** $i = k+1$ **to** p **do**
8: **for** $j = k+1$ **to** i **do**
9: Update: $F_{i,j} \leftarrow$ RL-Update$(F_{i,j}, \widetilde{F}_{i,k}, \widetilde{F}_{j,k})$
10: **end for**
11: **end for**
12: **end for**
13: **for** $i = p_{fs}+1$ **to** p **do**
14: **for** $k = p_{fs}+1$ **to** i **do**
15: **if** CB compression is activated **then**
16: Compress: $F_{i,k} \approx \widetilde{F}_{i,k} = X_{i,k} Y_{i,k}^T$
17: **end if**
18: **end for**
19: **end for**

(line 2) forms an *update contribution* $\widetilde{C}_{i,j}^{(k)}$, which can be either full-rank or low-rank depending on whether $F_{i,k}$ and $F_{j,k}$ are both full-rank or not, respectively. Because the updated block $F_{i,j}$ must be kept full-rank, when $\widetilde{C}_{i,j}^{(k)}$ is low-rank, it is decompressed into a full-rank update contribution $C_{i,j}^{(k)}$ via an Outer Product step (line 3), before being summed in full-rank (line 4) with $F_{i,j}$.

Algorithm 2.2 Right-looking RL-Update step.

1: /* **Input:** a block $F_{i,j}$ to be updated by blocks $\widetilde{F}_{i,k}$ and $\widetilde{F}_{j,k}$. */
2: Inner Product: $\widetilde{C}_{i,j}^{(k)} \leftarrow X_{i,k}(Y_{i,k}^T D_{k,k} Y_{j,k}) X_{j,k}^T$
3: Outer Product: $C_{i,j}^{(k)} \leftarrow \widetilde{C}_{i,j}^{(k)}$
4: $F_{i,j} \leftarrow F_{i,j} - C_{i,j}^{(k)}$

Once the partial factorization of the front is finished, the L (and U in the unsymmetric case) factors are compressed, while the CB is still full-rank. In some contexts, we may want to also compress the CB (lines 13-19). The CB compression is discussed in Section 2.5.

2.2.2 The Left-looking UFSC variant

The FSCU algorithm is presented in its Right-looking form. Its Left-looking version, referred to as UFSC, is presented in Algorithm 2.3. First, the fully-summed part of the front is factorized (lines 2-10); then the contribution block is updated (lines 11-18).

Algorithm 2.3 Frontal BLR LDL^T (Left-looking) factorization: UFSC variant.

1: /* **Input:** a $p \times p$ block frontal matrix F; $F = [F_{i,j}]_{i=1:p,j=1:p}$; $p = p_{fs} + p_{nfs}$ */
2: **for** $k = 1$ **to** p_{fs} **do**
3: **for** $i = k$ **to** p **do**
4: Update: $F_{i,k} \leftarrow$ LL-Update($F_{i,k}$)
5: **end for**
6: Factor+Solve: $F_{k:p,k} \leftarrow L_{k:p,k} D_{k,k} L_{k,k}^T$
7: **for** $i = k+1$ **to** p **do**
8: Compress: $F_{i,k} \approx \widetilde{F}_{i,k} = X_{i,k} Y_{i,k}^T$
9: **end for**
10: **end for**
11: **for** $i = p_{fs}+1$ **to** p **do**
12: **for** $k = p_{fs}+1$ **to** i **do**
13: Update: $F_{i,k} \leftarrow$ LL-Update($F_{i,k}$)
14: **if** CB compression is activated **then**
15: Compress: $F_{i,k} \approx \widetilde{F}_{i,k} = X_{i,k} Y_{i,k}^T$
16: **end if**
17: **end for**
18: **end for**

The left-looking version of the Update step is provided in Algorithm 2.4. Note that the number of blocks used to update $F_{i,k}$ is either $k-1$ or p_{fs}, depending on whether $F_{i,k}$ belongs to the factors or the contribution block part of the front, respectively. Also note that at step $k = 1$, there is nothing to do; this leads to an empty loop (because $k - 1 = 0$ at line 2 of Algorithm 2.4), which is implicitly skipped.

Algorithm 2.4 Left-looking LL-Update step.

1: /* **Input:** a block $F_{i,k}$ to be updated. */
2: **for** $j = 1$ **to** $\min(k-1, p_{fs})$ **do**
3: Inner Product: $\widetilde{C}_{i,k}^{(j)} \leftarrow X_{i,j}(Y_{i,j}^T D_{j,j} Y_{k,j}) X_{k,j}^T$
4: Outer Product: $C_{i,k}^{(j)} \leftarrow \widetilde{C}_{i,k}^{(j)}$
5: $F_{i,k} \leftarrow F_{i,k} - C_{i,k}^{(j)}$
6: **end for**

While the Right- and Left-looking versions of each variant are numerically equivalent, we will show in Chapter 5 that the Left-looking version outperforms the Right-looking one for a number of reasons. Therefore, we present the rest of the BLR factorization variants using the Left-looking terminology. We refer the reader to Table 2.1 for the corresponding Right-looking terminology.

2.2.3 How to handle numerical pivoting in the BLR factorization

Algorithms 2.1 and 2.3 are fully compatible with threshold partial pivoting (cf. Section 1.2.4). The pivots are selected inside the BLR blocks; to assess their quality,

they are compared to the pivots of the entire column. Therefore, as explained before, to perform numerical pivoting, the Solve step is merged with the Factor step. The postponed pivots are merged with the next BLR block, whose size consequently increases, as illustrated in Figure 2.1. At the end of the last BLR panel, if uneliminated variables remain, they are then delayed to the parent front, just as in the standard algorithm presented in Section 1.2.4.2.

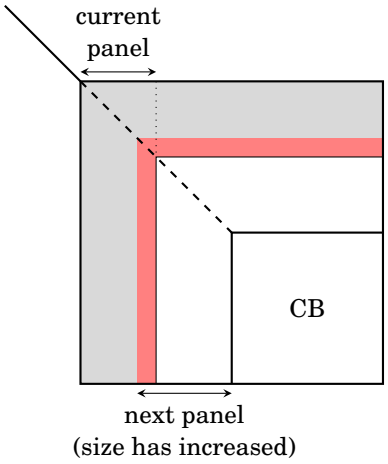
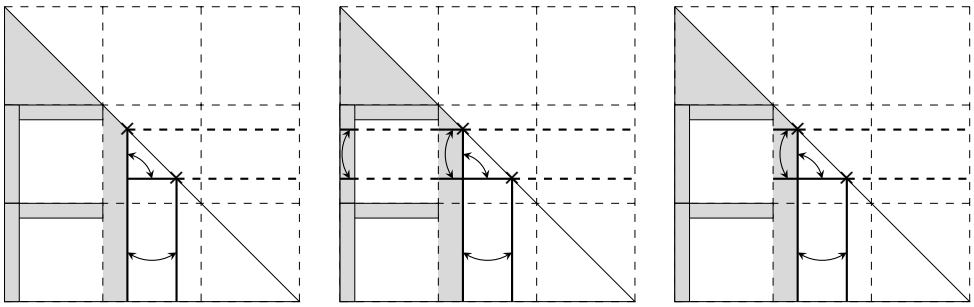


Figure 2.1 – BLR factorization with numerical pivoting scheme. Postponed pivots after the elimination of the current panel are in red; they are merged with the next panel, whose size increases.

Thus, in situations where many postponed pivots occur, the size of the BLR blocks may be considerably different from the size of the originally computed cluster. While this does not require any additional operations, one could think it may degrade the compression rate since the clusters after pivoting may not be admissible.

The row swaps involved in the BLR factorization are quite different depending on the choice of pivoting style (LINPACK or LAPACK, cf. Figure 1.5), as illustrated in Figure 2.2. With the LINPACK style (Figure 2.2a), the only part to swap is the trailing submatrix part, which is still represented in full-rank and can thus be done exactly as in the Full-Rank factorization (Figure 1.5). The same holds for the hybrid style where the LAPACK style is used inside the diagonal BLR blocks (Figure 2.2c). With the LAPACK style (Figure 2.2b), the entire row is swapped, which corresponds to parts of the frontal matrix already compressed. As long as there are no postponed pivots, this does not pose any particular problem: swapping the rows or columns of a low-rank block $\widetilde{B} = XY^T$ can be achieved easily by swapping the rows of X or Y, respectively.

However, in the presence of pivots postponed from one BLR block to the other, the LAPACK style becomes much more complex to use, as illustrated in Figure 2.3. Indeed, the low-rank blocks already compressed that lie on the left are not necessarily aligned with the current diagonal block. Therefore, two rows inside the current diagonal block may correspond to rows belonging to different low-rank or full-rank blocks in the previous panels, as illustrated in Figure 2.3. Of course, swapping rows from two full-rank blocks is straightforward. However, if one or both blocks



(a) LINPACK style (b) LAPACK style, without postponed pivots (c) Hybrid style

Figure 2.2 – LINPACK and LAPACK pivoting styles in BLR (symmetric case). The shaded area represents the part of the matrix that is already factored (and also indicates the BLR blocks).

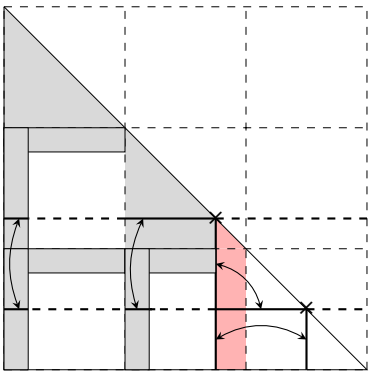


Figure 2.3 – LAPACK pivoting style in BLR, with postponed pivots (symmetric case). Eliminated pivots are in gray while postponed pivots are in red.

are low-rank, then there is no simple way to swap the rows. Two strategies can be distinguished:

- FR-swap: the low-rank blocks are decompressed back to full-rank; the swap can then be easily performed in full-rank. Then the blocks are compressed again; to avoid decompressing and compressing the same blocks at each swap, one should wait until enough pivots have been eliminated to guarantee no more decompressions will be necessary to compress a given block again.

- LR-swap: as shown in Figure 2.4, it is possible to swap rows from two different low-rank blocks by increasing their rank by 1. Indeed, let us consider two blocks $\widetilde{B}_1 = X_1 Y_1^T$ and $\widetilde{B}_2 = X_2 Y_2^T$; one can swap row i_1 from $\widetilde{B}_1$ with row i_2 from $\widetilde{B}_2$ as follows:

$$X_1^{new} \leftarrow \begin{pmatrix} \widehat{X}_1 & e_{i_2} \end{pmatrix} \text{ and } Y_1^{new} \leftarrow \begin{pmatrix} Y_1 & Y_2 (X_2)_{i_2,:}^T \end{pmatrix}, \tag{2.1}$$

$$X_2^{new} \leftarrow \begin{pmatrix} \widehat{X}_2 & e_{i_1} \end{pmatrix} \text{ and } Y_2^{new} \leftarrow \begin{pmatrix} Y_2 & Y_1 (X_1)_{i_1,:}^T \end{pmatrix}, \tag{2.2}$$

where e_i is the standard basis vector with a 1 at coordinate i and 0 everywhere else, and $\widehat{X}_1 = X_1 \dashv (-X_1)_{i_1,:}$ is equal to X_1 where row i_1 has been replaced by zeroes (and similarly for $\widehat{X}_2$). This strategy can also be applied to swap rows between a full-rank and a low-rank block. At the end of the factorization, one should recompress the blocks to obtain the most compact representation possible.

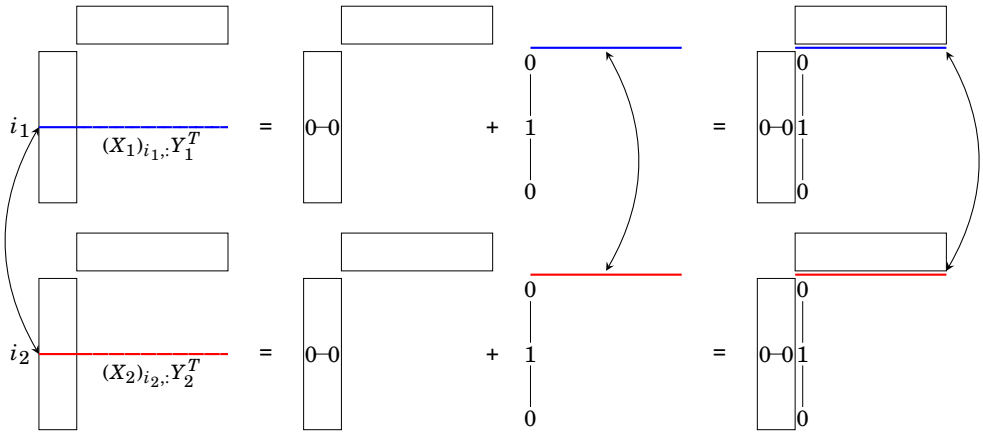


Figure 2.4 – LR-swap strategy.

In this thesis, we will not perform an experimental comparison of the different pivoting styles and swap strategies. We will use the LAPACK style with the FR-swap strategy. This should be the object of further research, as mentioned in Chapter 9 (Section 9.2).

2.2.4 Is the UFSC standard variant enough?

In Weisbecker (2013), the potential of the standard UFSC variant was demonstrated. This variant was furthermore presented as the best compromise between savings and robustness for the following three reasons:

- The number of operations needed for the Solve step is much lower than for the Update step and so we can focus on the latter without using a low-rank solve operation.
- The Solve step is done accurately which avoids a twofold approximation of the factors.
- As described in Section 2.2.3, standard pivoting strategies can be used, while it is not obvious how numerical pivoting can be performed with low-rank off-diagonal blocks.

We will show in this manuscript that:

- While the UFSC variant can already achieve significant gains with respect to the full-rank solver, compressing before the Solve is critical to capture the best complexity possible, as explained in Chapter 4, Section 4.5.

- The twofold approximation of the factors barely degrades the accuracy of the factorization and is therefore much more efficient in terms of flops/accuracy ratio.

- Standard pivoting strategies can be efficiently and effectively adapted to be performed on low-rank sub-blocks, as explained in Section 2.3.2 (so-called UCFS variant).

Therefore, the other BLR factorization variants can and must be considered.

2.3 Compress before Solve: the UFCS and UCFS variants

2.3.1 When pivoting can be relaxed: the UFCS variant

In the previous variants, as described in Section 2.2.3, threshold partial pivoting is performed by merging together the Factor and Solve steps. For many problems, numerical pivoting can be restricted to a smaller area of the panel (for example, the diagonal BLR blocks). In this case, the Solve step can be separated from the Factor step and applied directly on the entire BLR panel, just as in the FR factorization (Algorithm 1.5). As said before, this makes the Solve step more efficient because it solely relies on BLAS-3 operations.

Furthermore, in BLR, when numerical pivoting is restricted, it is natural and more efficient to perform the Compress before the Solve, thus leading to the so-called UFCS factorization, described in Algorithm 2.5. The UFCS variant makes further use of the low-rank property of the blocks since the Solve step can then be performed in low-rank as shown at line 11.

In Section 2.3.2.3, we compare the accuracy of the UFSC and UFCS variants on a set of problems where numerical pivoting can be restricted with no significant loss of accuracy. The goal of this comparison is to assess whether a two-fold approximation significantly degrades the residual, as feared in Weisbecker (2013). Indeed, performing the Compress step earlier can decrease the ranks of the blocks; however, we will show this does not have a significant impact on the scaled residual.

We will prove in Chapter 4 that the UFCS variant improves the asymptotic complexity of the factorization since it decreases the cost of the full-rank part of the computations. We will also show in Chapters 5 and 6 how this translates into a significant performance gain in parallel settings. Therefore, compressing before the Solve is crucial to make the BLR factorization efficient and scalable. However, it is also necessary to preserve the ability to perform pivoting, as numerous problems require it, and since it is one of the main advantages and originality of Block Low-Rank solvers.

This double requirement leads us to introduce the UCFS variant, that allows us to capture the improved compression gain without sacrificing the ability to pivot.

Algorithm 2.5 Frontal BLR LDL^T (Left-looking) factorization: UFCS variant.

/* **Input:** a $p \times p$ block frontal matrix F; $F = [F_{i,j}]_{i=1:p,j=1:p}$; $p = p_{fs} + p_{nfs}$ */
for $k = 1$ **to** p_{fs} **do**
  for $i = k$ **to** p **do**
    Update: $F_{i,k} \leftarrow$ LL-Update$(F_{i,k})$
  end for
  Factor: $F_{k,k} \leftarrow L_{k,k} D_{k,k} L_{k,k}^T$
  for $i = k+1$ **to** p **do**
    Compress: $F_{i,k} \approx \widetilde{F}_{i,k} = X_{i,k} Y_{i,k}^T$
  end for
  for $i = k+1$ **to** p **do**
    Solve: $\widetilde{F}_{i,k} \leftarrow \widetilde{F}_{i,k} L_{k,k}^{-T} D_{k,k}^{-1} = X_{i,k} (Y_{i,k}^T L_{k,k}^{-T} D_{k,k}^{-1})$
  end for
end for
for $i = p_{fs} + 1$ **to** p **do**
  for $k = p_{fs} + 1$ **to** i **do**
    Update: $F_{i,k} \leftarrow$ LL-Update$(F_{i,k})$
    if CB compression is activated **then**
      Compress: $F_{i,k} \approx \widetilde{F}_{i,k} = X_{i,k} Y_{i,k}^T$
    end if
  end for
end for

2.3.2 When pivoting is still required: the UCFS variant

We call UCFS the variant where the Compress is performed before the Factor+Solve step, which allows the latter to take into account the entries belonging to the low-rank blocks when assessing the quality of a pivot. Note that, technically, performing the Compress step before the Factor step does not necessarily imply that numerical pivoting is performed, as one could very well perform a "UCFS" factorization with restricted pivoting, which would then be equivalent to a UFCS factorization. However, because there is no particular reason to perform the Compress before the Factor other than using the low-rank information to pivot, we adopt the convention that numerical pivoting is always performed in the UCFS variant, while it is always restricted in the UFCS one.

2.3.2.1 Algorithm description

Algorithm 2.6 describes the UCFS factorization, where the Factor+Solve step takes as input a BLR panel that is already compressed. Thus, the Factor+Solve step must be modified to take into account the low-rank blocks of the panel. We have to guarantee that if there is no growth in Y_i, there will not be either in $\widetilde{F}_i = X_i Y_i^T$. This is of course not true in general, but holds in the case where X_i is an orthonormal matrix (which we assume because a truncated QR factorization with column pivoting is used to perform the compression). Indeed, let $\widetilde{B} = \widetilde{F}_i = XY^T$ be

Algorithm 2.6 Frontal BLR LDL^T (Left-looking) factorization: UCFS variant.

/* **Input:** a $p \times p$ block frontal matrix F; $F = [F_{i,j}]_{i=1:p,j=1:p}$; $p = p_{fs} + p_{nfs}$ */
for $k = 1$ **to** p_{fs} **do**
  for $i = k$ **to** p **do**
    Update: $F_{i,k} \leftarrow$ LL-Update($F_{i,k}$)
  end for
  for $i = k+1$ **to** p **do**
    Compress: $F_{i,k} \approx \widetilde{F}_{i,k} = X_{i,k} Y_{i,k}^T$
  end for
  Factor+Solve: $\widetilde{F}_{k:p,k} \leftarrow \widetilde{L}_{k:p,k} D_{k,k} L_{k,k}^T$
end for
for $i = p_{fs}+1$ **to** p **do**
  for $k = p_{fs}+1$ **to** i **do**
    Update: $F_{i,k} \leftarrow$ LL-Update($F_{i,k}$)
    if CB compression is activated **then**
      Compress: $F_{i,k} \approx \widetilde{F}_{i,k} = X_{i,k} Y_{i,k}^T$
    end if
  end for
end for

a low-rank block, then the maximum norm of $\widetilde{B}_{:,k}$ can be bounded by

$$\|\widetilde{B}_{:,k}\|_\infty \leq \|\widetilde{B}_{:,k}\|_2 = \|X Y_{k,:}^T\|_2 = \|X\|_2 \|Y_{k,:}^T\|_2 = \|Y_{k,:}^T\|_2, \tag{2.3}$$

where $\|\cdot\|_\infty$ and $\|\cdot\|_2$ denote the maximum norm and the 2-norm, respectively. Note that we cannot directly use the maximum norm to bound $\|\widetilde{B}_{:,k}\|_\infty$ because it is not a submultiplicative norm (i.e. $\|AB\|_\infty \leq \|A\|_\infty \|B\|_\infty$ does not hold). Furthermore, we cannot use other submultiplicative norms such as the 1-norm or the infinity norm, because they are not unitarily invariant (i.e. $\|X\| \neq 1$). Using the 2-norm has a double consequence:

- First, computing the 2-norm of $\|Y_{k,:}^T\|$ can be expensive since it requires $2r$ operations (where r is the rank of $\widetilde{B}$), compared to r operations for the 1-norm and only comparisons (no operations) for the maximum norm. This is however acceptable since we expect to reduce the overall number of operations by performing the Solve in low-rank, at a cost of br operations instead of b^2, where b denotes the number of rows of $\widetilde{B}$.

- Second, equation (2.3) may overestimate the maximum norm by a factor as large as $\sqrt{b}$:

  $$\|\widetilde{B}_{:,k}\|_\infty \leq \|\widetilde{B}_{:,k}\|_2 \leq \sqrt{b} \|\widetilde{B}_{:,k}\|_\infty. \tag{2.4}$$

  This can therefore potentially lead to an increased amount of (unnecessary) postponed and delayed pivots. We will quantify this effect in Section 2.3.2.3 and show in most practical cases it is not problematic. In some cases where it might, this could be avoided by choosing a lower threshold τ for partial pivoting (to account for the overestimation). Alternatively, we could decom-

press the column $\widetilde{B}_{:,k}$ to compute its exact maximum norm, but only when the inequality above cannot guarantee the pivot to be safe.

This results in a new Factor+Solve algorithm, described in Algorithm 2.7. We denote the (j,j)-th entry of the i-th block by $(F_i)_{j,j}$. Since Algorithm 2.7 is written in its symmetric form, the pivot search is restricted to the diagonal block F_1. For the sake of simplicity, we only consider 1×1 pivots, and thus only diagonal entries $(F_1)_{j,j}$ are assessed as pivot candidates. The algorithm mainly differs from the standard Factor+Solve algorithm (described in Algorithm 1.9) on how the maximal element m on column j is computed. For each block F_i, we compute a bound m_i on the maximum or 2-norm of column j, depending on whether F_i is full-rank or low-rank, respectively (lines 13-17). Note that, in the case of the diagonal block F_1, the block is symmetric, and therefore, at line 11, the maximum norm of column j is computed as

$$\max\left(\max_{i<j}(F_1)_{j,i}\ ,\ \max_{i>j}(F_1)_{i,j}\right).$$

The algorithm loops until a pivot candidate $(F_1)_{j,j}$ belonging to the diagonal block F_1 is found. Then, columns and rows k and j are swapped (line 22). If $\widetilde{F}_i = X_i Y_i^T$ is low-rank, this corresponds to swapping the rows of Y_i, as illustrated in Figure 2.5. Finally, pivot k is eliminated (lines 23-31). For the low-rank off-diagonal blocks $\widetilde{F}_i = X_i Y_i^T$, the update resulting from this elimination is applied on the rows of Y_i.

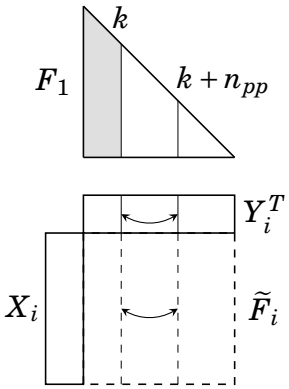


Figure 2.5 – Column swap in the UCFS factorization. Swapping the columns of $\widetilde{F}_i$ is equivalent to swapping the rows Y_i.

2.3.2.2 Dealing with postponed and delayed pivots

At the termination of Algorithm 2.7, a panel $\widetilde{F}_{k:p,k}$ has been factorized and n_{pp} pivots have been postponed, as illustrated in Figure 2.6a. Let us note J_{pp} the set of postponed columns. For each off-diagonal low-rank block $\widetilde{F}_{i,k} = X_{i,k} Y_{i,k}$, these postponed columns must be merged with the adjacent block $F_{i,k+1}$ (which is still represented in full-rank) from the next panel, just as for the previous BLR vari-

Algorithm 2.7 Factor+Solve step adapted for the UCFS factorization (symmetric case)

/* **Input:** a panel $\widetilde{F}$ with n_r block-rows and n_c columns $\widetilde{F} = [\widetilde{F}_i]_{i=1:n_r}$. */
for $k = 1$ **to** n_c **do**
  $j \leftarrow k-1$
  /* Loop until an acceptable pivot is found, or until no pivot candidates are left. */
  repeat
    $j \leftarrow j+1$
    if $j > n_c$ **then**
      /* No pivot candidates left, exit. */
      go to 33
    end if
    $m_1 \leftarrow \max\left(\max_{i<j}(F_1)_{j,i}\ ,\ \max_{i>j}(F_1)_{i,j}\right)$
    for $i = 2$ **to** n_r **do**
      if $\widetilde{F}_i$ is low-rank **then**
        $m_i \leftarrow \|(Y_i)_{j,:}\|_2$
      else
        $m_i \leftarrow \|(F_i)_{:,j}\|_\infty = \max_{i'} |(F_i)_{i',j}|$
      end if
    end for
    $m \leftarrow \max_{i\in[1;n_r]} m_i$
  until $|(F_1)_{j,j}| \geq \tau m$
  /* Pivot candidate has been chosen in column j: swap it with current column k. */
  Swap columns k and j and rows k and j
  for $i = 1$ **to** n_r **do**
    if $\widetilde{F}_i$ is low-rank **then**
      $(Y_i)_{k,:} \leftarrow (Y_i)_{k,:}/(F_1)_{k,k}$
      $(Y_i)_{k+1:n_c,:} \leftarrow (Y_i)_{k+1:n_c,:} - (Y_i)_{k,:}(Y_i)^T_{k,k+1:n_c}$
    else
      $(F_i)_{:,k} \leftarrow (F_i)_{:,k}/(F_1)_{k,k}$
      $(F_i)_{:,k+1:n_c} \leftarrow (F_i)_{:,k+1:n_c} - (F_i)_{:,k}(F_i)^T_{k+1:n_c,k}$
    end if
  end for
end for
/* $n_{pp} = n_c - k + 1$ is the number of postponed pivots. */

ants[1]. However, in the UCFS factorization, the postponed columns $(\widetilde{F}_{i,k})_{:,J_{pp}}$ have already been compressed and are thus represented in low-rank. In this case, the merge requires extra computations. As illustrated in Figure 2.6, three strategies can be considered:

- FR-merge: first, the postponed columns are decompressed via the matrix-matrix product $(\widetilde{F}_{i,k})_{:,J_{pp}} = X_{i,k}(Y_{i,k})^T_{J_{pp},:}$. Then, the two full-rank blocks $(F_{i,k})_{:,J_{pp}}$

[1]Note that, in the left-looking factorization, the next panel $F_{:,k+1}$ must first be updated before being merged with the postponed pivots.

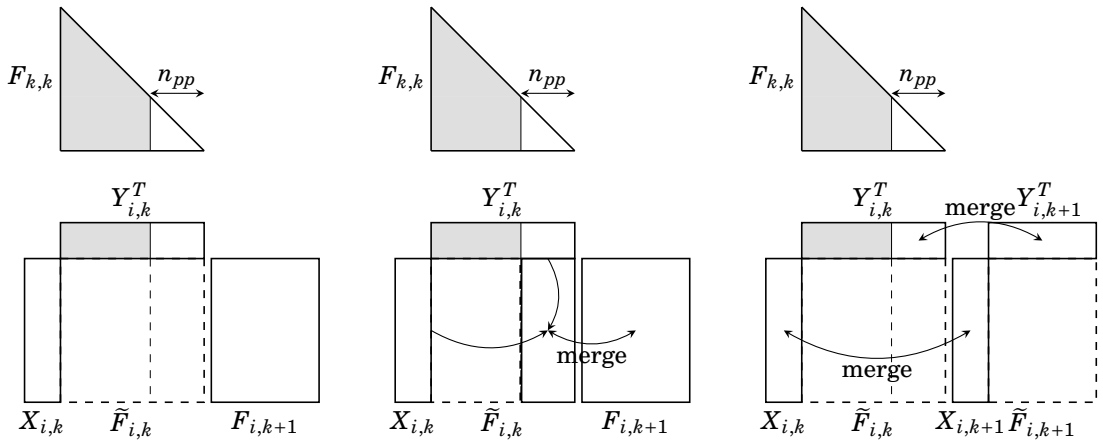
and $F_{i,k+1}$ are adjacent and can be compressed as a single block $F_{i,k+1} \leftarrow \left((F_{i,k})_{:,J_{pp}} \;\; F_{i,k+1}\right)$. This is illustrated in Figure 2.6b.

- LR-merge: first, the next panel block $F_{i,k+1}$ is compressed: $F_{i,k+1} \approx \widetilde{F}_{i,k+1} = X_{i,k+1}Y_{i,k+1}^T$. Then, the two low-rank blocks $(\widetilde{F}_{i,k})_{:,J_{pp}}$ and $\widetilde{F}_{i,k+1}$ can be merged by means of a low-rank block agglomeration operation (cf. Bebendorf (2008), Section 1.1.6):

$$\begin{aligned}\left((\widetilde{F}_{i,k})_{:,J_{pp}} \;\; \widetilde{F}_{i,k+1}\right) &= \left(X_{i,k}(Y_{i,k})_{J_{pp},:}^T \;\; X_{i,k+1}Y_{i,k+1}^T\right) \\ &= \left(X_{i,k} \;\; X_{i,k+1}\right)\begin{pmatrix} (Y_{i,k})_{J_{pp},:}^T & 0 \\ 0 & Y_{i,k+1}^T \end{pmatrix} = X_{agg}Y_{agg}^T. \end{aligned} \tag{2.5}$$

  The agglomerated low-rank block $X_{agg}Y_{agg}^T$ is then recompressed to obtain the final low-rank representation of $\widetilde{F}_{i,k+1}$. The merge step is illustrated in Figure 2.6c.

- No-merge: the postponed columns are not merged but rather permuted to the end of the frontal matrix. We will try to eliminate them as independent panels, hoping that their elimination will become possible after the rest of the panels have been treated.



(a) Before merge (b) FR-merge (c) LR-merge

Figure 2.6 – Strategies to merge postponed pivots with the next panel in the UCFS factorization. $\widetilde{F}_{i,k}$ has been factored and n_{pp} pivots have been postponed; they must be merged with the next panel $F_{i,k+1}$.

In the case where the panel $\widetilde{F}_{k:p,k}$ is the last panel and there are still some postponed pivots, they become delayed pivots and must be passed to the parent front.

2.3.2.3 Experimental study on matrices that require numerical pivoting

In Table 2.2, we list some matrices from the UFSMC that require numerical pivoting during the factorization. We measure the scaled residual (taking as right-

hand side the vector such that the solution is the vector containing only ones) resulting from the FR factorization with standard threshold partial pivoting (with a threshold $\tau = 0.01$) and with restricted threshold partial pivoting (restricted to diagonal blocks).

Two cases can be distinguished. The first category of problems (delimited by the dashed lines in Table 2.2) achieves a scaled residual with restricted pivoting that is as good as that using standard pivoting. This is often related to the fact that they require only a moderate amount of numerical pivoting (the number of delayed pivots is under 10% of the order of the matrix). Conversely, for the second category of problems, much more pivoting is needed (delayed pivots up to 80% of the order of the matrix); in this case, restricted pivoting either achieves a much higher scaled residual, or simply breaks down due to numerical issues. Note that the number of delayed pivots can be much higher than the order of the matrix because a given variable can be delayed on multiple fronts before being eliminated.

In Table 2.3, we compare the UFSC, UFCS (restricting pivoting to the diagonal BLR blocks), and UCFS factorizations on the matrices listed in Table 2.2. We use a low-rank threshold arbitrarily set to $\varepsilon = 10^{-6}$ for all matrices, with the exception of those coming from our EDF application (matrices perf009{d,ar}, cf. Section 1.5.2.2), for which we know the required threshold is $\varepsilon = 10^{-9}$, as well as matrix c-big, for which $\varepsilon = 10^{-4}$ is necessary to achieve some compression. In the case of the UCFS factorization, postponed pivots are merged using the FR-merge strategy (cf. Section 2.3.2.2). The threshold for partial pivoting is still set to $\tau = 0.01$.

For the first category of matrices (those for which restricted pivoting is acceptable in FR), the UFCS factorization with restricted pivoting also results in most cases in an acceptable residual. Note that the residual is often slightly larger than that of the standard UFSC factorization. The fact that the UCFS factorization (with non-restricted pivoting) does not improve the residual with respect to UFCS shows that this is not due to the restricted pivoting, but rather to the higher compression rate (not provided in the table). There are two exceptions to this fact: matrix_9 and matrix-new_3. For these two matrices, even though restricted pivoting is acceptable in the full-rank case, the UFCS factorization results in a poor residual with respect to UFSC; with UCFS, using non-restricted pivoting retrieves the same quality as UFSC. It would therefore seem there is some amplification of the pivoting errors due to the BLR approximations.

For the second category of matrices (those for which restricted pivoting is not acceptable in FR), the UFCS factorization also fails to achieve a satisfying residual. However, with the non-restricted pivoting algorithm designed in the previous sections, the UCFS variant manages to retrieve a good residual for all matrices. Furthermore, the good news is it does so without increasing significantly the number of delayed pivots with respect to the UFSC variant. This may illustrate the fact that numerical pivoting is needed to reject pivot candidates which are several orders of magnitude smaller than the off-diagonal entries; thus, even if the norm of the latter is overestimated by a factor $\sqrt{b}$ (where b denotes the block size, cf. equation (2.4)), this does not create additional, unnecessary delayed pivots. Note that this may not remain true for much larger block sizes, such as those which arise in the hierarchical framework; this makes the BLR format particularly suitable for handling numerical pivoting based on block low-rank information. There are

matrix	n	nnz	flops	scaled residual		delayed pivots	
				TPP	RP	TPP	RP
3Dspectralwave	681k	33.7M	18.4 TF	2e-11	6e-11	13	0
af_shell10	1508k	52.7M	755.8 GF	5e-16	5e-16	0	0
barrier2-10	116k	3.9M	221.6 GF	2e-14	8e-16	1,456	0
bmw3_2	227k	11.3M	49.3 GF	9e-16	1e-15	774	0
dielFilterV2real	1547k	48.5M	2.0 TF	2e-16	2e-16	0	0
ecology1	1000k	5.0M	26.2 GF	5e-16	7e-16	0	0
Ga41As41H72	268k	18.5M	84.4 TF	2e-12	2e-12	0	0
gas_sensor	67k	1.7M	35.9 GF	1e-16	1e-16	0	0
helm2d03	392k	2.7M	8.8 GF	1e-12	2e-12	0	0
Lin	256k	1.8M	520.6 GF	4e-12	4e-12	0	0
matrix_9	103k	2.1M	271.4 GF	2e-16	2e-16	1,812	0
matrix-new_3	125k	2.7M	429.7 GF	2e-16	2e-16	8,415	0
nlpkkt80	1062k	28.7M	30.0 TF	5e-12	1e-11	0	0
para-10	156k	5.4M	362.2 GF	1e-14	8e-16	2,134	0
PR02R	161k	8.2M	105.6 GF	1e-14	3e-15	28	0
c-73	169k	1.3M	1.1 GF	2.1e-14	1.3e-11	22,122	0
c-73b	169k	1.3M	0.8 GF	3.7e-15	1.1e-09	766	0
c-big	345k	2.3M	264.9 GF	2.5e-16	1.1e-09	6,162	0
cont-300	181k	1.0M	32.4 GF	1e-07	fail	143,438	—
darcy003	390k	2.1M	1.0 GF	5e-14	fail	293,620	—
d_pretok	183k	1.6M	9.6 GF	3e-15	fail	90,204	—
kkt_power	2063k	14.6M	2.0 TF	4e-13	fail	234,927	—
mario002	390k	2.1M	1.0 GF	5e-14	fail	293,620	—
perf009d	803k	55.6M	870.5 GF	2e-16	5e-08	20,069	20,332
perf009ar	5413k	412.3M	36.6 TF	3e-16	9e-02	82,332	82,069
TSOPF_FS_b39_c30	120k	3.1M	4.0 GF	6e-14	4e-05	61,433	33,399
turon_m	190k	1.7M	8.4 GF	6e-15	fail	91,797	—

Table 2.2 – Set of matrices that require pivoting. All matrices are factorized by means of a LU factorization in double real (d) arithmetic. Scaled residual and number of delayed pivots are provided for the factorization with threshold partial pivoting with threshold $\tau = 0.01$ (TPP) and with restricted pivoting (RP) to the diagonal blocks. The term "fail" indicates the factorization broke down due to numerical issues. The problems can be classified into two categories (separated by the dashed lines), depending on whether restricted pivoting is acceptable (top) or not (bottom).

also some exceptions in the second category: for matrices c-73, c-73b, and c-big, the residual remains of the same order with restricted pivoting (UFCS factorization), whereas in the FR case, the residual grows by several orders of magnitude when using restricted pivoting. A possible explanation for this behavior is that the initial ordering is different in BLR, due to the clustering.

Thus, five matrices (in bold in Table 2.3) have switched category in BLR with respect to FR. It therefore seems that the errors associated with BLR approximations and off-diagonal entry growth have an effect on each other (amplification or the inverse). This effect would deserve to be further studied and formalized by means of an error analysis of the BLR factorization. Note that for some matrices (cont-300, d_pretok, turon_m), the residual with UCFS is significantly better than with UFSC; this is currently unexplained and should be investigated.

We now assess the gains in flops due to compressing before the Solve step, and

matrix	flops (% of FR)			scaled residual			delayed pivots		
	UFSC	UFCS	UCFS	UFSC	UFCS	UCFS	UFSC	UFCS	UCFS
3Dspectralwave	46.6	39.9	39.9	9e-05	2e-05	2e-05	13	0	13
af_shell10	29.9	22.7	22.7	2e-06	5e-06	4e-06	0	0	0
barrier2-10	24.8	16.5	16.1	1e-10	2e-09	2e-09	3,248	0	3,248
bmw3_2	70.3	60.5	60.7	3e-09	5e-10	8e-11	855	0	855
dielFilterV2real	44.3	35.8	35.8	7e-07	1e-06	1e-06	0	0	0
ecology1	33.5	21.2	21.2	5e-06	2e-05	2e-05	0	0	0
Ga41As41H72	6.7	4.6	4.6	6e-06	2e-06	2e-06	0	0	0
gas_sensor	55.3	47.1	47.1	2e-08	2e-07	2e-07	0	0	0
helm2d03	56.2	43.5	43.5	6e-06	7e-06	7e-06	0	0	1
Lin	24.0	18.5	18.5	4e-05	4e-05	4e-05	0	0	0
matrix_9	42.2	34.2	38.7	7e-11	1e-07	9e-11	5,737	0	5,818
matrix-new_3	27.8	22.3	29.7	5e-12	1e-10	5e-11	18,374	0	18,546
nlpkkt80	22.4	23.3	23.3	2e-07	1e-07	1e-07	0	0	0
para-10	20.5	13.4	13.0	5e-10	1e-09	5e-10	4,099	0	4,099
PR02R	50.8	43.3	45.4	2e-09	2e-11	2e-11	1,159	0	1,189
c-73	84.2	65.0	73.0	2e-05	4e-08	1e-07	22,122	0	22,122
c-73b	100.8	97.0	97.1	2e-07	3e-06	2e-06	767	0	767
c-big	80.1	67.6	68.0	5e-04	1e-03	1e-03	6,167	0	6,167
cont-300	62.8	—	84.7	4e-01	fail	6e-06	148,315	—	148,131
darcy003	82.8	—	72.2	2e-06	fail	1e-06	292,620	—	292,620
d_pretok	67.1	—	57.8	1e-07	fail	4e-12	90,204	—	90,204
kkt_power	65.7	64.9	64.6	4e-12	9e-02	4e-14	242,732	42,558	242,708
mario002	82.8	—	72.2	2e-06	fail	1e-06	292,620	—	292,620
perf009d	48.4	43.3	43.0	2e-13	4e-05	1e-10	20,368	20,272	20,368
perf009ar	26.0	22.7	22.1	3e-13	1e-01	9e-11	82,123	80,659	82,123
TSOPF_FS_b39_c30	78.6	—	70.6	4e-10	fail	8e-12	60,797	—	60,797
turon_m	60.4	—	49.3	1e-06	fail	7e-13	91,433	—	91,433

Table 2.3 – Comparison of the UFSC, UFCS, and UCFS factorization variants on the set of matrices of Table 2.2. All matrices are factorized by means of a LU factorization in double real (d) arithmetic. UFSC and UCFS use threshold partial pivoting with threshold $\tau = 0.01$; UFCS uses restricted pivoting to the diagonal blocks. Flops are given as a percentage of the full-rank ones. The term "fail" indicates the factorization broke down due to numerical issues. The problems can be classified into two categories (separated by the dashed lines), depending on whether restricted pivoting is acceptable in the FR case (top) or not (bottom); in BLR, some problems (indicated in bold) switch categories.

also measure the overhead cost of UCFS with respect to UFCS. For the matrices for which UFCS is acceptable, the gains in flops with respect to standard UFSC are considerable, higher than 30% for several matrices. Moreover, the UCFS variant only leads a small increase in flops with respect to UFCS. This overhead cost is associated with the FR-merge operations, as well as the possibly lower compression rate (due to the higher amount of pivoting, which can perturb the original clustering). For most matrices, the overhead is negligible (compare column "flops" for UFCS and UCFS). For those for which it is noticeable (PR02R, c-73), we have observed (results not shown here) that it is due to a lower compression rate rather than the FR-merge operations. Therefore, on this set of problems, it is probably not worth designing more sophisticated merge strategies (such as LR-merge).

From this study, it results that the UCFS factorization can achieve comparable gains with respect to UFCS, while maintaining the stability of UFSC. This is confirmed by the analysis on the second category of matrices, for which UFCS fails, whereas UCFS is able to achieve a satisfying residual while considerably reducing the number of operations for the factorization.

2.4 Compress as soon as possible: the CUFS variant

The CUFS algorithm, described in Algorithm 2.8, is the final variant of the BLR factorization. It performs the compression as early as possible. In Right-looking, the entire matrix needs to be accessed at step 1, and therefore the entire matrix must be compressed before starting the factorization: this is the meaning of the "C;FSU" notation (cf. Table 2.1). On the contrary, in Left-looking, at step k, the panels $k+1$ to p have not been accessed yet and therefore have not been compressed yet either. Therefore, the Compress is the first step performed on each panel, but is still interlaced with the other steps panel after panel. Note that a similar Left-looking "C;UFS" variant (where the matrix would be first compressed entirely) is possible but is unnecessary and is likely to have worse locality than the CUFS variant.

Algorithm 2.8 Frontal BLR LDL^T (left-looking) factorization: CUFS variant.

1: /* **Input:** a $p \times p$ block frontal matrix F; $F = [F_{i,j}]_{i=1:p,j=1:p}$; $p = p_{fs} + p_{nfs}$ */
2: **for** $k = 1$ **to** p_{fs} **do**
3: **for** $i = k+1$ **to** p **do**
4: Compress: $F_{i,k} \approx \widetilde{F}_{i,k} = X_{i,k} Y_{i,k}^T$
5: **end for**
6: **for** $i = k$ **to** p **do**
7: Update: $\widetilde{F}_{i,k} \leftarrow$ CUFS-Update($\widetilde{F}_{i,k}$)
8: **end for**
9: Factor+Solve: $\widetilde{F}_{k:p,k} \leftarrow L_{k:p,k} D_{k,k} L_{k,k}^T$
10: **end for**
11: **for** $i = p_{fs}+1$ **to** p **do**
12: **for** $k = p_{fs}+1$ **to** i **do**
13: **if** CB compression is activated **then**
14: Compress: $F_{i,k} \approx \widetilde{F}_{i,k} = X_{i,k} Y_{i,k}^T$
15: Update: $\widetilde{F}_{i,k} \leftarrow$ CUFS-Update($\widetilde{F}_{i,k}$)
16: **else**
17: Update: $F_{i,k} \leftarrow$ LL-Update($F_{i,k}$)
18: **end if**
19: **end for**
20: **end for**

Then, the blocks belonging to the L (and U in the unsymmetric case) factors part of the front and those belonging to the CB part must be considered separately.

- For the L factors blocks, the original entries of the matrix F_{ik} are compressed before the Update step, and thus this step must be modified as shown in Algorithm 2.9. At line 4, the update contribution $\widetilde{C}_{i,k}^{(j)}$ is summed with the compressed original entries $\widetilde{F}_{i,k}$; this operation is therefore a low-rank matrix sum, described in Section 1.4.1.3. However, the sum of all the contributions might not be low-rank anymore since its rank is the sum of the ranks of each summed matrix. This is why the result is then recompressed (as shown at line 6) to obtain the final low-rank approximation of block $F_{i,k}$. The Outer Product step can thus be avoided. The recompression is an essential part of the CUFS factorization, as the recompression is used to compute the low-rank blocks of the L factors. This is further discussed in Section 3.4.

- For the CB blocks, the situation is different since after being updated, the blocks must be assembled into the parent front. Therefore, the Update step depends on whether the CB is compressed. If, just as for the previous variants, the CB is not compressed, then its Update is still applied on full-rank blocks and is therefore still based on the LL-Update kernel (Algorithm 2.4), as shown at line 17 of Algorithm 2.8. In particular the update contributions are still decompressed by means of an Outer Product. On the contrary, if the CB is compressed, as shown at line 14, then its blocks can be treated just as the blocks in the L factors; we discuss the benefits and drawbacks of compressing the CB in Section 2.5.

Algorithm 2.9 CUFS-Update step.

1: /* **Input:** a block $\widetilde{F}_{i,k}$ to be updated. */
2: **for** $j = 1$ **to** $\min(k-1, p_{fs})$ **do**
3: Inner Product: $\widetilde{C}_{i,k}^{(j)} \leftarrow X_{i,j}(Y_{i,j}^T D_{j,j} Y_{k,j}) X_{k,j}^T$
4: $\widetilde{F}_{i,k} \leftarrow \widetilde{F}_{i,k} - \widetilde{C}_{i,k}^{(j)}$
5: **end for**
6: $\widetilde{F}_{i,k} \leftarrow$ Recompress($\widetilde{F}_{i,k}$)

Note that, technically, the Inner Product performed to compute the update contributions $\widetilde{C}_{i,k}^{(j)}$ (line 3 of Algorithm 2.9) is independent from the Compress step performed to compute $\widetilde{F}_{i,k}$ and could therefore be performed before it; however, this would significantly increase the memory consumption since both the update contributions and the full-rank panel containing the original entries would need to be allocated and stored at the same time.

In this thesis, we will consider the CUFS variant from a theoretical point of view only. We will show in Chapter 4 that it does not improve the asymptotic complexity of the BLR factorization with respect to the previously presented UFCS/UCFS variants. Therefore, we leave the implementation of the CUFS variant and its experimental analysis for future work, as mentioned in the conclusion chapter.

2.5 Compressing the contribution block

In all BLR variants, we have the flexibility to choose whether the CB is compressed or not. Here, we discuss the benefits and drawbacks depending on which metric is considered.

- Flops: in general, compressing the CB represents an overhead cost which does not contribute to reducing the global number of operations. With the CUFS variant, it might become beneficial, because the Outer Product can be skipped; however, this is only true if the assembly operations are also performed in low-rank (fully-structured case, discussed below).

- Time: in terms of time, the picture looks even worse, since we are exchanging an efficient kernel (the Outer Product, a matrix-matrix multiplication) for much slower operations (the CB Compress, based on some compression kernel). Therefore, the time overhead is expected to be worse than the flop overhead, due to a lower GF/s rate.

- Memory: as explained in Section 1.3.2.5, the memory consumption in the multifrontal method is the sum of two types of memory, the factors and active memories. The factors memory can be reduced by compressing the L factors. However, the active memory mainly consists of the CB memory, and is thus not reduced unless the CB is compressed. In a parallel context, the active memory scales much worse than the factors memory, and it might thus become critical to compress the CB. This is further discussed in Section 9.3.

- Communications: in a parallel context (Chapter 6), we will also show that the volume of communications can be reduced both by compressing the L and CB blocks.

- Assembly type: if the CB is compressed, then the assembly must be modified to be performed on low-rank blocks. This involves quite complex low-rank extend-add operations, described in Section 1.4.3.2, which we may prefer to avoid.

For these reasons, the following three strategies are worth considering:

- Do not compress CB, full-rank assembly (CB_{FR}+Asm_{FR}): this should be the strategy of choice in a sequential context where there is enough memory to perform the factorization while storing the CB in uncompressed form. Indeed, in such a context, we aim to minimize the time for factorization and have no reason to compress the CB.

- Compress CB, full-rank assembly (CB_{LR}+Asm_{FR}): this strategy is of interest in a parallel context, or in a sequential context where memory is the limiting factor. The CB is first compressed and sent to the parent front, which allows for reducing both memory consumption and the volume of communications. Then, it is decompressed just before the assembly, which is thus efficiently performed in full-rank.

- Compress CB, low-rank assembly (CB_{LR}+Asm_{LR}): to avoid decompressing the CB blocks as in the previous strategy, the assembly can be performed in low-rank, which may or may not result in a gain in flops and/or time.

The CB_{LR}+Asm_{LR} strategy is usually referred to as *fully-structured* (or sometimes matrix-free), which means that the factorization can be performed without needing to store the matrix in full-rank at any point of the factorization. The fully-structured factorization is often considered when the input matrix is already under compressed form, or when its compressed form can be computed at a negligible cost. This is for example the case if A is sparse, in which case its original entries can be compressed via some sparse compression kernel (e.g. sparse SVD or randomized sampling (Halko et al., 2011) via a sparse matrix-vector product).

In Chapter 4, we show that skipping the Outer Product (performed to decompress the blocks) brings no asymptotical gain. Therefore, in this work, we will not consider the fully-structured CB_{LR}+Asm_{LR} strategy. We will use the CB_{FR}+Asm_{FR} strategy in the sequential case (Chapters 4 and 5) and compare it to the CB_{LR}+Asm_{FR} strategy in our parallel experiments (Chapter 6). Moreover, note that using a full-rank assembly does not necessarily imply an asymptotic increase of the memory consumption, because the assembly can be performed panel by panel (or even block by block) and interlaced with the compression.

2.6 Low-rank Updates Accumulation and Recompression (LUAR)

In this section, we introduce a modification of the Update step in order to reduce the cost of the Outer Product operation. It is applicable to all variants that use the LL-Update kernel (Algorithm 2.4), by replacing it by the LUAR-Update kernel, presented in Algorithm 2.10.

Algorithm 2.10 LUAR-Update step.

1: /* **Input:** a block $F_{i,k}$ to be updated. */
2: Initialize $\widetilde{C}_{i,k}^{(acc)}$ to zero
3: **for** $j = 1$ **to** $\min(k-1, p_{fs})$ **do**
4: Inner Product: $\widetilde{C}_{i,k}^{(j)} \leftarrow X_{i,j}(Y_{i,j}^T D_{j,j} Y_{k,j}) X_{k,j}^T$
5: Accumulate update: $\widetilde{C}_{i,k}^{(acc)} \leftarrow \widetilde{C}_{i,k}^{(acc)} + \widetilde{C}_{i,k}^{(j)}$
6: $\widetilde{C}_{i,k}^{(acc)} \leftarrow \text{Recompress}(\widetilde{C}_{i,k}^{(acc)})$
7: **end for**
8: Outer Product: $C_{i,k}^{(acc)} \leftarrow \widetilde{C}_{i,k}^{(acc)}$
9: $F_{i,k} \leftarrow F_{i,k} - C_{i,k}^{(acc)}$

LUAR stands for *Low-rank Updates Accumulation and Recompression*. It consists in accumulating the matrices $\widetilde{C}_{i,k}^{(j)}$ together, as shown at line 5 of Algorithm 2.10:

$$\widetilde{C}_{i,k}^{(acc)} \leftarrow \widetilde{C}_{i,k}^{(acc)} + \widetilde{C}_{i,k}^{(j)}.$$

We refer to $\widetilde{C}^{(acc)}_{i,k}$ as *accumulators*. The + sign in the previous equation denotes a low-rank sum operation. Specifically, noting $A = C^{(acc)}_{i,k}$ and $B = C^{(j)}_{i,k}$, we have

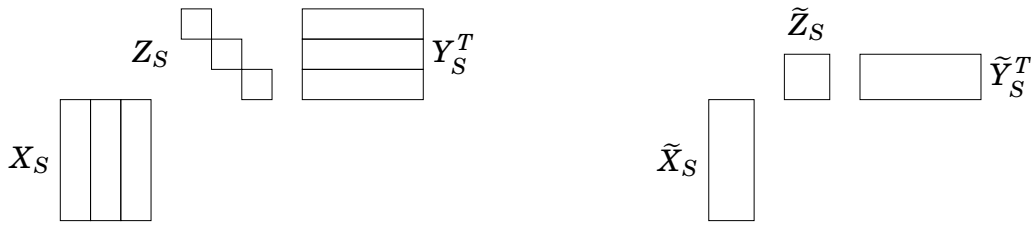
$$\widetilde{B} = \widetilde{C}^{(j)}_{i,k} = X_{i,j}(Y^T_{i,j} D_{j,j} Y_{j,k}) X^T_{j,k} = X_B Z_B Y^T_B,$$

with $X_B = X_{i,j}$, $Z_B = Y^T_{i,j} D_{j,j} Y_{k,j}$, and $Y_B = X_{k,j}$. Similarly, $\widetilde{A} = \widetilde{C}^{(acc)}_{i,k} = X_A Z_A Y^T_A$. Then, as explained in Section 1.4.1.3, the low-rank sum of A and B can be performed as follows:

$$\widetilde{A} + \widetilde{B} = X_A Z_A Y^T_A + X_B Z_B Y^T_B = (X_A \quad X_B) \begin{pmatrix} Z_A & \\ & Z_B \end{pmatrix} (Y_A \quad Y_B)^T = X_S Z_S Y^T_S = \widetilde{S},$$

where $\widetilde{S}$ is a low-rank approximant of $S = A + B$. A visual representation is given in Figure 2.7.

This algorithm has two advantages: first, accumulating the update contributions together leads to higher granularities in the Outer Product step, which is thus performed more efficiently. This will be analyzed in the context of the multithreaded factorization in Chapter 5. Second, it allows for additional compression, as the accumulated updates $\widetilde{C}^{(acc)}_{i,k}$ can be recompressed (as shown at line 6) before the Outer Product. This second aspect is more complex as there are many strategies to perform the recompression. The analysis and comparison of these strategies is the object of Chapter 3.



(a) Accumulated updates before recompression

(b) Accumulated updates after recompression

Figure 2.7 – Low-rank Updates Accumulation and Recompression.

2.7 Chapter conclusion

In this chapter, we have presented the Block Low-Rank factorization. We have the freedom to choose when the compression is performed, and this flexibility leads to a number of variants which we have described.

We have briefly mentioned the FSUC variant, which performs an offline compression once the factorization of each front is finished, to reduce the memory required to store the factors. We will not consider this variant further.

We have then described the standard UFSC (FSCU in right-looking) variant, introduced by Amestoy et al. (2015a). This variant reduces the number of operations

performed in the Update step, and can easily handle numerical pivoting. However, it performs the Solve step of the factorization phase in full-rank, whose cost is thus not reduced.

For this reason, we have considered two variants which perform the Compress before the Solve step, and can thus accelerate the latter. When numerical pivoting can be restricted, the UFCS variant can compress before the Solve the blocks which are not used for pivoting. To preserve the ability to perform numerical pivoting, critical in many applications, we have designed the UCFS variant which first compresses the blocks and then uses their low-rank representation to perform pivoting.

We have also described the CUFS variant, which performs the compression as soon as possible. In particular, this variant allows for a fully-structured factorization, where the matrix is not stored in full-rank at any point of the factorization.

Regardless of the variant considered, we may choose to compress the LU factors only, or to also compress the contribution block (CB) of the frontal matrices. Compressing the CB does not contribute in general to reducing the global number of operations, but can be useful to reduce the volume of communications or the memory consumption of the solver. These aspects will be studied in Sections 6.3 and 9.3, respectively.

Finally, we have proposed a so-called LUAR algorithm to reduce the cost of the Update step. This algorithm consists in accumulating and recompressing the low-rank updates. The object of the next chapter is to study and compare the different strategies to recompress these low-rank updates.

CHAPTER

3 Strategies to add and recompress low-rank matrices

In this chapter, we present, analyze, and compare different strategies to compute the sum of several low-rank matrices. In general, the rank of the sum can be as large as the sum of the ranks of each matrix. Some intermediate recompression operations are needed to avoid the rank growing too much.

This broad setting is in particular applicable to two special cases of the BLR factorization, in which the low-rank matrices that are to be summed are the update contributions $\widetilde{C}_{i,k}^{(j)}$.

- In the CUFS variant, the update contributions, together with the already compressed original entries of the matrix, are recompressed to compute the final low-rank representation of the blocks, as presented in Section 2.4.
- In the LUAR algorithm, the update contributions are first accumulated and recompressed before being decompressed, in order to reduce the cost of the Outer Product step, as presented in Section 2.6.

In this chapter, we place ourselves in the context of the LUAR variant, and analyze and compare the different recompression strategies based on their effectiveness to reduce the cost of the Outer Product operation. We denote the accumulators by XZY^T, as illustrated in Figure 2.7. Z is referred to as the *middle accumulator*, while X and Y are referred to as the *outer accumulators*.

These strategies have also been analyzed in the context of the CUFS variant in Anton, Ashcraft, and Weisbecker (2016). We summarize the main findings of that study in Section 3.4, and explain why they lead to different conclusions.

The different recompression strategies can be distinguished and classified based on the three questions below. Consider the update of a block $F_{i,k}$ with $k-1$ update contributions $\{\widetilde{C}_{i,k}^{(j)}\}_{j=1:k-1}$. Then, one must decide:

- **When** do we recompress the update contributions? Do we first compute all the Inner Product operations before recompressing the accumulator, or do we interlace both operations?
- **What** update contributions do we recompress together? Assuming more than two update contributions are available for recompression, do we recompress them all together or separately? And if separately, in which order?

- **How** do we recompress the update contributions? Do we recompress X, Z, or Y? And in which order?

Answering these three questions is the object of Sections 3.1 through 3.3. Then, before concluding, we present some experiments using these strategies on real-life sparse matrices in Section 3.5.

3.1 When: non-lazy, half-lazy, and full-lazy strategies

Algorithm 3.1 presents three different strategies to decide when to recompress the $k-1$ update contributions of a given block $F_{i,k}$:

- The first strategy consists in computing and accumulating all update contributions first and then recompressing the accumulator. This strategy, described in Algorithm 3.1a, is referred to as *full-lazy* recompression in the sense that the recompression is performed as late as possible.

- The second strategy consists in recompressing the accumulator each time a new update contribution is computed. This strategy, described in Algorithm 3.1b, is referred to as *non-lazy* recompression in the sense that the recompression is performed as soon as possible.

- The third strategy is an intermediary between the first two, where the computation of the update contributions and the recompressions are interlaced to some degree. It is referred to as *half-lazy* recompression.

As summarized in Table 3.1, these strategies can be compared based on two criteria: memory consumption and recompression granularity. Non-lazy recompression leads to the lowest memory consumption by recompressing the accumulators as often as possible, while full-lazy leads to the highest consumption. Conversely, full-lazy achieves the highest granularity for the recompression operation, while non-lazy achieves the lowest.

	memory	granularity
full-lazy	high	high
half-lazy	average	average
non-lazy	low	low

Table 3.1 – Comparison of the memory consumption and granularity for full-, half-, and non-lazy recompression.

There is no strategy better than the others in general (even if it is suggested in Anton et al. (2016) that half-lazy achieves the best compromise). However, in the special case of a left-looking factorization, it can be argued that full-lazy recompression is the best. Indeed, in a sequential (monothreaded) left-looking factorization,

Algorithm 3.1a Full-lazy recompression.

1: **for** $j = 1$ **to** $k-1$ **do**
2: Compute update contribution $\widetilde{C}_{i,k}^{(j)}$
3: Add it to accumulator $\widetilde{C}_{i,k}^{(acc)}$
4: **end for**
5: Recompress accumulator $\widetilde{C}_{i,k}^{(acc)}$

Algorithm 3.1b Non-lazy recompression.

1: **for** $j = 1$ **to** $k-1$ **do**
2: Compute update contribution $\widetilde{C}_{i,k}^{(j)}$
3: Add it to accumulator $\widetilde{C}_{i,k}^{(acc)}$
4: Recompress accumulator $\widetilde{C}_{i,k}^{(acc)}$
5: **end for**

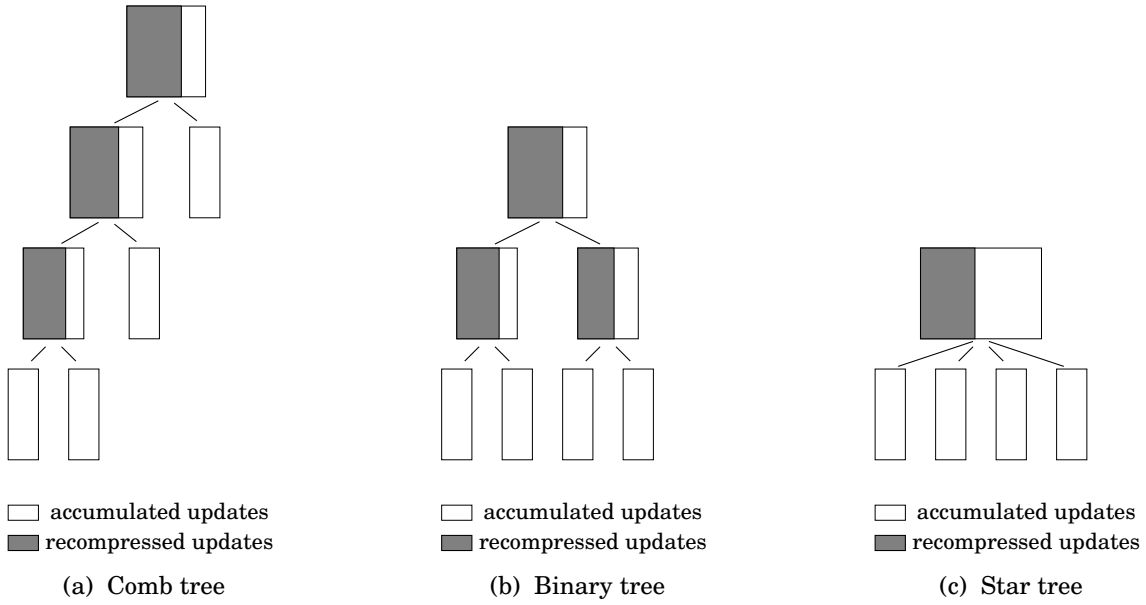
once we start updating a block $F_{i,k}$, its update contributions are all computed, accumulated, and recompressed before starting to update another block $F_{i',k'}$. Thus, only one accumulator is needed (in the multithreaded case, each thread updates a different block and thus the number of accumulators can still be bounded by the number of threads used). This allows us to easily control the memory consumption while benefiting from the higher granularity. More importantly, the full-lazy recompression gives the complete freedom to choose which update contributions are recompressed with which and in which order. As we are going to show in the next section, this freedom is crucial to limit the cost of the recompression; in particular, it allows us to sort the update contributions by increasing rank.

Therefore, since we consider a left-looking factorization, we will limit the rest of this study (Sections 3.2 and 3.3) to the full-lazy recompression strategy. However, part of our findings should still apply to non- and half-lazy recompression in the context of right-looking or task-based factorizations. Comparing them with our left-looking full-lazy approach is out of the scope of this work.

3.2 What: merge trees

We now consider the following context: $k-1$ update contributions $\{\widetilde{C}_{i,k}^{(j)}\}_{j=1:k-1}$ have been computed; we now investigate the best strategy to recompress them: which update contribution with which, and in which order? This problem can be modeled with so-called *merge trees*, built as follows:

- The update contributions $\widetilde{C}_{i,k}^{(j)}$ constitute the leaves of the tree.
- The other nodes are defined as the recompressed accumulation of their children.
- The root of the tree is the final accumulator on which the Outer Product step is performed.



(a) Comb tree (b) Binary tree (c) Star tree

Figure 3.1 – Merge trees comparison.

Note that this model can be easily generalized to forests by defining the final accumulator as the result of the accumulation (with no recompression) of all the roots.

In Figure 3.1, we present three examples of merge trees: comb tree, binary tree, and star tree.

- With a comb merge tree, we first recompress together $\widetilde{C}_{i,k}^{(1)}$ and $\widetilde{C}_{i,k}^{(2)}$ and obtain $\widetilde{C}_{i,k}^{(12)}$, which we in turn recompress with $\widetilde{C}_{i,k}^{(3)}$ to obtain $\widetilde{C}_{i,k}^{(123)}$, and so on until the final accumulator $\widetilde{C}_{i,k}^{(123\ldots k-1)}$ is computed.
- With a star merge tree, all $\widetilde{C}_{i,k}^{(j)}$ for $j \in [1;k-1]$ are recompressed together at the same time.
- With a binary merge tree, we compress the leaves two-by-two to obtain the next level made of $\lceil (k-1)/2 \rceil$ nodes, and so on until the root level. This can be easily generalized to q-ary trees for $q \geq 2$.

Note that there are some constraints on the shape of the merge tree depending on the laziness of the recompression strategy: while any shape is possible with full-lazy (hence our choice), the non-lazy strategy for example imposes to use a comb merge tree.

3.2.1 Merge tree complexity analysis

We now present a theoretical complexity analysis for comb, star, and q-ary merge trees. We first consider a uniform rank assumption: we consider n_ℓ leaves of size b and rank r. We show that under that uniform assumption, the star tree leads to the lowest complexity. We then generalize the analysis to non-uniform rank distributions, for which we show other merge trees can be more effective.

3.2.1.1 Uniform rank complexity analysis

We assume the recompression is performed by means of a QR factorization with column pivoting. We also assume $n_\ell r \leq b$; removing this assumption would lead to the same conclusions but would complexify the following computations.

Worst-case complexity analysis In the worst case, there is no recompression at all. The cost of compressing a given node is then

$$\mathscr{C}_{rec}^{worst}(R) = c_{merge} b R^2. \tag{3.1}$$

where c_{merge} is a constant that depends on which merge kernel is used (cf. Section 3.3) and R is the sum of the ranks of the children of the node to be recompressed. Then, we compute the worst-case complexity $\mathscr{C}_{star}^{worst}$, $\mathscr{C}_{comb}^{worst}$, and $\mathscr{C}_{qary}^{worst}$ when using a star, comb, and q-ary merge tree, respectively. We have

$$\mathscr{C}_{star}^{worst}(n_\ell) = \mathscr{C}_{rec}^{worst}(n_\ell r) = c_{merge} b r^2 n_\ell^2, \tag{3.2}$$

$$\begin{aligned}\mathscr{C}_{comb}^{worst}(n_\ell) &= \sum_{\ell=1}^{n_\ell - 1} \mathscr{C}_{rec}^{worst}((\ell+1)r) = c_{merge} b r^2 \sum_{\ell=1}^{n_\ell - 1} (\ell+1)^2 \\ &= c_{merge} b r^2 (n_\ell (n_\ell + 1)(2n_\ell + 1)/6 - 1),\end{aligned} \tag{3.3}$$

and, assuming n_ℓ is a power of q,

$$\begin{aligned}\mathscr{C}_{qary}^{worst}(n_\ell) &= \sum_{\ell=1}^{\log_q n_\ell} n_\ell q^{-\ell}\, \mathscr{C}_{rec}^{worst}(q^\ell r) = c_{merge} b r^2 n_\ell \sum_{\ell=1}^{\log_q n_\ell} q^\ell \\ &= c_{merge} b r^2 n_\ell (n_\ell - 1) q/(q-1).\end{aligned} \tag{3.4}$$

Therefore, the conclusion of the worst-case analysis is

$$\frac{\mathscr{C}_{comb}^{worst}(n_\ell)}{\mathscr{C}_{star}^{worst}(n_\ell)} \propto n_\ell/3, \quad \frac{\mathscr{C}_{qary}^{worst}(n_\ell)}{\mathscr{C}_{star}^{worst}(n_\ell)} \propto q/(q-1), \text{ and } \quad \frac{\mathscr{C}_{comb}^{worst}(n_\ell)}{\mathscr{C}_{qary}^{worst}(n_\ell)} \propto n_\ell (q-1)/3q. \tag{3.5}$$

Unsurprisingly, the star tree is the best option if no recompression gains can be achieved, since it limits the number (and thus the cost) of the recompressions.

Best-case complexity analysis In the best case, every intermediary node in the merge tree has rank r. The cost of compressing a given node is then

$$\mathscr{C}_{rec}^{best}(R) = c_{merge} b R r. \tag{3.6}$$

We compute

$$\mathscr{C}_{star}^{best}(n_\ell) = \mathscr{C}_{rec}^{best}(n_\ell r) = c_{merge} b r^2 n_\ell, \tag{3.7}$$

$$\mathscr{C}_{comb}^{best}(n_\ell) = \sum_{\ell=1}^{n_\ell - 1} \mathscr{C}_{rec}^{best}(2r) = 2 c_{merge} b r^2 (n_\ell - 1), \tag{3.8}$$

and, assuming n_ℓ is a power of q,

$$\mathscr{C}^{best}_{qary}(n_\ell) = \sum_{\ell=1}^{\log_q n_\ell} n_\ell q^{-\ell}\, \mathscr{C}^{best}_{rec}(qr) = c_{merge} b r^2 n_\ell \sum_{\ell=1}^{\log_q n_\ell} q^{1-\ell}$$
$$= c_{merge} b r^2 (n_\ell - 1) q/(q-1). \tag{3.9}$$

Therefore, the conclusion of the best-case analysis is

$$\frac{\mathscr{C}^{best}_{comb}(n_\ell)}{\mathscr{C}^{best}_{star}(n_\ell)} \propto 2, \quad \frac{\mathscr{C}^{best}_{qary}(n_\ell)}{\mathscr{C}^{best}_{star}(n_\ell)} \propto q/(q-1), \text{ and } \frac{\mathscr{C}^{best}_{comb}(n_\ell)}{\mathscr{C}^{best}_{qary}(n_\ell)} \propto 2(q-1)/q. \tag{3.10}$$

which means the star tree is also the best option in the best case, which is more interesting. The first conclusion of this complexity analysis is therefore that nodes of similar rank should be merged all together.

3.2.1.2 Non-uniform rank complexity analysis

The uniform rank analysis leads to the conclusion that the star tree recompression is always the best in terms of cost. However, this conclusion is in fact erroneous in practice as we will show in our numerical experiments. This is because for real problems, the rank distribution of the nodes is not uniform. We now extend the complexity analysis to non-uniform rank distributions to prove the star tree recompression may in fact be significantly worse for some particular rank distributions.

We assume there are n_ℓ leaves of rank r_ℓ. Since we have shown in the previous analysis that close ranks should be recompressed all together, we assume the the leaves are regrouped into n_c clusters of similar rank. For the sake of simplicity, we assume there are p leaves in each cluster, all of identical rank r_c, for $c = 1, \ldots, n_c$. This assumption may not be entirely realistic because in practice low-rank blocks are more numerous than high-rank ones, but it simplifies the computations and accomplishes the purpose of this section: to show that star tree recompression is not *necessarily* the best choice.

Under this assumption, we compute the best-case complexity of the star and comb tree recompression. In the comb tree case, we first recompress each uniform-rank cluster by means of a star tree recompression, and then use a comb tree recompression on the n_c remaining nodes. We have

$$\mathscr{C}^{best}_{star}(n_\ell) = c_{merge} b p \sum_{c=1}^{n_c} r_c r_{n_c}, \tag{3.11}$$

and

$$\mathscr{C}^{best}_{comb}(n_\ell) = \sum_{c=1}^{n_c} c_{merge} b p r_c^2 + \sum_{c=1}^{n_c-1} c_{merge} b (r_c + r_{c+1}) r_{c+1}$$
$$= c_{merge} b \left(\sum_{c=1}^{n_c} p r_c^2 + \sum_{c=1}^{n_c-1} (r_c + r_{c+1}) r_{c+1} \right). \tag{3.12}$$

To assess which merge tree achieves the lower complexity, we compute the differ-

ence

$$\mathscr{C}^{best}_{star}(n_\ell) - \mathscr{C}^{best}_{comb}(n_\ell) = c_{merge} b \sum_{c=1}^{n_c-1} \left(p r_c r_{n_c} - p r_c^2 - (r_c + r_{c+1}) r_{c+1} \right) \tag{3.13}$$

To proceed further, we need to assume a specific rank distribution. Let us assume $r_{c+1} = \alpha r_c = \alpha^c r$ (noting $r = r_1$), with $\alpha \geq 1$. Then, we have

$$\mathscr{C}^{best}_{star}(n_\ell) - \mathscr{C}^{best}_{comb}(n_\ell) = c_{merge} b r^2 \sum_{c=1}^{n_c-1} \left(\alpha^{c-1} \left(p \left(\alpha^{n_c-1} - \alpha^{c-1} \right) - \alpha^c (1+\alpha) \right) \right) \tag{3.14}$$

We omit the tedious but straightforward computation of the previous expression as a sum of geometric series, which leads to the final result

$$\operatorname{sign}\left(\mathscr{C}^{best}_{star}(n_\ell) - \mathscr{C}^{best}_{comb}(n_\ell) \right) = \operatorname{sign}\left(\alpha^{2n_c-1}(p-1-\alpha) - \alpha^{n_c-1}(\alpha+1)p + \alpha^2 + \alpha + p \right) \tag{3.15}$$

The sign of the difference thus behaves asymptotically (for large $n_c = n_\ell/p$) as the sign of $p - 1 - \alpha$. Figure 3.2 illustrates how either star or comb tree recompression can be the cheapest strategy depending on the values of p and α.

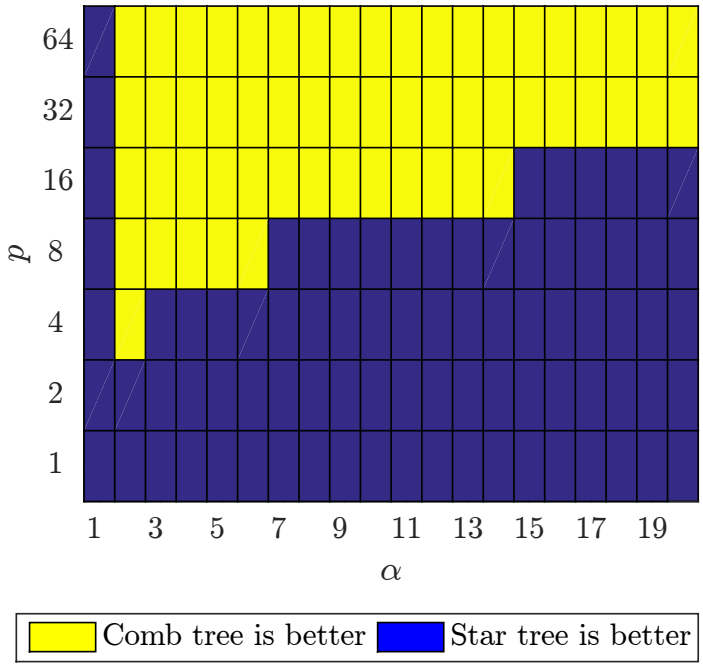


Figure 3.2 – Comparison of star and comb tree recompression for non-uniform rank distributions defined by $\forall c \in [1; n_c], \forall \ell \in [(c-1)p + 1; cp], r_\ell = r_c = \alpha^{c-1} r$.

We leave out the non-uniform complexity analysis of the q-ary merge trees.

3.2.2 Improving the recompression by sorting the nodes

Even for a fixed merge tree, we still have the freedom to choose which update contribution belongs to which node of the tree. One can of course simply assign the update contributions in their natural order (i.e. $C_{i,k}^{(j)}$ is assigned the j-th leaf of the tree), but there are better strategies, that we described in the following.

The rank of a parent node is necessarily greater than the biggest rank of its children; thus, to maximize the potential gain, we want to merge nodes of similar ranks: a rank-10 and a rank-90 nodes can only recompress to a rank-90 node, while two rank-50 nodes can potentially recompress to a rank-50 node. This is illustrated in Figure 3.3.

(a) Low potential (b) High potential

Figure 3.3 – Potential recompression is greater for similar ranks.

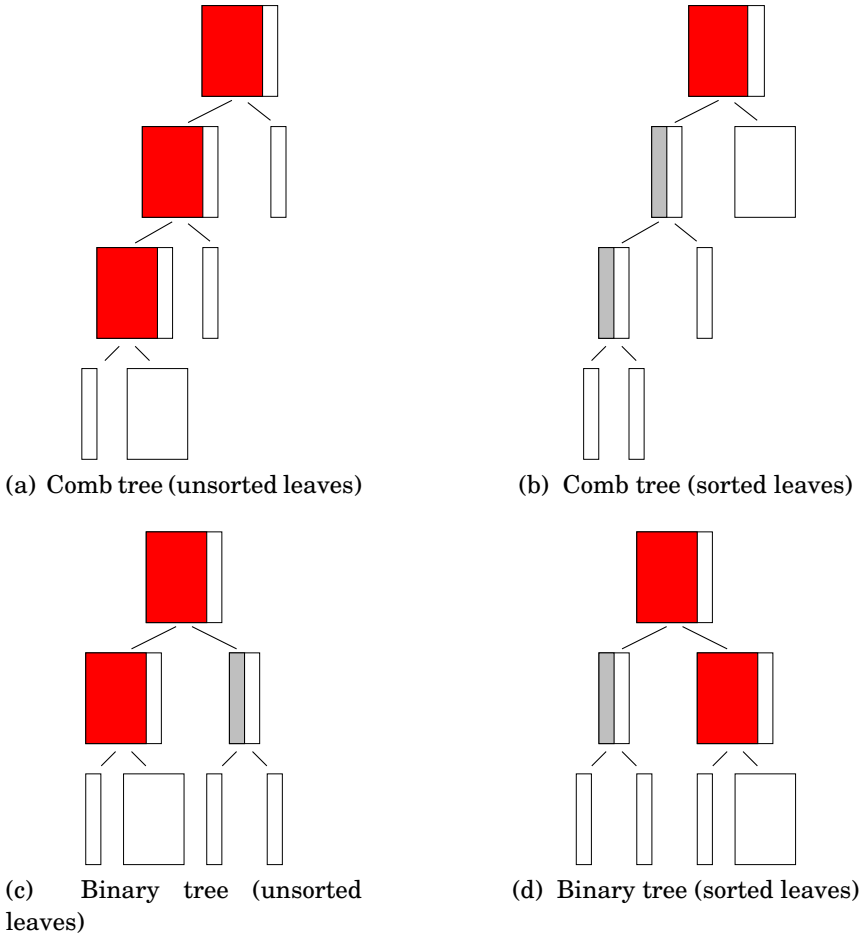
Therefore, a basic approach is to sort the leaves of the tree by (increasing or decreasing) rank. We want to avoid merging big rank with small rank nodes; therefore, sorting by decreasing rank is likely to be a poor strategy since all the big rank nodes will be merged first and will thus lead to an expensive merge with the subsequent small rank nodes. Conversely, if the small rank nodes are merged first, they can be merged at low cost, while the most costly merges will then be done as late as possible. This is illustrated in Table 3.2.

	recompression flops ($\times 10^9$)					
merge tree type	do not sort	sort leaves		sort at each level		savings
		inc.	dec.	inc.	dec.	
comb	3.56	2.39	14.30	2.39	14.30	33%
binary	2.64	2.50	3.30	2.47	3.30	6%
ternary	2.68	2.63	2.86	2.59	2.86	3%
quaternary	2.44	2.49	2.90	2.49	2.90	-2%
star	1.96	1.96	1.96	1.96	1.96	0%
dynamic	2.36					

Table 3.2 – Impact of sorting the nodes for different merge trees on the root node of a 128^3 Poisson problem. The reported savings correspond to the gains achieved by the best sorting strategy (which is here sorting at each level by increasing rank) with respect to the unsorted strategy.

For the same reason, this sorting strategy has a much stronger impact on the comb tree than the other types of merge tree: indeed, with a comb tree, as soon as a big rank node is encountered, it will be merged with all the subsequent nodes,

which increases the cost of the recompression. In a q-ary tree, a big rank node is not merged with all subsequent nodes, but simply those lying in the same subtree. This is illustrated in Figure 3.4 and shown by the results in Table 3.2: the recompression cost is reduced by 33% with a comb merge tree, while it is barely improved with q-ary trees. Finally, note that star tree recompression is obviously not influenced by the leaves order.



(a) Comb tree (unsorted leaves)

(b) Comb tree (sorted leaves)

(c) Binary tree (unsorted leaves)

(d) Binary tree (sorted leaves)

Figure 3.4 – Impact of sorting the leaves of the merge tree: gray merges are inexpensive while red are costly; sorting is more important for comb trees than binary trees.

A more advanced strategy consists in resorting the nodes at each level of the tree. This does not make a difference for the comb tree, since there are only two nodes on each level, but can slightly improve the cost with q-ary trees, as shown in Table 3.2. Finally, rather than guiding the recompression with a static merge tree, one could dynamically build the tree based on the ranks of the nodes: we initialize a pool of nodes with the leaves, merge together the q nodes of smallest rank, reinsert the result in the pool, and iterate until only one node is left. The result of this strategy with $q = 2$ is also reported in Table 3.2 (cf. row “dynamic”).

3.3 How: merge strategies

The last question regards how a given node of the merge tree is recompressed. Several merge kernels can be considered. A given node corresponds to an accumulator XZY^T made of q update contributions, as already illustrated in Figure 2.7.

The merge kernels differ in two ways:

- The part of the accumulator that is recompressed (X, Y, Z, or a combination of the three);
- If more than one part is recompressed, the order in which they are.

3.3.1 Weight vs geometry recompression

Consider the computation of a leaf node of the merge tree associated with an accumulator $\widetilde{C}_{i,k}^{(acc)} = XZY^T$. In the rest of this section, we place ourselves in the context where the compression kernel $\widetilde{F}_{i,j} = X_{i,j}Y_{i,j}^T$ used yields orthonormal $X_{i,j}$ matrices (e.g. a truncated QR with column pivoting). In that case, the information contained in the outer X,Y, and middle Z accumulators are of different nature.

Indeed, X is made of q orthonormal matrices $X_{i,j}$ (with $j \in [1;k-1]$) and thus contains geometric information. Similarly, Y is made of q orthonormal matrices $Y_{k,j}$. In the following, we refer to X and Y as *block-wise orthonormal columns* (BOC). Note that X and Y are BOC only for the leaf nodes of the merge tree, since the accumulators associated with nodes higher up in the tree depend on how nodes are merged together (so-called merge kernels, discussed below).

On the other hand, Z is a block-diagonal matrix where each block $Y_{i,j}^T D_{j,j} Y_{k,j}$ contains all the weight information relative to the j-th update contribution.

Each diagonal block of Z can be independently recompressed (since we use an absolute low-rank threshold), and this makes the recompression of Z particularly cheap (compared to that of X and Y) due to the small size of its diagonal blocks. In Table 3.3, we compare the following three approaches:

- No recompression at all: we simply multiply each $Z_{j,j}$ to whichever side (X or Y) minimizes the flops, as explained in Section 1.4.1.3.
- Exploit only weight information: we recompress Z (i.e. compute $Z_{j,j} = X_{Z_{j,j}} Y_{Z_{j,j}}^T$ for each diagonal block $Z_{j,j}$) and multiply $X_{Z_{j,j}}$ and $Y_{Z_{j,j}}$ with the corresponding columns of X and Y, respectively.
- Exploit both weight and geometry information: we first recompress Z as previously, but then also recompress X and Y.

We compare the number of operations performed in the Outer Product and Recompress steps, as well as the global impact on the total number of operations. When no recompression is performed, the Outer Product represents a significant part of the total computations (around 20%), which illustrates the importance of recompressing. Exploiting weight only reduces its cost by a factor 3, at almost no overhead, due to the small size of the diagonal blocks of Z. This reduces the total by a significant amount (around 15%). If additionally, geometry information is

	flops ($\times 10^9$)		
recompression type	Recompress	Outer Product	Total
none	0.0	21.0	99.6
weight only	0.1	6.9	86.6
weight+geometry	2.4	2.0	83.9

Table 3.3 – Comparison between exploiting weight only or both weight and geometry informations when recompressing the accumulators, on the root node of a 128^3 Poisson problem. Note that the total columns is not equal to the sum of the two other columns as it also includes other steps involved in the factorization.

recompressed, the cost of the Outer Product is further divided and becomes negligible; however, the recompression overhead in this case is significant, due to the relatively large size of X and Y. This leads to a marginal improvement of the total operations (around 3%).

Therefore, on this problem, recompression based on weight information only captures almost all the recompression gain while keeping the recompress overhead limited.

As a consequence, in the context of the unsymmetric LU factorization, it is important to compress U^T rather than U. Indeed, the LR-LR inner product operation yields a middle block of the form $Z = Y_{i,j}^T X_{j,k}$ or $Z = Y_{i,j}^T Y_{j,k}$ depending if U or U^T is compressed, respectively. It is preferable to recompress a middle block that contains all the weight information rather than part of it.

3.3.2 Exploiting BOC leaf nodes

As said before, on the leaf nodes of the merge tree, X and Y are column-wise orthonormal. This property can be exploited to efficiently recompress them by skipping the recompression of the first set of orthonormal columns: indeed, assume we want to recompress $X = (X_1 \dots X_q)$ and let us note $X_{2:q} = (X_2 \dots X_q)$. Then, recompressing X can be done as follows. First, we remove the span of X_1 from $X_{2:q}$:

$$W \leftarrow X_{2:q} - X_1 X_1^T X_{2:q}. \tag{3.16}$$

Then, we perform a truncated QR factorization with column pivoting of W:

$$W = X_W Y_W^T, \tag{3.17}$$

and this leads to the recompressed form of XZY^T:

$$XZY^T = (X_1 \;\; X_{2:q}) ZY^T = \left(X_1 \;\; (W + X_1 X_1^T X_{2:q})\right) ZY^T \tag{3.18}$$

$$= \left(X_1 \;\; (X_W Y_W^T + X_1 X_1^T X_{2:q})\right) ZY^T \tag{3.19}$$

$$= (X_1 \;\; X_W) \begin{pmatrix} I & X_1^T X_{2:q} \\ 0 & Y_W^T \end{pmatrix} ZY^T = X'Z'Y^T. \tag{3.20}$$

The impact of exploiting the column-wise orthonormality of X strongly depends on how large is X_1 compared to $X_{2:q}$, as shown by Table 3.4. The larger is X_1, the greater are the savings. Thus, the greatest savings can be achieved in the case of a comb merge tree (28% of the recompression cost); significant gains can also be achieved for q-ary trees, although the savings decrease when q increases: 19%, 13%, and 11%, for $q = 2$, 3, and 4, respectively. Finally, the savings in the star tree case are expectedly negligible (5%).

	recompression flops ($\times 10^9$)		
merge tree type	do not exploit	do exploit	savings (%)
comb	3.07	2.22	28%
binary	4.36	3.51	19%
ternary	4.10	3.55	13%
quaternary	3.93	3.49	11%
star	3.64	3.47	5%

Table 3.4 – Usefulness of exploiting the orthonormality of X_1 and Y_1 when recompressing X and Y depending on the merge tree used, on the root node of a 128^3 Poisson problem.

We can similarly exploit the column-wise orthonormality of Y to efficiently recompress it.

3.3.3 Merge kernels

As said before, when recompressing more than one part (X, Y, and Z) of the accumulator (as is the case when exploiting both weight and geometry), we have some freedom on the order in which we recompress them.

Z should always be recompressed first, because, as we have seen, it captures most of the recompression and allows for a considerable reduction of the size of the X and Y parts and thus the recompression overhead. We therefore now consider the recompression of an accumulator XY^T. Note that either X or Y is BOC (but not both), depending on whether Z or Z^T has been compressed, respectively (because X_Z is orthonormal). In the following, if X or Y is BOC, we refer to the accumulator as left- or right-BOC, respectively.

Several *merge kernels* can be distinguished:

- mergeLeft and mergeRight (**mergeL** and **mergeR**): we recompress only one side. With mergeL, we recompress the left side $X = X_X Y_X^T$ and obtain the new accumulator $X \leftarrow X_X$ and $Y \leftarrow Y Y_X$. mergeR is the equivalent of mergeL but on the right side (Y recompression). mergeL and mergeR can exploit left- and right-BOC inputs, respectively.

- mergeBoth and mergeBothCompress (**mergeB** and **mergeBC**): we recompress both sides at the same time: $X = X_X Y_X^T$ and $Y = X_Y Y_Y^T$; therefore, both left- and right-BOC inputs can be exploited. Then, the product $W = Y_X^T Y_Y$ is formed. In mergeB, we then multiply W to whichever side minimizes the

flops; in mergeBC, we compress $W = X_W Y_W^T$ and the new accumulator is given by $X \leftarrow X_X X_W$ and $Y \leftarrow X_Y Y_W$.

- mergeRightLeft and mergeLeftRight (**mergeRL** and **mergeLR**): we also recompress both sides, but one after the other: in mergeRL, $Y = X_Y Y_Y^T$ is recompressed first; we update X: $X \leftarrow XY_Y$ and recompress it: X: $X_X Y_X^T$; finally, the new accumulator is given by $X \leftarrow X_X$ and $Y \leftarrow X_Y Y_X$. mergeRL can exploit a right-BOC input but not a left-BOC one because when the left side is compressed, $X \leftarrow XY_Y$ is not BOC anymore. Conversely, mergeLR can exploit a left-BOC input but not a right-BOC one.

Before comparing these merge kernels, let us discuss how to exploit BOC inputs on general nodes (not just the leaves).

3.3.4 Exploiting BOC general nodes

As said before, X or Y are BOC only for the leaf nodes of the merge tree. To preserve this property of nodes higher up the tree, one must recompress XY^T in such a way that either the new X or the new Y has orthonormal columns (OC). Similarly as for inputs, we distinguish left- and right-OC outputs.

Let us reexamine the merge kernels with that in mind:

- On output of mergeL, X is orthonormal and the output is thus left-OC. Conversely, mergeR yields a right-OC output.

- mergeBC yields an output that is either left- or right-OC, depending if W or W^T is compressed, respectively (because X_W is orthonormal). Since we can choose whether to compress W or W^T, we can easily control the kind of output yielded. With mergeB, the output depends on which side W is multiplied to, which is decided based on the side which minimizes the flops of the Inner and Outer Product (as explained in Section 1.4.1.3). Thus, while the output is necessarily either left- or right-OC, we cannot easily control which it effectively is. However, there are several ways to systematically enforce left- or right-OC at the cost of additional operations. For example, to enforce left-OC, one can:.

  - Force the multiplication of W to the right, even when it should be multiplied to the left;
  - When W is multiplied to the left, perform one more recompression on the left side;
  - When W should be multiplied to the left, compress W instead (i.e. switch to mergeBC).

  A right-OC output can be similarly enforced.

- mergeRL and mergeLR yield a left- and right-OC output, respectively. However, we have seen that they can exploit right- and left-BOC inputs, respectively. The fact that the desired input and yielded output are BOC/OC on opposite sides can be problematic as it will lead to left- and right-OC nodes

being merged together[1]. For this reason, we also introduce two additional kernels...

- mergeLeftRightTranspose and mergeRightLeftTranspose (**mergeLRT** and **mergeRLT**): in mergeLRT, we first compress X (and thus left-BOC inputs can be exploited), i.e compute $X = X_X Y_X^T$; we then update Y ($Y \leftarrow Y Y_X$) before compressing its transpose ($Y = Y_Y X_Y^T$). The final accumulator is then given by $X \leftarrow X_X X_Y$ and $Y \leftarrow Y_Y$, and is thus left-OC. Similarly, mergeRLT can exploit a right-BOC input and yields a right-OC output.

In Table 3.5, we summarize which kind of inputs can be exploited and which kind of output are yielded by each merge kernel.

merge kernel	can exploit _ input	can yield _ output
mergeL	left-BOC	left-OC
mergeR	right-BOC	right-OC
mergeB(C)	right-BOC left-BOC	right-OC left-OC
mergeRL	right-BOC	left-OC
mergeLR	left-BOC	right-OC
mergeLRT	left-BOC	left-OC
mergeRLT	right-BOC	right-OC

Table 3.5 – Comparison of what kind of input can exploited and what kind of output can be yielded by different merge kernels.

For the sake of exhaustivity, note that merge{RTL,LTR,RT,LT,RTLT,LTRT} kernels could also be defined. However, these kernels cannot exploit neither left- or right-BOC inputs, and, additionally, the latter four yield a general output (i.e. neither left- or right-OC). Therefore, we do not consider them.

In Table 3.6, we compare the gains obtained exploiting left-BOC inputs for different merge kernels. Expectedly, mergeL benefits the most, followed by mergeLRT, mergeB, and mergeBC; mergeRL cannot exploit left-BOC inputs and its recompression cost is thus not reduced. Interestingly, without exploiting the column-wise orthonormality of the input, mergeL results in an extremely expensive recompression, while mergeB for example, which also recompresses the left side, does not. Even though mergeB is necessarily more expensive than mergeL at the leaf nodes, it seems that recompressing the right side as well leads to smaller ranks on the nodes higher in the tree and thus considerably reduces the overall recompression cost. It might thus be important to recompress both sides.

Finally, in Figure 3.5, we compare the total factorization cost depending on the merge kernel used. Left-BOC inputs are exploited and thus only merge kernels yielding left-OC output are considered. Similar results would be obtained exploiting right-BOC inputs and using the corresponding merge kernels. We use a comb

[1]In this case, the input is neither left- or right-BOC, but can still be recompressed efficiently because the first set of columns of either X or Y is necessarily orthonormal. This is however somewhat complex and is ignored in the following.

merge kernel	recompression flops ($\times 10^9$) do not exploit	do exploit	savings (%)
mergeL	10.70	0.44	96%
mergeB	3.11	2.12	32%
mergeBC	3.09	2.13	31%
mergeRL	2.55	2.55	0%
mergeLRT	2.25	1.16	48%

Table 3.6 – Usefulness of exploiting the orthonormality of X_1 when recompressing X and Y depending on the merge kernel used, on the root node of a 128^3 Poisson problem. A comb merge tree with leaves sorted by increasing rank is used. We consider only kernels that yield left-OC output so that left-BOC inputs can be exploited on every node of the merge tree, not just the leaves. Similar results would be obtained exploiting right-BOC inputs and using the corresponding merge kernels.

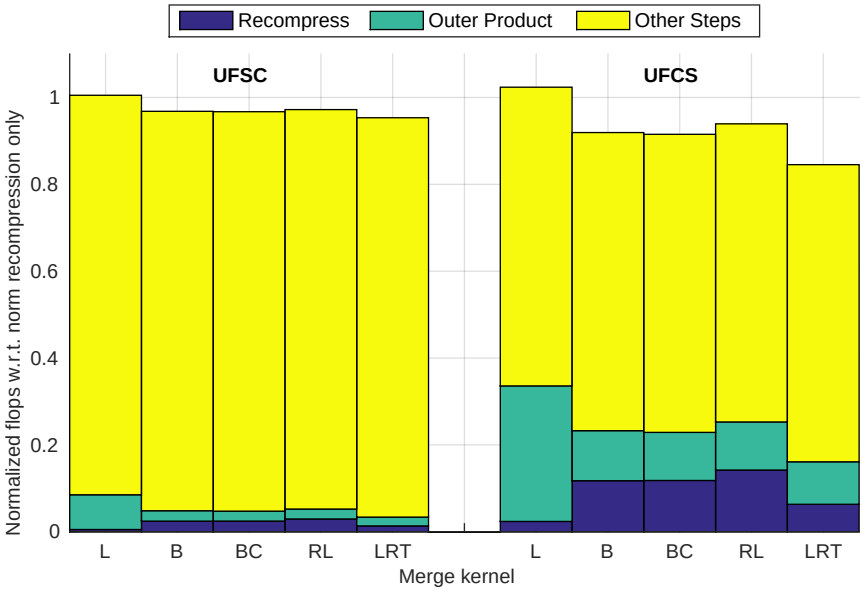
merge tree. We report the flops performed in the Recompression and the Outer Product steps, as well as the total factorization. The flops are normalized with respect to the total flops for the factorization with only recompression based on weight information (i.e. when X and Y are not recompressed). We consider two matrix sizes (root nodes of 128^3 and 256^3 Poisson problems) and two variants: UFSC and UFCS.

For the UFSC variant, the total flops are dominated by the Solve step and thus the impact of the merge kernels on the total is limited; it is more important for the UFCS variant. This effect is especially clear on the larger matrix (cf. Figure 3.5). The mergeL recompression is inexpensive but barely reduces the cost of the Outer Product and thus is not an effective way to exploit geometric information. mergeB, mergeBC, and mergeRL all achieve further compression compared to exploiting weight information only, but their overhead cost is important and leads to limited gains only. mergeLRT is in the end the most effective kernel, since its overhead cost is relatively low due to the possibility to efficiently recompress the accumulators; it achieves up to 30% gains with respect to the case where weight information only is exploited.

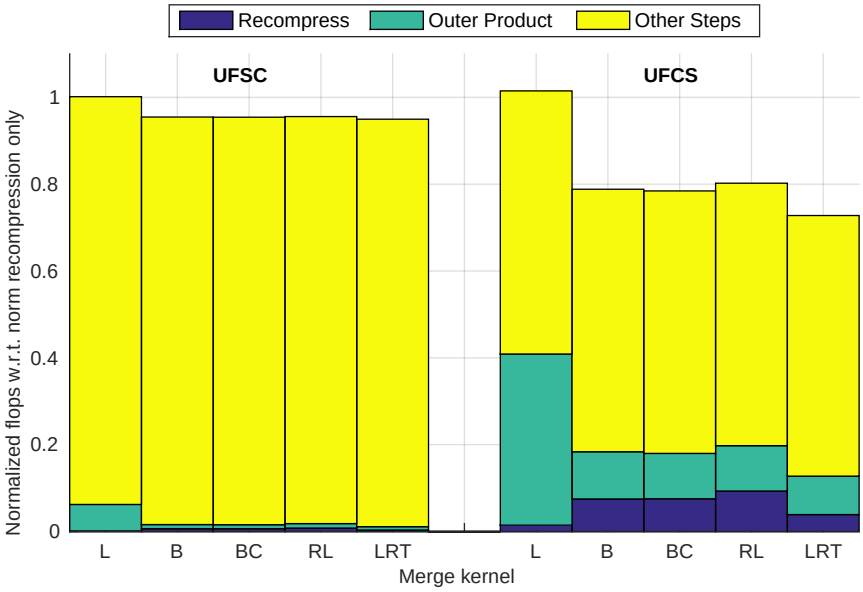
Note that these kernels lead not only to different recompression overhead costs but also to different accumulator ranks (i.e. Outer Product cost); this is currently unexplained. While it only leads to slight differences in this context, this makes a huge difference in the case of the CUFS factorization (as explained in Section 3.4) and should therefore be further investigated.

3.4 Recompression strategies for the CUFS variant

As explained in the beginning of this chapter, the recompression strategies that we have analyzed can also be applied to the CUFS factorization, presented in Section 2.4. In that context, they have a very different role. Indeed, while in the LUAR



(a) Root node of 128^3 Poisson problem.



(b) Root node of 256^3 Poisson problem.

Figure 3.5 – Merge kernels comparison on two problem sizes and two variants: UFSC and UFCS. The flops are given as a percentage of those achieved with weight recompression only.

context, the goal of is to reduce the cost of the Outer Product (by recompressing the accumulators), in the CUFS variant, there is no Outer Product. Instead, the update contributions and the original entries of the matrix are all recompressed together; the result is kept as a low-rank representation of the blocks, which is then used to perform the update of the subtrailing matrix. Therefore, in the CUFS context,

the goal of the recompression is to achieve as small a rank as possible. This leads to completely different conclusions on the comparison of the different recompression strategies. Such a study was carried out by Anton et al. (2016) and the main conclusions were:

- The global impact of the recompression is much more important than in the non fully-structured case, because even the smallest difference of rank makes a huge difference; this is because the computed low-rank blocks are subsequently reused to update the trailing submatrix. It is therefore absolutely necessary to recompress all the information, both on weight and geometry.

- For the same reason, big differences can be observed between different merge kernels, depending on the ranks obtained by the recompression. For example, mergeL is not a good choice because even though it keeps the merge overhead cost low, it achieves higher ranks than other merge kernels because it does not recompress both sides of the accumulator.

- Finally, because low-rank blocks are computed from the recompressed updates, it is important to enforce the output to be left-orthonormal in L and right-orthonormal in U, to maximize the weight recompression. This is not necessary with the LUAR algorithm because the blocks are first decompressed back to full-rank; then, the outer-orthonormality can be easily enforced by compressing L and U^T, as explained in Section 3.3.1

3.5 Experimental results on real-life sparse problems

We conclude this chapter with some experiments on real-life sparse problems, reported in Figure 3.6. We consider the UFSC variant. We measure the flops for the factorization with LUAR recompression normalized with respect to the flops without recompressing. We compare weight only and weight+geometry recompression. For the geometry recompression, we have tested several merge trees and merge kernels, and report the best for each problem. The merge tree nodes are sorted by increasing rank, and the column-wise orthonormality of the outer accumulators is exploited when possible. Important gains are achieved, up to 40% with weight recompression only and 50% with additionally geometry recompression.

It can be observed that the gain due to the recompression is greater for larger problems: for example, 24%, 29%, and 34% gain for the 5Hz, 7Hz, and 10Hz matrices (matrix ID 1-3), respectively. This illustrates the fact that the LUAR algorithm reduces the asymptotic complexity of the BLR factorization, which will be proved in Chapter 4 (Section 4.5). This also applies, to a lesser extent, to the gains due to the geometry recompression (compared to weight recompression only), and will also be discussed in Chapter 4 (cf. Figure 4.6).

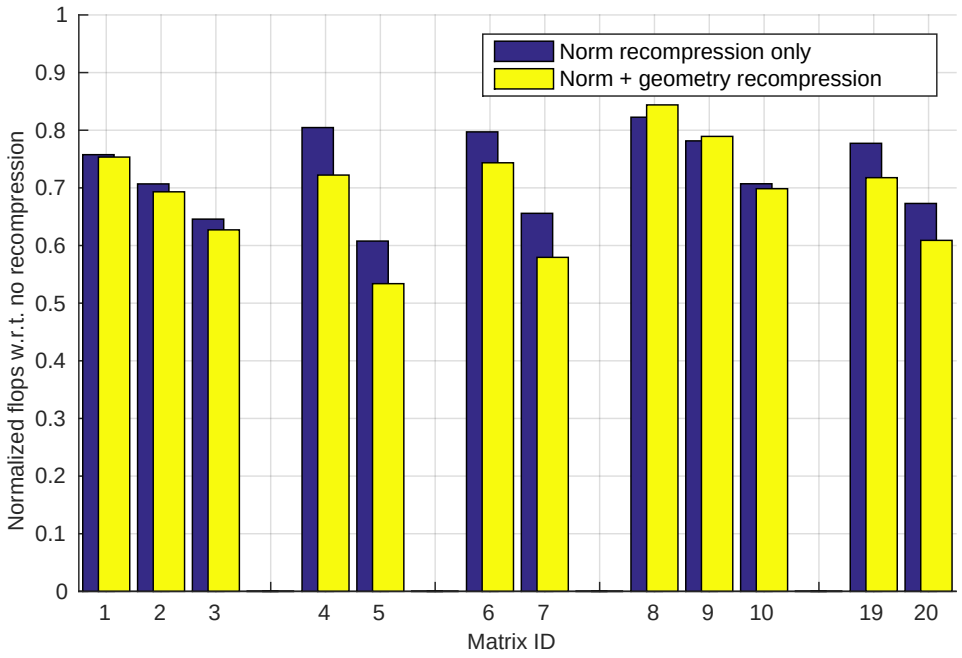


Figure 3.6 – Gains due to LUAR on real-life sparse problems using the UFSC factorization variant. The flop achieved by weight only and weight+geometry recompression are given as a percentage of the flops without recompression. For the geometry recompression, we have tested several merge trees and merge kernels, and report the best for each problem. The merge tree nodes are sorted by increasing rank, and the column-wise orthonormality of the outer accumulators is exploited when possible. The matrices ID have been assigned in Table 1.3.

3.6 Chapter conclusion

We have presented and analyzed strategies to add and recompress low-rank updates. We have applied these strategies to the BLR factorization in order to reduce the cost of the Outer Product step (so-called LUAR algorithm).

We have considered different strategies based on three criteria: when do we recompress the low-rank updates? What do we recompress? And how?

First, we have explained why, in a left-looking factorization, it is best to recompress once all updates have been computed, so as to have more flexibility.

Then, we formalized the problem of choosing which updates are recompressed together with the concept of merge tree: each node represents an update, and siblings are recompressed together. Our complexity analysis and experiments show that no merge tree is always better than the others. Moreover, we have underlined the importance of sorting the nodes by increasing rank.

Finally, we have analyzed several merge kernels, which differ on what type of information is recompressed. Weight information can be efficiently exploited by recompressing the middle accumulator (Z_S in Figure 2.7), which is of small size. Geometry information can be exploited by recompressing the outer accumulators (X_S and Y_S); even though it leads to a higher recompression cost, we have shown

we can exploit the orthonormality of X_S or Y_S to limit the overhead.

All in all, we have shown the accumulation and recompression of low-rank updates allows for significant gains in the factorization of large sparse real-life matrices, up to a factor 2 in number of operations. In the next chapters, we will come back to these recompression strategies to analyze their asymptotic complexity (Chapter 4), and their performance in shared- and distributed-memory parallel settings (Chapters 5 and 6).

CHAPTER

4 Complexity of the BLR Factorization

Unlike hierarchical formats, the theoretical complexity of the BLR factorization was unknown until recently. It remained to be proved that the BLR format does not only reduce the computations by a constant, i.e. leads to dense LU factorization of complexity $\mathcal{O}(m^3)$, just like in the full-rank case.

The main objective of this chapter is to compute the complexity of the BLR multifrontal factorization. We will first prove that the BLR UFSC (or FSCU in right-looking) factorization does provide a non-constant gain, and then show how the BLR factorization variants (described in Chapter 2) influence its complexity.

As explained in Chapter 1, the BLR format is strongly admissible, which means that off-diagonal blocks are allowed to be full-rank. One of the key results of this chapter (Lemma 4.1) is that the number of these off-diagonal full-rank blocks on any row can be bounded by a constant, even for a strong admissibility condition for which theoretical bounds on the ranks have been proven. This will allow us to compute the theoretical complexity of the BLR factorization.

We now briefly describe the contents of each section. In Section 4.1, we present the context of this complexity study, the solution of elliptic PDEs with the Finite Element (FE) method. In Section 4.2, we compute the theoretical bounds on the numerical ranks of the off-diagonal blocks in BLR matrices arising in our context. First, we briefly review the work done on hierarchical matrices and the complexity of their factorization. Then, we explain why applying this work to BLR matrices (which can be seen as a very particular kind of hierarchical matrices) does not provide a satisfying result. We then give the necessary ingredients to extend this work to the BLR case. In Section 4.3, we use the rank bounds computed in Section 4.2 to compute the theoretical complexity of the standard dense UFSC factorization. Then, we explain in Section 4.4 how the dense UFSC factorization can be used within each node of the tree associated with a sparse multifrontal factorization, and we compute the complexity of the corresponding BLR multifrontal factorization. We then analyze in Section 4.5 the other variants of the BLR factorization and show how they can further reduce its complexity. In Section 4.6, we support our theoretical complexity bounds with numerical experiments and analyze the influence on complexity of each variant of the BLR factorization and of two other parameters: the low-rank threshold and the block size. We conclude this chapter with some remarks in Section 4.7.

4.1 Context of the study

We describe the context in which we place ourselves for the complexity study, which is the same as in Bebendorf and Hackbusch (2003).

We consider a partial differential equation of the form:

$$\begin{aligned} Lu &= f \quad \text{in } \Omega \subset \mathbb{R}^d, \Omega \text{ convex }, d \geq 2, \\ u &= g \quad \text{on } \partial\Omega, \end{aligned} \tag{4.1}$$

where L is a uniformly elliptic operator in divergence form:

$$Lu = -\text{div}[C\nabla u + \boldsymbol{c_1} u] + \boldsymbol{c_2} \cdot \nabla u + c_3 u.$$

C is a $d \times d$ matrix of functions, such that $\forall x, C(x) \in \mathbb{R}^{d\times d}$ is symmetric positive definite with entries $c_{i,j} \in L^\infty(\Omega)$. Furthermore, $\boldsymbol{c_1}(x), \boldsymbol{c_2}(x) \in \mathbb{R}^d$ and $c_3(x) \in \mathbb{R}$.

We consider the solution of problem (4.1) by the Finite Element (FE) method. Let $\mathscr{D} = H_0^1(\Omega)$ be the domain of definition of operator L. We consider a FE discretization, with step size h, that defines the associated approximation of $\mathscr{D}$, the space $\mathscr{D}_h$. Let $n = N^d = \dim \mathscr{D}_h$ be its dimension and $\{\varphi_i\}_{i\in\mathscr{I}}$ the basis functions, with $\mathscr{I} = [1, n]$ the index set. Similarly as in Bebendorf and Hackbusch (2003), we assume that a quasi-uniform and shape-regular triangulation is used. We define X_i, the support of φ_i, and generalize the definition of support to subdomains:

$$X_\sigma = \bigcup_{i\in\sigma} X_i.$$

Please note that with this definition, the admissibility conditions defined in Section 1.4.2.1 should be rewritten by replacing σ and τ by X_σ and X_τ in the distance and diameter calls, respectively. For example, the least-restrictive strong admissibility condition (1.27) becomes

$$\sigma \times \tau \text{ is admissible } \Leftrightarrow \text{dist}(X_\sigma, X_\tau) > 0. \tag{4.2}$$

We note J the bijection defined by

$$J : \begin{array}{ccc} \mathbb{R}^n & \to & \mathscr{D}_h \\ x & \mapsto & \sum_{i\in\mathscr{I}} x_i \varphi_i \end{array}. \tag{4.3}$$

Equation (4.1) can be approximately solved by solving the discretized problem, which takes the form of a sparse linear system $Ax = b$ where A is the stiffness matrix defined by $A = J^* L J$. We consider the solution of that system using the multifrontal method to factorize A. We also define $B = J^* L^{-1} J$ and $M = J^* J$. B is the Galerkin discretization of L^{-1} and M the mass matrix.

A matrix of the form

$$S = A_{\Psi,\Psi} - A_{\Psi,\Phi} A_{\Phi,\Phi}^{-1} A_{\Phi,\Psi}, \tag{4.4}$$

for some $\Phi, \Psi \subset \mathscr{I}$ such that $\Phi \cup \Psi = \mathscr{I}$ is called a Schur complement of A. One of the main results of Bebendorf (2007) (Section 3) states that the Schur complements

of A can be approximated if an approximant of the inverse stiffness matrix A^{-1} is known.

Therefore, we are interested in finding $\widetilde{A}^{-1}$, approximant of the inverse stiffness matrix A^{-1}. The following result from FE theory will be used (cf. Bebendorf and Hackbusch (2003), Subsection 5.2): the discretization of the inverse of the operator is approximated by the inverse of the discretized operator, i.e.,

$$\|A^{-1} - M^{-1}BM^{-1}\|_2 \leq \mathcal{O}(\varepsilon_h), \tag{4.5}$$

where ε_h is the accuracy associated with the step size h of the FE discretization. In the following, for the sake of simplicity, we assume that the low-rank threshold ε is set to be equal to ε_h.

Then, assuming we can find $\widetilde{M}^{-1}$ and $\widetilde{B}$, approximants of the inverse mass matrix M^{-1} and of the B matrix, we have (cf. Bebendorf and Hackbusch (2003), Subsection 5.3):

$$\begin{aligned} M^{-1}BM^{-1} - \widetilde{M}^{-1}\widetilde{B}\widetilde{M}^{-1} = (M^{-1} - \widetilde{M}^{-1})BM^{-1} \\ + \widetilde{M}^{-1}(B - \widetilde{B})M^{-1} + \widetilde{M}^{-1}\widetilde{B}(M^{-1} - \widetilde{M}^{-1}). \end{aligned} \tag{4.6}$$

Thus $M^{-1}BM^{-1}$ can be approximated by $\widetilde{M}^{-1}\widetilde{B}\widetilde{M}^{-1}$ and therefore so can A^{-1}.

The proofs in Bebendorf and Hackbusch (2003), on which this paper is based, rely on the strong admissibility. In Hackbusch, Khoromskij, and Kriemann (2004), it is shown that using the weak block-admissibility condition instead leads to a smaller constant in the complexity estimates. The extension to the weak admissibility condition in the BLR case is out of the scope of this work. Therefore, we assume that a strong block-admissibility condition is used for computing our theoretical complexity bounds, that we will simply note (Adm_b).

4.2 From Hierarchical to BLR bounds

The existence of $\mathscr{H}$-matrix approximants of the Schur complements of A has been shown in Bebendorf and Hackbusch (2003) and Bebendorf (2007). In this section, we summarize the main ideas of the proof and give the necessary ingredients to extend it to the BLR case. The reader can refer to Bebendorf and Hackbusch (2003), Bebendorf (2005), and Bebendorf (2007) for the details of the proof for hierarchical matrices.

4.2.1 $\mathscr{H}$-admissibility and properties

We define the so-called *sparsity constant*

$$c_{sp} = \max\left(\max_i \#\{I_j; I_i \times I_j \in \mathfrak{S}(\mathscr{I} \times \mathscr{I})\}, \max_j \#\{I_i; I_i \times I_j \in \mathfrak{S}(\mathscr{I} \times \mathscr{I})\}\right), \tag{4.7}$$

where $\#E$ denotes the cardinality of a set E (we will use this notation throughout the rest of this chapter). Thus, the sparsity constant is the maximum number of blocks of a given level in the block cluster tree that are in the same row or column

of the matrix. For example, in Figure 1.16b, c_{sp} is equal to 2, and in Figures 1.15b and 1.17b, it is equal to 4. Under the assumption of a partitioning $\mathfrak{S}(\mathscr{I}\times\mathscr{I})$ defined by a geometrically balanced tree, the sparsity constant can be bound by $\mathscr{O}(1)$ (cf. Grasedyck and Hackbusch (2003), Lemma 4.5). A geometrically balanced tree is a block cluster tree resulting from a partitioning computed as the intersection between the domain Ω (defined below in (4.1)) and a hierarchical cubic domain. For a formal description, see Construction 4.3 in Grasedyck and Hackbusch (2003).

We remind the definition of the $\mathscr{H}$-admissibility condition:

$$\mathfrak{S}(\mathscr{I}\times\mathscr{I}) \text{ is admissible } \Leftrightarrow \forall \sigma\times\tau\in\mathfrak{S}(\mathscr{I}\times\mathscr{I}),\quad (Adm_b) \text{ is satisfied,} \qquad (Adm_{\mathscr{H}})$$
$$\text{or } \min(\#\sigma,\#\tau)\leq c_{min}.$$

where c_{min} is a positive constant. The partitioning associated with the $\mathscr{H}$-admissibility condition ($Adm_{\mathscr{H}}$) can thus roughly be obtained by the following algorithm: for a given initial partition, for each block $\sigma\times\tau$, if (Adm_b) is satisfied, the block is admissible and is added to the final partition; if not, the block is subdivided, until either (Adm_b) is satisfied or the block becomes small enough. This often leads to non-uniform partitionings, such as the example shown on Figure 1.17b.

We note $\mathscr{H}(\mathfrak{S}(\mathscr{I}\times\mathscr{I}),r)$ the set of hierarchical matrices such that r is the maximal rank of the blocks defined by the admissible partition $\mathfrak{S}(\mathscr{I}\times\mathscr{I})$.

In Bebendorf and Hackbusch (2003), Bebendorf (2005), and Bebendorf (2007), the proof that the Schur complements of A possess $\mathscr{H}$-approximants is derived using (4.5).

It is first established that B and M^{-1} possess $\mathscr{H}$-approximants (cf. Bebendorf and Hackbusch (2003), Theorems 3.4 and 4.3). More precisely, they can be approximated with accuracy ε by $\mathscr{H}$-matrices $\widetilde{B}$ and $\widetilde{M}^{-1}$ such that

$$\widetilde{B}\in\mathscr{H}(\mathfrak{S}(\mathscr{I}\times\mathscr{I}),r_G), \tag{4.8}$$
$$\widetilde{M}^{-1}\in\mathscr{H}(\mathfrak{S}(\mathscr{I}\times\mathscr{I}),|\log\varepsilon|^d), \tag{4.9}$$

where $\mathfrak{S}(\mathscr{I}\times\mathscr{I})$ is an $\mathscr{H}$-admissible partition and r_G is the rank resulting from the approximation of the degenerate Green function's kernel. r_G can be shown to be small for many problem classes (Bebendorf and Hackbusch, 2003; Bebendorf, 2005).

Then, the following $\mathscr{H}$-arithmetics theorem is used.

Theorem 4.1 ($\mathscr{H}$-matrix product, Theorem 2.20 in Grasedyck and Hackbusch (2003))**.** *Let H_1 and H_2 be two hierarchical matrices of order n, such that $H_1\in\mathscr{H}(\mathfrak{S}(\mathscr{I}\times\mathscr{I}),r_1)$ and $H_2\in\mathscr{H}(\mathfrak{S}(\mathscr{I}\times\mathscr{I}),r_2)$. Then, their product is also a hierarchical matrix and it holds*

$$H_1H_2\in\mathscr{H}(\mathfrak{S}(\mathscr{I}\times\mathscr{I}),c_{sp}\max(r_1,r_2)\log n).$$

In Theorem 4.1, c_{sp} is the sparsity constant, defined by (4.7).

Then, using the fact that $r_G>|\log\varepsilon|^d$ (Bebendorf and Hackbusch, 2003), and applying (4.5), (4.6), and Theorem 4.1, it is established (cf. Bebendorf and Hackbusch (2003), Theorem 5.4) that

$$\widetilde{A}^{-1}\in\mathscr{H}(\mathfrak{S}(\mathscr{I}\times\mathscr{I}),r_{\mathscr{H}}),\quad \text{with } r_{\mathscr{H}}=c_{sp}^2 r_G\log^2 n. \tag{4.10}$$

Furthermore, if an approximant $\widetilde{A}^{-1}$ exists, then for any $\Phi \subset \mathcal{I}$, an approximant of $A^{-1}_{\Phi,\Phi}$ must also exist, since $A_{\Phi,\Phi}$ is simply the restriction of A to the subdomain X_Φ (Bebendorf, 2007).

Thus, using (4.4), in combination to the fact that the stiffness matrix A can also be approximated by $\widetilde{A} \in \mathcal{H}(\mathfrak{S}(\mathcal{I} \times \mathcal{I}), \mathcal{O}(1))$, the existence of $\widetilde{S}$, $\mathcal{H}$-approximant of any Schur complement S of A, is guaranteed by (4.10) and it is shown (Bebendorf, 2007) that the maximal rank of the blocks of $\widetilde{S}$ is $r_{\mathcal{H}}$, i.e.

$$\widetilde{S} \in \mathcal{H}(\mathfrak{S}(\mathcal{I} \times \mathcal{I}), r_{\mathcal{H}}). \tag{4.11}$$

Finally, it can be shown that the complexity of factorizing an $\mathcal{H}$-matrix of order m and of maximal rank r is (Grasedyck and Hackbusch, 2003; Hackbusch, 1999):

$$\mathcal{C}(m) = \mathcal{O}(c_{sp}^2 r^2 m \log^2 m). \tag{4.12}$$

Equation (4.12) relies on the assumption that the factorization is fully-structured, i.e. the compressed form $\widetilde{A}$ of A is available at no cost.

To conclude, in the $\mathcal{H}$ case, applying (4.12) to the (dense) factorization of $\widetilde{S}$ leads to a cost which is almost linear when $r = \mathcal{O}(1)$ and almost $\mathcal{O}(m^2)$ when $r = \mathcal{O}(\sqrt{m})$. As will be explained in Section 4.4, both cases lead to near-linear complexity of the multifrontal (sparse) factorization (Xia, 2013a).

4.2.2 Why this result is not suitable to compute a complexity bound for BLR

One might think that, since BLR is a specific type of $\mathcal{H}$-matrix, the previous result can be used to derive the complexity of the BLR factorization. However, the bound obtained by simply applying $\mathcal{H}$-matrix theory to BLR is useless, as explained below.

Applying the result on $\mathcal{H}$-matrices to BLR is equivalent to bounding *all the ranks* $k^{\varepsilon}_{i,j}$ by *the same bound* r, the maximal rank. The problem is that this necessarily implies $r = b$, because there will always be some blocks of size b such that $\mathrm{dist}(X_\sigma, X_\tau) = 0$ (i.e., non-admissible blocks, which will be considered full-rank). Thus, the best we can say about a BLR matrix is that it belongs to $\mathcal{H}(\mathfrak{S}(\mathcal{I}), b)$, which is obvious and overly pessimistic.

In addition, with a BLR partitioning, the sparsity constant c_{sp} (defined by (4.7)) is not bounded, as it is equal to $p = m/b$. Thus, (4.12) leads to a factorization complexity bound $\mathcal{O}((m/b)^2 b^2 m \log^2 m) = \mathcal{O}(m^3 \log^2 m)$, which is even worse than the full-rank factorization.

4.2.3 BLR-admissibility and properties

To compute a meaningful complexity bound for BLR, we divide the BLR blocks into two groups: the blocks that satisfy the block-admissibility condition (whose rank r can be bounded by a meaningful bound), and those that do not, which we assume are left in full-rank form. We show that the number of non-admissible blocks in A can be asymptotically negligible, provided an appropriate partitioning

$\mathfrak{S}(\mathscr{I})$. This leads us to introduce the notion of BLR-admissibility of a partition $\mathfrak{S}(\mathscr{I})$, and we establish for such a partition a bound on the maximal rank of the admissible blocks.

In the following, we note $\mathscr{B}_A$ the set of admissible blocks. We also define

$$N_{na} = \max_{\sigma \in \mathfrak{S}(\mathscr{I})} \#\{\tau \in \mathfrak{S}(\mathscr{I}), \sigma \times \tau \notin \mathscr{B}_A\}, \tag{4.13}$$

the maximum number of non-admissible blocks on any row. Note that, because we have assumed for simplicity that the row and column partitioning are the same, N_{na} is also the maximum number of non-admissible blocks on any column.

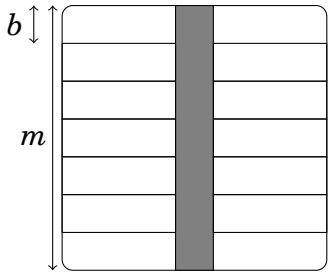
We then recast the $\mathscr{H}$-admissibility of a partition to the BLR uniform blocking. We propose the following BLR-admissibility condition:

$$\mathfrak{S}(\mathscr{I}) \text{ is admissible } \Leftrightarrow N_{na} \leq q, \tag{Adm_{BLR}}$$

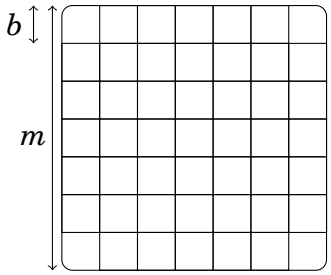
where q is a positive constant. With (Adm_{BLR}), we want the number of blocks (on any row or column) that are not admissible (and thus whose rank is not bounded by r), to be itself bounded by q.

For example, if the least-restrictive strong block-admissibility condition (1.27) is used, (Adm_{BLR}) means that a partition is admissible if for any subdomain, its number of neighbors (i.e. number of subdomains at distance zero) is smaller than q. The BLR-admissibility condition is illustrated in Figure 4.1, where we have assumed that (1.27) is used for simplicity. In Figure 4.1a, the vertical subdomain (in gray) is at distance zero of $\mathscr{O}(m/b)$ blocks and thus N_{na} is not constant. In Figure 4.1b, the maximal number of blocks at distance zero of any block is at most 9 and thus the partition is BLR-admissible for $q \geq 9$. Note that if a general strong admissibility condition (1.25) is used, the same reasoning applies, as in Figure 4.1b, N_{na} only depends on η and d, which are both constant.

We note $\mathscr{BLR}(\mathfrak{S}(\mathscr{I}), r, q)$ the set of BLR matrices such that r is the maximal rank of the admissible blocks defined by the BLR-admissible partition $\mathfrak{S}(\mathscr{I})$.



(a) Example of a non BLR-admissible partition. Here, the gray subdomain has $N_{na} = \mathscr{O}(m/b) \neq \mathscr{O}(1)$ neighbors.



(b) Example of a BLR-admissible partition when $q \geq N_{na} = 9 = \mathscr{O}(1)$, the maximal number of neighbors of any block.

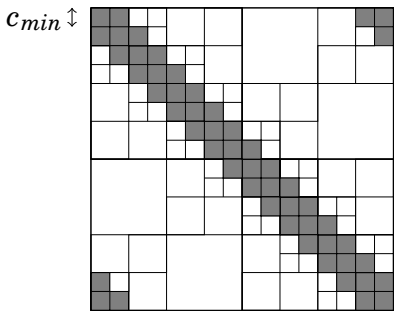
Figure 4.1 – Illustration of the BLR-admissibility condition.

We now prove the following lemma.

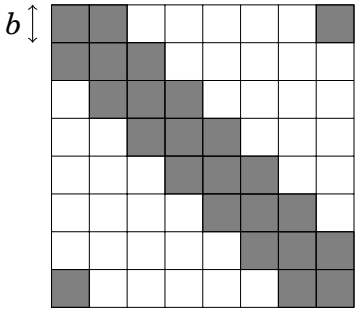
Lemma 4.1. *Let $\mathfrak{S}(\mathcal{I}\times\mathcal{I})$ be a given $\mathcal{H}$-partitioning and let $\mathfrak{S}(\mathcal{I})$ be the corresponding BLR partitioning obtained by refining the $\mathcal{H}$ one. Let us note $N_{na}^{(\mathcal{H})}$ and $N_{na}^{(BLR)}$ the value of N_{na} for the $\mathcal{H}$ and BLR partitionings, respectively. Then: (a) Provided $b \geq c_{min}$, it holds $N_{na}^{(BLR)} \leq N_{na}^{(\mathcal{H})}$; (b) Under the assumption that $\mathfrak{S}(\mathcal{I}\times\mathcal{I})$ is defined by a geometrically balanced block cluster tree, it holds $N_{na}^{(\mathcal{H})} = \mathcal{O}(1)$.*

Proof. (a) We provide Figure 4.2 (where non-admissible blocks are in gray) to illustrate the following proof. The BLR partitioning is simply obtained by refining the $\mathcal{H}$ one. Since non-admissible $\mathcal{H}$-blocks are of size $c_{min} \leq b$, they will not be refined, and thus the BLR refining only adds more admissible blocks to the partitioning. Furthermore, if c_{min} is strictly inferior to b, the non-admissible $\mathcal{H}$-blocks will be merged as a single BLR-block of size b and thus $N_{na}^{(BLR)}$ may in fact be smaller than $N_{na}^{(\mathcal{H})}$. (b) Since all non-admissible blocks necessarily belong to the same level of the block cluster tree (the last one), it holds by definition that $N_{na}^{(\mathcal{H})} \leq c_{sp}$. We conclude with the fact that in the $\mathcal{H}$ case, the sparsity constant is bounded for geometrically balanced block cluster trees (Grasedyck and Hackbusch, 2003). □

As a corollary, we assume in the following that the partition $\mathfrak{S}(\mathcal{I})$ is defined by a geometrically balanced cluster tree and is thus admissible for $q = N_{na} = \mathcal{O}(1)$.



(a) Example of $\mathcal{H}$-partitioning, with $N_{na}^{(\mathcal{H})} = 4 \leq c_{sp} = 6 = \mathcal{O}(1)$. Here, we have assumed the bottom-left and top-right blocks are non-admissible for illustrative purposes.



(b) BLR refining of the $\mathcal{H}$-partitioning 4.2a. $c_{sp} = \mathcal{O}(m/b)$ is not bounded anymore but $N_{na}^{(BLR)} = 3 = \mathcal{O}(1)$ remains constant.

Figure 4.2 – Illustration of Lemma 4.1 (proof of the boundedness of N_{na}).

The next step is to find BLR approximants of B and M^{-1}.

The construction of $\widetilde{B}$ is the same for a BLR or an $\mathcal{H}$-partitioning, and we can thus rely on the work of Bebendorf and Hackbusch (2003).

The main idea behind this construction is to exploit the smoothness property of Green functions. As shown in Bebendorf and Hackbusch (2003) (see their Theorem 3.4), for any admissible block $\sigma\times\tau \in \mathcal{B}_A$, $B_{\sigma\times\tau}$ can be approximated by a low-rank matrix $B^{\varepsilon}_{\sigma\times\tau}$ of numerical rank less than r_G.

Therefore, we construct $\widetilde{B}$ as follows:

$$\forall \sigma\times\tau \in \mathfrak{S}(\mathcal{I})^2,\ \widetilde{B}_{\sigma\times\tau} = \begin{cases} B^{\varepsilon}_{\sigma\times\tau} & \text{if } \sigma\times\tau \in \mathcal{B}_A \\ B_{\sigma\times\tau} & \text{otherwise} \end{cases}, \tag{4.14}$$

which leads to

$$\widetilde{B} \in \mathscr{BLR}(\mathfrak{S}(\mathscr{I}), r_G, N_{na}). \tag{4.15}$$

The construction of $\widetilde{M}^{-1}$ is also very similar to the one in Bebendorf and Hackbusch (2003). The main idea is that the inverse mass matrix asymptotically tends towards a block-diagonal matrix. More precisely, it is shown that, for any block $\sigma \times \tau \in \mathfrak{S}(\mathscr{I})^2$,

$$\|M^{-1}_{\sigma\times\tau}\| \leq \mathscr{O}(\sqrt{\#\sigma\#\tau}\; c^{\sqrt[2d]{\#\sigma\#\tau}\,\mathrm{dist}(X_\sigma, X_\tau)})\|M^{-1}\|,$$

where $c < 1$ (cf. Bebendorf and Hackbusch (2003), Lemma 4.2). Therefore, $\|M^{-1}_{\sigma\times\tau}\|$ tends towards zero when $\#\sigma, \#\tau$ tend towards infinity (which is the case for a non-constant block size b), as long as $\mathrm{dist}(X_\sigma, X_\tau) > 0$, i.e., as long as $\sigma \times \tau \in \mathscr{B}_A$.

Therefore, we construct $\widetilde{M}^{-1} \in \mathscr{BLR}(\mathfrak{S}(\mathscr{I}), 0, N_{na})$ as follows:

$$\forall \sigma \times \tau \in \mathfrak{S}(\mathscr{I})^2,\ \widetilde{M}^{-1}_{\sigma\times\tau} = \begin{cases} 0 & \text{if } \sigma \times \tau \in \mathscr{B}_A \\ M^{-1}_{\sigma\times\tau} & \text{otherwise} \end{cases}, \tag{4.16}$$

which leads to

$$\widetilde{M}^{-1} \in \mathscr{BLR}(\mathfrak{S}(\mathscr{I}), 0, N_{na}). \tag{4.17}$$

(4.15) and (4.17) are the BLR equivalents of (4.8) and (4.9), respectively. It now remains to derive a BLR arithmetic property similar to Theorem 4.1. which takes the form of the following theorem.

Theorem 4.2 (BLR matrix product)**.** *If $A \in \mathscr{BLR}(\mathfrak{S}(\mathscr{I}), r_A, q_A)$ and $B \in \mathscr{BLR}(\mathfrak{S}(\mathscr{I}), r_B, q_B)$ are BLR matrices then their product $P = AB$ is a BLR matrix such that*

$$P \in \mathscr{BLR}(\mathfrak{S}(\mathscr{I}), r_P, q_P),$$

with $r_P = c_{sp}\min(r_A, r_B) + q_A r_B + q_B r_A$ and $q_P = q_A q_B$.

Proof. Let $A \in \mathscr{BLR}(\mathfrak{S}(\mathscr{I}), r_A, q_A)$ and $B \in \mathscr{BLR}(\mathfrak{S}(\mathscr{I}), r_B, q_B)$ be two $c_{sp} \times c_{sp}$ BLR matrices and let $P = AB$ be their product.

For all $(i,j) \in [1; c_{sp}]^2$, we note $A_{i,j}$, $B_{i,j}$, and $P_{i,j}$ the (i,j)-th subblock of matrix A, B, and P, respectively. We also note $\mathrm{Rk}_\varepsilon(X)$ the numerical rank of a matrix X at accuracy ε.

We define

$$q_A(i) = \{k \in [1; c_{sp}]; A_{i,k} \notin \mathscr{B}_A\},$$
$$q_B(j) = \{k \in [1; c_{sp}]; B_{k,j} \notin \mathscr{B}_A\},$$

and thus $q_A = \max_{i\in[1;c_{sp}]} \#q_A(i)$ and $q_B = \max_{j\in[1;c_{sp}]} \#q_B(j)$.

Then, for any $(i,j) \in [1; c_{sp}]^2$, it holds

$$P_{i,j} = \sum_{k=1}^{c_{sp}} A_{i,k}B_{k,j} = \overbrace{\sum_{\substack{A_{i,k}\in\mathscr{B}_A \\ B_{k,j}\in\mathscr{B}_A}} A_{i,k}B_{k,j}}^{S_1} + \overbrace{\sum_{\substack{A_{i,k}\in\mathscr{B}_A \\ B_{k,j}\notin\mathscr{B}_A}} A_{i,k}B_{k,j}}^{S_2} + \overbrace{\sum_{\substack{A_{i,k}\notin\mathscr{B}_A \\ B_{k,j}\in\mathscr{B}_A}} A_{i,k}B_{k,j}}^{S_3} + \overbrace{\sum_{\substack{A_{i,k}\notin\mathscr{B}_A \\ B_{k,j}\notin\mathscr{B}_A}} A_{i,k}B_{k,j}}^{S_4}.$$

We then seek a bound on the rank of each of the four terms S_1 to S_4.

1.

$$\begin{aligned}\mathrm{Rk}_\varepsilon(S_1) &\leq \sum_{\substack{A_{i,k}\in\mathscr{B}_A\\ B_{k,j}\in\mathscr{B}_A}} \mathrm{Rk}_\varepsilon\left(A_{i,k}B_{k,j}\right) \leq \sum_{\substack{A_{i,k}\in\mathscr{B}_A\\ B_{k,j}\in\mathscr{B}_A}} \min(\mathrm{Rk}_\varepsilon\left(A_{i,k}\right),\mathrm{Rk}_\varepsilon\left(B_{k,j}\right))\\ &\leq \sum_{\substack{A_{i,k}\in\mathscr{B}_A\\ B_{k,j}\in\mathscr{B}_A}} \min(r_A,r_B) \leq c_{sp}\min(r_A,r_B).\end{aligned}$$

2.

$$\begin{aligned}\mathrm{Rk}_\varepsilon(S_2) &\leq \sum_{\substack{A_{i,k}\in\mathscr{B}_A\\ B_{k,j}\notin\mathscr{B}_A}} \mathrm{Rk}_\varepsilon\left(A_{i,k}B_{k,j}\right) \leq \sum_{\substack{A_{i,k}\in\mathscr{B}_A\\ B_{k,j}\notin\mathscr{B}_A}} \min(\mathrm{Rk}_\varepsilon\left(A_{i,k}\right),\mathrm{Rk}_\varepsilon\left(B_{k,j}\right))\\ &\leq \sum_{\substack{A_{i,k}\in\mathscr{B}_A\\ B_{k,j}\notin\mathscr{B}_A}} r_A \leq \#q_B(j)\, r_A \leq q_B r_A.\end{aligned}$$

3.

$$\begin{aligned}\mathrm{Rk}_\varepsilon(S_3) &\leq \sum_{\substack{A_{i,k}\notin\mathscr{B}_A\\ B_{k,j}\in\mathscr{B}_A}} \mathrm{Rk}_\varepsilon\left(A_{i,k}B_{k,j}\right) \leq \sum_{\substack{A_{i,k}\notin\mathscr{B}_A\\ B_{k,j}\in\mathscr{B}_A}} \min(\mathrm{Rk}_\varepsilon\left(A_{i,k}\right),\mathrm{Rk}_\varepsilon\left(B_{k,j}\right))\\ &\leq \sum_{\substack{A_{i,k}\notin\mathscr{B}_A\\ B_{k,j}\in\mathscr{B}_A}} r_B \leq \#q_A(i)\, r_B \leq q_A r_B.\end{aligned}$$

4. It holds that

$$\forall i\in[1;c_{sp}],\quad \#\{j\in[1;c_{sp}];q_A(i)\cap q_B(j)\neq\emptyset\}\leq q_Aq_B, \tag{4.18}$$

$$\forall j\in[1;c_{sp}],\quad \#\{i\in[1;c_{sp}];q_A(i)\cap q_B(j)\neq\emptyset\}\leq q_Aq_B. \tag{4.19}$$

Equation (4.18) states that the number of non-admissible blocks on any row of P is bounded by q_Aq_B, while equation (4.19) states that the number of non-admissible blocks on any column of P is also bounded by q_Aq_B. Thus, putting (4.18) and (4.19) together, we have $q_P = q_Aq_B$. Finally, we prove equation (4.18). Let $i\in[1;c_{sp}]$. For all $k\in q_A(i)$, it holds

$$\#\{j\in[1;c_{sp}];B_{k,j}\notin\mathscr{B}_A\}\leq q_B,$$

and since $\#q_A(i)\leq q_A$, we conclude

$$\#\{j\in[1;c_{sp}];P_{i,j}\notin\mathscr{B}_A\}\leq q_Aq_B.$$

The proof of (4.19) works in the same way. In conclusion, $S_4 = 0$ and thus $\mathrm{Rk}_\varepsilon(S_4) = 0$, except for $q_P = q_Aq_B$ blocks whose rank is not bounded. For the rest, their rank is thus bounded by

$$r_P = c_{sp}\min(r_A,r_B) + q_Br_A + q_Ar_B.$$

□

Note that the sparsity constant c_{sp} is not bounded but only appears in the term $c_{sp}\min(r_A, r_B)$ that will disappear when one of r_A or r_B is zero.

Since A^{-1} can be approximated by $M^{-1}BM^{-1}$ (equation (4.5)), applying Theorem 4.2 on (4.15) and (4.17) leads to

$$\widetilde{A}^{-1} \in \mathscr{BLR}(\mathfrak{S}(\mathscr{I}), N_{na}^2 r_G, N_{na}^3), \tag{4.20}$$

and from this approximant of A^{-1} we can derive an approximant of $A_{\Phi,\Phi}^{-1}$ for any $\Phi \subset \mathscr{I}$.

The stiffness matrix A satisfies the following property:

$$\forall \sigma, \tau \in \mathfrak{S}(\mathscr{I}),\ A_{\sigma,\tau} \neq 0 \Leftrightarrow \operatorname{dist}(\sigma, \tau) = 0,$$

and thus $A \in \mathscr{BLR}(\mathfrak{S}(\mathscr{I}), 0, N_{na})$. A fortiori, for any $\Phi, \Psi \subset \mathscr{I}$, we have $A_{\Phi,\Psi} \in \mathscr{BLR}(\mathfrak{S}(\mathscr{I}), 0, N_{na})$.

Therefore applying Theorem 4.2 on (4.4) and (4.20) implies in turn

$$\widetilde{S} \in \mathscr{BLR}(\mathfrak{S}(\mathscr{I}), N_{na}^4 r_G, N_{na}^5). \tag{4.21}$$

As a result, there are at most $N_{na}^5 = \mathscr{O}(1)$ non-admissible blocks that are not considered low-rank candidates and are left full-rank.

The rest are low-rank and their rank is bounded by $N_{na}^4 r_G$. In addition to the bound r_G, which is already quite large (Bebendorf, 2005), the constant N_{na}^4 can be very large. However, our bound is extremely pessimistic. In Section 4.6, we will experimentally validate that, in reality, the ranks are much smaller. Similarly, the bound N_{na}^5 on the number of non-admissible blocks is also very pessimistic.

In conclusion, the ranks are bound by $\mathscr{O}(r_G)$, i.e. the BLR bound only differs from the hierarchical one by a constant.

In the following, the bound $N_{na}^4 r_G$ will be simply referred to as r.

4.3 Complexity of the dense standard BLR factorization

We now compute the complexity of the dense BLR UFSC factorization. The extension to the multifrontal case is the object of Section 4.4, while the other factorization variants are dealt with in Section 4.5. Note that the complexity of the BLR factorization is the same in LU or LDL^T, up to a constant.

Note that, as long as a bound on the ranks holds, similar to the one we have established in Section 4.2, the complexity computations reported in this section hold, and thus, the following results may be applicable to a broader context than the solution of discretized PDEs.

First, we compute the complexity of factorizing a dense frontal matrix of order m. The cost of the main steps Factor, Solve, Compress, Inner and Outer Product necessary to compute the factorization of a matrix of order m are shown in Table 4.1

(third column). This cost depends on the type (full-rank or low-rank) of the block(s) on which the operation is performed (second column). Note that the Inner Product operation can take the form of a product of two full-rank blocks (FR-FR), two low-rank blocks (LR-LR) or a low-rank block and a full-rank one (LR-FR). In the last two cases, the Inner Product yields a low-rank block that is decompressed by means of the Outer Product operation. We note b the block size and $p = m/b$ the number of blocks per row and/or column. We remind that the cost of the classic linear algebra operations on low-rank matrices has been discussed in Section 1.4.1.3.

We assume here that the cost of compressing an admissible block is $\mathcal{O}(b^2 r)$. For example, this is true when the Compress is performed by means of a truncated QR factorization with column pivoting which, as explained in Subsection 4.6.1, will be used in our numerical experiments. This assumption does not hold for Singular Value Decomposition (SVD) for which the Compress cost is $\mathcal{O}(b^3)$; however, this would not change the final complexity of this standard variant, as the complexity of the Compress step would then be of the same order as that of the Solve step.

We can then use (4.21) to compute the cost of the factorization. The boundedness of $N_{na}^5 = \mathcal{O}(1)$ ensures that only a constant number of blocks on each row are full-rank. From that we derive the fourth column of Table 4.1, which counts the number of blocks on which the step is performed.

step	type	cost	number	$\mathscr{C}_{step}(b,p)$	$\mathscr{C}_{step}(m,x)$
Factor	FR	$\mathcal{O}(b^3)$	$\mathcal{O}(p)$	$\mathcal{O}(pb^3)$	$\mathcal{O}(m^{1+2x})$
Solve	FR-FR	$\mathcal{O}(b^3)$	$\mathcal{O}(p^2)$	$\mathcal{O}(p^2b^3)$	$\mathcal{O}(m^{2+x})$
Compress	LR	$\mathcal{O}(b^2r)$	$\mathcal{O}(p^2)$	$\mathcal{O}(p^2b^2r)$	$\mathcal{O}(m^2r)$
Inner Product	LR-LR	$\mathcal{O}(br^2)$	$\mathcal{O}(p^3)$	$\mathcal{O}(p^3br^2)$	$\mathcal{O}(m^{3-2x}r^2)$
	LR-FR	$\mathcal{O}(b^2r)$	$\mathcal{O}(p^2)$	$\mathcal{O}(p^2b^2r)$	$\mathcal{O}(m^2r)$
	FR-FR	$\mathcal{O}(b^3)$	$\mathcal{O}(p)$	$\mathcal{O}(pb^3)$	$\mathcal{O}(m^{1+2x})$
Outer Product	LR	$\mathcal{O}(b^2r)$	$\mathcal{O}(p^3)$	$\mathcal{O}(p^3b^2r)$	$\mathcal{O}(m^{3-x}r)$

Table 4.1 – Main operations for the BLR (standard UFSC variant) factorization of a dense matrix of order m, with blocks of size b, and low-rank blocks of rank at most r. We note $p = m/b$. *type*: type of the block(s) on which the operation is performed. *cost*: cost of performing the operation once. *number*: number of times the operation is performed. $\mathscr{C}_{step}(b,p)$: obtained by multiplying the *cost* and *number* columns (equation (4.22)). $\mathscr{C}_{step}(m,x)$: obtained with the assumption that $b = \mathcal{O}(m^x)$ (and thus $p = \mathcal{O}(m^{1-x})$), for some $x \in [0,1]$.

The BLR factorization cost of each step is then equal to

$$\mathscr{C}_{step}(b,p) = cost_{step} * number_{step}, \tag{4.22}$$

and is reported in the fifth column of Table 4.1. Then, we assume the block size b is of order $\mathcal{O}(m^x)$, where x is a real value in $[0,1]$, and thus the number of blocks p per row and/or column is of order $\mathcal{O}(m^{1-x})$. Then by substituting b and p by their value, we compute $\mathscr{C}_{step}(m,x)$ in the last column.

We can then compute the total flop complexity of the dense BLR factorization as the sum of the cost of all steps:

$$\mathscr{C}(m,x) = \mathcal{O}(rm^{3-x} + m^{2+x}). \tag{4.23}$$

Similarly, the factor size complexity of a dense BLR matrix can be computed as

$$\mathcal{O}(N_{LR} * br + N_{FR} * b^2) = \mathcal{O}(p^2 br + N_{na}^5 pb^2) = \mathcal{O}(p^2 br + pb^2), \tag{4.24}$$

where $N_{LR} = \mathcal{O}(p^2)$ and $N_{FR} = \mathcal{O}(p)$ are the number of low-rank and full-rank blocks in the matrix, respectively. Thus, the factor size complexity is:

$$\mathscr{M}(m,x) = \mathcal{O}(rm^{2-x} + m^{1+x}). \tag{4.25}$$

It then remains to compute the optimal x^* which minimizes the complexity. We consider a general rank bound $r = \mathcal{O}(m^\alpha)$, with $\alpha \in [0,1]$. Equations (4.23) and (4.25) become

$$\mathscr{C}(m,x) = \mathcal{O}(m^{3+\alpha-x} + m^{2+x}), \tag{4.26}$$
$$\mathscr{M}(m,x) = \mathcal{O}(m^{2+\alpha-x} + m^{1+x}), \tag{4.27}$$

respectively. Then, the optimal x^* is given by

$$x^* = \frac{1+\alpha}{2}, \tag{4.28}$$

which leads to optimal complexities

$$\mathscr{C}(m) = \mathscr{C}(m,x^*) = \mathcal{O}(m^{2.5+\alpha/2}) = \mathcal{O}(m^{2.5}\sqrt{r}), \tag{4.29}$$
$$\mathscr{M}(m) = \mathscr{M}(m,x^*) = \mathcal{O}(m^{1.5+\alpha/2}) = \mathcal{O}(m^{1.5}\sqrt{r}). \tag{4.30}$$

It is remarkable that the value of x^* is the same for both the flop and factor size complexities, i.e. that both complexities are minimized by the same x. This was not guaranteed, and is a desirable property as we do not need to choose which complexity to minimize at the expense of the other.

In particular, the case $r = \mathcal{O}(1)$ leads to complexities $\mathcal{O}(m^{2.5})$ for flops and $\mathcal{O}(m^{1.5})$ for factor size, while the case $r = \mathcal{O}(\sqrt{m})$ leads to $\mathcal{O}(m^{2.75})$ for flops and $\mathcal{O}(m^{1.75})$ for factor size. The link between dense and sparse rank bounds will be made in Section 4.4.2.

Note that the fully-structured BLR factorization (when A is available under compressed form at no cost, i.e. when the Compress step does not need to be performed) has the same complexity as the non fully-structured factorization, since the Compress is asymptotically negligible with respect to the Solve step. This is not the case in the hierarchical case, where the construction of the compressed matrix, whose cost is $\mathcal{O}(m^2 r)$ (Grasedyck and Hackbusch, 2003), becomes the bottleneck when it has to be performed.

4.4 From dense to sparse BLR complexity

We first describe in Subsection 4.4.1 how the BLR clustering is computed in the context of the multifrontal method and the relation between frontal matrices and BLR approximants of the Schur complements of the stiffness matrix A. Then, we

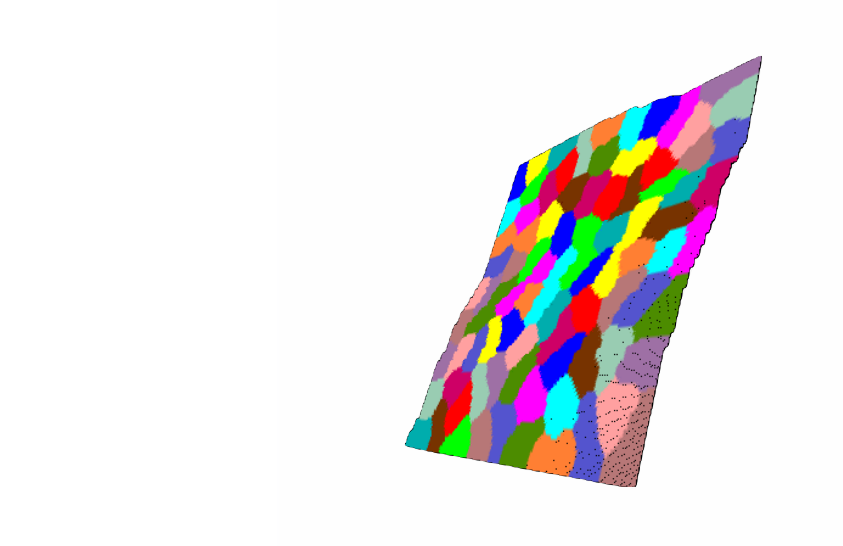


Figure 4.3 – BLR clustering of the root separator of a 128^3 Poisson problem.

extend in Subsection 4.4.2 the computation of the factorization complexity to the sparse multifrontal case.

4.4.1 BLR clustering and BLR approximants of frontal matrices

In a multifrontal context, the computation of the BLR clustering and therefore the resulting complexity of the BLR factorization strongly depend on how the assembly tree is built, as explained in Section 1.4.3.1.

Let us assume that the assembly tree is built by means of a nested dissection (George, 1973). Because of the bottom-up traversal of the assembly tree, the rows and columns of the fully-summed variables of a frontal matrix associated with a separator S thus belong to the Schur complement of the variables of the two domain subgraphs separated by S. From this and the existence of low-rank approximants of the Schur complements of the stiffness matrix A, which was established in Section 4.2, it results that the fronts can be approximated by BLR matrices. This remains true for general orderings other than nested dissection.

The admissibility condition ($\mathscr{H}$ or BLR) requires geometric information to compute the diameter and distances. To remain in a purely algebraic context, as explained in Section 1.4.3.1, we use the adjacency graph of the matrix A instead. The BLR clustering is computed with a k-way partitioning of each separator subgraph. This may lead to suboptimal complexity if the admissibility condition is not respected. An example of BLR clustering obtained with this method is shown in Figure 4.3. In particular, we note that the clustering respects the BLR-admissibility condition (Adm_{BLR}).

The impact of algebraic orderings (as opposed to geometric) will be analyzed in Section 4.6.

4.4.2 Computation of the sparse BLR multifrontal complexity

The BLR multifrontal complexity in the sparse case can be directly derived from the dense complexity.

The flop and factor size complexities $\mathscr{C}_{MF}(N)$ and $\mathscr{M}_{MF}(N)$ of the BLR multifrontal factorization are reported in Table 4.2. We assume a nested dissection ordering (George, 1973) (with separators in cross shape).

At each level ℓ of the separators tree, we need to factorize $(2^d)^\ell$ fronts of order $\mathscr{O}((\frac{N}{2^\ell})^{d-1})$, for ℓ ranging from 0 to $L = \log_2(N)$. Therefore, the flop complexity $\mathscr{C}_{MF}(N)$ to factorize a sparse matrix of order N^d is

$$\mathscr{C}_{MF}(N) = \sum_{\ell=0}^{L} \mathscr{C}_\ell(N) = \sum_{\ell=0}^{L} (2^d)^\ell \mathscr{C}((\frac{N}{2^\ell})^{d-1}), \tag{4.31}$$

where $\mathscr{C}_\ell(N)$ is the cost of factorizing all the fronts on the ℓ-th level, i.e. $\mathscr{C}_\ell(N) = (2^d)^\ell \mathscr{C}(m_\ell)$ with $m_\ell = (\frac{N}{2^\ell})^{d-1}$. Using the dense complexity equation (4.29), we compute and report the value of $\mathscr{C}_\ell(N)$ in Table 4.2 (second column). $\mathscr{C}_\ell(N)$ takes the form $\mathscr{O}(2^{\beta\ell} N^\gamma)$ where β and γ depend on the dimension d and the rank bound α parameter. Thus, $\mathscr{C}_{MF}(N)$ is a geometric series of common ratio $Q = 2^{\beta\ell}$. Its computation depends on the sign of β:

- If $\beta < 0$, then $Q < 1$ and $\mathscr{C}_{MF}(N) = \mathscr{O}(\frac{1-Q^{L+1}}{1-Q} N^\gamma) = \mathscr{O}(N^\gamma)$.
- If $\beta = 0$, then $Q = 1$ and $\mathscr{C}_{MF}(N) = \mathscr{O}(LN^\gamma) = \mathscr{O}(N^\gamma \log N)$.
- If $\beta > 0$, then $Q > 1$ and $\mathscr{C}_{MF}(N) = \mathscr{O}(\frac{Q^{L+1}-1}{Q-1} N^\gamma) = \mathscr{O}(2^{\beta \log N} N^\gamma) = \mathscr{O}(N^{\beta+\gamma})$.

The corresponding value of $\mathscr{C}_{MF}(N)$ is reported in the third column of Table 4.2. Note that, to obtain an expression as a function of r rather than α, the relation $r = \mathscr{O}(m^\alpha) = \mathscr{O}(N^{(d-1)\alpha})$ is used, i.e. N^α is equal to $\mathscr{O}(r)$ in 2D and $\mathscr{O}(\sqrt{r})$ in 3D.

Using (4.30), we similarly compute the factor size complexity:

$$\mathscr{M}_{MF}(N) = \sum_{\ell=0}^{L} \mathscr{M}_\ell(N) = \sum_{\ell=0}^{L} (2^d)^\ell \mathscr{M}((\frac{N}{2^\ell})^{d-1}), \tag{4.32}$$

and report the results in Table 4.2.

4.5 The other BLR variants and their complexity

In this section, we analyze the other BLR factorization variants, and prove they achieve lower complexity than the standard UFSC variant. For the sake of clarity, we refer to the different variants using their left-looking name. Of course, the right-looking variants have the same complexity as their left-looking counterpart.

Let us start by the LUAR algorithm, which has been described in Section 2.6 (cf. Algorithm 2.10). To compute its complexity, we first need to compute the cost of the Recompress; we thus need to bound the rank of the accumulators X_S and Y_S. If the Compress is done with an SVD or RRQR operation, then each accumulated

d	$\mathscr{C}_\ell(N)$	$\mathscr{C}_{MF}(N)$	
2D	$\mathcal{O}(2^{-(\ell+\alpha)/2}N^{2.5+\alpha/2})$	$\mathcal{O}(N^{2.5+\alpha/2}) = \mathcal{O}(N^{2.5}\sqrt{r})$	
3D	$\mathcal{O}(2^{-(2+\alpha)\ell}N^{5+\alpha})$	$\mathcal{O}(N^{5+\alpha}) = \mathcal{O}(N^{5}\sqrt{r})$	
d	$\mathscr{M}_\ell(N)$	$\mathscr{M}_{MF}(N)$	
		$\alpha = 0$	$\alpha > 0$
2D	$\mathcal{O}(2^{\ell(1-\alpha)/2}N^{1.5+\alpha/2})$	$\mathcal{O}(N^2)$	$\mathcal{O}(N^2)$
3D	$\mathcal{O}(2^{-\alpha\ell}N^{3+\alpha})$	$\mathcal{O}(N^3 \log N)$	$\mathcal{O}(N^{3+\alpha}) = \mathcal{O}(N^3\sqrt{r})$

Table 4.2 – Flop and factor size complexity of the BLR (standard UFSC variant) multifrontal factorization of a sparse matrix of order N^d. d: dimension. $\mathscr{C}_\ell(N)/\mathscr{M}_\ell(N)$: flop/factor size complexity at level ℓ in the separator tree, computed using the dense complexity equations (4.29) and (4.30). $\mathscr{C}_{MF}(N)/\mathscr{M}_{MF}(N)$: total multifrontal flop/factor size complexity, computed using equations (4.31) and (4.32).

update in X_S and Y_S is an orthonormal basis of an admissible block. Thus, the accumulator is a basis of a superblock which is itself admissible (because the union of admissible blocks remains admissible) and thus the rank of the accumulator is bounded by r.

Then, by recompressing the accumulators, we only need to do one Outer Product (of size r) per block, instead of $\mathcal{O}(p)$ (one for each update matrix). This leads to a substantial theoretical improvement, as it lowers the cost of the Outer Product from $\mathcal{O}(b^2r) * \mathcal{O}(p^3) = \mathcal{O}(p^3b^2r)$ to $\mathcal{O}(b^2r) * \mathcal{O}(p^2) = \mathcal{O}(p^2b^2r)$ (cf. Table 4.3, column $\mathscr{C}_{step}(b,p)$), even though the recompressions of the accumulated updates (Recompress operation) introduce an overhead cost, equal to $\mathcal{O}(pbr^2) * \mathcal{O}(p^2) = \mathcal{O}(p^3br^2)$.

Next, let us analyze the UFCS variant, described in Algorithm 2.5. In this variant, we perform the Compress before the Solve. The Solve step can thus be performed on low-rank blocks (as shown at line 11) and its cost is thus reduced to $\mathcal{O}(p^2b^2r + pb^3)$ (as reported in Table 4.3, column $\mathscr{C}_{step}(b,p)$). The UCFS (Algorithm 2.6) variant simply changes the way numerical pivoting is handled and therefore has the same complexity as the UFCS one.

Finally, with the CUFS variant (Algorithm 2.8), we can suppress the Outer Product step, whose cost becomes zero. Furthermore, in a fully-structured context, the cost of the Compress may also be made negligible (e.g. if the original entries of the matrix can be compressed with a fast matrix-vector operation).

The computation of the complexity of the BLR variants is very similar to that of the standard version, computed in Section 4.3. We provide the equivalent of Tables 4.1 and 4.2 for the BLR variants in Tables 4.3 and 4.4, respectively.

In Table 4.3, we report the cost of each step of the factorization. These costs have been explained in the previous subsection. For each variant, the steps whose cost has changed with respect to the previous variant (in the order: UFSC, UFSC+LUAR, UFCS+LUAR, CUFS) have been grayed out.

By summing the cost of all steps, we obtain the flop complexity of the dense factorization. In the UFSC+LUAR variant, it is given by

$$\mathscr{C}(m,x) = \mathcal{O}(r^2 m^{3-2x} + m^{2+x}). \tag{4.33}$$

UFSC+LUAR variant					
step	type	cost	number	$\mathscr{C}_{step}(b,p)$	$\mathscr{C}_{step}(m,x)$
Factor	FR	$\mathcal{O}(b^3)$	$\mathcal{O}(p)$	$\mathcal{O}(pb^3)$	$\mathcal{O}(m^{1+2x})$
Solve	FR-FR	$\mathcal{O}(b^3)$	$\mathcal{O}(p^2)$	$\mathcal{O}(p^2b^3)$	$\mathcal{O}(m^{2+x})$
Compress	LR	$\mathcal{O}(b^2r)$	$\mathcal{O}(p^2)$	$\mathcal{O}(p^2b^2r)$	$\mathcal{O}(m^2r)$
Inner Product	LR-LR	$\mathcal{O}(br^2)$	$\mathcal{O}(p^3)$	$\mathcal{O}(p^3br^2)$	$\mathcal{O}(m^{3-2x}r^2)$
	LR-FR	$\mathcal{O}(b^2r)$	$\mathcal{O}(p^2)$	$\mathcal{O}(p^2b^2r)$	$\mathcal{O}(m^2r)$
	FR-FR	$\mathcal{O}(b^3)$	$\mathcal{O}(p)$	$\mathcal{O}(pb^3)$	$\mathcal{O}(m^{1+2x})$
Recompress	LR	$\mathcal{O}(bpr^2)$	$\mathcal{O}(p^2)$	$\mathcal{O}(p^3br^2)$	$\mathcal{O}(m^{3-2x}r^2)$
Outer Product	LR	$\mathcal{O}(b^2r)$	$\mathcal{O}(p^2)$	$\mathcal{O}(p^2b^2r)$	$\mathcal{O}(m^2r)$

UFCS/UCFS+LUAR variant					
step	type	cost	number	$\mathscr{C}_{step}(b,p)$	$\mathscr{C}_{step}(m,x)$
Factor	FR	$\mathcal{O}(b^3)$	$\mathcal{O}(p)$	$\mathcal{O}(pb^3)$	$\mathcal{O}(m^{1+2x})$
Solve	FR-FR	$\mathcal{O}(b^3)$	$\mathcal{O}(p)$	$\mathcal{O}(pb^3)$	$\mathcal{O}(m^{1+2x})$
	LR-FR	$\mathcal{O}(b^2r)$	$\mathcal{O}(p^2)$	$\mathcal{O}(p^2b^2r)$	$\mathcal{O}(m^2r)$
Compress	LR	$\mathcal{O}(b^2r)$	$\mathcal{O}(p^2)$	$\mathcal{O}(p^2b^2r)$	$\mathcal{O}(m^2r)$
Inner Product	LR-LR	$\mathcal{O}(br^2)$	$\mathcal{O}(p^3)$	$\mathcal{O}(p^3br^2)$	$\mathcal{O}(m^{3-2x}r^2)$
	LR-FR	$\mathcal{O}(b^2r)$	$\mathcal{O}(p^2)$	$\mathcal{O}(p^2b^2r)$	$\mathcal{O}(m^2r)$
	FR-FR	$\mathcal{O}(b^3)$	$\mathcal{O}(p)$	$\mathcal{O}(pb^3)$	$\mathcal{O}(m^{1+2x})$
Recompress	LR	$\mathcal{O}(bpr^2)$	$\mathcal{O}(p^2)$	$\mathcal{O}(p^3br^2)$	$\mathcal{O}(m^{3-2x}r^2)$
Outer Product	LR	$\mathcal{O}(b^2r)$	$\mathcal{O}(p^2)$	$\mathcal{O}(p^2b^2r)$	$\mathcal{O}(m^2r)$

CUFS variant (fully-structured)					
step	type	cost	number	$\mathscr{C}_{step}(b,p)$	$\mathscr{C}_{step}(m,x)$
Factor	FR	$\mathcal{O}(b^3)$	$\mathcal{O}(p)$	$\mathcal{O}(pb^3)$	$\mathcal{O}(m^{1+2x})$
Solve	FR-FR	$\mathcal{O}(b^3)$	$\mathcal{O}(p)$	$\mathcal{O}(pb^3)$	$\mathcal{O}(m^{1+2x})$
	LR-FR	$\mathcal{O}(b^2r)$	$\mathcal{O}(p^2)$	$\mathcal{O}(p^2b^2r)$	$\mathcal{O}(m^2r)$
Compress	LR	—	—	—	—
Inner Product	LR-LR	$\mathcal{O}(br^2)$	$\mathcal{O}(p^3)$	$\mathcal{O}(p^3br^2)$	$\mathcal{O}(m^{3-2x}r^2)$
	LR-FR	$\mathcal{O}(b^2r)$	$\mathcal{O}(p^2)$	$\mathcal{O}(p^2b^2r)$	$\mathcal{O}(m^2r)$
	FR-FR	$\mathcal{O}(b^3)$	$\mathcal{O}(p)$	$\mathcal{O}(pb^3)$	$\mathcal{O}(m^{1+2x})$
Recompress	LR	$\mathcal{O}(bpr^2)$	$\mathcal{O}(p^2)$	$\mathcal{O}(p^3br^2)$	$\mathcal{O}(m^{3-2x}r^2)$
Outer Product	LR	—	—	—	—

Table 4.3 – Main operations for the factorization of a dense matrix of order m, with blocks of size b, and low-rank blocks of rank at most r. We note $p = m/b$. *type*: type of the block(s) on which the operation is performed. *cost*: cost of performing the operation once. *number*: number of times the operation is performed. $\mathscr{C}_{step}(b,p)$: obtained by multiplying the *cost* and *number* columns (equation (4.22)). $\mathscr{C}_{step}(m,x)$: obtained with the assumption that $b = \mathcal{O}(m^x)$ (and thus $p = \mathcal{O}(m^{1-x})$), for some $x \in [0,1]$. Note that the non fully-structured CUFS variant can also be considered, where the Compress step is still needed; its asymptotic complexity is the same as the fully-structured CUFS variant.

Compared to (4.23), the low-rank term of the complexity has thus been reduced from $\mathcal{O}(rm^{3-x})$ to $\mathcal{O}(r^2m^{3-2x})$ thanks to the recompression of the accumulated updates. The full-rank term $\mathcal{O}(m^{2+x})$ remains the same. By recomputing the value of x^*, we achieve flop complexity gains: For $r=\mathcal{O}(m^\alpha)$, $\mathscr{C}(m)$ becomes

$$\mathscr{C}(m)=\mathcal{O}(m^{2+(2\alpha+1)/3})=\mathcal{O}(m^{7/3}r^{2/3}), \tag{4.34}$$

which yields in particular $\mathcal{O}(m^{7/3})$ for $r=\mathcal{O}(1)$ and $\mathcal{O}(m^{8/3})$ for $r=\mathcal{O}(\sqrt{m})$.

In the same way, the flop complexity for the dense factorization with the UFCS+LUAR variant is given by

$$\mathscr{C}(m,x)=\mathcal{O}(r^2m^{3-2x}+m^{1+2x}). \tag{4.35}$$

This time, the full-rank term has been reduced from $\mathcal{O}(m^{2+x})$ to $\mathcal{O}(m^{1+2x})$. By recomputing x^*, we achieve further flop complexity gains:

$$\mathscr{C}(m)=\mathcal{O}(m^{2+\alpha})=\mathcal{O}(m^2r), \tag{4.36}$$

which yields in particular $\mathcal{O}(m^2)$ for $r=\mathcal{O}(1)$ and $\mathcal{O}(m^{2.5})$ for $r=\mathcal{O}(\sqrt{m})$.

Note that for the UFCS+LUAR variant, the Compress step has become asymptotically dominant and thus the assumption that its cost is $\mathcal{O}(b^2r)$ is now necessary to obtain the complexity reported in equation (4.35).

Finally, the CUFS variant achieves the same asymptotic complexity as the UFCS+LUAR one, because even though the Compress and Outer Product steps have been suppressed, the Solve step remains dominant in both UFCS and CUFS.

Note that the factor size complexity is not affected by the BLR variant used.

The sparse flop complexities are derived from the dense ones in the same way as they are for the standard UFSC variant. The results are reported in Table 4.4.

UFSC+LUAR		
d	$\mathscr{C}_\ell(N)$	$\mathscr{C}_{MF}(N)$
2D	$\mathcal{O}(2^{-(1+2\alpha)\ell/3}N^{2+(2\alpha+1)/3})$	$\mathcal{O}(N^{2+(2\alpha+1)/3})=\mathcal{O}(N^{7/3}r^{2/3})$
3D	$\mathcal{O}(2^{-(5+4\alpha)\ell/3}N^{4+(4\alpha+2)/3})$	$\mathcal{O}(N^{4+(4\alpha+2)/3})=\mathcal{O}(N^{14/3}r^{2/3})$

UFCS+LUAR/UCFS+LUAR/CUFS			
d	$\mathscr{C}_\ell(N)$	$\mathscr{C}_{MF}(N)$	
		$\alpha=0$	$\alpha>0$
2D	$\mathcal{O}(2^{-\alpha\ell}N^{2+\alpha})$	$\mathcal{O}(N^2\log N)$	$\mathcal{O}(N^{2+\alpha})=\mathcal{O}(N^2r)$
3D	$\mathcal{O}(2^{-(1+2\alpha)\ell}N^{4+2\alpha})$	$\mathcal{O}(N^{4+2\alpha})=\mathcal{O}(N^4r)$	

Table 4.4 – Flop and factor size complexity of the BLR multifrontal factorization of a sparse matrix of order N^d. d: dimension. $\mathscr{C}_\ell(N)$: flop complexity at level ℓ in the separator tree, computed using the dense complexity equations (4.34) and (4.36) for the UFSC+LUAR and UFCS+LUAR variants, respectively. $\mathscr{C}_{MF}(N)$: total multifrontal flop complexity, computed using equation (4.31).

A summary of the sparse complexities for all BLR variants, as well as the full-rank and $\mathscr{H}$ complexities, is given in Table 4.5. The complexity formulas sometimes have a max term in their expression because, depending on the value of r with respect to n, the complexity formula is a geometric series with common ratio >1

or < 1. Therefore, to give more readable expressions, which can be easily compared to the experimental ones obtained in the next section, we provide in Table 4.6 the BLR complexity formulas in the case $r = \mathcal{O}(1)$ and $r = \mathcal{O}(\sqrt{m})$, respectively.

	operations		factor size	
	2D	3D	2D	3D
FR	$\mathcal{O}(n^{1.5})$	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^{4/3})$
BLR UFSC	$\mathcal{O}(n^{1.25}\sqrt{r})$	$\mathcal{O}(n^{5/3}\sqrt{r})$	$\mathcal{O}(n)$	$\mathcal{O}(n \max(\sqrt{r}, \log n))$
BLR UFSC+LUAR	$\mathcal{O}(n^{7/6} r^{2/3})$	$\mathcal{O}(n^{14/9} r^{2/3})$	$\mathcal{O}(n)$	$\mathcal{O}(n \max(\sqrt{r}, \log n))$
BLR UFCS+LUAR	$\mathcal{O}(n \max(r, \log n))$	$\mathcal{O}(n^{4/3} r)$	$\mathcal{O}(n)$	$\mathcal{O}(n \max(\sqrt{r}, \log n))$
$\mathcal{H}$	$\mathcal{O}(n \max(r, \log n))$	$\mathcal{O}(n^{4/3} r)$	$\mathcal{O}(n)$	$\mathcal{O}(\max(n, n^{2/3} r))$
$\mathcal{H}$ (fully-structured)	$\mathcal{O}(\max(n, \sqrt{n} r^2))$	$\mathcal{O}(\max(n, n^{2/3} r^2))$	$\mathcal{O}(n)$	$\mathcal{O}(\max(n, n^{2/3} r))$

Table 4.5 – Flop and factor size complexities of the BLR multifrontal factorization of a system of n unknowns, with maximal rank r and an optimal choice of b. In the fully-structured case, the original matrix is assumed to be already compressed. In the non fully-structured case, the cost of compressing the original matrix is included. We remind that the complexity of the BLR fully-structured factorization is the same as that of the non fully-structured one, i.e. UFCS and CUFS have the same asymptotic complexity.

	operations		factor size	
	2D	3D	2D	3D
$r = \mathcal{O}(1)$				
BLR UFSC	$\mathcal{O}(n^{1.25})$	$\mathcal{O}(n^{1.67})$	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$
BLR UFSC+LUAR	$\mathcal{O}(n^{1.17})$	$\mathcal{O}(n^{1.56})$	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$
BLR UFCS+LUAR	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^{1.33})$	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$
$r = \mathcal{O}(\sqrt{m})$				
BLR UFSC	$\mathcal{O}(n^{1.5})$	$\mathcal{O}(n^{1.83})$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^{1.17})$
BLR UFSC+LUAR	$\mathcal{O}(n^{1.5})$	$\mathcal{O}(n^{1.78})$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^{1.17})$
BLR UFCS+LUAR	$\mathcal{O}(n^{1.5})$	$\mathcal{O}(n^{1.67})$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^{1.17})$

Table 4.6 – Flop and factor size complexities of the BLR multifrontal factorization of a system of n unknowns, considering the case $r = \mathcal{O}(1)$ and $r = \mathcal{O}(\sqrt{m}) = \mathcal{O}(N^{d-1})$.

4.6 Numerical experiments

In this section we compare the experimental complexity of the full-rank solver with each of the BLR variants previously presented. We also discuss the choice of two parameters, the low-rank threshold ε and the block size b, and their impact on the complexity.

4.6.1 Description of the experimental setting

All the experiments in this chapter were performed on **brunch** (see description in Section 1.5.3).

To compute our complexity estimates, we use least-squares estimation to compute the coefficients $\{\beta_i\}_i$ of a regression function f such that $X_{fit} = f(N, \{\beta_i\}_i)$ fits the observed data X_{obs}. We use the following regression function:

$$X_{fit} = e^{\beta_1^*} N^{\beta_2^*} \text{ with } \beta_1^*, \beta_2^* = \underset{\beta_1,\beta_2}{\operatorname{argmin}} \| \log X_{obs} - \beta_1 - \beta_2 \log N \|^2. \tag{4.37}$$

We provide the experimental complexities for two different problems: the Poisson problem and the Helmholtz problem, described in 1.5.2.1.

For the Poisson problem, the rank bound is $\mathcal{O}(1)$ (Bebendorf and Hackbusch, 2003). For the Helmholtz problem, although there is no rigorous proof of it, the rank bound is assumed to be $\mathcal{O}(\sqrt{m})$ in the related literature (Xia, 2013a; Wang, Li, Rouet, Xia, and De Hoop, 2016; Engquist and Ying, 2011). Thus, we will use the Poisson and Helmholtz problems to experimentally validate the complexities computed in Table 4.6.

For Poisson, we will use a low-rank threshold ε varying from 10^{-14} to 10^{-2} to better understand its influence on the complexity, with no particular application in mind. For Helmholtz, we will use a low-rank threshold ε varying from 10^{-5} to 10^{-3} because we know these are the values for which the result is meaningful for the application, as explained in Section 7.1.

For both Poisson and Helmholtz, in all the following experiments, the backward error is in good agreement with the low-rank threshold used.

Note that both the Poisson and Helmholtz problems were discretized using the finite-difference method rather than the finite-elements one, but this is acceptable as both methods are equivalent on equispaced meshes (Peiró and Sherwin, 2005).

4.6.2 Flop complexity of each BLR variant

In Figures 4.4 and 4.5, we compare the flop complexity of the full-rank solver with each of the BLR variants previously presented (UFSC, UFSC+LUAR, UFCS+LUAR) for the Poisson problem, and the Helmholtz problem, respectively. Note that in these experiments, the LUAR variant only exploits weight information to recompress the accumulators. In Section 4.6.3, we report experiments with geometry recompression, as well as with the CUFS variant.

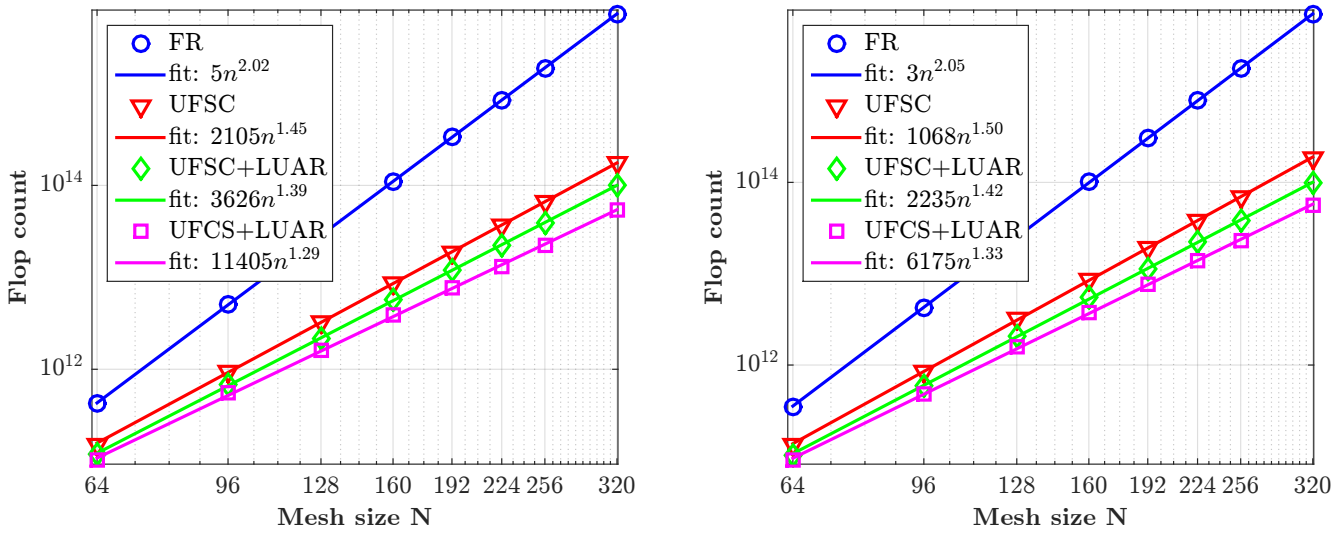
The results show that each new variant improves the complexity. Note that we obtain the well-known quadratic complexity of the full-rank version. Results with both geometric nested dissection (Figures 4.4a and 4.5a) and with a purely algebraic ordering computed by METIS (Figures 4.4b and 4.5b) are also reported.

We first analyze the results obtained with geometric nested dissection and compare them with our theoretical results. For Poisson, the standard BLR (UFSC) version achieves a complexity $\mathcal{O}(n^{1.45})$. Moreover, the constant in the big $\mathcal{O}()$ is equal to 2105, which is quite reasonable, and leads to a substantial improvement of the number of flops performed with respect to the full-rank version. This confirms that the theoretical rank bounds ($N_{na}^4 r_G$) are very pessimistic, as the experimental

constants are in fact much smaller. Further compression in the UFSC+LUAR variant lowers the complexity to $\mathcal{O}(n^{1.39})$, while the UFCS+LUAR reaches the lowest complexity of the variants, $\mathcal{O}(n^{1.29})$. Although the constants increase with the new variants, they also remain relatively small and they effectively reduce the number of operations with respect to the standard variant, even for the smaller mesh sizes. The same trend is observed for Helmholtz, with complexities $\mathcal{O}(n^{1.85})$ for UFSC, $\mathcal{O}(n^{1.79})$ for UFSC+LUAR, and finally $\mathcal{O}(n^{1.74})$ for UFCS+LUAR. Thus, the numerical results are in good agreement with the theoretical bounds reported in Table 4.6.

We also analyze the influence of the ordering on the complexity. We observe that even though the METIS ordering slightly degrades the complexity, results remain close to the geometric nested dissection ordering and still in good agreement with the theoretical bounds. This is a very important property of the BLR factorization as it allows us to remain in a purely algebraic (black box) framework, an essential property for a general purpose solver.

For the remaining experiments, we use the METIS ordering.



(a) Nested Dissection ordering (geometric)

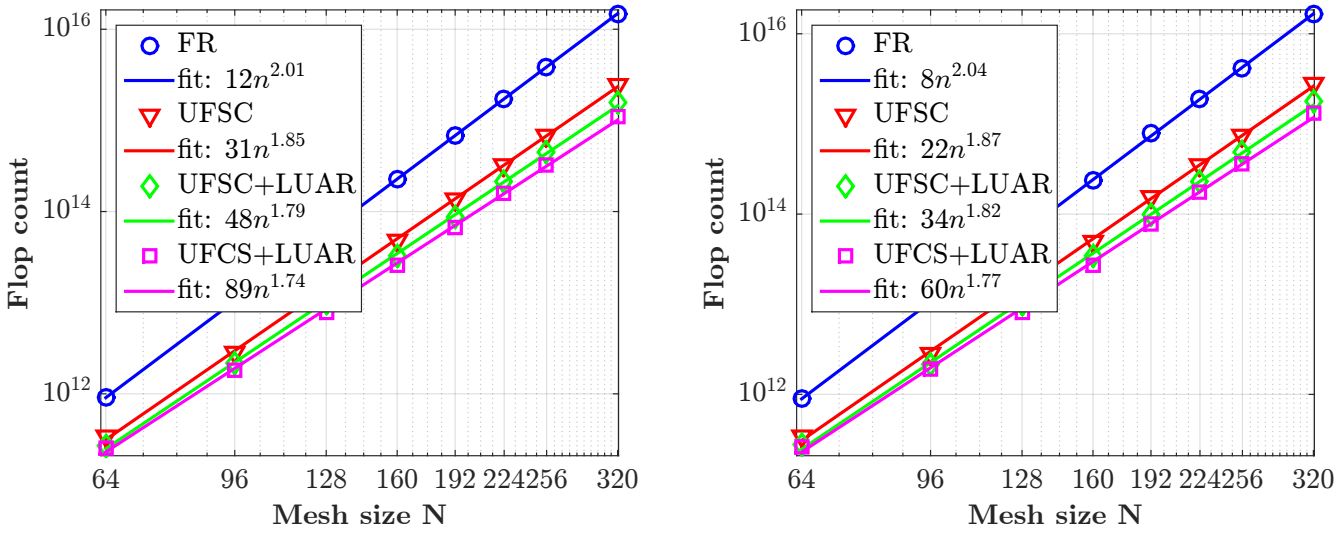
(b) METIS ordering (purely algebraic)

Figure 4.4 – Flop complexity of each BLR variant (Poisson, $\varepsilon = 10^{-10}$).

4.6.3 LUAR and CUFS complexity

In the previous experiments, the LUAR algorithm recompressed the accumulated updates exploiting only weight information (i.e. the Z part of the accumulator), which captures most of the recompression potential while keeping the recompress overhead cost limited, as explained in Section 2.6.

However, it is interesting to investigate how the additional recompression obtained recompressing geometry information (i.e. the X, Y parts of the accumulator) evolves with the size of the problem. This analysis is performed in Figure 4.6, for the UFCS variant on the Poisson problem and a low-rank threshold equal to $\varepsilon = 10^{-10}$.



(a) Nested Dissection ordering (geometric)

(b) METIS ordering (purely algebraic)

Figure 4.5 – Flop complexity of each BLR variant (Helmholtz, $\varepsilon = 10^{-4}$).

Recompressing geometry information is not beneficial for the smaller problems: the recompress overhead is greater than the additional recompression gained. In this case, gains are obtained for problems of size $N = 192$ or greater. This shows that the LUAR algorithm based on both weight and geometry recompression has lower complexity than the one based on weight only, but a (much) larger prefactor. This is confirmed by the fitting shown in Figure 4.6: compared to the asymptotic complexity of $\mathcal{O}(n^{1.30})$ obtained with weight recompression only, geometry recompression achieves a $\mathcal{O}(n^{1.21})$ complexity[1]. This is therefore a marginal asymptotic improvement that will only pay off for very large problems. For example, for $N = 320$, the geometry recompression brings a gain of 19% with respect to the weight recompression only version.

The fact that the geometry recompression is only beneficial for large problems means that it is probably beneficial for large fronts only. Therefore, a possible improvement is to exploit geometry information only for large enough fronts (i.e. of size greater than some n_{min}). We have tested several values of n_{min} and taken the minimal number of flops obtained for each problem size. The result is shown in Figure 4.6 (green diamond curve). This improved version allows us to reduce the prefactor and thus to achieve gains on smaller problems (of size $N = 128$ or greater), while maintaining almost the same asymptotic behavior ($\mathcal{O}(n^{1.22})$).

Finally, in Figure 4.7, we study the CUFS variant. and compare it to the UFCS+LUAR variant. The recompression is based on both weight and geometry information (the weight only recompression strategy is not suitable for the CUFS variant because it would lead to too large ranks, as explained in Section 3.4).

In the CUFS variant, the Outer Product step can be suppressed, and this leads to a lower prefactor (red triangle curve); however, the exponent is almost unchanged

[1]Note that the smaller problems for which the geometry recompression is not beneficial have not been taken into account for the fit, as we are interested in the asymptotic behavior.

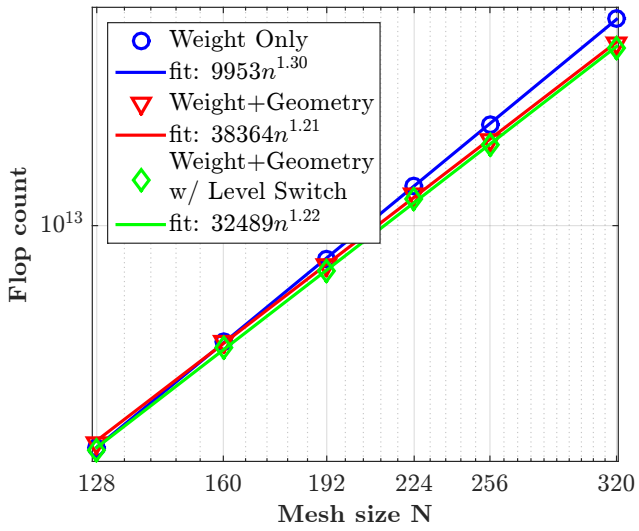


Figure 4.6 – Flop complexity of the UFCS+LUAR variant, depending on the recompression strategy (Poisson, $\varepsilon = 10^{-10}$).

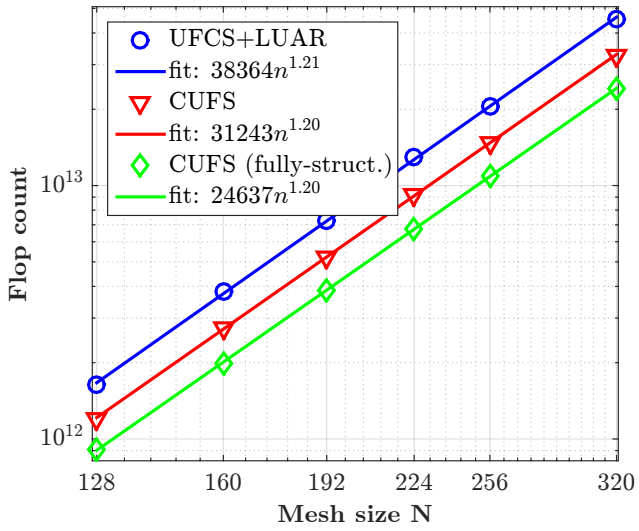


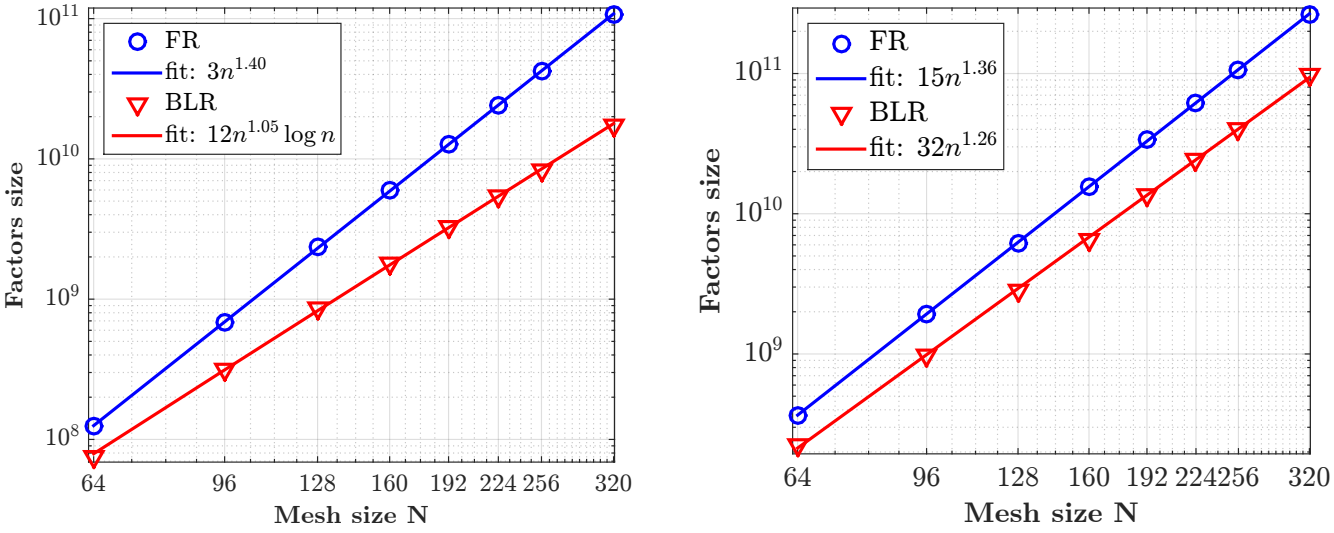
Figure 4.7 – Flop complexity of the CUFS variant (Poisson, $\varepsilon = 10^{-10}$).

($\mathcal{O}(n^{1.20})$), as predicted by the theory. Furthermore, if we assume the Compress step can be performed for free (fully-structured case, green diamond curve), the prefactor is further decreased, but again, no asymptotic gain is achieved, as expected.

4.6.4 Factor size complexity

To compute the factor size complexity of the BLR solver, we study the evolution of the number of entries in the factors, i.e., the compression rate of L and U. Note that the global compression rate would be even better, because the local matrices that need to be stored during the multifrontal factorization compress more than the factors.

In Figure 4.8, we plot the factor size complexity using the METIS ordering for both the Poisson and Helmholtz problems. The different BLR variants do not impact the factor size complexity. Here again, the results are in good agreement with the bounds computed in Table 4.6. The complexity is of order $\mathcal{O}(n^{1.05}\log n)$ for Poisson and $\mathcal{O}(n^{1.26})$ for Helmholtz.



(a) Poisson problem ($\varepsilon = 10^{-10}$)

(b) Helmholtz problem ($\varepsilon = 10^{-4}$)

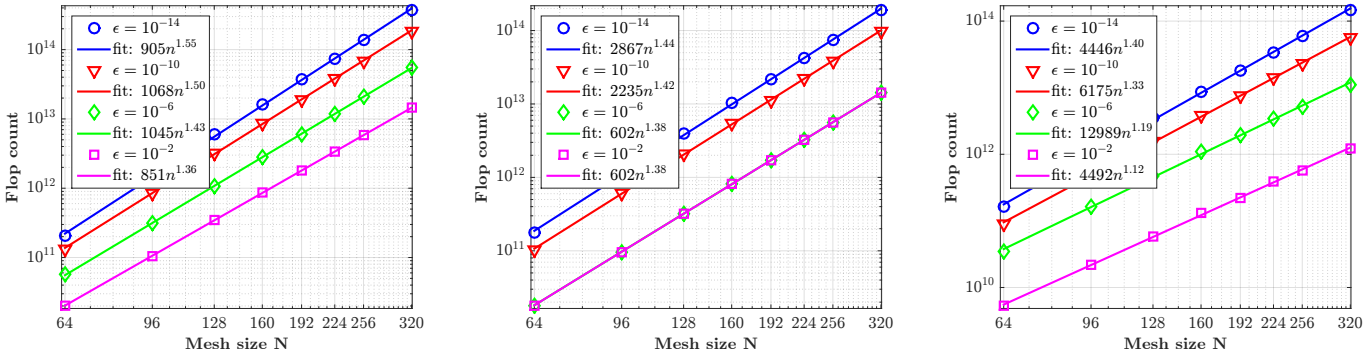
Figure 4.8 – Factor size complexity with METIS ordering.

4.6.5 Low-rank threshold

The theory (Bebendorf and Hackbusch, 2003; Bebendorf, 2005), through the bound r_G, which increases as $|\log\varepsilon|^{d+1}$, states the threshold ε should only play a role in the constant factor of the complexity.

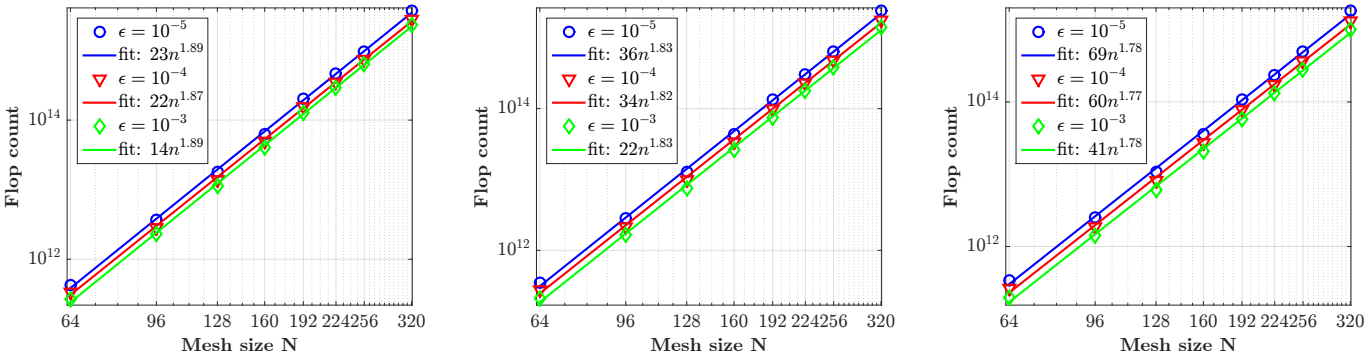
However, that is not exactly what the numerical experiments show. In Figures 4.9 and 4.10, we compare the flop complexity for different values of ε, for the Poisson and Helmholtz problems, respectively. For Helmholtz, the threshold does seem to play a role only in the constant factor, as the complexity exponent remains around $\mathcal{O}(n^{1.87})$, $\mathcal{O}(n^{1.82})$, and $\mathcal{O}(n^{1.77})$, for the UFSC, UFSC+LUAR, and UFCS+LUAR variants, respectively. However, for Poisson, an interesting trend appears: the complexity gradually lowers as the threshold increases. For example, the UFSC variant achieves a complexity of the order of $\mathcal{O}(n^{1.55})$ to $\mathcal{O}(n^{1.50})$, $\mathcal{O}(n^{1.46})$ and $\mathcal{O}(n^{1.36})$ with a threshold of 10^{-14}, 10^{-10}, 10^{-6}, and 10^{-2}, respectively. A similar behavior is observed for the UFSC+LUAR and UFCS+LUAR variants. The factor size complexity analysis, reported in Figure 4.11, leads to the same observation: varying the threshold leads to a high asymptotic variation for Poisson but not for Helmholtz.

Our analysis is that the complexity exponent is related to the processing of zero-rank blocks. With absolute tolerance, it is possible for blocks to have a numerical rank equal to zero. However, the bound on the ranks r_G is strictly positive, and



(a) UFSC variant (b) UFSC+LUAR variant (c) UFCS+LUAR variant

Figure 4.9 – Flop complexities for different thresholds ϵ (Poisson problem, METIS ordering).



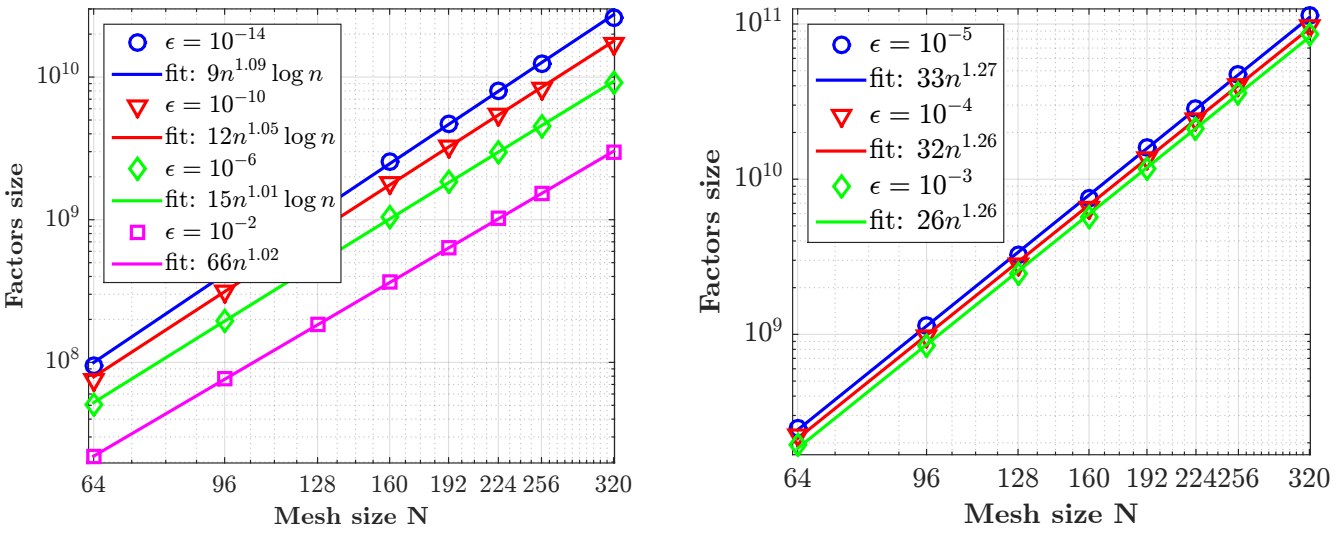
(a) UFSC variant (b) UFSC+LUAR variant (c) UFCS+LUAR variant

Figure 4.10 – Flop complexities for different thresholds ϵ (Helmholtz problem, METIS ordering).

thus the theory does not account for zero-rank blocks. This leads to a theoretical sparsity constant c_{sp} equal to $p = m/b$ (i.e. all blocks are considered nonzero-rank), while in fact the actual value of c_{sp} (number of nonzero-rank blocks) may be much less than p.

Clearly, the number of zero-rank blocks increases with the threshold ϵ and with the mesh size N. What is more interesting, as shown in Table 4.7, is that the number of zero-rank blocks (N_{ZR}) has a much faster rate of increase with respect to the mesh size than the number of nonzero low-rank blocks (N_{LR}). For example, for $\epsilon = 10^{-2}$, the number of zero-rank blocks represents 74% of the total for $N = 64$ while it represents 97% for $N = 320$.

This could suggest that, asymptotically, the major part of the blocks are zero-rank blocks. Then, the number of low-rank blocks in Table 4.1 would not be $\mathcal{O}(p)$ but rather $\mathcal{O}(p^\alpha)$, with $\alpha < 1$. For example, for $\epsilon = 10^{-2}$, the complexity of $\mathcal{O}(n^{1.36})$ could be explained by a number of nonzero-rank blocks of order $\mathcal{O}(1)$: indeed, if we assume the number of nonzero-rank blocks per row and/or column remains constant, then the dense flop complexity equation (4.23) is only driven by full-rank operations and



(a) Poisson problem

(b) Helmholtz problem

Figure 4.11 – Factor size complexities for different thresholds ε (METIS ordering).

		N							
		64	96	128	160	192	224	256	320
$\varepsilon = 10^{-14}$	N_{FR}	40.8	35.5	31.3	30.3	26.4	26.4	23.6	13.4
	N_{LR}	59.2	64.5	68.6	69.6	73.6	73.6	76.4	86.6
	N_{ZR}	0.0	0.0	0.1	0.1	0.0	0.0	0.0	0.0
$\varepsilon = 10^{-10}$	N_{FR}	21.3	19.1	16.6	17.0	14.6	14.6	12.8	5.8
	N_{LR}	78.6	80.9	83.4	82.9	85.4	85.4	87.1	94.2
	N_{ZR}	0.0	0.0	0.1	0.1	0.0	0.0	0.0	0.0
$\varepsilon = 10^{-6}$	N_{FR}	2.9	3.2	3.0	3.1	2.5	2.5	2.1	0.6
	N_{LR}	97.0	96.5	96.7	96.4	96.4	95.7	95.3	93.3
	N_{ZR}	0.1	0.3	0.3	0.5	1.0	1.7	2.5	6.1
$\varepsilon = 10^{-2}$	N_{FR}	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	N_{LR}	26.2	17.4	12.2	9.4	7.6	6.4	5.5	3.0
	N_{ZR}	73.8	82.6	87.8	90.6	92.4	93.6	94.5	97.0

Table 4.7 – Number of full- (N_{FR}), low- (N_{LR}), and zero-rank (N_{ZR}) blocks in percentage of the total number of blocks.

becomes

$$\mathscr{C}(m,x) = \mathcal{O}(m^{2+x} + m^2 r) = \mathcal{O}(m^{2+x}), \tag{4.38}$$

since $r \leq b = \mathcal{O}(m^x)$. With the optimal $x^* = 0$, the previous equation leads to a dense complexity $\mathcal{O}(m^2)$, which, as shown in Section 4.5, Table 4.4, leads to a sparse complexity $\mathcal{O}(n^{1.33})$. Similarly, the same assumption on the number of nonzero-rank blocks leads to a factor size complexity of $\mathcal{O}(n)$, which matches very well the experimental $\mathcal{O}(n^{1.02})$ result with $\varepsilon = 10^{-2}$ in Figure 4.11.

4.6.6 Block size

For our theoretical complexity, we have assumed that the block size varies with the front size. Here, we want to show that the complexity of the BLR factorization is not strongly impacted by the choice of the block size, as long as this choice remains reasonable. The choice of a good block size is currently an ongoing research focus; the tuning of the block size for performance is out of the scope of this paper. In Figure 4.12, we show the number of operations for the BLR factorization of the root node (which is of size $m = N^2$) of the Poisson problem, for block sizes $b \in [128;640]$. Three trends can be observed.

First, for each matrix, there is a reasonably large range of block sizes around the optimal one that lead to a number of operations that is reasonable with respect to the minimal one. For example, for the UFSC variant and $m = 256^2$, the optimal block size among the ones tested is $b = 448$. However, any block size in the range $[320;640]$ (all those under the dashed black line) leads to a number of operations at most 10% greater than the minimal one. Thus, we have the flexibility to choose the block size which in turn gives the flexibility to tune the performance of the BLR factorization.

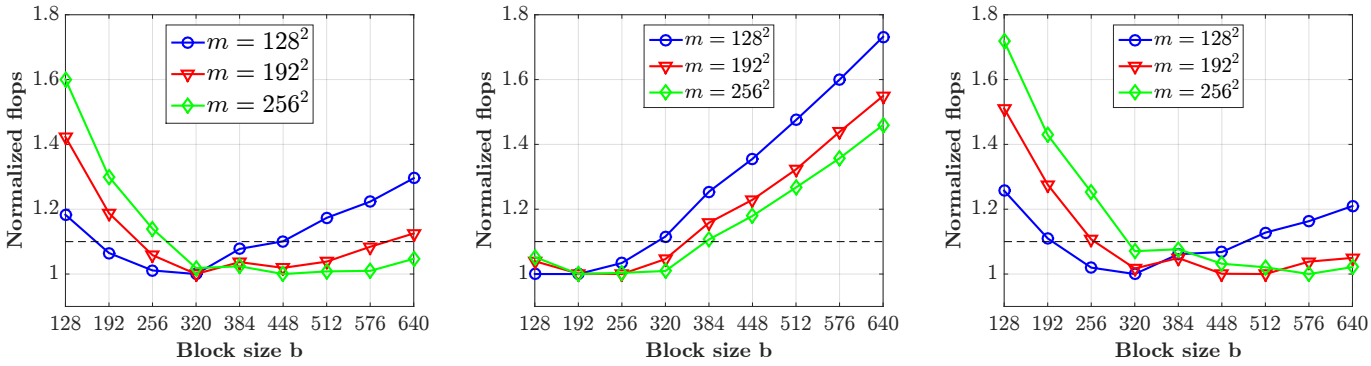
The second trend is that the range of acceptable block sizes (i.e., the block sizes for which the number of operations is not too far from the minimal one) increases with the size of the matrix m. For example, for the UFSC variant, the range of block sizes leading to a number of operations at most 10% greater than the minimal one is $[192;384]$ for $m = 128^2$, $[256;576]$ for $m = 192^2$ and $[320;640]$ for $m = 256^2$. This is expected and in agreement with the theory, at least under the assumption that the block size is of the form $b = \mathcal{O}(m^{x^*})$. This shows the importance of having a variable block size during the multifrontal factorization to adapt to the separators' size along the assembly tree, as opposed to fixed block size.

The third trend is observed when comparing the three factorization variants. Compared to the standard UFSC variant (Figure 4.12a), the UFSC+LUAR variant (Figure 4.12b) tends to favor smaller block sizes. This is because the low-rank term of the complexity equation (4.33) has been further reduced, and thus, in relative, the full-rank term (which increases with the block size) represents a greater part of the computations. This is accounted for by the theory with the optimal value x^* being equal to 1/3 instead of 1/2. In turn, compared to the UFSC+LUAR variant, the UFCS+LUAR (Figure 4.12c) benefits from bigger block sizes. This comes from the fact that the full-rank term has been reduced from (4.33) to (4.35). This is again consistent with the theory, which leads to $x^* = 1/2$.

A similar study on the Helmholtz problem shows very flat curves (i.e., the block size has little effect on the number of operations) as long as the block size remains reasonable.

4.7 Chapter conclusion

We have computed a bound on the numerical rank of the admissible blocks of BLR matrices arising from discretized elliptic PDEs. This bound is the same as in the hierarchical case (up to a constant), but cannot be obtained by applying directly



(a) UFSC variant (b) UFSC+LUAR variant (c) UFCS+LUAR variant

Figure 4.12 – Normalized flops (i.e., the minimal is 1) for different block sizes b (Poisson problem, $\varepsilon = 10^{-10}$, METIS ordering). The block sizes under the dashed black line are those for which the number of operations is at most 10% greater than the minimal one.

the theoretical work done on $\mathscr{H}$-matrices. The main idea of the extension to BLR matrices is to identify the blocks that are not low-rank, and to reformulate the admissibility condition of a partition to ensure that they are in negligible number for an admissible partition.

Under this bound assumption, we have computed the theoretical complexity of the BLR multifrontal factorization. The standard UFSC version can reach a complexity as low as $\mathcal{O}(n^{5/3})$ (in 3D, for constant ranks). We have shown how the other factorization variants can further reduce this complexity, down to $\mathcal{O}(n^{4/3})$. Our numerical results demonstrate an experimental complexity that is in good agreement with the theoretical bounds. The importance of zero-rank blocks and variable block sizes on the complexity has been identified and analyzed.

The object of the next chapter is to translate this complexity reduction into actual performance gains. This is a challenging problem, especially in a parallel setting.

CHAPTER

5 Performance of the BLR Factorization on Multicores

In this chapter, we present a multithreaded BLR factorization for multicore architectures and analyze its performance on a variety of problems coming from real-life applications. We explain why it is difficult to fully convert the reduction in the number of operations into a performance gain, especially in multicore environments, and describe how to improve the efficiency and the scalability of the BLR factorization.

We briefly describe the organization of this chapter. In Section 5.1, we describe our experimental setting. In Section 5.2, we motivate our work with an analysis of the performance of the FSCU algorithm in a sequential setting. We then present in Section 5.3 the parallelization of the BLR factorization in a shared-memory context, the challenges that arise, and the algorithmic choices made to overcome these challenges. In Section 5.4, we analyze the algorithmic variants of the BLR multifrontal factorization that were described in Chapter 2. We show how they can improve the performance of the standard algorithm. In Section 5.5, we provide a complete set of experimental results on a variety of real-life applications and in different multicore environments. We provide our concluding remarks in Section 5.6.

5.1 Experimental setting

5.1.1 Test machines

In this chapter, we use the **brunch** and **grunch** shared-memory, multicore machines. A detailed description of these machines is provided in Table 5.1. All the experiments reported in this article, except those of Table 5.14, were performed on **brunch**, which equipped with 1.5 TB of memory and four Intel 24-cores Broadwell processors running at a frequency varying between 2.2 and 3.4 GHz, due to the turbo technology. We consider as peak per core the measured performance of the dgemm kernel with one core, 47.1 GF/s. Bandwidth is measured with the STREAM benchmark. For all experiments on **brunch** where several threads are used, the threads are scattered among the four processors to exploit the full bandwidth of the machine.

To validate our performance analysis, we report in Table 5.14 additional experiments performed on **grunch**, which has a similar architecture but different proper-

ties (frequency and bandwidth), as described in Section 5.5.3. For the experiments on **grunch**, all 28 cores are used.

name	cpu model	np	nc	freq (GHz)	peak (GF/s)	bw (GB/s)	mem (GB)
brunch	E7-8890 v4	4	24	2.2–3.4*	47.1*	102	1500
grunch	E5-2695 v3	2	14	2.3	36.8	57	768

*frequency can vary due to turbo; peak is estimated as the dgemm peak

Table 5.1 – List of machines used for Table 5.14 and their properties: number of processors (np), number of cores (nc), frequency (freq), peak performance, bandwidth (bw), and memory (mem).

The above GF/s peak, as well as all the other GF/s values in this article, are computed counting flops in double-precision real (d) arithmetic, and assuming a complex flop corresponds to four real flops of the same precision.

5.1.2 Test problems

In the experiments of this chapter, we have used real life problems coming from the three applications described in Section 1.5.2.2, as well as additional matrices coming from the UFSMC. We remind that the complete set of matrices and their description is provided in Table 1.3.

We also remind that the low-rank threshold is chosen according to the application requirements: 10^{-3} for the seismic modeling matrices, 10^{-7} for the electromagnetics matrices, and 10^{-9} for the structural mechanics matrices. For the matrices from the UFSMC, we have arbitrarily set the low-rank threshold to $\varepsilon = 10^{-6}$, except for the more difficult matrix nlpkkt120 where we used $\varepsilon = 10^{-9}$ (cf. Section 5.5).

For all experiments, we have used a right-hand side b such that the solution x is the vector containing only ones.

We provide in Section 5.5 experimental results on the complete set of matrices. For the sake of conciseness, the performance analysis in the main body of this chapter (Sections 5.2 to 5.4) will focus on matrix S3 (described in Section 1.5.2.2).

5.2 Performance analysis of sequential FSCU algorithm

In this section, we analyze the performance of the FSCU algorithm (described in Algorithm 2.1) in a sequential setting. Our analysis underlines several issues, which will be addressed in subsequent sections.

In Table 5.2, we compare the number of flops and execution time of the sequential FR and BLR factorizations. While the use of BLR reduces the number of flops by a factor 7.7, the time is only reduced by a factor 3.3. Thus, the potential gain in terms of flops is not fully translated in terms of time.

	FR	BLR	ratio
flops ($\times 10^{12}$)	77.97	10.19	7.7
time (s)	7390.1	2241.9	3.3

Table 5.2 – Sequential run (1 thread) on matrix S3.

To understand why, we report in Table 5.3 the time spent in each step of the factorization, in the FR and BLR cases. The relative weight of each step is also provided in percentage of the total. In addition to the four main steps Factor, Solve, Compress and Update, we also provide the time spent in parts with low arithmetic intensity (LAI parts). This includes the time spent in assembly, memory copies and factorization of the fronts at the bottom of the tree, which are too small to benefit from BLR and are thus treated in FR.

	FR				BLR			
step	flops ($\times 10^{12}$)	%	time (s)	%	flops ($\times 10^{12}$)	%	time (s)	%
Factor+Solve	1.51	1.9	671.0	9.1	1.51	14.9	671.0	29.9
Update	76.22	97.8	6467.0	87.5	7.85	77.0	1063.7	47.4
Compress	0.00	0.0	0.0	0.0	0.59	5.8	255.1	11.4
LAI parts	0.24	0.3	252.1	3.4	0.24	2.3	252.1	11.2
Total	77.97	100.0	7390.1	100.0	10.19	100.0	2241.9	100.0

Table 5.3 – Performance analysis of sequential run of Table 5.2 on matrix S3.

The FR factorization is clearly dominated by the Update, which represents 87.5% of the total time. In BLR, the Update operations are done exploiting the low-rank property of the blocks and thus the number of operations performed in the Update is divided by a factor 9.7. The Factor+Solve and LAI steps remain in FR and thus do not change. From this result, we can identify three main issues with the performance of the BLR factorization:

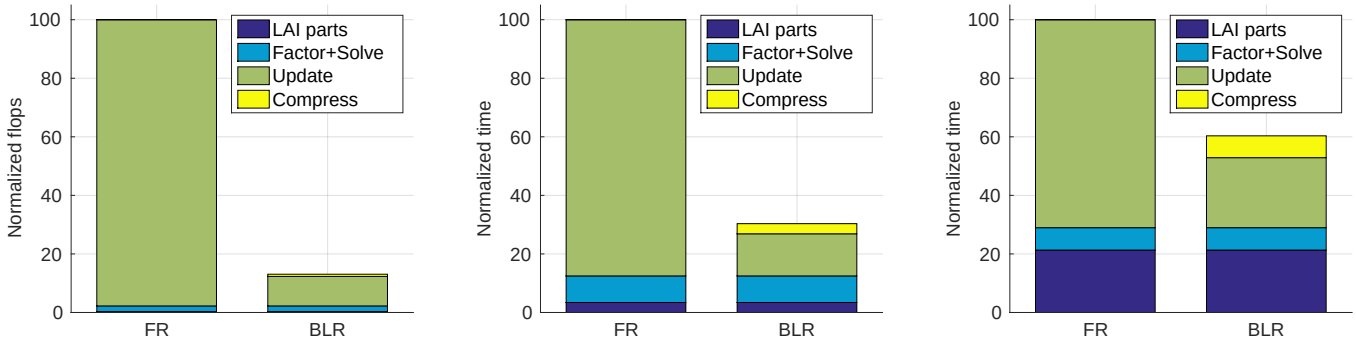
Issue 1 (lower granularity): the flop reduction by a factor 9.7 in the Update is not fully captured, as its execution time is only reduced by a factor 6.1. This is due to the lower granularity of the operations involved in low-rank products, which have thus a lower performance: the speed of the Update step is 47.1 GF/s in FR and 29.5 GF/s in BLR.

Issue 2 (higher relative weight of the FR parts): because the Update is reduced in BLR, the relative weight of the parts that remain FR (Factor, Solve, and LAI parts) increases from 12.5% to 41.1%. Thus, even if the Update step is further accelerated, one cannot expect the global reduction to follow as the FR part will become the bottleneck.

Issue 3 (cost of the Compress step): even though the overhead cost of the Compress step is negligible in terms of flops (5.8% of the total), it is a very slow operation (9.2 GF/s) and thus represents a non-negligible part of the total time (11.4%).

A visual representation of this analysis is given on Figure 5.1 (compare Figures 5.1a and 5.1b).

In the next section, we first extend the BLR factorization to the multithreaded case, for which previous observations are even more critical. **Issues 1** and **2** will then be addressed by the algorithmic variants of the BLR factorization in Section 5.4. **Issue 3** is a topic of research by itself; it is out of the scope of this thesis and we only comment on possible ways to reduce the cost of the Compress step in the conclusion chapter.



(a) Normalized flops (b) Normalized time (1 thread) (c) Normalized time (24 threads)

Figure 5.1 – Normalized (FR = 100%) flops and time on matrix S3.

5.3 Multithreading the BLR factorization

In this section, we describe the shared-memory parallelization of the BLR FSCU factorization (Algorithm 2.1).

5.3.1 Performance analysis of multithreaded FSCU algorithm

Our reference Full-Rank implementation is based on a fork-join approach combining OpenMP directives with multithreaded BLAS libraries. While this approach can have limited performance on very small matrices, on the set of problems considered, it achieves quite satisfactory speedups on 24 threads (around 20 for the largest problems) because the bottleneck consists of matrix-matrix product operations. This approach will be taken as a reference for our performance analysis.

In the BLR factorization, the operations have a finer granularity and thus a lower speed and a lower potential for exploiting efficiently multithreaded BLAS. To overcome this obstacle, more OpenMP-based multithreading exploiting serial BLAS has been introduced. This allows for a larger granularity of computations per thread than multithreaded BLAS on low-rank kernels. In our implementation, we simply parallelize the loops of the Compress and Update operations on different blocks (lines 4, and 7-8) of Algorithm 2.1. The Factor+Solve step remains full-rank, as well as the FR factorization of the fronts at the bottom of the assembly tree.

Because each block has a different rank, the task load of the parallel loops is very irregular in the BLR case. To account for this irregularity, we use the dynamic OpenMP schedule (with a chunk size equal to 1), which achieves the best performance.

In Table 5.4, we compare the execution time of the FR and BLR factorization on 24 threads. The multithreaded FR factorization achieves a speedup of 14.5 on 24 threads. However, the BLR factorization achieves a much lower speedup of 7.3. The gain factor of BLR with respect to FR is therefore reduced from 3.3 to 1.7.

	FR	BLR	ratio
time (1 thread)	7390.1	2241.9	3.3
time (24 threads)	508.5	306.8	1.7
speedup	14.5	7.3	

Table 5.4 – Multithreaded run on matrix S3.

The BLR multithreading is thus less efficient than the FR one. To understand why, we provide in Table 5.5 the time spent in each step for the multithreaded FR and BLR factorizations. We additionally provide for each step the speedup achieved on 24 threads.

	FR			BLR		
step	time	%	speedup	time	%	speedup
Factor+Solve	38.9	7.7	17.3	38.9	12.7	17.3
Update	361.2	71.0	17.9	121.6	39.6	8.8
Compress	0.0	0.0		37.9	12.4	6.7
LAI parts	108.4	21.3	2.3	108.4	35.3	2.3
Total	508.5	100.0	14.5	306.8	100.0	7.3

Table 5.5 – Performance analysis of multithreaded run (24 threads) of Table 5.4 on matrix S3.

From this analysis, one can identify two additional issues related to the multithreading of the BLR factorization:

Issue 4 (low arithmetic intensity parts become critical): the LAI parts expectedly achieve a very low speedup of 2.3. While their relative weight with respect to the total remains reasonably limited in FR, it becomes quite significant in BLR, with over 35% of time spent in them. Thus, the impact of the poor multithreading of the LAI parts is higher on the BLR factorization than on the FR one.

Issue 5 (scalability of the BLR Update): not only is the BLR Update less efficient than the FR one in sequential, it also achieves a lower speedup of 8.8 on 24 threads, compared to a FR speedup of 17.9. This comes from the fact that the BLR Update, due to its smaller granularities, is limited by the speed of

memory transfers instead of the CPU peak as in FR. As a consequence, the Outer Product operation runs at the poor speed of 8.8 GF/s, to compare to 35.2 GF/s in FR.

A visual representation of this analysis is given on Figure 5.1 (compare Figures 5.1b and 5.1c).

In the rest of this section, we will revisit our algorithmic choices to address both of these issues.

5.3.2 Exploiting tree-based multithreading

In our standard shared-memory implementation, multithreading is exploited at the node parallelism level only, i.e. different fronts are not factored concurrently. However, in multifrontal methods, multithreading may exploit both node and tree parallelism. Such an approach has been proposed, in the FR context, by L'Excellent and Sid-Lakhdar (2014) and relies on the idea of separating the fronts by a so-called $\mathscr{L}_0$ layer, as illustrated in Figure 5.2. Each subtree rooted at the $\mathscr{L}_0$ layer is treated sequentially by a single thread; therefore, below the $\mathscr{L}_0$ layer pure tree parallelism is exploited by using all the available threads to process concurrently multiple sequential subtrees. When all the sequential subtrees have been processed, the approach reverts to pure node parallelism: all the fronts above the $\mathscr{L}_0$ layer are processed sequentially (i.e., one after the other) but all the available threads are used to assemble and factorize each one of them.

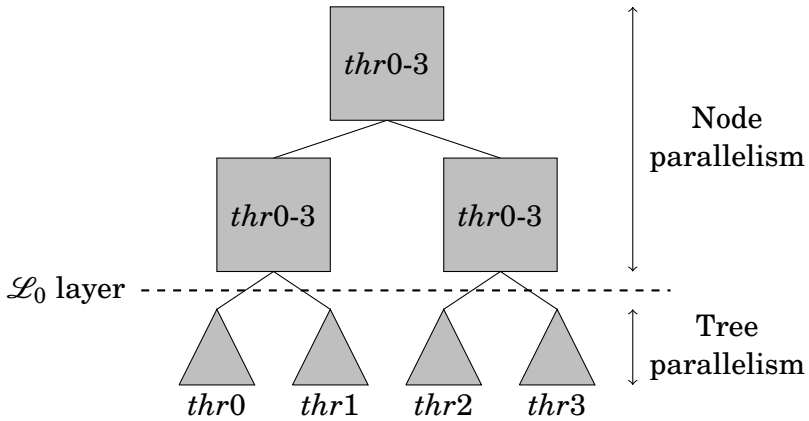


Figure 5.2 – Illustration with four threads of how both node and tree multithreading can be exploited.

In Table 5.6, we quantify and analyze the impact of this strategy on the BLR factorization. The majority of the time spent in LAI parts is localized under the $\mathscr{L}_0$ layer. Indeed, all the fronts too small to benefit from BLR are under it; in addition, the time spent in assembly and memory copies for the fronts under the $\mathscr{L}_0$ layer represents 60% of the total time spent in the assembly and memory copies. Therefore, the LAI parts are significantly accelerated, by a factor over 2, by exploiting tree multithreading.

In addition, the other steps (the Update and especially the Compress) are also accelerated thanks to the improved multithreading behavior of the relatively smaller BLR fronts under the $\mathscr{L}_0$ layer which do not expose much node parallelism.

	FR			BLR		
step	time	%	speedup	time	%	speedup
Factor+Solve	33.2	7.9	20.2	33.2	15.1	20.2
Update	331.7	79.4	19.5	110.2	50.0	9.7
Compress	0.0	0.0		24.1	10.9	10.6
LAI parts	53.0	12.7	4.8	53.0	24.0	4.8
Total	417.9	100.0	17.4	220.5	100.0	10.2

Table 5.6 – Execution time of FR and BLR factorizations on matrix S3 on 24 threads, exploiting both node and tree parallelism.

Please note that the relative gain due to introducing tree multithreading can be larger even in FR, for 2D or very small 3D problems, for which the relative weight of the LAI parts is important. However, for large 3D problems the relative weight of the LAI parts is limited, and the overall gain in FR remains marginal. In BLR, the weight of the LAI parts is much more important so that exploiting tree parallelism becomes critical: the overall gain is significant in BLR. We have thus addressed **Issue 4**, identified in Subsection 5.3.1.

The approach described in L'Excellent and Sid-Lakhdar (2014) additionally involves a so-called Idle Core Recycling (ICR) algorithm which consists in reusing the idle cores that have already finished factorizing their subtrees to help factorizing the subtrees assigned to other cores. This results in the use of both tree and node parallelism when the workload below the $\mathscr{L}_0$ layer is unbalanced.

The maximal potential gain of using ICR can be computed by measuring the difference between the maximal and average time spent under the $\mathscr{L}_0$ layer by the threads (this corresponds to the work unbalance). For the run of Table 5.6, the potential gain is equal to 3.3s in FR (i.e., 0.8% of the total) and 5.1s in BLR (i.e., 2.3% of the total). Thus, even though the potential gain in FR is marginal, it is higher in BLR, due to load unbalance generated by the irregularity of the compressions: indeed, the compression rate can greatly vary from front to front and thus from subtree to subtree.

Activating ICR brings a gain of 3.0s in FR and 4.7s in BLR; thus, roughly 90% of the potential gain is captured in both cases. While the absolute gain with respect to the total is relatively small even in BLR, this analysis illustrates that Idle Core Recycling becomes an even more relevant feature for the multithreaded BLR factorization.

Exploiting tree multithreading is thus very critical in the BLR context. It will be used for the rest of the experiments for both FR and BLR.

5.3.3 Right-looking vs. Left-looking

Algorithm 2.1 has been presented in its Right-looking (RL) version. In Table 5.7, we compare it to its Left-looking (LL) equivalent, referred to as UFSC (Algorithm 2.3). The RL and LL variants perform the same operations but in a different order, which results in a different memory access pattern (Dongarra, Duff, Sorensen, and Vorst, 1998).

		FR		BLR	
parallelism	step	RL	LL	RL	LL
1 thread	Update	6467.0	6549.8	1063.7	899.1
	Total	7390.1	7463.9	2241.9	2074.5
24 threads, node+tree//	Update	331.7	335.6	110.2	66.9
	Total	417.9	420.6	220.5	174.7

Table 5.7 – Execution time of Right- and Left-looking factorizations on matrix S3.

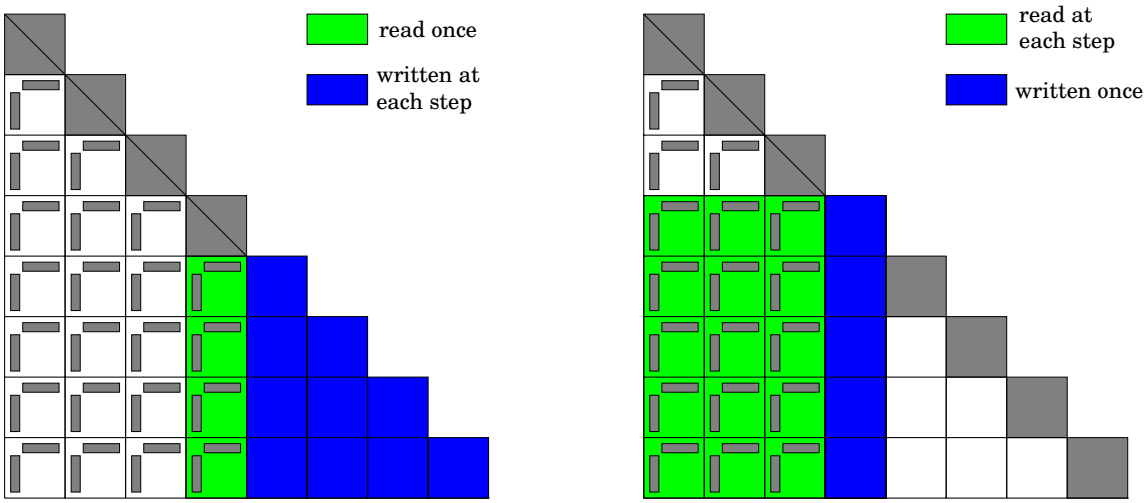
The impact of using a RL or LL factorization is mainly observed on the Update step. In FR, there is almost no difference between the two, RL being slightly (less than 1%) faster than LL. In BLR however, the Update is significantly faster in LL than in RL. This effect is especially clear on 24 threads (40% faster Update, which leads to a global gain of 20%).

We explain this result by a lower volume of memory transfers in LL BLR than RL BLR. As illustrated in Figure 5.3, during the BLR LDL^T factorization of a $p \times p$ block matrix, the Update will require loading the following blocks stored in main memory:

- in RL (Figure 5.3a), at each step k, the FR blocks of the trailing sub-matrix are written and therefore they are loaded many times (at each step of the factorization), while the LR blocks of the current panel are read once and never loaded again.
- in LL (Figure 5.3b), at each step k, the FR blocks of the current panel are written for the first and last time of the factorization, while the LR blocks of all the previous panels are read, and therefore they are loaded many times during the entire factorization.

Thus, while the number of loaded blocks is roughly the same in RL and LL (which explains the absence of difference between the RL FR and LL FR factorizations), the difference lies in the fact that the LL BLR factorization tends to load more often LR blocks and less FR blocks, while the RL one has the opposite behavior. To be precise:

- Under the assumption that one FR block and two LR blocks fit in cache, the LL BLR factorization loads $\mathcal{O}(p^2)$ FR blocks and $\mathcal{O}(p^3)$ LR blocks.
- Under the assumption that one FR block and an entire LR panel fit in cache (which is a stronger assumption so the number of loaded blocks may in fact



(a) RL factorization (b) LL factorization

Figure 5.3 – Illustration of the memory access pattern in the RL and LL BLR Update during step k of the factorization of a matrix of $p \times p$ blocks (here, $p = 8$ and $k = 4$).

be even worse), the RL BLR factorization loads $\mathcal{O}(p^3)$ FR blocks and $\mathcal{O}(p^2)$ LR blocks.

Thus, switching from RL to LL reduces the volume of memory transfers and therefore accelerates the BLR factorization, which addresses **Issue 5**, identified in Subsection 5.3.1.

Throughout the rest of this article, the best algorithm is considered: LL for BLR and RL for FR.

Thanks to both the tree multithreading and the Left-looking BLR factorization, the factor of gain due to BLR with respect to FR on 24 threads has increased from 1.7 (Table 5.4) to 2.4 (Table 5.7).

Next, we show how the algorithmic variants of the BLR factorization can further improve its performance.

5.4 BLR factorization variants

In this section, we study the UFSC+LUAR and UFCS+LUAR BLR factorization variants. In Chapter 4, we have proved that they lead to a lower theoretical complexity. In this section, we quantify the flop reduction achieved by these variants and how well this flop reduction can be translated into a time reduction. We analyze how they can improve the efficiency and scalability of the factorization.

5.4.1 LUAR: Low-rank Updates Accumulation and Recompression

We begin by the UFSC+LUAR variant, i.e. Algorithm 2.3 with the modified LUAR-Update of Algorithm 2.10.

The LUAR algorithm has two advantages: first, accumulating the update matrices together leads to higher granularities in the Outer Product step (line 8 of Algorithm 2.10), which is thus performed more efficiently. This should address **Issue 1**, identified in Section 5.2. Second, it allows for additional compression, as explained in Section 2.6.

In Table 5.8, we analyze the performance of the UFSC+LUAR variant. We separate the gain due to accumulation (UFSC+LUA, without recompression) and the gain due to the recompression (UFSC+LUAR). As explained in Chapter 3, two different types of recompression can be considered: weight recompression, which consists in recompressing the middle accumulator Z_S (cf. Figure 2.7); and geometry recompression, which consists in recompressing the outer accumulators X_S and/or Y_S. In the following, we compare both weight only recompression and weight+geometry recompression. We provide the flops, time and speed of both the Outer Product (which is the step impacted by this variant) and the total (to show the global gain). We also provide the average (inner) size of the Outer Product operation, which corresponds to the rank of $\widetilde{C}_{i,k}^{(acc)}$ at line 8 in Algorithm 2.10. It also corresponds to the number of columns of X_S and Y_S in Figure 2.7.

			UFSC	+LUA	+LUAR (weight only)	+LUAR (weight+geometry)
average size of Outer Product			16.5	61.0	32.8	9.1
flops	($\times 10^{12}$)	Outer Product	3.76	3.76	1.59	0.77
	($\times 10^{12}$)	Recompress	0.00	0.00	0.01	0.35
	($\times 10^{12}$)	Total	10.19	10.19	8.15	7.57
time (s)		Outer Product	21.4	14.0	6.0	2.7
		Recompress	0.0	0.0	1.2	11.6
		Total	174.7	167.1	160.0	168.3
speed (GF/s)		Outer Product	29.3	44.7	44.4	47.8
		Recompress			0.7	5.0
		Total	9.7	10.2	8.5	7.5

Table 5.8 – Performance analysis of the UFSC+LUAR factorization on matrix S3 on 24 threads.

Thanks to the accumulation, the average size of the Outer Product increases from 16.5 to 61.0. As illustrated by Figure 5.4, this higher granularity improves the speed of the Outer Product from 29.3 to 44.7 GF/s (compared to a peak of 47.1 GF/s) and thus accelerates it by 35%. The impact of accumulation on the total time depends on both the matrix and the computer properties and will be further discussed in Section 5.5.

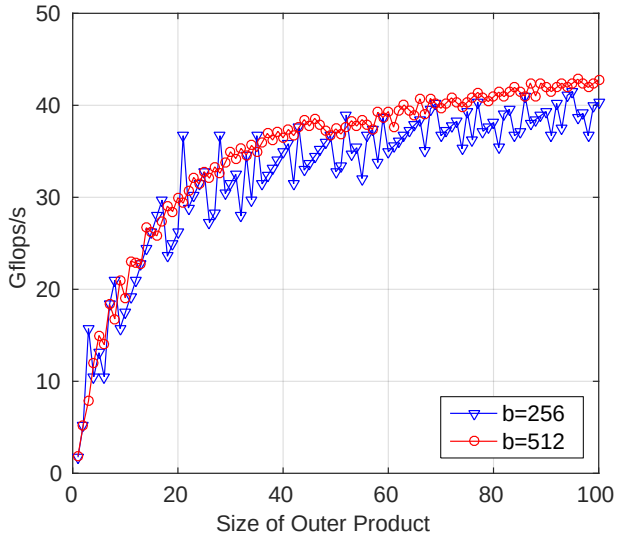


Figure 5.4 – Performance benchmark of the Outer Product step on **brunch**. Please note that the average sizes (first line) and speed values (eighth line) of Table 5.8 cannot be directly linked using this figure because the average size would need to be weighted by its number of flops.

Next, we analyze the gain obtained by recompressing the accumulated low-rank updates (Figure 2.7b). We begin by weight only recompression. While the total flops are reduced by 20%, the execution time is only accelerated by 5%. This is partly due to the fact that the Outer Product only represents a small part of the total, but could also come from two other reasons:

- The recompression decreases the average size of the Outer Product back to 32.8. As illustrated by Figure 5.4, its speed remains at 44.4 GF/s and is thus not significantly decreased, but it can be the case for other matrices or machines.

- The speed of the Recompress operation itself is 0.7 GF/s, an extremely low value. Thus, even though the Recompress overhead is negligible in terms of flops, it can limit the global gain in terms of time. Here, the time overhead is 1.2s for an 8s gain, i.e. 15% overhead.

The latter issue becomes even more noticeable when considering weight+geometry recompression. Indeed, even though the flops are further reduced, the efficient Outer Product operations are traded for much slower Recompress operations, which represent an important part of the total in the weight+geometry case. Overall, recompressing the geometry information leads to slowdowns. Therefore, we conclude that recompressing the middle accumulator Z_S only (weight information only) is the best strategy as, due to the small size of Z_S, it leads to a lower Recompress cost while capturing most of the recompression potential. In the rest of our experiments, we use weight only recompression.

5.4.2 UFCS algorithm

In all the previous experiments, threshold partial pivoting was performed during the FR and BLR factorizations, which means the Factor and Solve steps were merged together as described in Section 5.1. For many problems, numerical pivoting can be restricted to a smaller area of the panel (for example, the diagonal BLR blocks). In this case, the Solve step can be separated from the Factor step and applied directly on the entire panel, thus solely relying on BLAS-3 operations.

Furthermore, in BLR, when numerical pivoting is restricted, it is natural and more efficient to perform the Compress before the Solve (thus leading to the so-called UFCS factorization). Indeed UFCS makes further use of the low-rank property of the blocks since the Solve step can then be performed in low-rank as shown at line 11 in Algorithm 2.5.

Note that for the matrices where pivoting cannot be restricted, we can instead turn to the UCFS variant, discussed in Section 2.3.2. We do not evaluate the multicore performance of this variant in this thesis.

In Table 5.9, we report the gain achieved by UFCS and its accuracy. We measure the scaled residual $\frac{\|Ax-b\|_\infty}{\|A\|_\infty\|x\|_\infty}$. We first compare the factorization with either standard or restricted pivoting. Restricting the pivoting allows the Solve to be performed with more BLAS-3 and thus the factorization is accelerated. This does not degrade the solution because on this test matrix restricted pivoting is enough to preserve accuracy.

	standard pivoting		restricted pivoting		
	FR	UFSC +LUAR	FR	UFSC +LUAR	UFCS +LUAR
flops ($\times 10^{12}$)	77.97	8.15	77.97	8.15	3.95
time (s)	417.9	160.0	401.3	140.4	110.7
scaled residual	4.5e-16	1.5e-09	5.0e-16	1.9e-09	2.7e-09

Table 5.9 – Performance and accuracy of UFSC and UFCS variants on 24 threads on matrix S3.

We then compare UFSC and UFCS (with LUAR used in both cases). The flops for the UFCS factorization are reduced by a factor 2.1 with respect to UFSC. This can at first be surprising as the Solve step represents less than 20% of the total flops of the UFSC factorization.

To explain the relatively high gain observed in Table 5.9, we analyze in detail the difference between UFSC and UFCS in Table 5.10. By performing the Solve in low-rank, we reduce its number of operations of the Factor+Solve step by a factor 4.2, which translates to a time reduction of this step by a factor of 1.9. Furthermore, the flops of the Compress and Update steps are also significantly reduced, leading to a time reduction of 15% and 35%, respectively. This is because the Compress is performed earlier, which decreases the ranks of the blocks. On our test problem, the average rank decreases from 21.6 in UFSC to 16.2 in UFCS, leading a very small relative increase of the scaled residual. The smaller ranks also lead to a smaller average size of the Outer Product, which decreases from 32.8 (last column of

	flops ($\times 10^{12}$)		time (s)	
	UFSC	UFCS	UFSC	UFCS
Factor+Solve	1.52	0.36	12.4	6.6
Update	5.78	2.93	53.4	34.0
Compress	0.62	0.43	24.1	20.4
LAI parts	0.24	0.24	50.5	49.7
Total	8.15	3.95	140.4	110.7

Table 5.10 – Detailed analysis of UFSC and UFCS results of Table 5.9 on matrix S3.

Table 5.8) to 24.4. This makes the LUAR variant even more critical when combined with UFCS: with no accumulation, the average size of the Outer Product in UFCS would be 10.9 (to compare to 16.5 in UFSC, first column of Table 5.8).

Thanks to both the LUAR and UFCS variants, the factor of gain due to BLR with respect to FR on 24 threads has increased from 2.4 (Table 5.7) to 3.6 (Table 5.9).

5.5 Complete set of results

This section serves two purposes. First, we show that the results and the analysis reported on a representative matrix on a given computer hold for a large number of matrices coming from a variety of real-life applications and in different multicore environments. Second, we will further comment on specificities that depend on the matrix or machine properties.

The results on the matrices coming from the three real-life applications from SEISCOPE, EMGS and EDF (described in Section 1.5.2.2) are reported in Table 5.11. To demonstrate the generality and robustness of our solver, these results are completed with those of Table 5.12 on matrices from the UFSMC. We summarize the main results of Tables 5.11 and 5.12 with a visual representation in Figure 5.5. Then, for the biggest problems, we report in Table 5.13 results obtained using 48 threads instead of 24. We recall that the test matrices are described and assigned an ID in Table 1.3.

5.5.1 Results on the complete set of matrices

We report the flops and time on 24 threads for all variants of the FR and BLR factorizations and report the speedup and scaled residual $\frac{\|Ax-b\|_\infty}{\|A\|_\infty\|x\|_\infty}$ for the best FR and BLR variants. The scaled residual in FR is taken as a reference. In BLR, the scaled residual also depends on the low-rank threshold ε (whose choice of value is justified in Section 5.1). One can see in Tables 5.11 and 5.12 that in BLR the scaled residual correctly reflects the influence of the low-rank approximations with threshold ε on the FR precision. Matrix nlpkkt120 (matrix ID 22) is a numerically difficult problem for which the FR residual (1.9e-08) is several digits lower than the machine precision; on this matrix the low-rank threshold is set to a smaller value

low-rank threshold ε		10^{-3}			10^{-7}					10^{-9}		
matrix ID		1	2	3	4	5	6	7	8	9	10	11
flops ($\times 10^{12}$)	FR	69.5	471.1	2703.0	57.9	2188.0	78.0	3119.0	101.0	377.5	1616.0	23.6
	BLR	9.3	48.4	222.8	10.4	159.2	10.2	163.3	21.4	55.2	157.2	5.6
	+ LUAR	7.0	34.4	146.1	8.3	95.2	8.1	105.6	17.7	43.3	110.2	5.2
	+ UFCS	6.4	34.3	100.1	3.7	53.1	3.9	48.1	15.9	37.6	93.5	—
	flop ratio*	**10.8**	**13.7**	**27.0**	**15.8**	**41.2**	**19.7**	**64.8**	**6.4**	**10.0**	**17.3**	**4.2**
time (24 cores)	FR	235.2	1295.9	6312.5	376.4	10089.4	508.5	14362.3	211.7	662.2	2272.1	166.7
	+ tree//	196.2	1100.0	5844.8	321.4	9779.2	417.9	13979.7	174.3	577.3	2145.0	77.6
	+ rest. piv.	163.2	1013.0	5649.5	304.2	9655.6	401.3	13842.7	163.8	544.1	2066.9	—
	BLR	146.2	537.7	1998.8	229.0	2967.5	306.8	3702.9	161.5	416.7	1129.0	151.5
	+ tree//	92.8	373.6	1497.3	161.2	2586.7	220.5	3165.9	115.6	313.0	944.5	56.3
	+ UFSC	88.1	347.7	1334.3	150.1	1688.4	174.7	1971.6	99.3	245.2	632.9	50.0
	+ LUA	84.0	327.5	1245.4	145.6	1643.7	167.1	1856.6	97.3	232.2	570.2	49.1
	+ LUAR	91.0	362.5	1196.9	138.3	1509.4	160.0	—	92.7	216.5	515.8	79.6
	+ UFCS	49.7	194.8	773.7	91.0	652.7	110.7	736.1	78.2	176.7	377.9	—
	time ratio*	**3.3**	**5.2**	**7.3**	**3.3**	**14.8**	**3.6**	**18.8**	**2.1**	**3.1**	**5.5**	**1.6**
speedup (24 cores)	Best FR	17.8	18.8	—	17.7	—	18.2	—	15.9	17.5	—	14.2
	Best BLR	11.3	13.8	12.6	9.6	11.5	9.0	12.2	10.8	12.2	13.5	12.3
scaled residual	Best FR	1.7e-04	3.5e-04	2.9e-04	3.7e-16	7.0e-16	5.0e-16	8.1e-16	9.1e-15	5.2e-15	7.1e-15	1.4e-15
	Best BLR	3.1e-02	2.9e-02	4.2e-02	3.1e-10	2.8e-10	2.7e-09	2.0e-10	1.4e-08	3.7e-08	5.0e-08	4.5e-13

*between best FR and best BLR

Table 5.11 – Experimental results on real-life matrices from SEISCOPE, EMGS, and EDF.

(10^{-9}) to preserve a scaled residual comparable to those obtained with the other matrices from the UFSMC set.

On this set of problems, BLR always reduces the number of operations with respect to FR by a significant factor. This factor is never fully translated in terms of time, but the time gains remain important, even for the smaller problems.

Tree parallelism (tree//), the Left-looking factorization (UFSC) and the accumulation (LUA) always improve the performance of the BLR factorization. For some smaller problems where the factorization of the fronts at the bottom of the assembly tree represents a considerable part of the total computations, such as StocF-1465 and atmosmodd (matrix ID 12 and 13), exploiting tree parallelism is especially critical, even in FR.

Even though the recompression (LUAR) is always beneficial in terms of flops, it is not always the case in terms of time. Especially for the smaller problems, the low speed of the computations may lead to slowdowns. When LUAR is not beneficial (in terms of time), the "+UFCS" lines in Tables 5.11 and 5.12 correspond to a UFCS factorization without Recompression (LUA only).

For most of the problems, the UFCS factorization obtained a scaled residual of the same order of magnitude as the one obtained by UFSC. This was the case even for some matrices where pivoting cannot be suppressed, but can be restricted to the diagonal BLR blocks, such as perf008{d,ar,cr} (matrix ID 8-10). Only for problems perf009ar and nlpkkt{80,120} (matrix ID 11 and 21-22), standard threshold pivoting was needed to preserve accuracy and thus the restricted pivoting and UFCS results are not available. For these three matrices, we should instead use the UCFS variant, discussed in Section 2.3.2. We do not evaluate the multicore performance of this variant in this thesis.

low-rank threshold ε		10^{-6}					10^{-6}				10^{-6}	10^{-9}
matrix ID		12	13	14	15	16	17	18	19	20	21	22
flops ($\times 10^{12}$)	FR	4.7	13.8	1872.0	31.6	39.3	98.9	261.1	80.1	4066.0	15.1	248.4
	BLR	0.4	1.2	216.2	5.0	2.0	14.7	21.5	9.9	161.8	1.9	22.8
	+ LUAR	0.4	1.0	173.1	3.9	1.9	12.8	18.8	7.9	111.5	1.7	17.6
	+ UFCS	0.2	0.8	133.9	4.1	1.8	12.7	14.6	6.1	75.5	—	—
	flop ratio*	**27.0**	**17.1**	**14.0**	**7.7**	**22.3**	**7.8**	**17.9**	**13.1**	**53.8**	**7.8**	**10.9**
time (24 cores)	FR	31.0	36.7	2349.1	81.0	85.5	210.2	513.1	123.6	4930.4	54.9	592.2
	+ tree//	15.2	27.7	2247.9	60.8	65.7	168.5	420.3	114.8	4864.1	37.0	523.2
	+ rest. piv.	12.8	23.4	2237.5	56.9	60.2	158.1	388.7	109.6	4626.2	—	—
	BLR	26.8	28.8	1019.6	66.3	52.2	153.1	321.0	76.6	1393.8	42.1	334.2
	+ tree//	10.1	18.5	840.8	43.3	27.9	105.2	202.3	67.9	1321.1	22.9	263.3
	+ UFSC	10.2	16.6	729.1	36.8	29.0	87.7	161.9	58.8	817.1	21.7	226.3
	+ LUA	10.9	16.6	697.3	35.2	29.6	82.1	152.8	55.0	734.2	20.9	215.6
	+ LUAR	10.5	18.1	681.4	35.1	28.7	80.9	151.2	53.6	662.2	24.7	228.8
	+ UFCS	7.5	11.4	565.0	28.7	20.1	68.5	106.1	43.4	517.3	—	—
	time ratio*	**1.7**	**2.1**	**4.0**	**2.0**	**3.0**	**2.3**	**3.7**	**2.5**	**8.9**	**1.8**	**2.4**
speedup (24 cores)	Best FR	10.9	14.1	—	14.5	15.2	15.8	17.0	16.3	—	17.7	20.4
	Best BLR	5.1	7.1	12.4	7.8	7.1	10.1	8.6	8.6	10.4	12.4	13.6
scaled residual	Best FR	1.6e-16	1.5e-15	1.8e-14	1.2e-15	1.8e-16	2.4e-15	2.7e-16	1.6e-15	3.4e-15	6.6e-12	1.9e-08
	Best BLR	1.9e-09	1.4e-04	6.5e-08	7.9e-07	6.9e-07	1.1e-08	2.1e-08	2.0e-05	4.1e-05	6.4e-04	1.7e-04

*between best FR and best BLR

Table 5.12 – Experimental results on real-life matrices from the UFSMC

We now analyze how these algorithmic variants evolve with the size of the matrix, by comparing the results on matrices of different sizes from the same problem class, such as perf008{d,ar,cr} (matrix ID 8-10) or {5,7,10}Hz (matrix ID 1-3). Tree parallelism becomes slightly less critical as the matrix gets bigger, due to the decreasing weight of the bottom of the assembly tree. On the contrary, improving the efficiency of the BLR factorization (UFSC+LUA variant, with reduced memory transfers and increased granularities) becomes more and more critical (e.g., 16% gain on perf008d compared to 40% gain on perf008cr). Both the gains due to the Recompression (LUAR) and the Compress before Solve (UFCS) increase with the problem size (e.g., 20% gain on perf008d compared to 34% gain on perf008cr), which is due to the improved complexity of these variants (cf. Chapter 4).

We also analyze the parallel efficiency of the FR and BLR factorization by reporting the speedup on 24 threads. The speedup achieved in FR for the small and medium problems is of 16.4 in average and up to 20.4. As for the biggest problems, they would take too long to run in sequential in FR; this is indicated by a "—" in the corresponding row of Tables 5.11 and 5.12. However, for these problems, we can estimate the speedup assuming they would run at the same speed as the fastest problem of the same class that can be run in sequential. Under this assumption (which is conservative because the smaller problems already run very close to the CPU peak speed), these big problems all achieve a speedup close to or over 20. Overall, it shows that our parallel FR solver is a good reference to be compared with.

The speedups achieved in BLR are lower than in FR, but they remain satisfactory, averaging at 10.5 and reaching up to 13.8, and leading to quite interesting overall time ratios between the best FR and the best BLR variants. It is worthy to note that bigger problems do not necessarily lead to better speedups than smaller

ones, because they achieve higher compression and thus lower efficiency.

We summarize the main results of Tables 5.11 and 5.12 with a visual representation in Figure 5.5. We compare the time using 24 threads for four versions of the factorization: reference (ref.) FR and BLR, and improved (impr.) FR and BLR. Reference versions correspond to the initial versions of the factorization with only node parallelism, standard partial threshold pivoting and the standard FSCU variant for the BLR factorization. The improved FR version exploits tree parallelism and restricts numerical pivoting when possible. The improved BLR version additionally uses a UFCS factorization with accumulation (LUA), and possibly recompression (LUAR, only when beneficial). While the time ratio between the reference FR and BLR versions is only of 1.9 in average (and up to 6.9), that of the improved versions is of 4.6 in average (and up to 18.8).

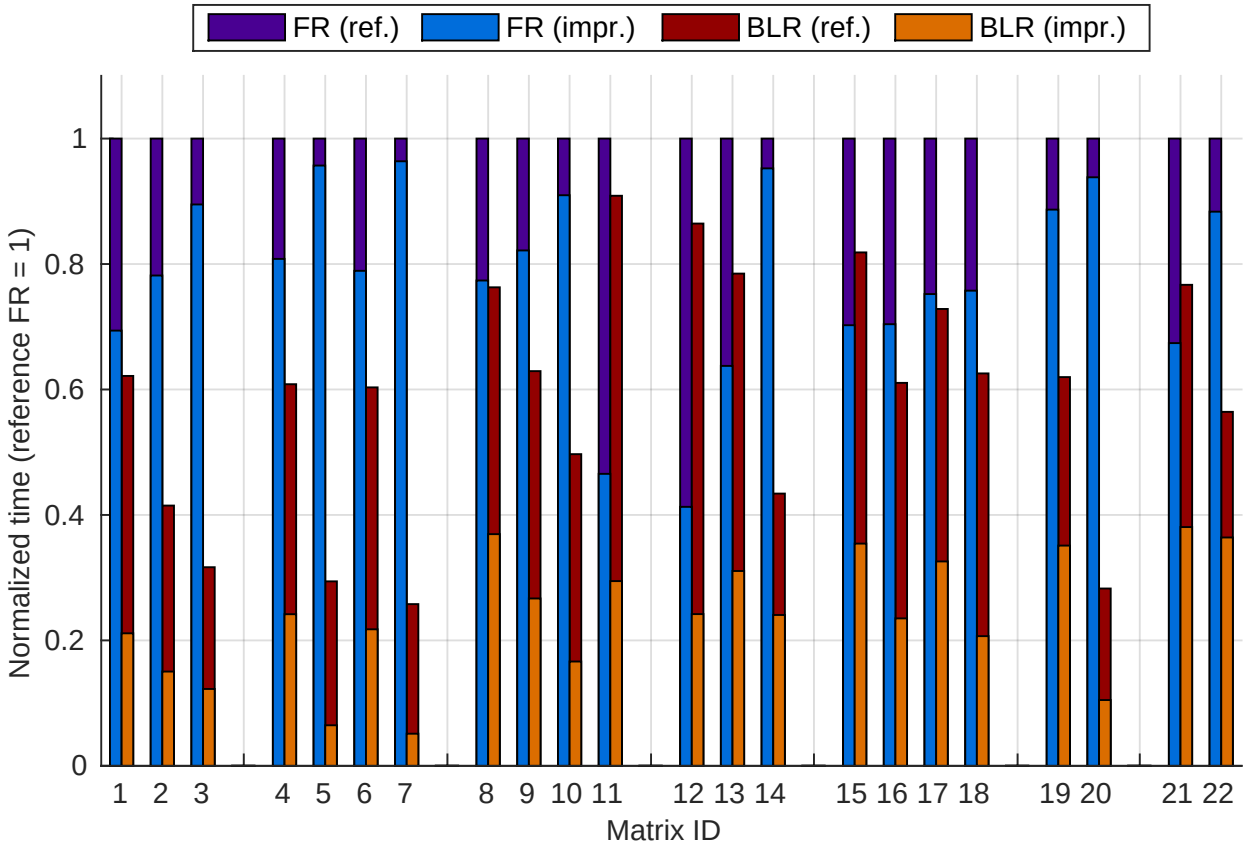


Figure 5.5 – Visual representation of summarized results of Tables 5.11 and 5.12 (ref.: reference; impr.: improved).

5.5.2 Results on 48 threads

Next, we report in Table 5.13 the results obtained using 48 threads on **brunch**. For these experiments, we have selected the biggest of our test problems: 10Hz, H17, S21, and perf008cr (matrix ID 3, 5, 7, and 10). On these big problems, we compute the speedup obtained using 48 threads with respect to 24 threads (and thus,

the optimal speedup value is 2). With the reference FR factorization, a speedup between 1.51 and 1.71 is achieved, which is quite satisfactory. The improved FR version, thanks to tree parallelism and restricted pivoting, increases the speedup to between 1.53 and 1.73, a relatively minor improvement.

	time (48 threads)				speedup w.r.t 24 threads			
matrix ID	3	5	7	10	3	5	7	10
FR	4100.5	5949.8	8387.0	1508.7	1.54	1.70	1.71	1.51
+ tree// + rest. piv.	3402.1	5599.0	7987.4	1350.4	1.66	1.72	1.73	1.53
BLR	1764.9	2478.3	3142.7	828.8	1.13	1.20	1.18	1.36
+ tree// + UFSC + LUA	1056.7	1071.3	1234.3	389.9	1.18	1.53	1.50	1.46
+ LUAR + UFCS	590.1	519.0	604.9	328.2	1.31	1.26	1.22	1.15
ratio best FR/best BLR	**6.1**	**10.8**	**13.2**	**4.1**				

Table 5.13 – Results on 48 threads.

The results are quite different for the BLR factorization. The speedup achieved by the reference version is much smaller: between 1.13 and 1.36, which illustrates that exploiting a high number of cores in BLR is a challenging problem. We then distinguish two types of improvements of the BLR factorization:

- The improvements that increase its scalability: tree parallelism but also the UFSC (i.e., Left-looking) factorization (due to a lower volume of memory accesses) and the LUA accumulation (due to increased granularities). All these changes combined lead to a major improvement of the achieved speedup, between 1.18 and 1.53, and illustrate the ability of the improved BLR factorization to scale reasonably well, even on higher numbers of cores.
- The improvements that increase its compression: the recompression (LUAR) and the UFCS factorization. By decreasing the number of operations, these changes may degrade the scalability of the factorization. This explains why the achieved speedup may be lower than that of the UFSC+LUA variant, or sometimes even that of the reference BLR version. Despite this observation, these changes do reduce the time by an important factor and illustrate the ability of the improved BLR factorization to achieve significant gains, even on higher numbers of cores.

5.5.3 Impact of bandwidth and frequency on BLR performance

In this Section, we report additional experiments performed on two machines and analyze the impact of their properties on the performance.

The machines and their properties are listed in Table 5.1. **brunch** is the machine used for all previous experiments. **grunch** is a machine with very similar architecture but with lower frequency and bandwidth.

In Table 5.14, we compare the results obtained on **brunch** and **grunch**. We report the execution time of the BLR factorization in Right-looking (RL), Left-looking

(LL), and with the LUA variant. On **brunch**, as observed and analyzed in Sections 5.3.3 and 5.4.1, the gain due to the LL factorization is significant while that of the LUA variant is limited. However, on **grunch**, we have the opposite effect, the difference between RL and LL is limited while the gain due to LUA is significant.

machine	RL	LL	LUA
brunch	220.5	174.7	167.1
grunch	247.7	228.3	196.8

Table 5.14 – Time (s) for BLR factorization on matrix S3 (on 24 threads on **brunch** and 28 threads on **grunch**).

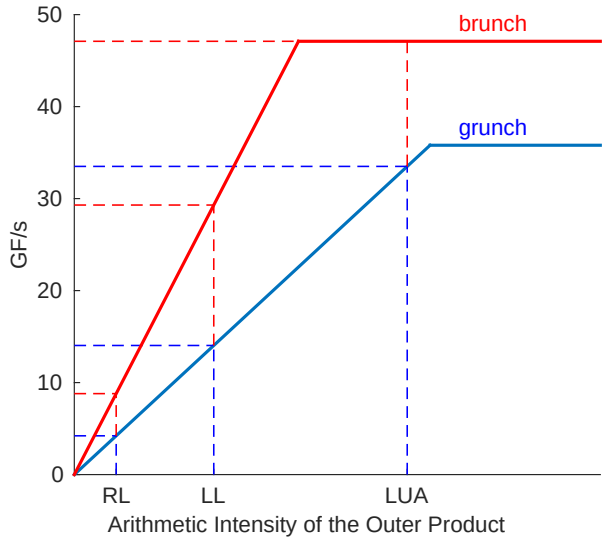


Figure 5.6 – Roofline model analysis of the Outer Product operation.

These results can be qualitatively analyzed using the Roofline Model (Williams, Waterman, and Patterson, 2009). This model provides an upper bound for the speed of an operation as a function of its arithmetic intensity, defined as the ratio between number of operations and number of memory transfers, the memory bandwidth and the CPU peak performance:

$$\begin{array}{c}\text{Attainable}\\ \text{GF/s}\end{array} = \min \left\{ \begin{array}{c} \begin{array}{c}\text{Peak Floating-point}\\ \text{Performance}\end{array} \\ \begin{array}{c}\text{Peak Memory}\\ \text{Bandwidth}\end{array} \times \begin{array}{c}\text{Operational}\\ \text{intensity}\end{array} \end{array} \right. .$$

The Roofline Model is plotted for the **grunch** and **brunch** machines in Figure 5.6 considering the bandwidth and CPU peak performance values reported in Table 5.1. Algorithms whose arithmetic intensity lies on the slope of the curve are commonly referred to as *memory-bound* because their performance is limited by the speed at which data can be transferred from memory; those whose arithmetic intensity lies on the plateau are referred to as *compute-bound* and can get close to the peak CPU speed.

Although it is very difficult to compute the exact arithmetic intensity for the algorithms presented above, the following relative order can be established:

- because of the unsuitable data access pattern (as explained in Section 5.3.3) and the low granularity of operations, the RL method is memory bound as proved by the fact that the Outer Product operation runs, on **brunch**, at the poor speed of 8.8 GF/s;
- as explained in Section 5.3.3, the LL method does the same operations as the RL one but in a different order which results is a lower volume of memory transfers. Consequently, the LL method enjoys a higher arithmetic intensity although it is still memory bound as shown by the fact that the Outer Product operation runs, on **brunch**, at 29.3 GF/s (cf. Table 5.8) which is still relatively far from the CPU peak;
- the LUA method is based on higher granularity operations; this likely allows for a better use of cache memories within BLAS operations which ultimately results in an increased arithmetic intensity; in conclusion the LUA method is compute-bound (or very close to) as shown by the fact that the Outer Product runs at 44.7 GF/s on **brunch** (cf. Table 5.8).

This leads to the following interpretation of the results of Table 5.14. Compared to **grunch**, **brunch** has a higher bandwidth; this translates by a steeper curve in the memory-bound area of the roofline figure. As a consequence, the difference between the RL and LL algorithms (which are both memory-bound) is greater on **brunch** than on **grunch**. However, the higher bandwidth also makes the LL factorization closer to being compute-bound on **brunch** than on **grunch**. Therefore, the difference between LL and LUA (for which the Outer Product is compute-bound) is greater on **grunch**.

5.6 Chapter conclusion

We have presented a multithreaded Block Low-Rank factorization for shared-memory multicore architectures.

We have first identified challenges of multithreaded performance in the use of BLR approximations within multifrontal solvers. This has motivated us to both revisit the algorithmic choices of our Full-Rank Right-looking solver based on node parallelism, and also to introduce algorithmic variants of the BLR factorization.

Regarding the algorithmic changes for the FR factorization, even though exploiting tree parallelism brings only a marginal gain in FR, we have shown that it is critical for the BLR factorization. This is because the factorization of the fronts at the bottom of the assembly tree is of much higher weight in BLR. We have then observed that, contrarily to the FR case, the Left-looking BLR factorization outperforms the Right-looking one by a significant factor. We have shown that it is due to a lower volume of memory transfers.

Regarding the BLR algorithmic variants, firstly we have shown that accumulating together the low-rank updates (so-called LUA algorithm) improves the granularity and the performance of the BLAS kernels. This approach also offers potential

for recompression (so-called LUAR algorithm) which can often be translated into time reduction. Secondly, for problems on which the constraint of numerical pivoting can be relaxed, we have presented the UFCS variant which improves both the efficiency and compression rate of the factorization.

An efficient multithreaded BLR factorization for multicores is the first building block of a scalable, parallel BLR solver. The next chapter deals with the extension to distributed-memory architectures.

CHAPTER

6 Distributed-memory BLR Factorization

In this chapter, we analyze the distributed-memory BLR factorization. We extend the algorithms presented previously to distributed-memory environments, and analyze the new issues raised in this context.

In Section 6.1, we describe our MPI implementation and how it is adapted to perform a BLR factorization. Then, in Section 6.2, we provide a strong scalability analysis of the standard FSCU factorization. The analysis raises a number of issues among which a high relative weight of communications, and load unbalance. In Sections 6.3 and 6.4, we propose some ways to overcome these issues. Then, in Sections 6.5 and 6.6, we present and analyze distributed-memory versions of the LUAR and UFCS factorization variants, introduced in Chapter 2. As in the previous chapter, we focus on one matrix (10Hz, matrix ID 3) in our analysis, before presenting results on our main set of problems (described in Section 1.5.2.2) in Section 6.7. We conclude in Section 6.8.

The **eos** supercomputer was used to run all experiments presented in this chapter, with the exception of one experiment of Section 6.3.2 which was run on **cori**.

6.1 Parallel MPI framework and implementation

In this section, we first describe a general MPI framework in which the algorithms and analysis presented in this chapter are valid. We then specify our implementation in MUMPS, and explain how it can be adapted to perform a BLR factorization.

6.1.1 General MPI framework

As explained in Section 1.3.2.4, two kinds of parallelism, referred to as tree parallelism and node parallelism, can be exploited. We consider a standard framework in the literature, where both types of parallelism are exploited.

This is illustrated in Figure 6.1. All processes are mapped on the root node of the assembly tree. Then, as we go down the tree, the processes are distributed on the child nodes, until each MPI process is left with one or more subtrees to process sequentially. There are several strategies to assign the processes to the child nodes.

In the following, we consider, without loss of generality, a proportional mapping strategy (Pothen and Sun, 1993).

In strict proportional mapping, the processes are recursively distributed over the assembly tree according to the following rule. Consider p_f processes assigned to the factorization of a given front f with n_c children. Denoting w_i the weight of the subtree rooted at child i, the number of processes assigned to child i is

$$p_i = \frac{w_i}{\sum_{j=1}^{n_c} w_j} p_f. \tag{6.1}$$

The metric used to compute the weights w_i is usually the workload (or sometimes the memory load).

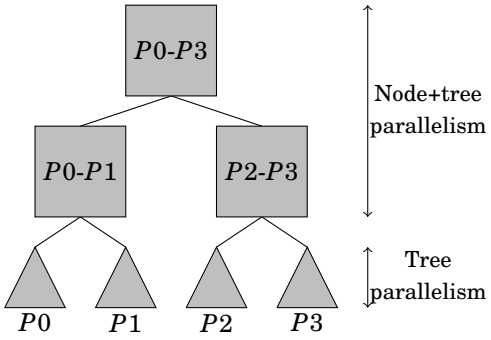


Figure 6.1 – Illustration with four MPI processes of our framework based on both tree and node parallelism.

We then describe in Figure 6.2 how node parallelism is handled in our framework. The MPI processes are mapped on a node following a one-dimensional row-wise partitioning. The mapping of the fully-summed rows (shaded area in Figure 6.2) is 1D-cyclic, so as to allow for the pipelining of computations. The mapping of the non fully-summed rows can also be cyclic or acyclic.

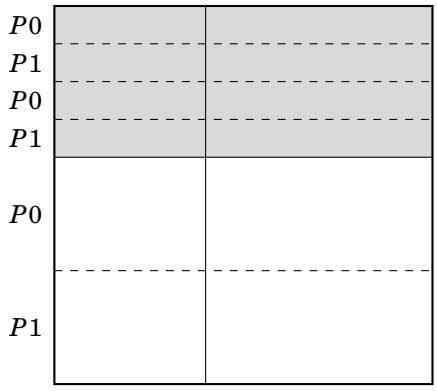


Figure 6.2 – Illustration with two MPI processes of how node parallelism is handled in our framework. The shaded area represents the fully-summed rows.

For the sake of simplicity, we base our analysis on the unsymmetric case. Results on symmetric matrices will be included in Section 6.7.

6.1.2 Implementation in MUMPS

The MPI parallelism designed in MUMPS is described in detail in Amestoy et al. (2001) and Amestoy et al. (2006). We summarize the most important aspects that are relevant in our context.

Both levels of parallelism are exploited with MPI, as illustrated by Figure 6.3. Fronts in different subtrees are processed by different processes, and the large enough ones are mapped on several processes: 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.

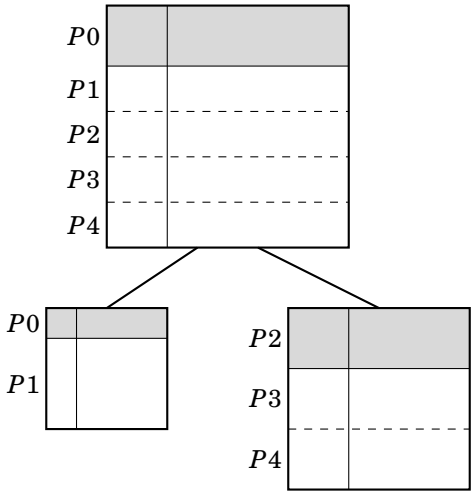


Figure 6.3 – Illustration of tree and node parallelism. The shaded part of each front represents its fully-summed rows. The fronts are row-wise partitioned in our implementation, but column-wise partitioning is also possible.

The fact that all the fully-summed rows are mapped on the same process simplifies many algorithmic aspects on the code, but would also lead to severe load unbalance. To overcome this issue, the nodes are *split* into chains so that the load of the master processes is roughly the same as that of the slaves, as illustrated in Figure 6.4. This splitting strategy is described in Rouet (2012) and Sid-Lakhdar (2014). We revisit it in Section 6.4, when analyzing the load balance of the BLR factorization.

There are two main types of messages that carry numerical values: the sending of LU factors between processes mapped on the same front, and the sending of a range of rows of the CB of a front to the processes mapped on the parent front. This is illustrated in Figure 6.5. The shaded area corresponds to the part of the matrix that is sent, and the arrows represent at least a message (and more commonly several messages, as the data to be sent is usually split into several smaller messages). The dashed arrows represent logical messages, which are not actually sent because the sender and receiver correspond to the same process. Note that the master processes of the child fronts may also have to send a message if they hold delayed pivots (non-eliminated variables due to numerical pivoting).

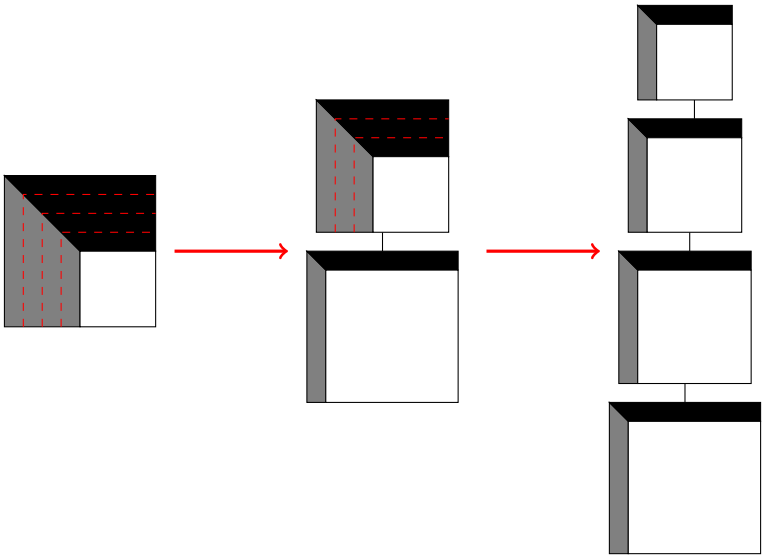
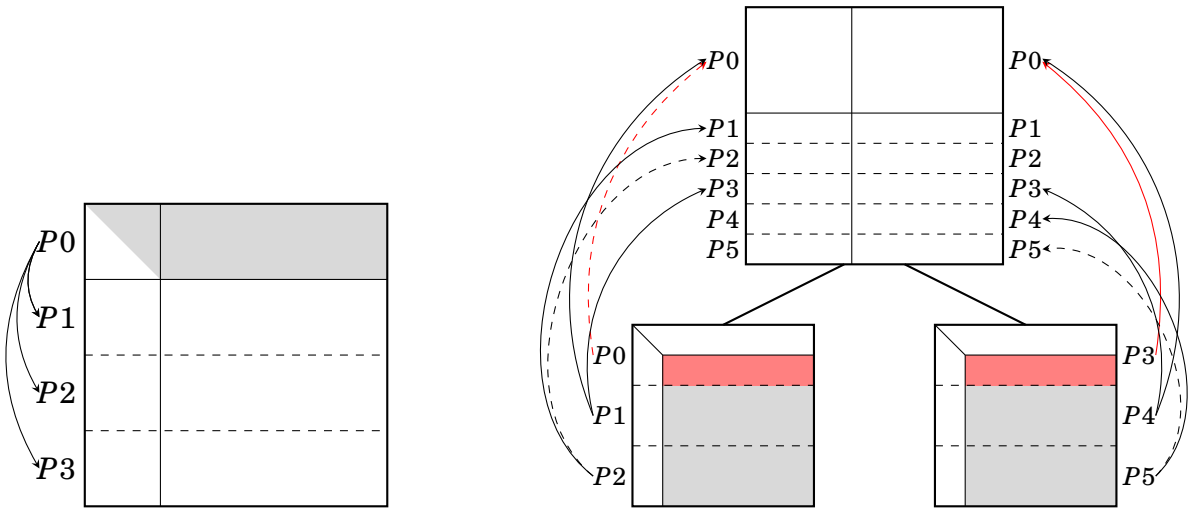


Figure 6.4 – Splitting of a front (left) into a split chain (right) to be processed bottom-up. L and U factors are in gray and black respectively, and Schur complements in white.



(a) LU messages. (b) CB messages.

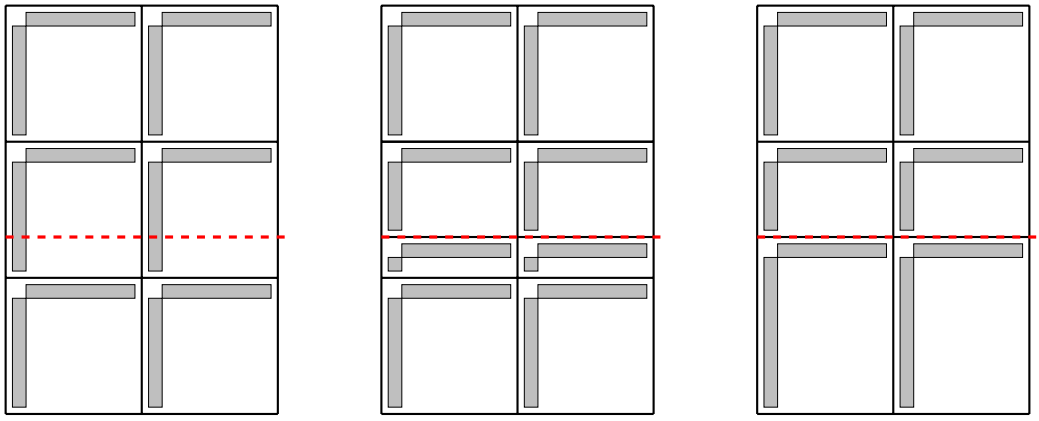
Figure 6.5 – The two main types of messages sent during the factorization. The shaded part represents the content of the messages. The shaded red part (right figure) represents delayed pivots.

Finally, note that both the master and slave processes perform a right-looking factorization in our current default implementation.

6.1.3 Adapting this framework/implementation to BLR factorizations

When considering a BLR front mapped on several processes, there are two options: either the MPI mapping is constrained to follow the BLR clustering, or the

inverse. Following Weisbecker (2013), we have chosen the latter option: the BLR clustering is first computed independently from the MPI partitioning; then, the BLR blocks mapped on more than one process are split into several blocks. If this produces too small blocks[1], we merge some blocks together. This is illustrated in Figure 6.6.



(a) BLR clustering computed independently of the MPI partitioning (dashed red line).

(b) BLR clustering constrained by the MPI partitioning.

(c) BLR clustering effectively used. Small blocks have been merged so as to obtain big enough blocks.

Figure 6.6 – BLR clustering constrained by MPI partitioning.

Similarly, the splitting of frontal matrices can be performed either before or after the BLR clustering. In our implementation, we perform it before; the separators are first split into subparts, which we then cluster independently. This assumes that the splitting strategy is aware of the graph, so as to avoid producing disconnected subparts (which would lead to a clustering of poor quality).

In the BLR factorization, the volume of the LU messages can be reduced thanks to the compression of the LU factors. Moreover, the volume of CB messages may also be reduced by compressing the CB, as described in Section 2.5. So far, we have not considered compressing the CB, because it does not contribute to reducing the global number of operations, and thus represents an overhead cost. However, in this chapter, we will compare the full-rank CB (noted CB_{FR}) and low-rank CB (noted CB_{LR}) strategies to assess the impact of compressing the CB on the volume of communications and on the overall performance of the solver.

Note that when the CB is compressed, two strategies are possible to perform the assembly operations, as described in Section 2.5. The assembly can be performed in full-rank by decompressing the CB low-rank blocks (CB_{LR}+Asm_{FR} strategy) or it can be performed in low-rank by padding and recompressing them (CB_{LR}+Asm_{LR} strategy, described in Section 1.4.3.2). In this work, we only consider the CB_{LR}+Asm_{FR} strategy.

When the CB is compressed, some CB low-rank blocks may be needed to assemble rows mapped on different processes on the parent front, as illustrated in

[1] In our implementation, too small is defined as less than half the target cluster size.

Figure 6.7. In this situation, the column basis Y must be sent to each process holding at least a row concerned by the block. Thus, if there are p such processes, a block of size $m \times n$ and rank k requires $\mathcal{O}((m+np)k)$ data to be sent.

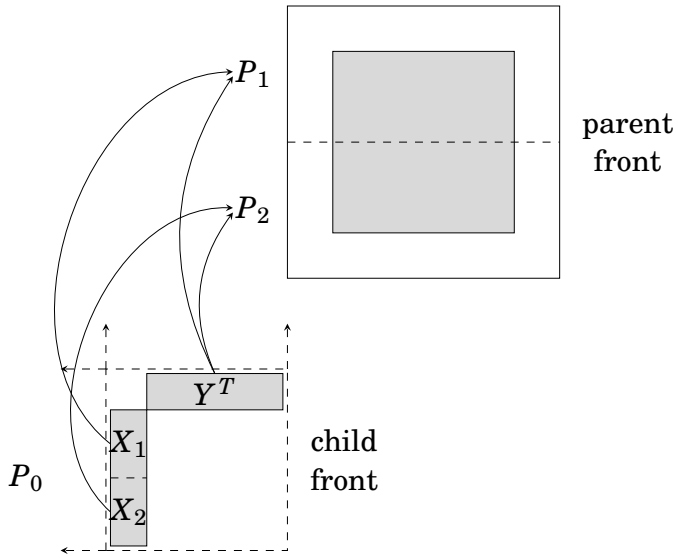


Figure 6.7 – CB block mapped on two processes on the parent front. The BLR block $\begin{pmatrix} X_1 \\ X_2 \end{pmatrix} Y^T$ is mapped on P_1 and P_2 on the parent front. P_0 must send X_1 to P_1, X_2 to P_2, and Y to both P_1 and P_2.

6.2 Strong scalability analysis

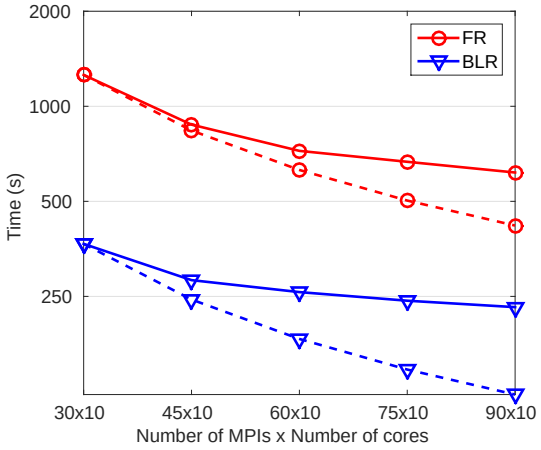
In Figure 6.8, we provide a strong scalability analysis of both the FR and BLR factorizations. This section focuses on the FSCU factorization variant, using the CB_{FR} strategy (i.e., the CB is not compressed).

We use the 10Hz matrix on an increasing number of nodes (from 30, the minimal number for the problem to fit in-core, to 90, the maximal number available). The number of threads is fixed to 10 per node. While both FR and BLR scale reasonably well, the FR strong scalability is clearly better than the BLR one.

To understand why, we provide an analysis of different metrics in Table 6.1: flops, load balance, memory consumption, and volume of communications. To estimate the quality of the load balance, we compute the flop ratio between the most loaded process and the least loaded one.

The analysis of these metrics raises the following issues:

Issue 1 (degradation of the compression rate): the compression rate slightly decreases with the number of processes. This is because the row-wise partitioning imposed by the distribution of the front onto several processes constraints the BLR clustering of the unknowns, as we want to ensure a given BLR block belongs to one process only (cf. Section 6.1.3). While this may become an issue on much larger numbers of processes, in this case it cannot explain the



(a) Strong scalability figure. Dashed lines represent ideal scalability.

	30×10	45×10	60×10	75×10	90×10
FR	1257.2	874.8	722.5	667.2	617.0
BLR	366.6	281.5	258.0	242.3	231.2
ratio	3.4	3.1	2.8	2.8	2.7

(b) Associated table.

Figure 6.8 – Strong scalability of the FR and BLR factorizations (10Hz matrix).

	30×10			90×10		
	FR	BLR	ratio	FR	BLR	ratio
time (s)	1257.2	366.6	3.4	617.0	231.2	2.7
flops ($\times 10^{14}$)	25.56	1.98	12.9	25.56	2.00	12.8
load unbalance	1.42	2.01		1.28	2.57	
communications (GB)	2060.3	935.6	2.2	5735.9	2587.2	2.2
avg. memory/proc (GB)	35.22	25.92	1.4	13.27	10.82	1.2

Table 6.1 – Analysis of the strong scalability of Figure 6.8 on 10Hz matrix (matrix ID 3).

lesser strong scalability of BLR, as the degradation of the compression rate is negligible (1.98 vs 2.00×10^{14} flops). We do not consider this issue further.

Issue 2 (higher relative weight of communications): both the flops and the volume of communications are reduced in BLR. However, while the flops are reduced by a factor around 12.8, the volume of communications is only reduced by a factor 2.2. Thus, the ratio of volume of communications over flops is much higher in BLR than in FR. We provide a theoretical and experimental communication analysis of the BLR factorization in Section 6.3.

Issue 3 (higher load unbalance): in our current implementation, the mapping and splitting strategies are based on the full-rank flops estimates, even for the BLR factorization. This is because the compression rate cannot be predicted in advance. As a consequence, the load balance in BLR is worse than in FR, with a significantly higher maximal load over minimal load ratio. We analyze the load balance in Section 6.4.

Issue 4 (lower memory efficiency): in FR, the average memory consumption per process decreases from 35.22 GB on 30 processes to 13.27 GB on 90 processes. It

is thus divided by a factor 2.65, which corresponds to a memory efficiency of 0.88 with respect to 30 processes. However, in BLR, the average memory is only divided by a factor 2.40, which corresponds to an efficiency of 0.80. We will analyze the reason behind this and how to improve the memory efficiency of the BLR factorization in Section 9.3.

Thus, **Issues 2** and **3** may both contribute to the lesser strong scalability observed for the BLR factorization. Moreover, **Issue 4** will limit the scope of BLR solvers on very large problems. In the next two sections, we analyze the **Issues 2** and **3**, and suggest some ways to address them. We will come back to **Issue 4** in Chapter 9.

6.3 Communication analysis

In this section, we analyze and compare the communications performed in the FR and BLR distributed-memory factorizations. This analysis if motivated by **Issue 2**: in the BLR factorization, the volume of communications has only been reduced by a factor of 2.2, which is relatively small with respect to the flops reduction factor (12.8).

In Table 6.2, we measure the data sent for each type of message. We focus on the *LU* and CB messages, since the data sent for other messages is negligible.

	30×10			90×10		
	FR	BLR	ratio	FR	BLR	ratio
LU messages	1219.3	100.1	12.2	3488.3	339.4	10.3
CB messages	840.7	835.2	1.0	2245.6	2245.8	1.0
other messages	0.3	0.3	1.0	2.0	2.0	1.0
total	2060.3	935.6	2.2	5735.9	2587.2	2.2

Table 6.2 – Volume of communications (in GB) for the FR and BLR factorizations on matrix 10Hz (matrix ID 3). In BLR, the CB is not compressed (CB_{FR} strategy).

In FR, the *LU* and CB messages represent a comparable part of the total volume of communications. The *LU* messages tend to dominate, especially when the number of processes grows. This will be explained in Section 6.3.1.

In BLR, the *LU* factors are compressed and therefore the volume of *LU* communications is reduced by a factor 12.2 on 30×10 cores. This might seem surprising at first because the compression rate of the factors is only of 22% on this matrix. However, this is the global compression rate (with respect to all compressed fronts), while most of the messages actually concern the bigger fronts at the top of the assembly tree, which are large enough to be mapped on several processes, and which are also likely to compress more. This also explains why the reduction factor of the *LU* messages decreases to 10.3 on 90×10 cores: with more processes, more fronts that are lower in the assembly tree, which compress less, are mapped on several processes.

In BLR, compressing the LU factors is thus not enough since the LU messages become negligible with respect to the total, and therefore the communications are largely dominated by the CB messages, which represent almost 90% of the total.

The total volume of communications is thus reduced by the underwhelming factor of 2.2. In particular, this means that the ratio between the number of operations and the volume of communications decreases compared to the FR factorization: on 900 cores, the FR factorization performs 446 GF for each GB of communication, whereas that value falls to 77 in BLR. This may explain the lesser scalability of the BLR factorization.

Therefore, compressing the CB deserves to be considered as a strategy to improve the parallel efficiency of the solver. Before providing experimental results in Section 6.3.2, we first motivate this idea with the following theoretical analysis.

6.3.1 Theoretical analysis of the volume of communications

For the purpose of this theoretical analysis, we make some stronger assumptions on our setting. We assume the assembly tree has been built by means of nested dissection. We use p processes mapped with a strict proportional mapping strategy (Pothen and Sun, 1993). We consider a domain of size N^d (where d denotes the dimension) and assume both N and p are powers of 2. We also assume $N \geq p$. We note $n = N^d$.

The LU and CB messages are of different type. The sending of the CB is a *many-to-many* communication (where "many" refers to all processes involved in the communication), while that of the LU factors is a *one-to-many* communication (a broadcast). There are two ways to measure the cost of a communication: the total volume $\mathcal{W}_{tot}$, and the critical volume $\mathcal{W}_{cri}$. $\mathcal{W}_{tot}$ is the total number of words that have to be transferred; $\mathcal{W}_{cri}$ is the number of words that have to be transferred along the critical path (representing communications as a graph).

In a one-to-many communication, the cost for a processor to send w words to p processors is $\mathcal{W}_{tot} = wp$ and $\mathcal{W}_{cri} = wp$, except if broadcasts follow a tree-based implementation, in which case $\mathcal{W}_{cri}$ is reduced to $w \log p$. We do not consider tree-based broadcasts in our analysis (nor in our current implementation), but mention how their use would influence the results at the end of this section. On the other hand, in a many-to-many communication, the cost for p processors to send w words each is $\mathcal{W}_{tot} = wp$ and $\mathcal{W}_{cri} = w$.

We begin by analyzing the communications performed in our two dense kernels: the partial LU factorization of a front and the sending of the CB of a front (i.e. the two communications depicted in Figure 6.5). We consider a frontal matrix of order m distributed over p processes following a 1D row-wise partitioning, as described in our framework (cf. Figure 6.2). We assume that m is a multiple of p and that the partitioning is perfectly regular, i.e. each process holds exactly m/p rows.

In FR, the LU partial factorization and the sending of the CB require one-to-many and many-to-many communications of $\mathcal{O}(m^2)$ data, respectively. Therefore, $\mathcal{W}_{tot}^{LU} = \mathcal{W}_{cri}^{LU} = \mathcal{O}(m^2 p)$, $\mathcal{W}_{tot}^{CB} = \mathcal{O}(m^2)$, and $\mathcal{W}_{cri}^{CB} = \mathcal{O}(m^2/p)$. Thus, $\mathcal{W}_{tot} = \mathcal{W}_{tot}^{LU} + \mathcal{W}_{tot}^{CB}$ is dominated by the LU messages, which transfer $\mathcal{O}(p)$ more words than the CB messages. This explains the results of Table 6.2. The weight of the LU messages is even more important in terms of critical volume ($\mathcal{O}(p^2)$ more words transferred).

This remark should be slightly attenuated by the fact that the ratio $\mathcal{W}_{tot}^{CB}/\mathcal{W}_{tot}^{LU}$ is proportional to $c = m_{nfs}/m_{fs}$, the ratio between the number of non fully-summed and fully-summed rows (c is usually of order $\mathcal{O}(10)$ in 3D nested dissection on a regular problem).

In BLR, we know from Chapter 4 that the factor size is reduced from $\mathcal{O}(m^2)$ to $\mathcal{O}(m^{1.5}\sqrt{r})$. Therefore, we obtain $\mathcal{W}_{tot}^{LU} = \mathcal{W}_{cri}^{LU} = \mathcal{O}(m^{1.5}\sqrt{r}p)$. If we compress the CB, and assuming the ranks of the CB blocks are also of order $\mathcal{O}(r)$, we obtain $\mathcal{W}_{tot}^{CB} = \mathcal{O}(m^{1.5}\sqrt{r})$ and $\mathcal{W}_{cri}^{CB} = \mathcal{O}(m^{1.5}\sqrt{r}/p)$.

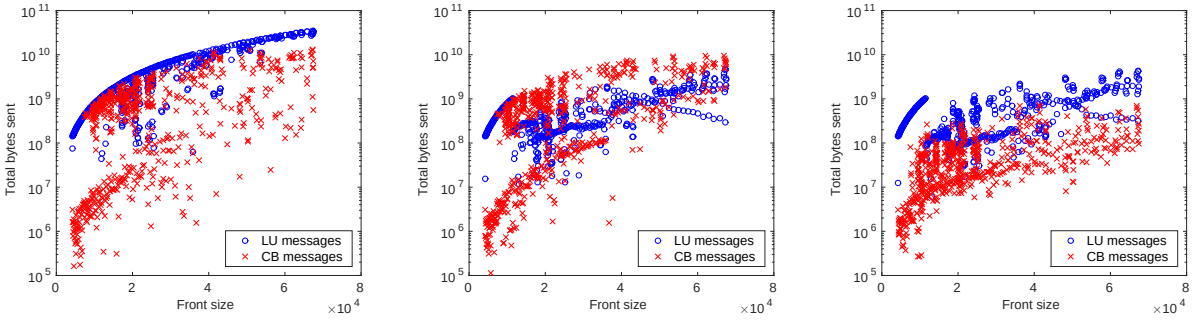
We summarize in Table 6.3 the total and critical volumes in FR, in BLR without compressing the CB (CB_{FR}), and in BLR compressing the CB (CB_{LR}).

	$\mathcal{W}_{tot}^{LU}(m,p)$	$\mathcal{W}_{tot}^{CB}(m,p)$	$\mathcal{W}_{cri}^{LU}(m,p)$	$\mathcal{W}_{cri}^{CB}(m,p)$
FR	$\mathcal{O}(m^2p)$	$\mathcal{O}(m^2)$	$\mathcal{O}(m^2p)$	$\mathcal{O}(m^2/p)$
BLR (CB_{FR})	$\mathcal{O}(m^{1.5}\sqrt{r}p)$	$\mathcal{O}(m^2)$	$\mathcal{O}(m^{1.5}\sqrt{r}p)$	$\mathcal{O}(m^2/p)$
BLR (CB_{LR})	$\mathcal{O}(m^{1.5}\sqrt{r}p)$	$\mathcal{O}(m^{1.5}\sqrt{r})$	$\mathcal{O}(m^{1.5}\sqrt{r}p)$	$\mathcal{O}(m^{1.5}\sqrt{r}/p)$

Table 6.3 – Theoretical number of words transferred in total ($\mathcal{W}_{tot}$) and along the critical path ($\mathcal{W}_{cri}$), for the LU and CB communications on a single front of order m mapped on p processes.

Let us now compute the total and critical volume of the entire multifrontal factorization, across all fronts in the assembly tree.

Similarly to the complexity computations in Chapter 4, we assume the separators are cross-shaped, without loss of generality. In this case, the number of mapped processes per front is divided by 2^d at each level of the tree, whereas the front size is divided by 2^{d-1}.



(a) FR factorization (b) BLR factorization (CB_{FR}) (c) BLR factorization (CB_{LR})

Figure 6.9 – Volume of communications as a function of the front size (matrix 10Hz).

To simplify the computations, we consider the 3D case ($d = 3$). We will also provide results for the 2D case, but leave out the associated computations for the sake of conciseness.

We begin by the computation of the total volume.

In the FR case, the total number of words $\mathcal{W}_{tot}(\ell)$ transferred at level $\ell < \log_8 p$ of the tree can be computed as

$$\mathcal{W}_{tot}(\ell) = 2^{3\ell}\mathcal{W}_{tot}\left(\frac{N^2}{2^{2\ell}}, \frac{p}{2^{3\ell}}\right) = 2^{3\ell}\left(\frac{N^4}{2^{4\ell}} * \frac{p}{2^{3\ell}} + \frac{N^4}{2^{4\ell}}\right) = \frac{N^4 p}{2^{4\ell}} + \frac{N^4}{2^{\ell}}. \tag{6.2}$$

Thus, the relative weight of the LU term ($\frac{N^4 p}{2^{4\ell}}$) with respect to the CB term ($\frac{N^4}{2^{\ell}}$) decreases when ℓ increases. Nevertheless, the LU term remains dominant as long as $\ell < \log_8 p$, i.e. for fronts mapped on more than one process. Using our example, we plot in Figure 6.9a the volume of communications for each front as a function of its size, and compare the volume for LU and for CB messages. The experiments match very well the theory: even though their relative weight gets smaller with the front size, the LU messages dominate for all fronts. This leads to the total cost over all the levels

$$\mathcal{W}_{tot}(n,p) = \sum_{\ell=0}^{\log_8 p} \mathcal{W}_{tot}(\ell) = \mathcal{O}(n^{4/3}p). \tag{6.3}$$

Let us now consider the BLR CB_{FR} case. First, it is interesting to estimate the front size for which the CB messages start to dominate over the LU ones in BLR CB_{FR}. We must have $m > p^2 r$. In our example, $p = 90$; our problem comes from the Helmholtz equation and therefore r is expected to behave as $\mathcal{O}(\sqrt{m})$. Thus, the CB communication becomes dominant for fronts of size $90^4 \approx 65$ million. At first glance, it would thus seem that the LU communication remains dominant for all practical sizes. However, as previously mentioned, we should also take into account the ratio c: the previous computation now yields $(90/c)^4 = \mathcal{O}(10^4)$, a much smaller front size which is largely attained in our problem (the biggest front is of order 67,500).

The following computations give further insight:

$$\mathcal{W}_{tot}(\ell) = 2^{3\ell}\mathcal{W}_{tot}\left(\frac{N^2}{2^{2\ell}}, \frac{p}{2^{3\ell}}\right) = 2^{3\ell}\left(\frac{N^3\sqrt{r}}{2^{3\ell}} * \frac{p}{2^{3\ell}} + \frac{N^4}{2^{4\ell}}\right) = \frac{N^3\sqrt{r}p}{2^{3\ell}} + \frac{N^4}{2^{\ell}}. \tag{6.4}$$

Now, the CB term dominates the LU one when $\ell \geq \log_4 \frac{\sqrt{r}p}{N}$, which is quite small (or even negative!). Thus, in BLR CB_{FR}, the CB term dominates the LU one for most (or all) fronts in the tree. The total cost over all the levels is

$$\mathcal{W}_{tot}(n,p) = \sum_{\ell=0}^{\log_8 p} \mathcal{W}_{tot}(\ell) = \mathcal{O}(n\sqrt{r}p + n^{4/3}). \tag{6.5}$$

Finally, we consider the BLR CB_{LR} case. We have

$$\mathcal{W}_{tot}(\ell) = 2^{3\ell}\mathcal{W}_{tot}\left(\frac{N^2}{2^{2\ell}}, \frac{p}{2^{3\ell}}\right) = 2^{3\ell}\left(\frac{N^3\sqrt{r}}{2^{3\ell}} * \frac{p}{2^{3\ell}} + \frac{N^3\sqrt{r}}{2^{3\ell}}\right) = \frac{N^3\sqrt{r}p}{2^{3\ell}} + N^3\sqrt{r}. \tag{6.6}$$

We are back to the case where the LU term dominates for all fronts mapped on more than one process ($\ell < \log_8 p$), regardless of the rank. Experimental results on our example (Figure 6.9c) match again very well this theoretical result (we further discuss the experimental results of the BLR CB_{LR} factorization in the next section).

The total cost over all the levels is

$$\mathcal{W}_{tot}(n,p) = \sum_{\ell=0}^{\log_8 p} \mathcal{W}_{tot}(\ell) = \mathcal{O}(n\sqrt{r}p). \tag{6.7}$$

We summarize the value of $\mathcal{W}_{tot}(n,p)$ in Table 6.4. We now turn to the computation of the critical volume.

The critical volume can be computed as the sum of the dense critical volumes associated with each front on the critical path of the assembly tree. Since we consider a regular problem, this simply corresponds to any branch of the tree. Thus, $\mathcal{W}_{cri}(n,p)$ can be computed by the same formulas as $\mathcal{W}_{tot}(n,p)$ where the $2^{3\ell}$ factor has been removed (to only count one front per level, i.e. one branch).

This leads to the results reported in Table 6.4. Because the LU volume dominates in the FR and BLR CB_{LR} cases, and since the LU total and critical volumes are the same, the sparse critical volume $\mathcal{W}_{cri}(n,p)$ is thus the same as its total counterpart in these two cases. However, in the BLR CB_{FR} case, the CB volume may dominate, and thus the critical volume may be lower than the total one. As a consequence, compressing the CB may reduce the total volume but not necessarily the critical one, depending on whether $n^{4/3}/p$ still dominates over the LU volume $n\sqrt{r}p$. We will come back to this key observation in the next section.

	2D ($d=2$)		3D ($d=3$)	
	$\mathcal{W}_{tot}(n,p)$	$\mathcal{W}_{cri}(n,p)$	$\mathcal{W}_{tot}(n,p)$	$\mathcal{W}_{cri}(n,p)$
FR	$\mathcal{O}(np)$	$\mathcal{O}(np)$	$\mathcal{O}(n^{4/3}p)$	$\mathcal{O}(n^{4/3}p)$
BLR (CB_{FR})	$\mathcal{O}(n^{3/4}\sqrt{r}p + n\log p)$	$\mathcal{O}(n^{3/4}\sqrt{r}p + n)$	$\mathcal{O}(n\sqrt{r}p + n^{4/3})$	$\mathcal{O}(n\sqrt{r}p + n^{4/3}/p)$
BLR (CB_{LR})	$\mathcal{O}(n^{3/4}\sqrt{r}p)$	$\mathcal{O}(n^{3/4}\sqrt{r}p)$	$\mathcal{O}(n\sqrt{r}p)$	$\mathcal{O}(n\sqrt{r}p)$

Table 6.4 – Theoretical number of words transferred in total ($\mathcal{W}_{tot}$) and along the critical path ($\mathcal{W}_{cri}$), for the overall multifrontal factorization of a sparse matrix of order $n = N^d$.

The conclusion of this communication analysis is that compressing the CB is necessary to truly reduce the volume of communications in the BLR factorization, because the CB messages become dominant on most levels of the assembly tree.

Note that if we used instead a tree-based broadcast implementation, such as the one described by Rouet (2012) and Sid-Lakhdar (2014), the cost of sending the LU messages to p processors would be reduced (the factor p would be replaced by $\log p$ in all the formulas), which would even further motivate the compression of the CB in the BLR factorization.

6.3.2 Compressing the CB to reduce the communications

We now return to our experimental example and analyze how compressing the CB can reduce the volume of communications and influence the overall performance of the solver.

We first analyze the overhead cost of the CB compression in terms of flops. In Table 6.5, we report an overhead of 11% and 4% on 30 and 90 processes, respec-

tively. This is a relatively small cost composed of two parts: the actual compression of the CB, and the decompression of the CB at the assembly, which is performed in FR (i.e. the so-called CB_{LR}+Asm_{FR} strategy described in Section 2.5). The latter term only represents 1% of the overhead, which is thus dominated by the actual compression.

	30×10			90×10		
	CB_{FR}	CB_{LR}	%	CB_{FR}	CB_{LR}	%
flops ($\times 10^{14}$)	1.98	2.20	+11%	2.00	2.09	+4%
total communications (GB)	935.6	177.0	-81%	2587.2	592.0	-77%
time (s)	358.7	363.4	+1%	229.2	282.5	+23%

Table 6.5 – Performance comparison of the BLR factorization with (CB_{LR}) and without (CB_{FR}) compression of the CB, on matrix 10Hz (matrix ID 3).

The average compression rate of the CB blocks is 11.2%. It is thus higher than the factors compression rate (around 22%), which is possibly due to the partially summed nature of the CB entries. As reported in Table 6.6, this CB compression results in a reduction of the volume of the CB messages, by a factor of 11.9 and 8.2 on 30 and 90 processes, respectively.

	30×10			90×10		
	CB_{FR}	CB_{LR}	ratio	CB_{FR}	CB_{LR}	ratio
LU messages	100.1	106.7	0.9	339.4	314.7	1.1
CB messages	835.2	70.0	11.9	2245.8	275.2	8.2
other messages	0.3	0.3	1.0	2.0	2.0	1.0
total	935.6	177.0	5.3	2587.2	592.0	4.4

Table 6.6 – Volume of communications (in GB) of BLR factorization with (CB_{LR}) and without (CB_{FR}) compression of the CB.

On 30 processes, the reduction factor of the communications is thus higher than the compression rate. This is a similar result as for the *LU* messages and the factors compression previously analyzed, and is probably due to the same reason: the communications mainly concern the bigger fronts at the top of the assembly tree, which tend to compress more than average.

On the other hand, on 90 processes, the reduction factor drops from 11.9 to 8.2. The reason for this is two-fold. First, with more processes, fronts lower in the assembly tree, which achieve a lower compression rate, are mapped on several processes and are thus concerned by communications. Second, the situation where a BLR block is mapped on more than one process on the parent front (described in Figure 6.7) occurs more frequently when the number of processes grows, which leads to a higher communication overhead cost (due to the sending on the column basis Y to each process concerned).

Note that the LU volume of communications is also slightly different from that of Table 6.2. This comes from slight variations of the LU compression rate depending on whether the CB is compressed or not.

As reported in Table 6.5, the reduction of the volume of communications is not enough to compensate the flop overhead and leads to slowdowns. This could be explained by the theoretical results obtained in the previous section. Indeed, while the total volume is asymptotically reduced by compressing the CB, the critical volume is not always asymptotically decreased, depending on whether the dominant term is the LU ($n\sqrt{r}p$) or CB ($n^{4/3}/p$) one (cf. Table 6.4). Since the performance of the solver is mainly driven by the critical volume, this may explain why no gains are observed. However, two facts could improve this result.

First, the CB compression in the context of the CUFS (or C;FSU) variant may have a much lower cost. Indeed, we have observed that if the CB blocks are compressed *before* being updated (as is the case in the CUFS variant), their rank is much smaller, which results in a much lower compression overhead. The compressed blocks must then be accumulated and recompressed together with the low-rank update contributions. The choice of recompression strategy (Chapter 3) is less critical in this case, because the CB blocks are not further involved in operations; a cheap strategy that captures most of the compression potential to reduce the communications is likely to be enough.

Second, and perhaps most importantly, a key parameter is the network speed with respect to the processor speed. We have run the same experiment on **cori**, which has a similar network speed (8 GB/s compared to 6.89 GB/s on **eos**) but twice faster processors (Haswell instead of Ivy Bridge). Using 32×16 cores, the run time for the BLR CB_{FR} factorization is 280s while that of the CB_{LR} factorization is only 272s. This shows that on a machine with a higher processor speed over network speed ratio, compressing the CB may improve the run time of the factorization. Since this ratio tends to increase on recent machines, we believe compressing the CB will become a necessity for BLR solvers to scale on modern architectures.

6.4 Load balance analysis

In this section, we propose some strategies to improve the load balance of the BLR factorization. This analysis is motivated by **Issue 3**: due to the low-rank compression, the flop ratio between the least and most loaded processes increases in BLR with respect to the FR factorization.

There are mainly two parameters that can influence the load balance: the mapping and the splitting strategies. The mapping controls which (and how many) processors are assigned to each frontal matrix in the assembly tree. The splitting can be used to to adjust the number of fully-summed rows in each front. Because the fully-summed rows are always assigned to a single processor, this can be used to balance the load. Thus, to simplify, mapping controls tree parallelism while splitting controls node parallelism.

Both the mapping and splitting strategies of our solver are based on full-rank cost estimates. Therefore, to improve the BLR factorization, we need to revisit them to take into account the low-rank compression.

We begin by adapting our proportional mapping (Pothen and Sun, 1993) strategy described in Section 6.1.1. Since we aim at minimizing the workload unbalance, the weights w_i in equation (6.1) measure the number of flops performed for the factorization of the subtree rooted at child i. The key question is then how to compute these weights w_i. In the FR factorization, this is relatively straightforward as the number of operations to factorize a given subtree can usually be predicted in advance (except in some cases such as when large amounts of numerical pivoting occur).

However, in the BLR case, the low-rank compression cannot be predicted. Furthermore, even if the compression rate could be known or estimated in advance, this information would be useless to improve the mapping without knowing the compression rate of each individual front. Indeed, if we know a given problem achieves a compression rate of, say, 10%, and if we assume each front achieves the same compression rate, all the weights w_i are divided by 10 and the processors are assigned exactly as in the full-rank case.

The problem with this reasoning is that we have assumed a *constant* compression rate. However, as proven in Chapter 4, the complexity is asymptotically reduced. With the standard FSCU variant, the factorization of a front of order m requires $\mathcal{O}(m^{2.5}\sqrt{r})$ operations rather than $\mathcal{O}(m^3)$ as in full-rank. This observation still makes no difference if the assembly tree is very regular (i.e. siblings fronts are of the same order), but may become important for unbalanced and/or irregular trees. Indeed, factorizing a front of order 1000 has the same cost as factorizing 1000 fronts of order 100 in FR, but only $10^{2.5} \approx 316$ fronts of order 100 in BLR (assuming $r = \mathcal{O}(1)$).

In Table 6.7, we assess the quality of this improved mapping, computed with the asymptotic formulas of Chapter 4. Specifically, we have used a dense complexity of order $\mathcal{O}(m^{2.75})$ (i.e. a rank bound $r = \mathcal{O}(\sqrt{m})$), because matrix 10Hz arises from the Helmholtz equation (cf. Section 1.5.2.2). We compare the load balance and the run time of the BLR factorization using a proportional mapping based on FR and LR estimates (Mapp_{FR} and Mapp_{LR}).

	30×10		90×10	
	Mapp_{FR}	Mapp_{LR}	Mapp_{FR}	Mapp_{LR}
load balance	2.01	1.98	2.57	2.07
time (s)	366.6	353.4	231.2	230.0

Table 6.7 – Comparison of the load balance (ratio between most and least loaded processor) and the time of the BLR factorization with the reference mapping based on full-rank cost estimates (Mapp_{FR}) and the improved mapping based on the asymptotic BLR complexities (Mapp_{LR}), on matrix 10Hz (matrix ID 3).

On both 300 and 900 cores, the load balance and time for factorization are improved by using the LR-based mapping. On 900 cores, the gains in time are negligible, whereas those on 300 cores are slightly better. This seems to indicate that the performance bottleneck lies somewhere else.

We therefore turn to the splitting strategy. Each front is split such that the master process holds $n_{master} = \alpha n_{slave}$ rows, where n_{slave} denotes the number of

rows held by the slave processes. In our solver, α is set by default to 1.0. Thus, the master and slave processes hold the same number of rows, which aims at balancing the memory consumption. Note that this means the master process has less work to perform than the slaves (Sid-Lakhdar, 2014). By tuning the value of α, some performance gains can be achieved. We have tested several values of α and report in Table 6.8 the load balance and run time for the FR and BLR factorizations for the choice of α that minimizes the run time, noted α^*.

	30×10				90×10			
	FR		BLR		FR		BLR	
α	1.0	$\alpha^* = 1.4$	1.0	$\alpha^* = 1.6$	1.0	$\alpha^* = 1.6$	1.0	$\alpha^* = 1.8$
load balance	1.42	1.41	1.98	1.89	1.28	1.27	2.07	2.13
time (s)	1257.2	1218.1	353.4	343.5	617.0	600.9	230.0	204.0

Table 6.8 – Influence of the splitting strategy on the FR and BLR factorizations, on matrix 10Hz (matrix ID 3).

In FR, some gains can be achieved by giving more work to the master, since the default behavior of our solver is to balance memory rather than flops. This slightly improves the load balance, and is translated into very modest time gains (less than 3% improvement).

In BLR, the optimal value α^* is different than in FR, which illustrates the effect of low-rank compression. With a tuned splitting, more significant gains are achieved, especially on 900 cores. At first glance, it might seem surprising that the optimal α^* value is higher in BLR than in FR (0.2 more on both 300 and 900 cores). Indeed, intuitively, the master is mainly dominated by the Factor and Solve steps, which are performed in full-rank; the slaves are dominated by the Update step, which is performed in low-rank. Therefore, we would expect the flop ratio between master and slaves to increase in BLR. This intuition is strengthened by the fact that, on 900 cores, the load balance is actually worse for $\alpha = \alpha^*$ than for $\alpha = 1.0$. We have performed some tests with $\alpha < 1.0$ which show some improvement of the load balance but lead to higher run times. Therefore, we believe this could be due to the fact that the full-rank operations performed by the master are faster (higher GF/s rate) than the low-rank ones performed by the slaves. There is thus a tradeoff between load balance and speed. This requires to be formalized, and is out of the scope of this thesis.

Overall, by revisiting the mapping and splitting strategies, we have accelerated the BLR factorization by 6% on 300 cores and by 11% on 900 cores. Thus, since the gains increase with the number of cores, we have improved the scalability of the factorization; on larger number of cores, higher gains could surely be expected. As a consequence, the time ratio with respect to the FR case has increased from 3.4 to 3.5 (300 cores) and from 2.7 to 2.9 (900 cores).

6.5 Distributed-memory LUAR

We now discuss how to benefit from the accumulation and recompression of the LUAR variant in a distributed-memory factorization. So far, the results we have presented were obtained using a right-looking factorization, both on the master and slave processes. However, to be accumulated (and possibly recompressed) together, the low-rank updates must be performed when they are all available, i.e. in left-looking. We must therefore revisit our choice of factorization pattern.

We refer to Table 6.9 for a comparison of the different options. In addition to the total time for the factorization, we also provide the average time spent in computations per MPI process. This value measures the time spent in calls to computational kernels, and can thus be used to assess the computational cost of the method without taking into account the effect of communications and synchronizations.

Factorization on master		RL	LL	
Factorization on slaves		RL	RL	LL
avg. time spent in computations (s)	FSCU/UFSC	132.4	133.5	126.4
	+LUA	—	127.6	114.3
	+LUAR	—	123.4	97.7
total time for factorization (s)	FSCU/UFSC	343.5	342.2	373.8
	+LUA	—	332.2	367.1
	+LUAR	—	335.7	333.6

Table 6.9 – Comparison between the different patterns of factorization on matrix 10Hz (matrix ID 3) on 30×10 cores. RL/LL: right-/left-looking. The average time spent in computations per MPI process evaluates the time spent in computational kernels, without taking into account idle times due to communications or synchronizations.

Using a LL instead of a RL factorization on the master process has been previously considered in the FR context (Sid-Lakhdar, 2014, Section 5.3). It can perform comparably or, in some cases, better, because the master produces panels faster at the beginning and thus feeds the slaves faster, preventing them from starving. On our matrix, using 300 cores, the run time for the factorization with a LL pattern on the master processes is 1210.3s (not shown in Table 6.9), and thus achieves a comparable performance as when using a RL pattern (1218.1s). The same behavior is observed in BLR, with a run time of 342.2s compared to 343.5s. In BLR, the difference is that LUAR can now be used, even if only on the master. This reduces the average time spent in computations, and leads to a modest overall gain, down to 332.3s.

To fully exploit the potential of the LUAR variant, the slaves processes should also switch their pattern from RL to LL. However, that is not a good idea in general: it worsens the pipeline between the master and slaves, the latter having more work towards the end of the factorization (in particular, the slaves only start updating the CB after the master has completed its work). Thus, even though the LUAR

algorithm significantly decreases the computational cost of the factorization (the time spent in computations decreases from 123.4 to 97.1), the total time increases.

Similar results are obtained on 900 cores (not shown here). With LUAR on the master processes only, the time for the BLR factorizations decreases from 204.0s to 195.1s. Overall, the distributed-memory LUAR variant has increased the time ratio between BLR and FR from 3.5 to 3.7 on 300 cores, and from 2.9 to 3.1 on 900 cores.

6.6 Distributed-memory UFCS

In the distributed-memory factorization, the non fully-summed variables are mapped on the slaves and therefore not available to the master, who performs the numerical pivoting. To avoid communicating during pivoting, which would be very inefficient, we use the approach proposed by Duff and Pralet (2007) to restrict the pivoting to the fully-summed variables, and use an estimate of the maximal pivots among the non fully-summed variables, as described in Section 1.3.2.6.

Thus, in BLR, the slave processes can switch from a UFSC to a UFCS factorization without compromising the ability to perform numerical pivoting. The result obtained is reported in Table 6.10 (compare second and third columns). The number of operations is reduced by a 23%. while the scaled residual remains comparable. The reduction of flops translates to a modest time gain.

	standard pivoting		restricted pivoting
	UFSC	UFCS on slaves only	UFCS on master+slaves
flops ($\times 10^{14}$)	2.00	1.54	1.40
time (s)	332.2	323.0	214.0
scaled residual	4.6e-02	4.3e-02	4.5e-02

Table 6.10 – Performance and accuracy of UFSC and UFCS on 10Hz matrix on 30×10 cores.

Furthermore, if numerical pivoting is not needed or can be further restricted, the master can also perform a UFCS factorization. We report in the fourth column of Table 6.10 the result using pivoting restricted to the diagonal blocks. The UFCS factorization allows for an additional gain in flops and leads to a time improvement of 34%, while keeping the residual almost constant.

Similar results are obtained on 900 cores (not shown here). With the UFCS variant, the time for the BLR factorization decreases to 123.4. Therefore, the distributed-memory UCFS variant has increased the time ratio between BLR and FR from 3.7 to 5.6 on 300 cores, and from 3.1 to 4.9 on 900 cores.

Note that the UCFS algorithm introduced in Section 2.3.2 to compress before the Solve step while maintaining the ability to perform pivoting can naturally be extended to the distributed-memory case. We do not present distributed-memory UCFS results in this thesis.

6.7 Complete set of results

As in the previous chapter (cf. Section 5.5), we now present results on our main set of problems, described in Section 1.5.2.2. We have chosen some of the larger problems of the set, and report the results for the FR and BLR factorizations on 900 (90×10) cores in Table 6.11.

low-rank threshold ε		10^{-3}	10^{-7}		10^{-9}	10^{-6}		
matrix ID		3	5	7	10	14	18	20
flops ($\times 10^{14}$)	FR	25.56	17.64	24.88	15.20	16.71	2.61	35.55
	BLR	2.00	1.43	1.69	2.62	2.19	0.22	1.11
	+ LUAR	1.95	1.30	1.53	1.32	1.72	0.20	0.98
	+ UFCS	1.35	0.31	0.32	1.15	1.34	0.12	0.77
	flop ratio*	**19.0**	**57.6**	**47.0**	**13.2**	**12.5**	**21.5**	**46.0**
time (s)	FR	617.0	960.0	1307.4	289.3	373.4	70.5	1488.1
	+ load bal.	600.9	919.3	1242.6	263.0	352.8	65.4	1438.8
	BLR	231.2	358.5	454.3	156.9	162.8	38.4	311.4
	+ load bal.	204.0	286.4	339.2	128.8	145.0	31.8	262.1
	+ LUAR	195.1	295.7	340.0	130.2	140.8	32.7	250.0
	+ UFCS	123.4	183.6	233.8	104.9	130.8	23.5	236.4
	time ratio*	**4.9**	**5.0**	**5.3**	**2.5**	**2.7**	**2.8**	**6.1**

*between best FR and best BLR

Table 6.11 – Experimental results on real-life matrices from SEISCOPE, EMGS, EDF, and the UFSMC, using 90×10 cores.

On this set of large problems, the BLR gains with respect to the FR solver in flops are very important (average flop gain of 13.9). Only a small fraction of this potential gain is translated into time gains with the standard BLR variant, with an average time gain of 2.7. With the improved load balance, the average time gain increases to 3.1. Finally, the lower complexity LUAR and UFCS variants further increase the flop gain to 31.0, which leads to the final average time gain of 4.2.

Compared to the multicore results presented in Section 5.5 (cf. Tables 5.11 and 5.12), the time gains with respect to the FR solver on 90×10 cores are lower than on 24 threads. This shows that, although the gains are already important, there is still some margin for further improvement of the performance and scalability of the distributed-memory BLR factorization.

6.8 Chapter conclusion

We have presented and analyzed the distributed-memory BLR factorization on a set of large real-life problems. Our strong scalability analysis underlines a number of issues for which we have proposed some beginnings of a solution.

First, the volume of communications of the BLR factorization has a higher relative weight compared to the FR case. We showed both theoretically and experimentally that this is because the messages associated with the contribution block of

the frontal matrices become dominant. By compressing these contribution blocks, we have greatly reduced the volume of communications; moreover, even though it represents an overhead cost which does not contribute to reducing the overall number of operations, it can potentially improve the run time depending on the relative speed of the network with respect to that of the processors.

Second, the BLR factorization suffers from a higher load unbalance, due to the unpredictability of the low-rank compressions. We have proposed some ways to improve the load balance by revisiting our mapping and splitting strategies.

Finally, we analyzed the distributed-memory version of the LUAR and UFCS algorithms, which both improve the performance of the BLR factorization.

CHAPTER

7 Application to real-life industrial problems

In this chapter, we present the use of BLR solvers in two real-life industrial applications: 3D frequency-domain Full Waveform Inversion (Section 7.1) and 3D Controlled-source electromagnetic inversion (Section 7.2).

We have already presented results using BLR approximations on matrices arising in these two applications ({5,7,10}Hz matrices and H{3,17},S{3,21} matrices, respectively) in Chapters 5 and 6. Therefore, in this chapter, we mostly focus on the applications and the overall gains that can be expected with the use of BLR solvers. We refer to the two previously mentioned chapters for a detailed performance analysis of the algorithms.

7.1 3D frequency-domain Full Waveform Inversion

7.1.1 Applicative context

Full Waveform Inversion (FWI) (Tarantola, 1984)) is now routinely used in the oil industry as part of the seismic imaging work-flow, at least in soft geological environments such as the North Sea that make the acoustic parameterization of the subsurface acceptable (cf. Barnes and Charara (2009) and Plessix and Perez Solano (2015) for a discussion on this latter issue). However, it remains a computational challenge due to the huge number of full-waveform seismic modelings to be performed over the iterations of the FWI optimization. Seismic modeling and FWI can be performed either in the time domain or in the frequency domain (e.g., Plessix (2007), Vigh and Starr (2008b), and Virieux and Operto (2009)). Hybrid approaches are also possible where seismic modeling and inversion are performed in the time domain and in the frequency domain, respectively (Sirgue, Etgen, and Albertin, 2008). In these hybrid approaches, the monochromatic wavefields required to perform the inversion in the frequency domain are built on the fly in the loop over time steps by discrete Fourier transform during the time-domain modeling (Nihei and Li, 2007). Today, most FWI codes are fully implemented in the time domain because the good scalability and the moderate memory demand of the initial-condition evolution problem underlying the forward problem allow one to tackle a wide range of

applications in terms of target dimension, survey design and wave physics. However, the time domain formulation also requires significant computational resources to be efficient when thousands or tens of thousands of (reciprocal) seismic sources are processed in parallel. A common parallel strategy consists in distributing these seismic sources over processors. This embarrassingly parallel strategy can be combined with shared-memory parallelism and/or domain decomposition of the computational mesh if enough computational resources are available. Strategies based on source sub-sampling (Warner et al., 2013) or source blending with random or deterministic (i.e., plane-wave decomposition) encoding (Vigh and Starr, 2008a; Krebs et al., 2009) are commonly used to reduce the number of seismic modelings per FWI iteration. However, these strategies, which reduce the data fold or add random noise in the FWI gradient at each iteration, require increasing the number of iterations to reach a sufficient signal-to-noise ratio in the subsurface models. Considering a computational mesh with N^3 degrees of freedom, an acquisition with N^2 seismic sources and assuming that the number of time steps scales to N, the time complexity of time-domain modeling scales to $\mathcal{O}(N^6)$ (Plessix, 2007).

Alternatively, both seismic modeling and FWI can be performed in the frequency domain (e.g., Pratt (1999) and Virieux and Operto (2009)). A few applications of 3D frequency-domain FWI on synthetic or real data are presented in Ben Hadj Ali, Operto, and Virieux (2008), Plessix (2009), Petrov and Newman (2014), and Operto et al. (2015). Solving the time-harmonic wave equation is a stationary boundary-value problem which requires to solve a large and sparse complex-valued system of linear equations with multiple right-hand sides per frequency (e.g., Marfurt (1984)). The sparse right-hand sides of these systems are the seismic sources, the solutions are monochromatic wavefields and the coefficients of the so-called impedance matrix depend on the frequency and the subsurface properties we want to image. This linear system can be solved either with sparse direct methods, namely, Gaussian elimination techniques (e.g. Operto et al. (2007)), iterative solvers (e.g., Riyanti et al. (2006), Plessix (2007), Petrov and Newman (2012), and Li, Métivier, Brossier, Han, and Virieux (2015)) or a combination of both in the framework of domain decomposition methods (e.g., Sourbier et al. (2011)). One pitfall of iterative methods is the design of an efficient preconditioner considering that the wave-equation operator is indefinite. More precisely, the iterative approach is competitive with the time-domain approach in terms of operation count as long as the number of iterations can be made independent of the frequency, i.e., the problem size (Plessix, 2007). It seems that this objective has not yet been fully achieved, although using damped wave equation as a preconditioner or as such for early-arrival modeling decreases the iteration count efficiently (Erlangga and Nabben, 2008; Petrov and Newman, 2012). Processing a large number of right-hand sides leads us more naturally toward direct methods because the computation of the solutions by forward/backward substitutions is quite efficient once a LU factorization of the impedance matrix has been performed. The pitfalls of the direct methods are the memory demand and the limited scalability of the LU factorization that result from the fill-in of the impedance matrix generated during the LU factorization. Fill-reducing matrix orderings based on nested dissections are commonly used to reduce the memory complexity by one order of magnitude, that is $\mathcal{O}(N^4)$ instead of $\mathcal{O}(N^5)$ (George, 1973). The time complexity of the substitution step for N^2 right-hand sides scales to $\mathcal{O}(N^6)$

accordingly and is the same as the complexity of time-domain modeling. The time complexity of one LU factorization (for sparse matrices as those considered in this study) also scales to $\mathcal{O}(N^6)$. The conclusions that can be drawn about the relevancy of the direct solver-based approach from this complexity analysis are two fold: the identity between the time complexity of the LU factorization and the solution step for N^2 right-hand sides requires the number of right-hand sides to scale to N^2. Moreover, the identity between the time complexity of one LU factorization and time-domain modeling for N^2 right-hand sides requires to limit the inversion to a few discrete frequencies. Both requirements are fulfilled by wide-azimuth long-offset acquisitions implemented with stationary-receiver geometries (ocean bottom cable or ocean bottom node acquisitions in marine environment). On the one hand, stationary-receiver geometries involve a large number of sources and associated receivers in the computational domain from which the LU factorization is performed. On the other hand, the wide-azimuth long-offset coverage provided by these geometries generate a strong redundancy in the wavenumber sampling of the subsurface target. This multi-fold wavenumber coverage results from the redundant contribution of finely-sampled (temporal) frequencies and a broad range of finely-sampled scattering angles provided by dense point-source acquisitions (Pratt and Worthington, 1990). The strategy consisting in coarsening the frequency sampling in the data space to remove the redundancy of the wavenumber sampling in the model space has been referred to as efficient FWI by Sirgue and Pratt (2004).

The focus of this study is to present an up-to-date status of the computational efficiency of 3D frequency-domain FWI based on sparse direct solvers with a real OBC data case study from the North Sea. A first application of 3D frequency-domain FWI on the North Sea OBC dataset has been presented in Operto et al. (2015), which is focused on a detailed quality control of the FWI results based on seismic modeling and source wavelet estimation. In this study, we focus on the benefits resulting from the use of low-rank approximations. We solve the linear system resulting from the discretization of the time-harmonic wave equation with the Block Low-Rank multifrontal solver presented in the previous chapters. Other low-rank formats such as the Hierarchically Semi-Separable (HSS) format have also been proposed for seismic modeling by Wang, Hoop, and Xia (2011), Wang, Xia, Hoop, and Li (2012b), and Wang, Hoop, Xia, and Li (2012a).

The rest of this section is organized as follows. Section 7.1.2 first briefly reviews the finite-difference stencil with which the time-harmonic wave equation is discretized. This stencil must satisfy some specifications so that the fill-in of the impedance matrix is minimized during the LU factorization. The reader is referred to Operto et al. (2014) for a more detailed description of this finite-difference stencil. Then, Section 7.1.3 presents the application on the OBC data from the North Sea. Finally, Section 7.1.4 analyzes the results obtained with the BLR multifrontal solver and compares it to the FR solver. We first show the nature of the errors introduced by the BLR approximation in the monochromatic wavefields and quantify the backward errors for different frequencies. For a given BLR threshold, the ratio between the backward errors obtained with the BLR and the full-rank (FR) solvers decreases with frequency. This suggests that more aggressive threshold can be used as frequency increases hence leading to more efficient compression as the problem size grows. Then, we show that the modeling errors have a negligible im-

pact in the FWI results. The cost of the FWI in terms of memory and time confirms that the computational saving provided by the BLR solver relative to the FR solver increases with frequency (namely, matrix size).

The limited computational resources that have been used to perform this case study and the limited computational time required to perform the FWI in the 3.5Hz-10Hz frequency band using all the sources and receivers of the survey at each FWI iteration highlights the computational efficiency of our frequency-domain approach to process stationary recording surveys in the visco-acoustic VTI approximation.

7.1.2 Finite-difference stencils for frequency-domain seismic modeling

Frequency-domain seismic modeling with direct solvers defines some stringent specifications whose objective is to minimize the computational burden of the LU factorization. The first specification aims to minimize the dependencies in the adjacency graph of the matrix and hence the fill-in of the impedance matrix during the LU factorization. The second one aims to obtain accurate solutions for a coarse discretization that is matched to the spatial resolution of the FWI (i.e., four grid points per wavelengths according to a theoretical resolution of half a wavelength). The first specification can be fulfilled by minimizing the numerical bandwidth and optimizing the sparsity of the impedance matrix using finite-difference stencils with compact spatial support. This precludes using conventional high-order accurate stencils. Instead, accurate stencils are designed by linearly combining different second-order accurate stencils that are built by discretizing the differential operators on different coordinate systems with a spreading of the mass term over the coefficients of the stencil (Jo, Shin, and Suh, 1996; Stekl and Pratt, 1998; Min, Sin, Kwon, and Chung, 2000; Hustedt, Operto, and Virieux, 2004; Gosselin-Cliche and Giroux, 2014). Such stencils are generally designed for second-order acoustic/elastic wave equations, by opposition to first-order velocity-stress equations, as the second-order formulation involves fewer wavefield components, hence limiting the dimension of the impedance matrix accordingly. For the 3D visco-acoustic wave equation, the resulting finite-difference stencil has 27 nonzero coefficients distributed over two grid intervals and provides accurate solution for arbitrary coarse grids provided that optimal coefficients are estimated by fitting the discrete dispersion equation in homogeneous media (Operto et al., 2007; Brossier, Etienne, Operto, and Virieux, 2010). The visco-acoustic 27-point stencil was recently extended in order to account for VTI anisotropy without generating significant computational overhead (Operto et al., 2014). The visco-acoustic VTI 27-point stencil results from the discretization of the following equation

$$A_h\, p_h = s', \tag{7.1}$$

$$p_v = A_v\, p_h + s'', \tag{7.2}$$

$$p = \frac{1}{3}(2p_h + p_v), \tag{7.3}$$

where p_h, p_v and p are the so-called horizontal pressure, vertical pressure and (physical) pressure wavefields, respectively. The operators A_h and A_v are given by

$$\begin{aligned} A_h &= \omega^2 \left[\frac{\omega^2}{\kappa_0} + (1+2\epsilon)(\mathscr{X}+\mathscr{Y}) + \sqrt{1+2\delta}\,\mathscr{Z} \frac{1}{\sqrt{1+2\delta}} \right] + 2\sqrt{1+2\delta}\,\mathscr{Z} \frac{\kappa_0(\epsilon-\delta)}{\sqrt{1+2\delta}} (\mathscr{X}+\mathscr{Y}), \\ A_v &= \frac{1}{\sqrt{1+2\delta}} + \frac{2(\epsilon-\delta)\kappa_0}{\omega^2\sqrt{1+2\delta}} (\mathscr{X}+\mathscr{Y}), \end{aligned} \tag{7.4}$$

where $\kappa_0 = \rho V_0^2$, ρ is density, V_0 is vertical wavespeed, ω is the angular frequency, δ and ϵ are Thomsen's parameters. Differential operators $\mathscr{X}$, $\mathscr{Y}$ and $\mathscr{Z}$ are given by $\partial_{\tilde{x}} b \partial_{\tilde{x}}$, $\partial_{\tilde{y}} b \partial_{\tilde{y}}$ and $\partial_{\tilde{z}} b \partial_{\tilde{z}}$, respectively, where $b = 1/\rho$ is buoyancy and $(\tilde{x}, \tilde{y}, \tilde{z})$ define a complex-valued coordinate system in which perfectly-matched layers absorbing boundary condition are implemented (Operto et al., 2007). The right-hand side vectors s' and s'' have the following expression:

$$\begin{aligned} s'(x,\omega) &= \omega^4 s(\omega) \frac{s_h(x)}{\kappa_0(x)} \tilde{\delta}(x-x_s) \\ &\quad -\omega^2 s(\omega) \sqrt{1+2\delta(x)}\,\mathscr{Z} \left(s_v(x) - \frac{1}{\sqrt{1+2\delta(x)}} s_h(x) \right) \tilde{\delta}(x-x_s), \end{aligned} \tag{7.5}$$

$$s''(x,\omega) = \left(s_v(x) - \frac{1}{\sqrt{1+2\delta(x)}} s_h(x) \right) \tilde{\delta}(x-x_s), \tag{7.6}$$

where $\tilde{\delta}$ denotes the Dirac delta function, x_s denotes the source position, $s(\omega)$ is the source excitation term and s_h, s_v are two quantities that depend on the Thomsen's parameters.

The expression of the operator A_h shows that the VTI wave equation has been decomposed as a sum of an elliptically anisotropic wave equation (term between brackets) plus an anelliptic correcting term involving the factor $(\epsilon - \delta)$. The elliptic part can be easily discretized by plugging the Thomsen parameters δ and ϵ in the appropriate coefficients of the isotropic 27-point stencil, while the anelliptic correcting term is discretized with conventional second-order accurate stencil to preserve the compactness of the overall stencil.

The direct solver is used to solve the linear system involving the matrix A_h, equation (7.1). In a second step, the vertical pressure p_v is explicitly inferred from the expression of p_h, equation (7.2), before forming the physical pressure wavefield p by linear combination of p_h and p_v, equation (7.3).

The accuracy of the 27-point stencil for the visco-acoustic isotropic and VTI equations is assessed in details in Operto et al. (2007), Brossier et al. (2010) and Operto et al. (2014) and allows for accurate modeling with a discretization rule of four grid points per minimum wavelength.

7.1.3 Description of the OBC dataset from the North Sea

The subsurface target and the dataset are the same as in Operto et al. (2015). The FWI experimental setup is also similar except that we perform FWI with a smaller number of discrete frequencies (6 instead of eleven) and we fix the maximum number of FWI iterations per frequency as stopping criterion of iterations. A

brief review of the target, dataset and experimental setup is provided here. The reader is referred to Operto et al. (2015) for a more thorough description.

7.1.3.1 Geological target

The subsurface target is the Valhall oil field located in the North Sea in a shallow water environment (70 m water depth) (Sirgue et al., 2010; Barkved et al., 2010). The reservoir is located at around 2.5 km depth. The overburden is characterized by soft sediments in the shallow part. A gas cloud, whose main zone of influence is delineated in Figure 7.1a, makes seismic imaging at the reservoir depths challenging. The parallel geometry of the wide-azimuth OBC acquisition consists of 2302 hydrophones which record 49,954 explosive sources located 5 m below the sea surface (Figure 7.1a). The seismic sources cover an area of 145 km^2 leading to a maximum offset of 14.5 km. The maximum depth of investigation in the FWI model is 4.5 km. The seismograms recorded by a receiver for a shot profile intersecting the gas cloud and the receiver position are shown in Figure 7.1b. The wavefield is dominated by the diving waves which mainly propagate above the gas zone, the reflection from the top of the gas cloud (Figure 7.1b, white arrow) and the reflection from the reservoir (Figure 7.1b, black arrow) (Prieux et al., 2011; Prieux, Brossier, Operto, and Virieux, 2013a; Prieux, Brossier, Operto, and Virieux, 2013b; Operto et al., 2015). In Figure 7.1b, the solid black arrow points on the pre-critical reflection from the reservoir while the dash black arrow points on the critical and post-critical reflection. The discontinuous pattern in the time-offset domain of this wide-angle reflection highlights the complex interaction of the wavefield with the gas cloud.

7.1.3.2 Initial models

The vertical-velocity (V_0) and the Thomsen's parameter models δ and ϵ, which are used as initial models for FWI, were built by reflection traveltime tomography (courtesy of BP) (Figure 7.2(a-d) and 7.3a,d). The V_0 model describes the long wavelengths of the subsurface except at the reservoir level which is delineated by a sharp positive velocity contrast at around 2.5 km depth (Figures 7.3a,d). We do not smooth this velocity model before FWI for reasons explained in Operto et al. (2015) (their figure 7). The velocity model allows us to match the first-arrival traveltimes as well as those of the critical reflection from the reservoir (cf. Operto et al. (2015), Figure 8), hence providing a suitable framework to prevent cycle skipping during FWI. A density model was built from the initial vertical velocity model using a polynomial form of the Gardner law given by $\rho = -0.0261V_0^2 + 0.373V_0 + 1.458$ (Castagna, 1993) and was kept fixed over iterations. A homogeneous model of the quality factor was used below the sea bottom with a value of $Q = 200$.

7.1.3.3 FWI experimental setup

We review here the experimental setup that was used to apply frequency-domain FWI on the OBC dataset. The discrete frequencies, the computational resources and the finite-difference grid dimensions that are used to perform FWI are reviewed

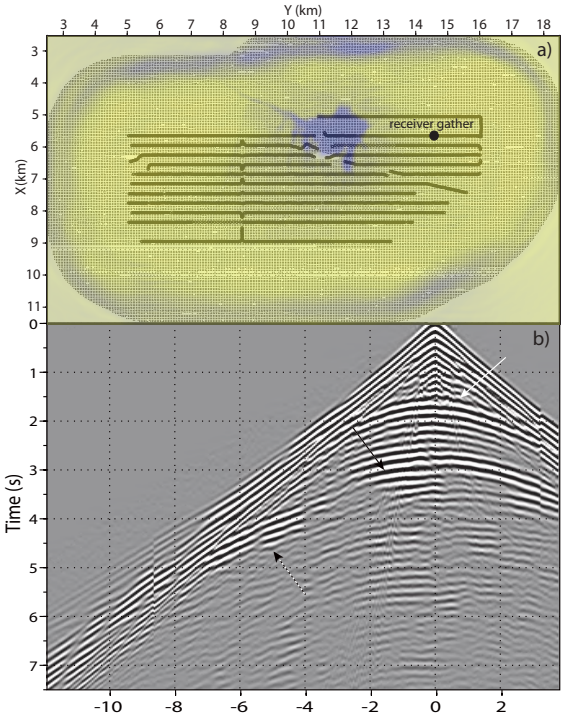


Figure 7.1 – North Sea case study. a) Acquisition layout. The green lines are cables. The dot pattern shows the area covered by the 50,000 explosive sources. The black circle shows the position of the receiver whose records are shown in b). A depth slice at 1 km depth across the gas cloud is superimposed in transparency to show the zone of influence of this gas cloud. b) Common receiver gather for a shot profile cross-cutting the receiver position (circle in a) and the gas cloud. The white arrow points on reflection from top of the gas. The black arrows point on pre-critical (solid) and post-critical (dash) reflections from the reservoir.

in Table 7.1. We perform FWI for six discrete frequencies in the 3.5Hz-10Hz frequency band. Only one frequency is processed at a time and the inversion proceeds sequentially from the lowest frequency to the highest one following a frequency-driven multi-scale approach (Pratt, 1999). The grid interval in the subsurface models is periodically matched to the frequency f to minimize the computational time and regularize the inversion by avoiding over-parameterization. We use a grid interval of 70m for 3.5Hz $\leq f \leq$ 5Hz, 50m for f = 7Hz and 35m for f = 10Hz. This roughly leads to 4 grid points per wavelength for a minimum wavespeed of 1400 m/s. The number of degrees of freedom in the 70m, 50m and 35m finite-difference grids are 2.9, 7.2 and 17.4 million, respectively, after adding perfectly matched absorbing layers (PMLs) (Bérenger, 1996; Operto et al., 2007).

We perform FWI on 12, 16 and 34 nodes of the **licallo** supercomputer for the 70m, 50m and 35m grids, respectively (Table 7.1). The operations are performed in single precision complex arithmetic (c), for which the peak performance of the machine is 10 GF/s/core (which corresponds to a double precision (d) peak of 20 GF/s/core). We launch two MPI processes per node (that is, one MPI process per processor) and use 10 threads per MPI process such that the number of MPI processes times the

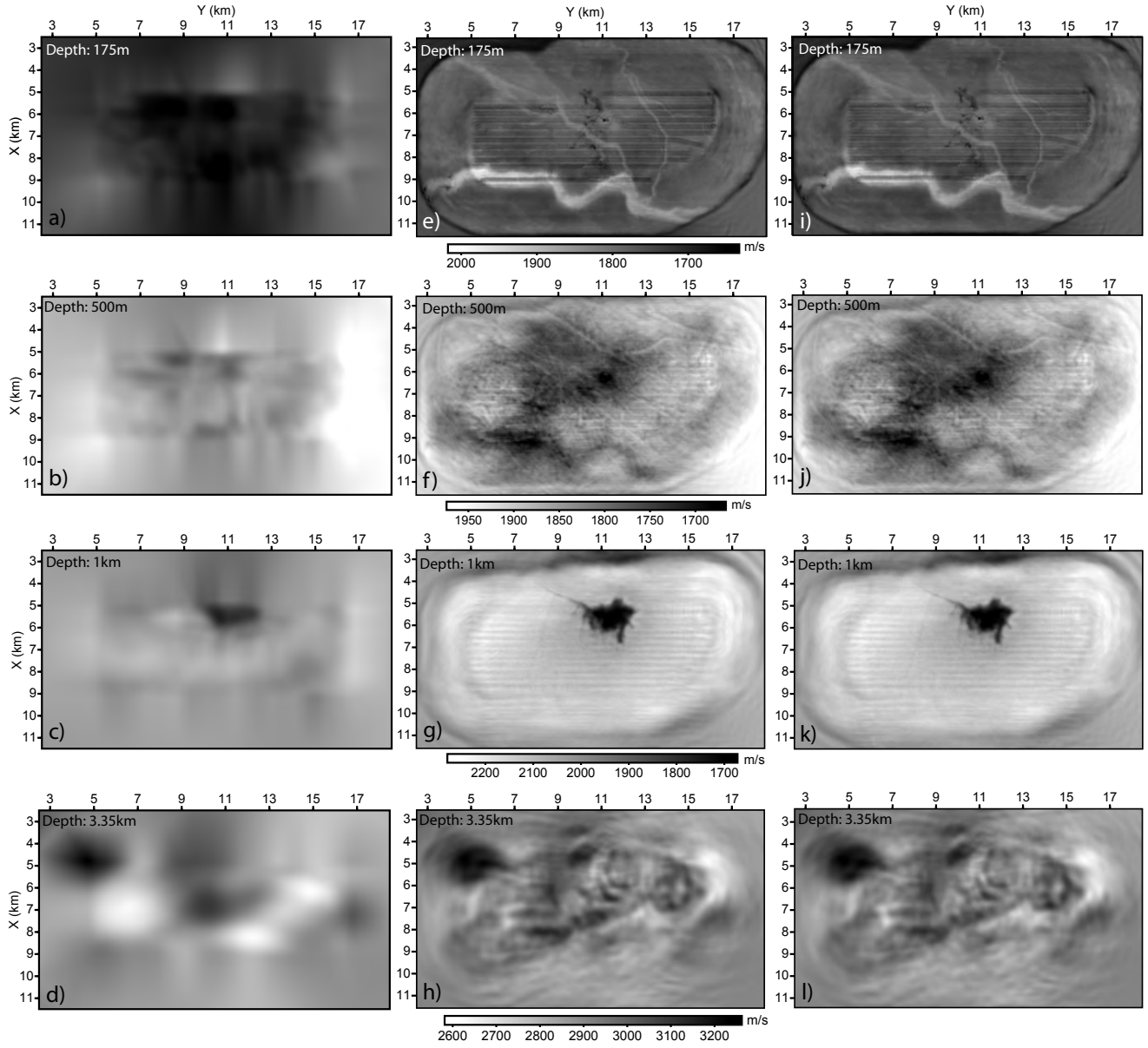


Figure 7.2 – North Sea case study. (a-d) Depth slice extracted from the initial model at 175 m (a), 500 m (b), 1 km (c) and 3.35 km (d) depths. (e-h) Same as (a-d) for depth slices extracted from the FWI model obtained with the FR solver. (i-l) Same as (a-d) for depth slices extracted from the FWI model obtained with the BLR solver.

number of threads is equal to the number of cores on the node. The number of nodes for the three grids (12, 16 and 34) were chosen pragmatically to find the best compromise between the computational efficiency, the memory use and the fast access to the computational resources according to the operating rule of the available cluster.

We use all the (reciprocal) shots and receivers at each FWI iteration, whatever the grid interval and the frequency. This implies that 4604 wavefield solutions need to be computed for each FWI gradient computation. Seismic modeling is performed with a free surface on top of the finite-difference grid during inversion. Therefore,

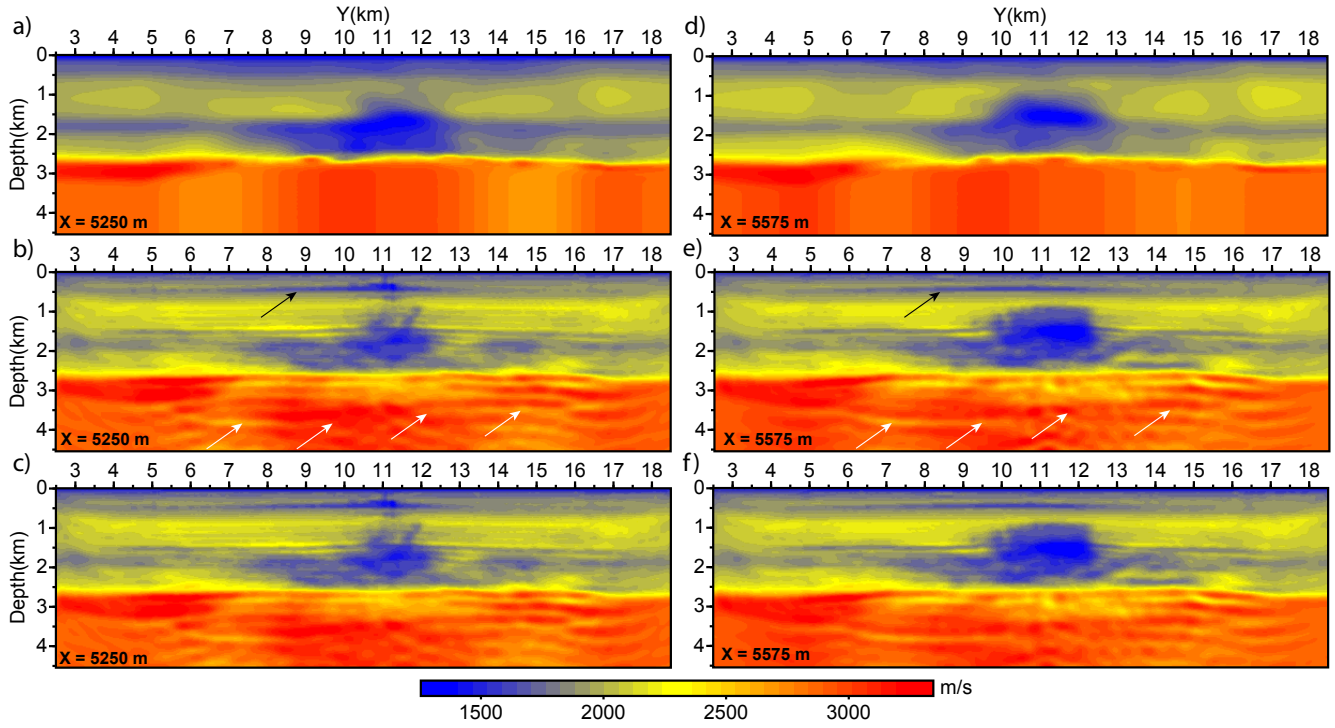


Figure 7.3 – North Sea case study. (a,d) Vertical slices extracted from the initial model at X=5250 m (a) and X=5575 m (d). The slice in (d) crosscuts the gas cloud while the slice in (a) is located in its periphery (cf. Figure 7.2). (b,e) Same as (a,d) for the final FWI model obtained with the FR solver. (c,f) Same as (b,e) for the final FWI model obtained with the BLR solver ($\varepsilon = 10^{-3}$). The back arrow points on a low-velocity reflector at 500 m depth (cf. Figure 7.2f,j for the lateral extension of this reflector at 500 m depth, while the white arrows point on the base Cretaceous reflector.

free surface multiples and ghosts were left in the data and accounted for during FWI. We only update V_0 during inversion, while ρ, Q, δ and ϵ are kept to their initial values. We use the preconditioned steepest-descent method implemented in the SEISCOPE toolbox (Métivier and Brossier, 2016) to perform FWI where the preconditioner is provided by the diagonal terms of the so-called pseudo-Hessian (Shin, Jang, and Min, 2001). No regularization and no data weighting were used.

The stopping criterion of iterations consists of fixing the maximum iteration to 15 for the 3.5Hz and 4Hz frequencies, 20 for the 4.5Hz, 5Hz and 7Hz frequencies and 10 for the 10Hz frequency. We use a limited number of iterations at 3.5Hz and 4Hz because of poor signal to noise ratio. Although this stopping criterion of iteration might seem quite crude, we show that a similar convergence rate was achieved by FWI performed with the full-rank and BLR solvers at the 7Hz and 10Hz frequencies, leading to very similar final FWI results. Therefore, the FWI computational cost achieved with the full-rank and BLR solvers in this study are provided for FWI results of similar quality.

Frequencies (Hz)	h(m)	Grid dimensions	n_{pml}	$\#u$	$\#n$	$\#MPI$	$\#th$	$\#c$	$\#rhs$
3.5, 4, 4.5, 5	70	$66 \times 130 \times 230$	8	2.9	12	24	10	240	4604
7	50	$92 \times 181 \times 321$	8	7.2	16	32	10	320	4604
10	35	$131 \times 258 \times 458$	4	17.4	34	68	10	680	4604

Table 7.1 – North Sea case study. Problem size and computational resources (for the results of Tables 7.3 and 7.4 only). h(m): grid interval. n_{pml}: number of grid points in absorbing perfectly-matched layers. $\#u(10^6)$: number of unknowns. $\#n$: number of computer nodes. $\#MPI$: number of MPI process. $\#th$: number of threads per MPI process. $\#c$: number of cores. $\#rhs$: number of right-hand sides processed per FWI gradient.

7.1.4 Results

7.1.4.1 Nature of the modeling errors introduced by the BLR solver

We show now the nature of the errors introduced in the wavefield solutions by the BLR approximation. Figures 7.4a, 7.5a and 7.6a show a 5Hz, 7Hz and 10Hz monochromatic common-receiver gather computed with the FR solver in the FWI models obtained after the 5Hz, 7Hz and 10Hz inversions (the FWI results are shown in the next section). Figures 7.4(b-d), 7.5(b-d) and 7.6(b-d) show the differences between the common-receiver gathers computed with the FR solver and those computed with the BLR solver using $\varepsilon = 10^{-5}$, $\varepsilon = 10^{-4}$ and $\varepsilon = 10^{-3}$ (the same subsurface model is used to perform the FR and the BLR simulations). These differences are shown after multiplication by a factor 10. A direct comparison between the FR and the BLR solutions along a shot profile intersecting the receiver position is also shown. Three conclusions can be drawn for this case study: for these values of ε the magnitude of the errors generated by the BLR approximation relative to the reference full-rank solutions is small. Second, these relative errors mainly concern the amplitude of the wavefields, not the phase. Third, for a given value of ε, the magnitude of the errors decreases with the frequency. This last statement can be more quantitatively measured by the ratio between the scaled residual obtained with the BLR and the full-rank (FR) solver where the scaled residual is given by $\delta = \frac{\|A_h \tilde{p_h} - b\|_\infty}{\|A_h\|_\infty \|\tilde{p_h}\|_\infty}$ and $\tilde{p_h}$ denotes the computed solution. We show that, for a given value of ε, δ_{BLR}/δ_{FR} decreases with frequency (Table 7.2).

F(Hz)/h(m)	$\delta(FR)$	$\delta(BLR, \varepsilon = 10^{-5})$	$\delta(BLR, \varepsilon = 10^{-4})$	$\delta(BLR, \varepsilon = 10^{-3})$
5Hz/70m	2.3×10^{-7} (1)	4.6×10^{-6} (20)	6.7×10^{-5} (291)	5.3×10^{-4} (2292)
7Hz/50m	7.5×10^{-7} (1)	4.6×10^{-6} (6)	6.9×10^{-5} (92)	7.5×10^{-4} (1000)
10Hz/35m	1.3×10^{-6} (1)	2.9×10^{-6} (2.3)	3.0×10^{-5} (23)	4.3×10^{-4} (331)

Table 7.2 – North Sea case study. Modeling error introduced by BLR for different low-rank threshold ε and different frequencies F. δ: scaled residuals defined as $\frac{\|A_h \tilde{p_h} - b\|_\infty}{\|A_h\|_\infty \|\tilde{p_h}\|_\infty}$, for b being for one of the RHSs in B. The number between bracket is δ_{BLR}/δ_{FR}. Note that, for a given ε, this ratio decreases as frequency increases.

7.1.4.2 FWI results

The FWI results obtained with the FR solver are shown in Figures 7.2(e-h) and 7.3b,e. Comparison between the initial model and the FWI model highlights the resolution improvement achieved by FWI. The structures reviewed below are also described in Sirgue et al. (2010), Barkved et al. (2010), and Operto et al. (2015) for comparison. The depth slice at 175 m depth shows high-velocity glacial sand channel deposits as well as small-scale low velocity anomalies (Figure 7.2e). The depth slice at 500 m depth shows linear structures interpreted as scrapes left by drifting icebergs on the paleo seafloor as well as a wide low velocity zone (Figure 7.2f) represented by a horizontal reflector in the vertical sections (Figure 7.3b,e, black arrow). The depth slice at 1 km depth (Figure 7.2g) and the vertical section at X=5575 m (Figure 7.3e) crosscuts the gas cloud whose geometry has been nicely refined by FWI. We also show a vertical section at X=5250 m near the periphery of the gas cloud (Figure 7.3b), which highlights some low-velocity sub-vertical structures also identifiable in the 1 km depth slice (Figure 7.2g). The depth slice at 3.35 km depth crosscuts the base Cretaceous reflector (Barkved et al., 2010) whose geometry is highlighted by the white arrows in the vertical sections (Figure 7.3b,e).

The final FWI model obtained with the BLR solver ($\varepsilon = 10^{-3}$) shown in Figures 7.2(i-l) and 7.3c,f) does not show any obvious differences with the one obtained with the FR solver (Figures 7.2(e-h) and 7.3b,e. This is indeed also the case when the BLR solver is used with $\varepsilon = 10^{-4}$ and $\varepsilon = 10^{-5}$ (not shown here).

The data fit achieved with the BLR solver ($\varepsilon = 10^{-3}$) is illustrated in Figure 7.7 for the receiver, the position of which is given in Figure 7.1a. The figure shows the real 5Hz, 7Hz and 10Hz monochromatic receiver gathers, the modeled ones computed in the FWI model inferred from the inversion of the frequency in question and the difference between the two. We also show a direct comparison between the recorded and modeled wavefields along the dip and cross profiles intersecting the position of the receiver. The data fit is very similar to the one achieved with the FR solver (not shown here, cf. Operto et al. (2015), Figures 15, 16 and 17) and is quite satisfactory, in particular in terms of phase. We already noted that the modeled amplitudes tend to be overestimated at long offsets when the wavefield has propagated through the gas cloud in the dip direction, unlike in the cross direction (Figure 7.7b, ellipse). In Operto et al. (2015), we interpret these amplitude mismatches as the footprint of attenuation whose absorption effects have been underestimated during seismic modeling with a uniform Q equal to 200.

The misfit functions versus the iteration number obtained with the FR and BLR ($\varepsilon = 10^{-5}, 10^{-4}, 10^{-3}$) solvers for the six frequencies are shown in Figure 7.8. The convergence curves obtained with the FR solver and the BLR solver for $\varepsilon = 10^{-4}$ and $\varepsilon = 10^{-5}$ are very similar. In contrast, we show that the convergence achieved by the BLR solver with $\varepsilon = 10^{-3}$ is alternatively better (4Hz, 5Hz) and worse (3.5Hz, 4.5Hz) than for the three other FWI runs when the inversion jumps from one frequency to the next within the 3.5Hz-5Hz frequency band. However, for the last two frequencies (7Hz and 10Hz) that have the best signal-to-noise ratio, all of the four convergence curves show a similar trend and reach a similar misfit function value at the last iteration. This suggests that the crude stopping criterion of iteration that is used in this study by fixing a common maximum iteration count is reason-

able for a fair comparison of the computational cost of each FWI run. The different convergence behavior at low frequencies shown for $\varepsilon = 10^{-3}$ probably reflects the sensitivity of the inversion to the noise introduced by the BLR approximation at low frequencies (Figure 7.4) although this noise remains sufficiently weak to not alter the FWI results. The higher footprint of the BLR approximation at low frequencies during FWI is consistent with the former analysis of the relative modeling errors which decrease as frequency increases (Table 7.2, ratio δ_{BLR}/δ_{FR}).

7.1.4.3 Computational cost

We now show how the BLR solver can reduce the overall time of the FWI modeling. This expensive simulation was carried out in 2016 and the results were published in Amestoy et al. (2016b). At the time, the BLR solver used the standard FSCU factorization and the solution phase was still performed in FR and therefore not accelerated in BLR.

We summarize the results obtained in Amestoy et al. (2016b); then, we provide a performance projection of the FWI modeling with the UFCS+LUAR variant (based on the results obtained in the previous chapters on the same matrices) and with the use of a reasonably optimized BLR solution phase (cf. Section 9.2). These projected results are reported in Table 7.5.

The reduction of the operation count and factorization time obtained with the BLR approximation are outlined in Table 7.3 for the 5Hz, 7Hz and 10Hz frequencies.

Compared to the FR factorization, when the BLR solver (with $\varepsilon = 10^{-3}$) is used, the number of flops for the factorization (field F_{LU} in Table 7.3) decreases by a factor 8, 10.7 and 13.3 for the 5Hz, 7Hz and 10Hz frequencies, respectively. Moreover, the LU factorization time is decreased by a factor 1.9, 2.7 and 2.7 (field T_{LU} in Table 7.3). The time reduction achieved by the BLR solver tends to increase with the frequency.

The elapsed time to compute one wavefield once the LU factorization has been performed is small (field T_s in Table 7.3). The two numbers provided in Table 7.3 for T_s are associated with the computation of the incident and adjoint wavefields. In the latter case, the source vectors are far less sparse which leads to a computational overhead during the solution phase. This results in an elapsed time of respectively 262s, 598s and 1542s to compute the 4604 wavefields required for the computation of one FWI gradient (field T_{ms} in Table 7.3). Despite the high efficiency of the substitution step, the elapsed time required to compute the wavefield solutions by substitution is significantly higher than the time required to perform the LU factorization, especially when the latter is accelerated by the use of BLR. However, the rate of increase of the solution step is smaller than that of the factorization time when increasing the frequency. In other words, the real time complexity of the LU factorization is higher than that of the solution phase, although the theoretical time complexities are both equal to $\mathcal{O}(N^6)$ for N^2 right-hand sides. This is shown by the decrease of the ratio T_{ms}/T_{LU} as the problem size increases (3.4, 1.86, 1.34 for the 70m, 50m and 35m grids, respectively when the FR solver is used). The fact that the speed-up of the factorization phase achieved by the BLR approximation increases

F(Hz)/h(m)	ε	$F_{LU}(\times 10^{12})$	$T_{LU}(s)$	$T_s(s)$	$T_{ms}(s)$	$T_g(mn)$
5Hz/70m (240 cores)	FR	66 (1.0)	78 (1.0)	0.051/0.063	262	9.9
	10^{-5}	17 (3.8)	48 (1.6)			9.4
	10^{-4}	12 (5.3)	46 (1.7)			9.1
	10^{-3}	8 (8.0)	41 (1.9)			9.1
7Hz/50m (320 cores)	FR	410 (1.0)	322 (1.0)	0.12/0.14	598	21.2
	10^{-5}	90 (4.5)	157 (2.1)			17.7
	10^{-4}	63 (6.5)	136 (2.4)			17.3
	10^{-3}	38 (10.7)	121 (2.7)			17.1
10Hz/35m (680 cores)	FR	2600 (1.0)	1153 (1.0)	0.26/0.41	1542	69.0
	10^{-5}	520 (4.9)	503 (2.3)			48.6
	10^{-4}	340 (7.5)	442 (2.6)			48.9
	10^{-3}	190 (13.3)	424 (2.7)			49.5

Table 7.3 – North Sea case study. Computational savings provided by the BLR solver during the factorization step. Factor of improvement due to BLR is indicated between parenthesis. The elapsed times required to perform the multi-rhs substitution step and to compute the gradient are also provided. F_{LU}: flops for the LU factorization. $T_{LU}(s)$: elapsed time for the LU factorization. $T_s(s)$: average time for 1 solve. The first and second numbers are related to the incident and adjoint wavefields, respectively. $T_{ms}(s)$: elapsed time for 4604 solves (incident + adjoint wavefields). $T_g(mn)$: elapsed time to compute the FWI gradient. This time also includes the IO tasks.

with the problem size will balance the higher complexity of the LU factorization relative to the solution phase and help to address large-scale problems.

As reported in column $T_g(mn)$ of Table 7.3, the elapsed times required to compute one gradient with the FR solver are of the order of 9.9mn, 21.3mn, and 69mn for the 5Hz, 7Hz and 10Hz frequencies, respectively, while those with the BLR solver are of the order of 9.1mn, 17.1mn, and 49.5mn, respectively.

For a fair assessment of the FWI speedup provided by the BLR solver, it is also important to check the impact of the BLR approximation on the line search and hence the number of FWI gradients computed during FWI (Table 7.4). On the 70m grid where the impact of the BLR errors is expected to be the largest one (as indicated in Table 7.2), the inversion with the BLR solver ($\varepsilon = 10^{-3}$) computes 82 gradients against 77 gradients for the three other settings (FR solver and BLR solver with $\varepsilon = 10^{-4}$ and $\varepsilon = 10^{-5}$) for a total of 70 FWI iterations. On the 7Hz grid, the FR solver and the BLR solver with $\varepsilon = 10^{-5}$ compute 39 gradients against 30 gradients with the BLR solver with $\varepsilon = 10^{-4}$ and $\varepsilon = 10^{-3}$ for a total of 20 FWI iterations. On the 10Hz grid, the four inversions compute 16 gradients for a total of 10 FWI iterations.

The elapsed time to perform the FWI is provided for each grid in Table 7.4. The entire FWI application takes 49hr, 40hr, 36hr and 37.8hr with the FR solver and the BLR solver with $\varepsilon = 10^{-5}$, $\varepsilon = 10^{-4}$ and $\varepsilon = 10^{-3}$, respectively. We remind that

the FWI application performed with the BLR solver with $\varepsilon = 10^{-3}$ takes more time than with $\varepsilon = 10^{-4}$ because more gradients were computed on the 70m grid.

h(m)	Freq(Hz)	#c	ε	$\#it$	$\#g$	T_{FWI} (hr)
70	3.5, 4, 4.5, 5	240	FR	70	77	14.0
			10^{-5}	70	77	12.9
			10^{-4}	70	77	12.7
			10^{-3}	70	82	14.0
50	7	320	FR	20	39	14.5
			10^{-5}	20	39	12.0
			10^{-4}	20	30	9.1
			10^{-3}	20	30	9.0
35	10	680	FR	10	16	20.7
			10^{-5}	10	16	14.5
			10^{-4}	10	16	14.2
			10^{-3}	10	16	14.8

Table 7.4 – North Sea case study. FWI cost. $\#it$ and $\#g$ are the number of FWI iterations and the number of gradients computed on each grid. T_{FWI} is the elapsed time for FWI on each grid. The total time for the FWI application is 49.2hr, 39.4hr, 36hr and 37.8hr for the FR, $BLR(10^{-5})$, $BLR(10^{-4})$ and $BLR(10^{-3})$ solvers, respectively.

We conclude from this analysis that, for this case study, the BLR solver with $\varepsilon = 10^{-4}$ provides the best trade-off between the number of FWI gradients required to reach a given value of the misfit function and the computational cost of one gradient computation at low frequencies. At the 7Hz and 10Hz frequencies, the BLR solver with $\varepsilon = 10^{-3}$ provides the smaller computational cost without impacting the quality of the FWI results.

Finally, we conclude by providing a projected performance if the entire FWI simulation was carried out again with the new FR and BLR solvers. These projected results are reported in Table 7.5. Both the FR and BLR factorizations have been improved; we assume the UFCS+LUAR variant is used for the BLR solver. Moreover, while the FR solution phase has not changed, we estimate the performance of the BLR solution phase once it will have been reasonably optimized, as described in Section 9.2. We assume the time gains are equal to half the potential gain due to the factor size reduction. Finally, we assume that the IO tasks have been divided by 2 due to code optimizations. Under these assumptions, we obtain a projected total cost for the FWI application of 31.9hr and 19.4hr using the FR and BLR ($\varepsilon = 10^{-3}$) solvers, respectively.

7.1.5 Section conclusion

While 3D frequency-domain FWI based on sparse direct methods is generally considered intractable, we have shown in this study its high computational efficiency to process OBC data in the visco-acoustic VTI approximation with quite

F(Hz)/h(m)	ε	$T_{LU}(s)$	$T_{ms}(s)$	$T_g(mn)$	$T_{FWI}(hr)$
5Hz/70m	FR	52	262	7.2	9.2
(240 cores)	10^{-3}	19	173	5.2	7.1
7Hz/50m	FR	226	598	16.5	10.7
(320 cores)	10^{-3}	70	382	10.3	5.2
10Hz/35m	FR	699	1542	44.9	12.0
(680 cores)	10^{-3}	181	961	26.5	7.1

Table 7.5 – North Sea case study. Projected computational saving provided by the BLR solver using the UFCS+LUAR factorization variant and the BLR solution phase. T_g is computed assuming the time for IO tasks has been divided by a factor 2 due to code optimizations. T_{FWI} is computed assuming the number of iterations and computed gradient would be the same as in Table 7.4. The total projected time for the FWI application is 31.9hr and 19.4hr for the FR and BLR ($\varepsilon = 10^{-3}$) solvers, respectively.

limited computational resources. The computational efficiency of the frequency-domain FWI relies on a suitable choice of a few discrete frequencies, and on the exploitation of Block Low-Rank approximations to accelerate the solver. This offers a suitable framework to preserve the fold resulting from dense seismic acquisitions during the stack procedure underlying FWI and hence build subsurface model with a high signal to noise ratio.

Although the FWI was limited to a maximum frequency of 10 Hz, it is probably reasonable to try to push the inversion up to a frequency of 15 Hz in the near future. For the case study presented here, this would require to manage computational grids with up to 60 million unknowns. Some preliminary performance and memory results are reported in Section 9.4. On the other hand, although the frequency decimation underlying efficient frequency-domain FWI is relatively neutral for mono-parameter FWI, the impact of this decimation will have to be assessed with care in the framework of multi-parameter reconstruction. This comment however also applies to time-domain FWI which is generally applied on a quite narrow frequency bandwidth. Frequency-domain FWI remains limited to a relatively narrow range of applications in terms of wave physics and acquisition geometries. Extension of our finite-difference stencil to TTI anisotropy is not straightforward and should lead to a significant computational overhead. Application to short-spread narrow-azimuth streamer data might not be beneficial since the cost of the LU factorizations might become prohibitive relative to the one of the solution steps and the number of frequencies to be managed should be increased to prevent wraparound artefacts. Despite these limitations, 3D frequency-domain FWI on OBC data based on sparse direct solver can also be viewed as an efficient tool to build an initial visco-acoustic VTI subsurface model for subsequent elastic FWI of multi-component data. Feasibility of frequency-domain visco-elastic modeling based on sparse direct solver for multi-component/multiparameter FWI applications at low frequencies needs to be assessed and will be the aim of future work.

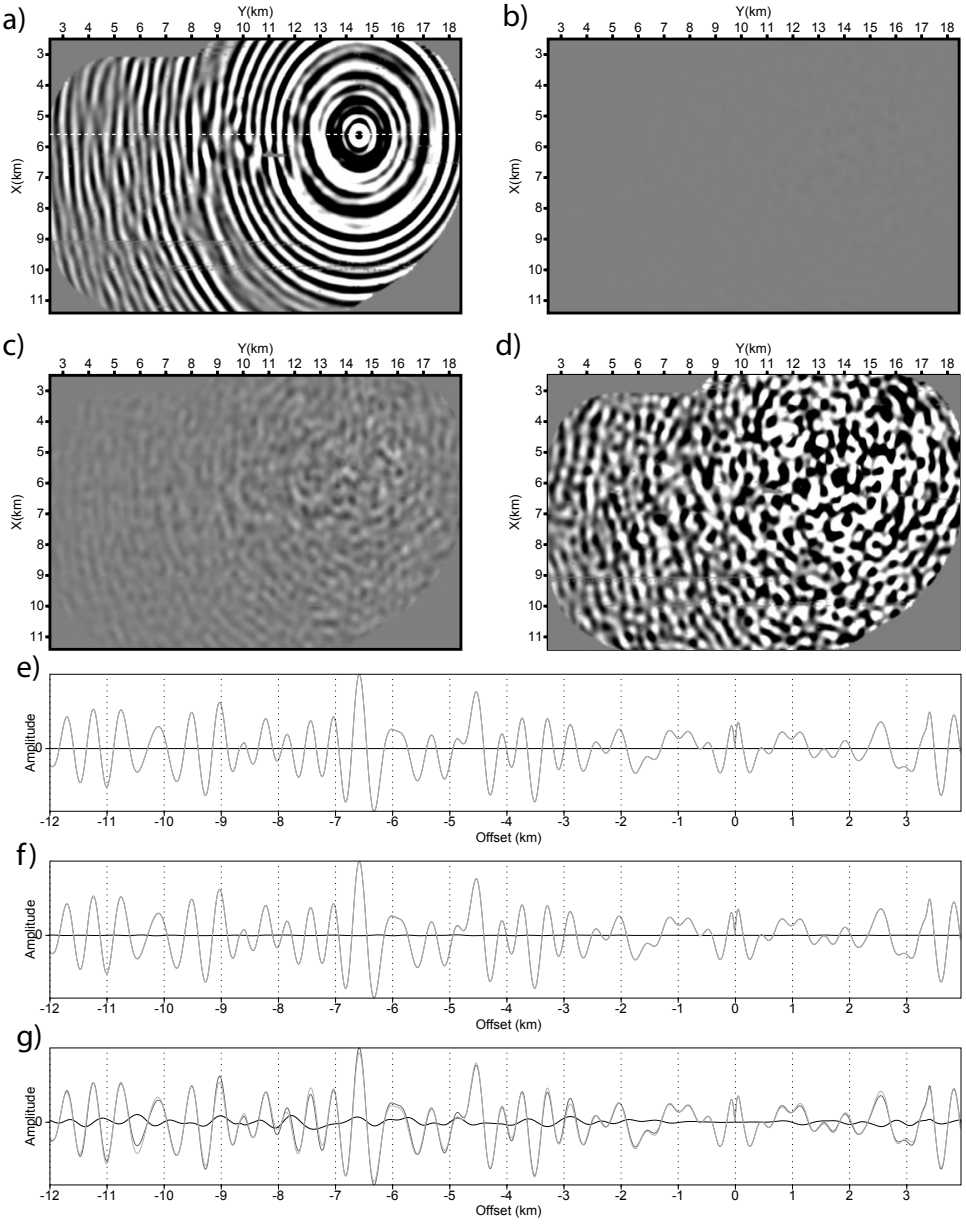


Figure 7.4 – North Sea case study. BLR modeling errors. (a) 5Hz receiver gather (real part) computed with the FR solver. (b) Difference between the receiver gathers computed with the BLR ($\varepsilon = 10^{-5}$) and the FR (a) solvers. (c) Same as (b) for $\varepsilon = 10^{-4}$. (d) Same as (b) for $\varepsilon = 10^{-3}$. Residual wavefields in (b-d) are multiplied by a factor 10 before plot. The FR wavefield (a) and the residual wavefields after multiplication by a factor 10 (b-d) are plotted with the same amplitude scale defined by a percentage of clip equal to 85 of the FR-wavefield amplitudes (a). (e-g) Direct comparison between the wavefields computed with the FR solver (dark gray) and the BLR solver (light gray) for $\varepsilon = 10^{-5}$ (e), $\varepsilon = 10^{-4}$ (f), $\varepsilon = 10^{-3}$ (g) along a X profile intersecting the receiver position (dash line in (a)). The difference is shown by the thin black line. Amplitudes are scaled by a linear gain with offset.

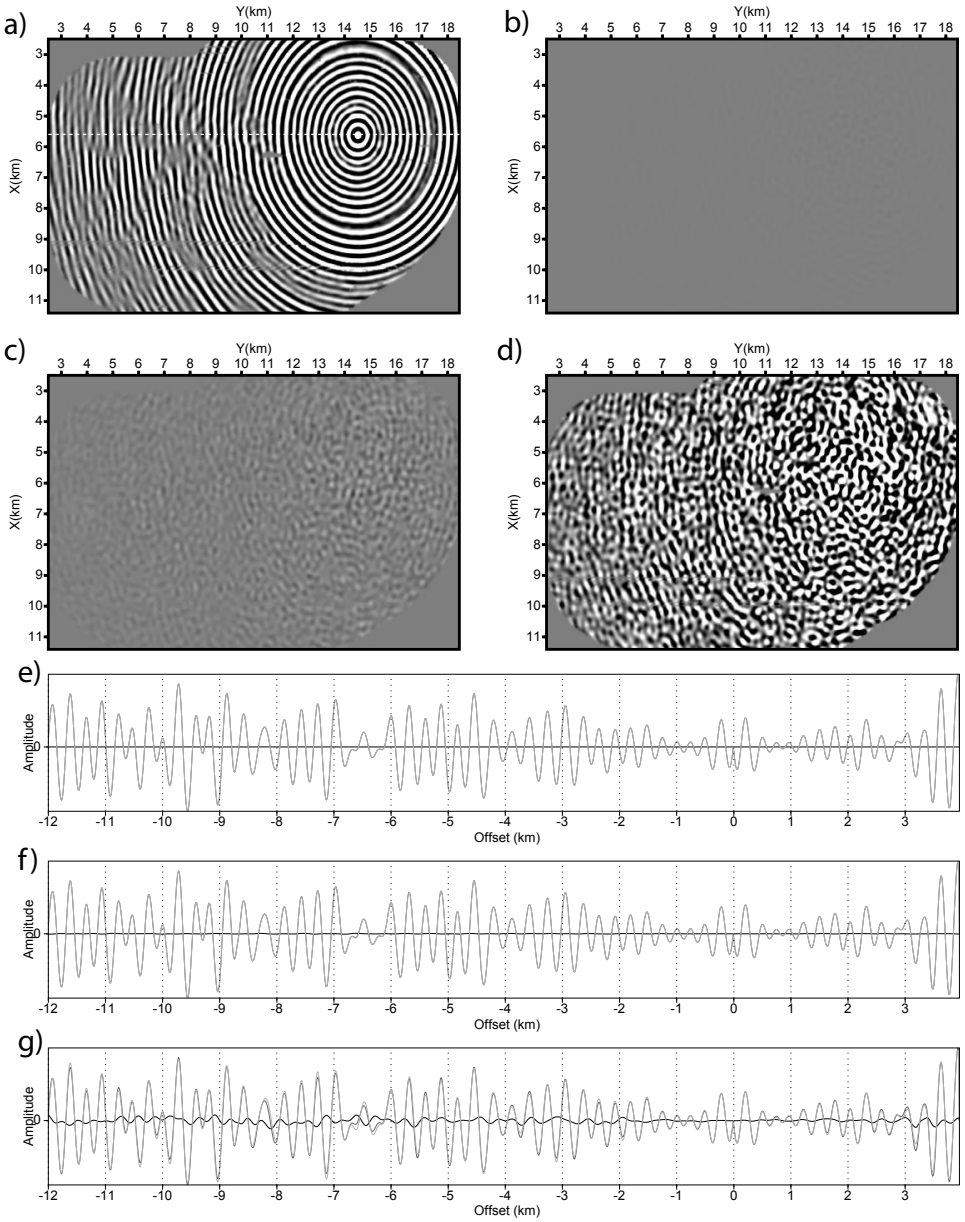


Figure 7.5 – North Sea case study. BLR modeling errors. Same as Figure 7.4 for the 7Hz frequency. The simulations are performed in the same subsurface model obtained after a 7Hz inversion (not shown here). The same percentage of clip (85%) of the FR-wavefield amplitudes and the same amplitude scaling of the residuals wavefields (multiplication by a factor 10 before plot) than those used in Figure 7.4 are used for plot.

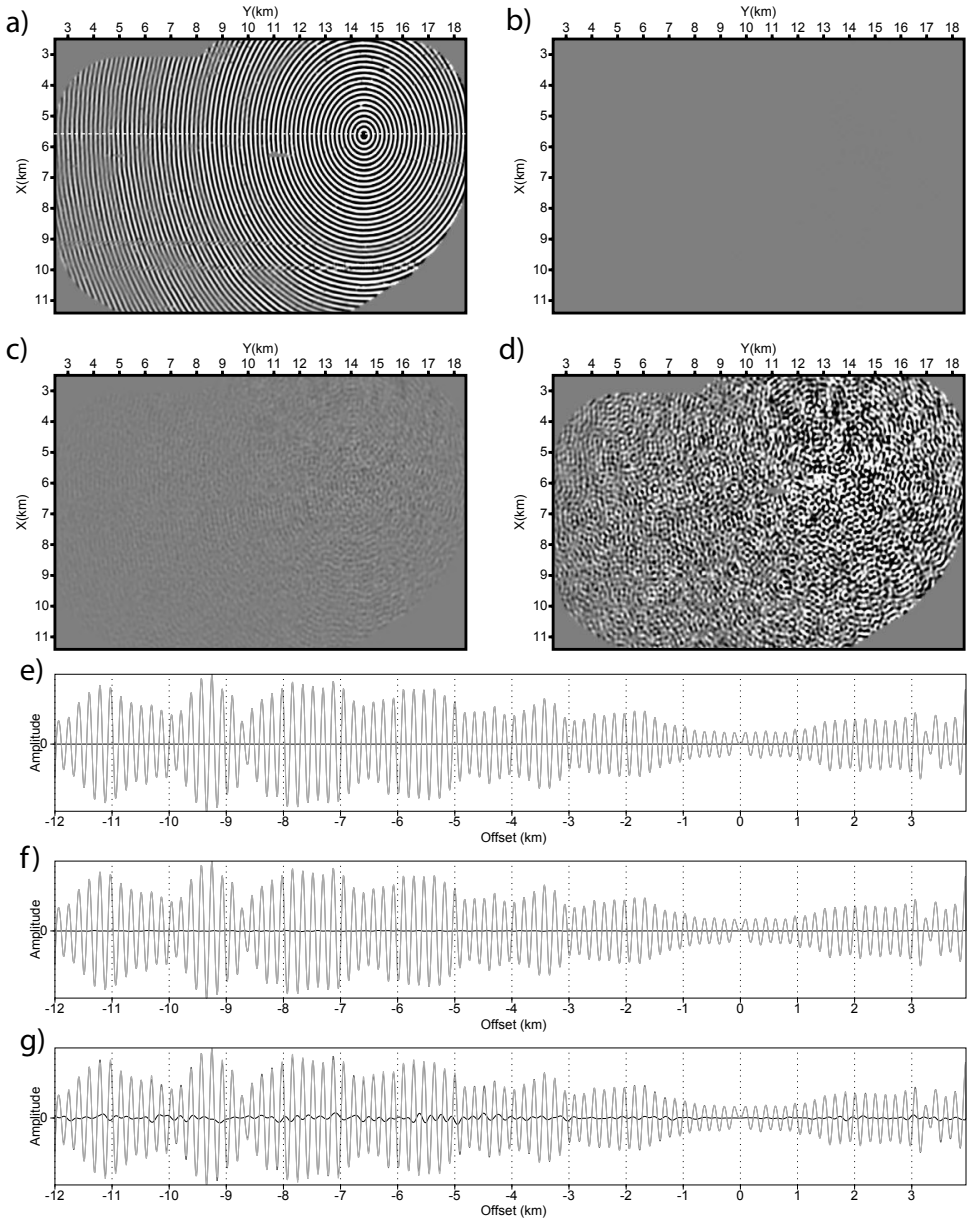


Figure 7.6 – North Sea case study. BLR modeling errors. Same as Figure 7.4 for the 10Hz frequency. The simulations are performed in the same subsurface model obtained after a 10Hz inversion. The same percentage of clip (85%) of the FR-wavefield amplitudes and the same amplitude scaling of the residuals wavefields (multiplication by a factor 10 before plot) than those used in Figure 7.4 are used for plot.

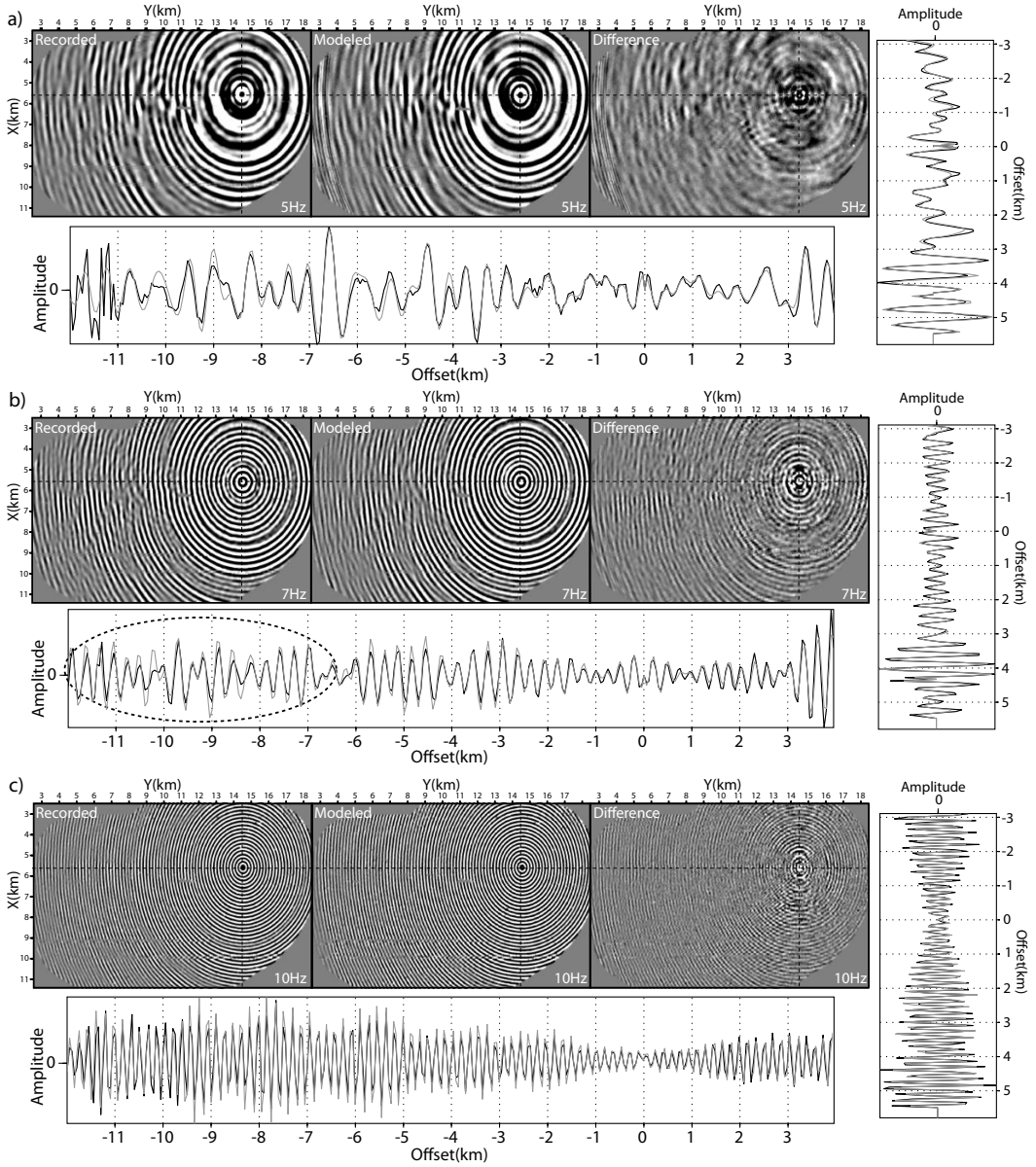


Figure 7.7 – North Sea case study. Data fit achieved with the BLR solver ($\varepsilon = 10^{-3}$). (a-c) 5Hz(a), 7Hz(b) and 10Hz(c) frequencies. Left (X-Y) panel shows the recorded data. Middle panel shows the modeled data computed in the FWI model inferred from the inversion of the current frequency. Right panel is the difference. Bottom and right amplitude-offset panels show direct comparison between the recorded (black) and the modeled (gray) data (real part) along a dip and a cross profiles intersecting the receiver position (dash lines in (X-Y) panels). Amplitudes are corrected for geometrical spreading. The ellipse delineates an offset range for which modeled amplitudes are overestimated (see text for interpretation).

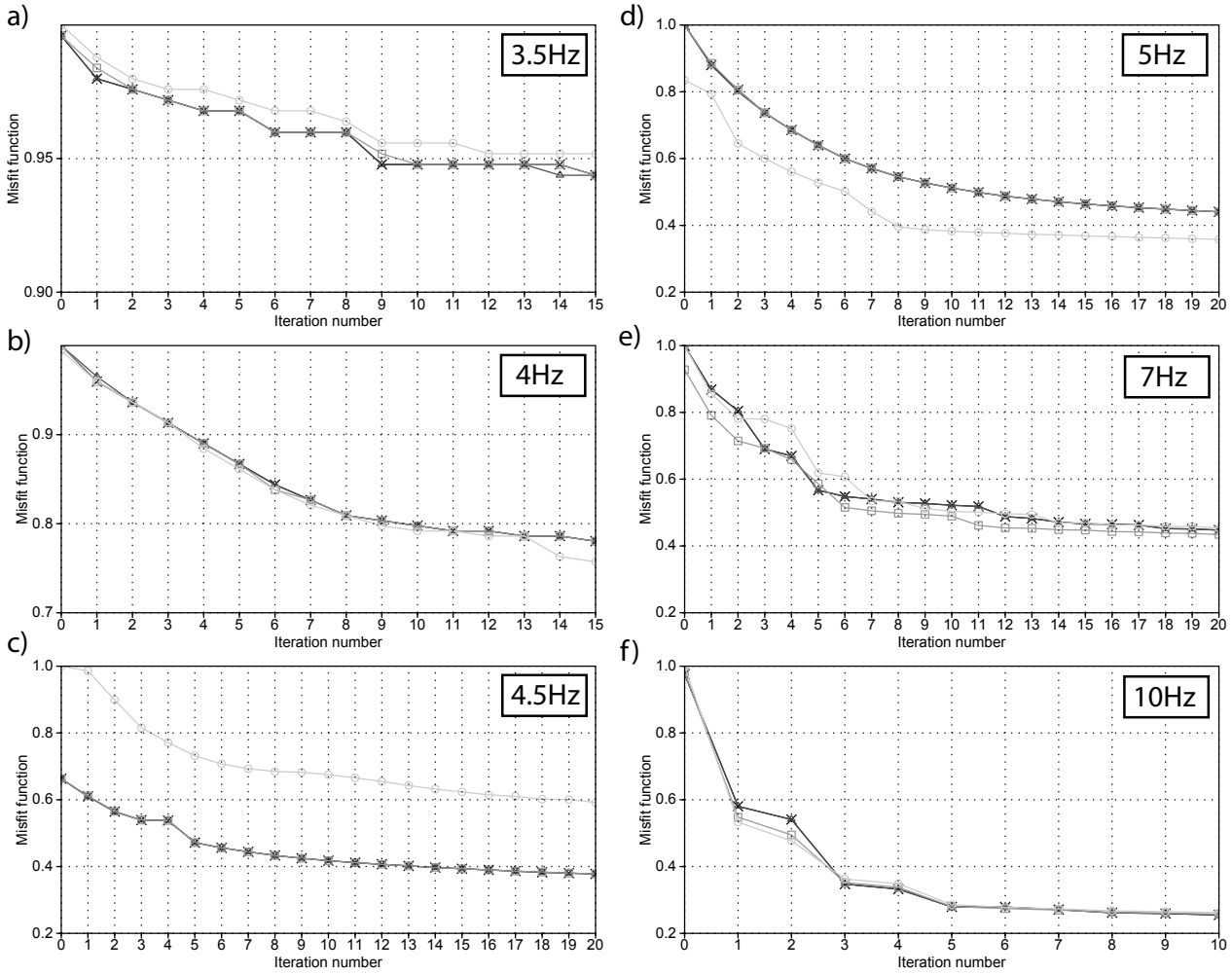


Figure 7.8 – North Sea case study. Misfit function versus iteration number achieved with the FR solver (black cross) and the BLR solver with $\varepsilon = 10^{-5}$(dark gray triangle), 10^{-4}(gray square), 10^{-3}(light gray circle). (a) 3.5Hz Inversion. (b) 4Hz inversion. (c) 4.5Hz inversion. (d) 5Hz inversion. (e) 7Hz inversion. (f) 10Hz inversion.

7.2 3D Controlled-source Electromagnetic inversion

In this section, we apply the BLR algorithms developed in the previous chapters to large-scale 3D CSEM problems. We first describe our frequency-domain finite-difference electromagnetic (EM) modeling approach. We search for the optimal threshold for the low-rank approximation that provides an acceptable accuracy for EM fields in the domain of interest, and analyze the reductions in flops, matrix factor size and computation time compared to the FR approach for linear systems of different sizes, up to 21 million unknowns. It is also shown that the gains due to low-rank approximation are significantly larger in a deep water setting (that excludes highly resistive air) than in shallow water. Finally, for a realistic 3D CSEM inversion scenario, we compare the computational cost of using the BLR multifrontal solver to that of running the inversion using an iterative solver.

7.2.1 Applicative context

Marine controlled-source electromagnetic (CSEM) surveying is a widely used method for detecting hydrocarbon reservoirs and other resistive structures embedded in conductive formations (Ellingsrud et al., 2002; Constable, 2010; Key, 2012). The conventional method uses a high powered electric dipole as a current source, which excites low-frequency (0.1-10 Hz) EM fields in the surrounding media, and the responses are recorded by electric and magnetic seabed receivers. In an industrial CSEM survey, data from a few hundred receivers and thousands of source positions are inverted to produce a 3D distribution of subsurface resistivity.

In order to invert and interpret the recorded EM fields, a key requirement is to have an efficient 3D EM modeling algorithm. Common approaches for numerical modeling of the EM fields include the finite-difference (FD), finite-volume (FV), finite-element (FE) and integral equation methods (see reviews e.g. by Avdeev (2005), Börner (2010), and Davydycheva (2010)). In the frequency domain, these methods reduce the governing Maxwell equations to a system of linear equations $Mx = s$ for each frequency, where M is the system matrix defined by the medium properties and grid discretization, x is a vector of unknown EM fields, and s represents the current source and boundary conditions. For the FD, FV and FE methods, the system matrix M is sparse, and hence the corresponding linear system can be efficiently solved using sparse iterative or direct solvers.

Iterative solvers have long dominated 3D EM modeling algorithms, see for example, Newman and Alumbaugh (1995), Smith (1996), Mulder (2006), Puzyrev et al. (2013), and Jaysaval, Shantsev, Kethulle de Ryhove, and Bratteland (2016), among others. This is due to the fact they are relatively cheap and provide better scalability in parallel environments. However, their robustness usually depends on the numerical properties of M and they often require problem-specific preconditioners. In addition, their computational cost grows linearly with an increasing number of sources (i.e. the right-hand side s vectors) (Blome, Maurer, and Schmidt, 2009; Oldenburg, Haber, and Shekhtman, 2013), and this number may reach many thousands in an industrial CSEM survey.

Direct solvers, on the other hand, are in general more robust, reliable, and well suited for multi-source simulations. Unfortunately, the amount of memory and floating point operations required for the factorization can become huge as the problem size grows. Therefore the application of direct solvers to 3D problems has traditionally been considered computationally too demanding. However, recent advances in sparse matrix factorization packages, along with the availability of modern parallel computing environments, have created the necessary conditions to attract interest in direct solvers in the case of 3D EM problems of moderate size, see for example, Blome et al. (2009), Streich (2009), Silva, Morgan, MacGregor, and Warner (2012), Puzyrev, Koric, and Wilkin (2016), and Jaysaval, Shantsev, and Kethulle de Ryhove (2014).

There have so far been no reports on the application of multifrontal solvers with low-rank approximations to 3D EM problems. EM fields in geophysical applications usually have a diffusive nature which makes the underlying equations fundamentally different from those describing seismic waves. They are also very different from the thermal diffusion equations since EM fields are vectors. Most importantly, the scatter of material properties in EM problems is exceptionally large, for example, for marine CSEM applications resistivities of seawater and resistive rocks often differ by four orders of magnitude or more. On top of that, the air layer has an essentially infinite resistivity and should be included in the computational domain unless water is deep. Thus, elements of the system matrix may vary by many orders of magnitude, which can affect the performance of low-rank approaches for matrix factorization.

Indeed, the geometrical principle behind the admissibility condition may not work for EM problems as well as it does for seismic ones since extreme variations in electrical resistivity over the system lead to vastly different matrix elements. Thus, for two clusters to be weakly connected and have a low rank interaction, it is not sufficient that the corresponding unknowns be located far from each other in space. One should also require that the medium between them not be highly resistive. We will see in the next sections that this nuance may strongly affect the BLR gains for CSEM problems involving the highly resistive air layer. Note that, from a user perspective, it is very desirable that the approach is able to automatically adapt the amount of low-rank compression to the physical properties of the medium.

In all simulations the system matrix was generated using the finite-difference modeling code presented in Jaysaval et al. (2014). The simulations were carried out on either the **eos** supercomputer or the **farad** machine.

7.2.2 Finite-difference electromagnetic modeling

If the temporal dependence of the EM fields is $e^{-i\omega t}$ where ω denotes the angular frequency, the frequency-domain Maxwell equations in the presence of a current source J are

$$\nabla \times E = i\omega\mu H, \tag{7.7}$$

$$\nabla \times H = \bar{\sigma} E - i\omega\epsilon E + J, \tag{7.8}$$

where E and H are, respectively, the electric and magnetic field vectors, and μ and ϵ are, respectively, the magnetic permeability and dielectric permittivity. The value of μ is assumed to be constant and equal to the free space value $\mu_0 = 4\pi 10^{-7}$ H/m. $\bar{\sigma}$ is the electric conductivity tensor and can vary in all the 3 dimensions. In a vertical transverse isotropic (VTI) medium, $\bar{\sigma}$ takes the form

$$\bar{\sigma} = \begin{bmatrix} \sigma_H & 0 & 0 \\ 0 & \sigma_H & 0 \\ 0 & 0 & \sigma_V \end{bmatrix}, \tag{7.9}$$

where σ_H (or $1/\rho_H$) and σ_V (or $1/\rho_V$) are, respectively, the horizontal and vertical conductivities (inverse resistivities).

The magnetic field can be eliminated from equations (7.7) and (7.8) by taking the curl of equation (7.7) and substituting it in equation (7.8). This yields a vector Helmholtz equation for the electric field

$$\nabla \times \nabla \times E - i\omega\mu\bar{\sigma}E - \omega^2\mu\epsilon E = i\omega\mu J. \tag{7.10}$$

For typical CSEM frequencies in the range of 0.1 to 10 Hz, the displacement current is negligible as $\sigma_H, \sigma_V \gg \omega\epsilon$. Therefore, equation (7.10) becomes

$$\nabla \times \nabla \times E - i\omega\mu\bar{\sigma}E = i\omega\mu J. \tag{7.11}$$

We assume that the bounded domain $\Omega \subset \mathbb{R}^3$ where equation (7.11) holds is big enough for EM fields at the domain boundaries $\partial\Omega$ to be negligible and allow the perfectly conducting Dirichlet boundary conditions

$$\hat{n} \times E|_{\partial\Omega} = 0 \text{ and } \hat{n} \cdot H|_{\partial\Omega} = 0 \tag{7.12}$$

to be applied, where $\hat{n}$ is the outward normal to the boundary of the domain.

In order to compute electric fields, equation (7.11) is approximated using finite differences on a Yee grid (Yee, 1966) following the approach of Newman and Alumbaugh (1995). This leads to a system of linear equations

$$Mx = s, \tag{7.13}$$

where M is the system matrix of order $3N$ for a modeling grid with $N = N_x \times N_y \times N_z$ cells, x is the unknown electric field vector of dimension $3N$, and s, also of dimension $3N$, is the source vector resulting from the right-hand side of equation (7.11). The matrix M is a complex-valued sparse matrix, having up to 13 nonzero entries in a row. In general, M is unsymmetric but it can easily be made symmetric (but non-Hermitian) by simply applying scaling factors to the discretized finite difference equations (cf. e.g. Newman and Alumbaugh (1995)). For all simulations in this section, we consider the matrix symmetric because it reduces the solution time of equation (7.13) by approximately a factor of two and increases the maximum feasible problem size. Finally, after computing the electric field by solving the matrix equation (7.13), Faraday's law (equation (7.7)) can be used to calculate the magnetic field.

7.2.3 Models and system matrices

In order to examine the performance of the BLR solver, let us consider the two earth resistivity models depicted in Figures 7.9 and 7.10.

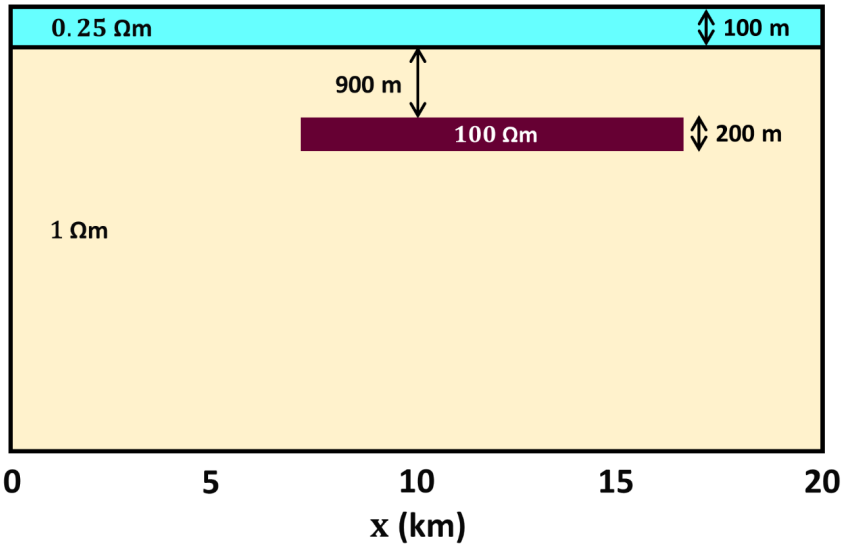


Figure 7.9 – Vertical cross-section through a simple isotropic 3D resistivity model (H-model) at y = 10 km.

The model in Figure 7.9 (which we hereafter call the H-model) is a simple isotropic half-space earth model in which a 3D reservoir of resistivity 100 Ωm and dimension $10 \times 10 \times 0.2$ km^3 is embedded in a uniform background of resistivity 1 Ωm. It is a shallow-water model: the seawater is 100 m deep and has a resistivity of 0.25 Ωm. The dimensions of the H-model are $20 \times 20 \times 10$ km^3. A deep-water variant where the water depth is increased to 3 km (hereafter referred to as the D-model) is also considered and will be described further later. The H and the D models lead to matrices with the same size and structure but different numerical properties.

The model in Figure 7.10 (hereafter the S-model) is the SEAM (SEG Advanced Modeling Program) Phase I salt resistivity model. It is a complex 3D earth model designed by the hydrocarbon exploration community and widely used to test 3D modeling and inversion algorithms. The S-model is representative of the geology in the Gulf of Mexico, its dimension is $35 \times 40 \times 8.6$ km^3, and it includes an isotropic complex salt body of resistivity 100 Ωm and several hydrocarbon reservoirs (Stefani, Frenkel, Bundalo, Day, and Fehler, 2010). The background formation has VTI anisotropy and horizontal ρ_H and vertical ρ_V resistivities varying mostly in the range 0.5 – 0.6 m. The seawater is isotropic with resistivity 0.3 Ωm, and thickness varying from 625 to 2250 m. However, we chose to remove 400 m from the water column in the original SEAM model, thereby resulting in water depths varying from 225 to 1850 m, to make sure that the air-wave (the signal components propagating from source to receiver via the air) has a significant impact on the data.

The top boundary of both models included an air layer of thickness 65 km and resistivity 10^6 Ωm. On the five other boundaries 30 km paddings were added to make

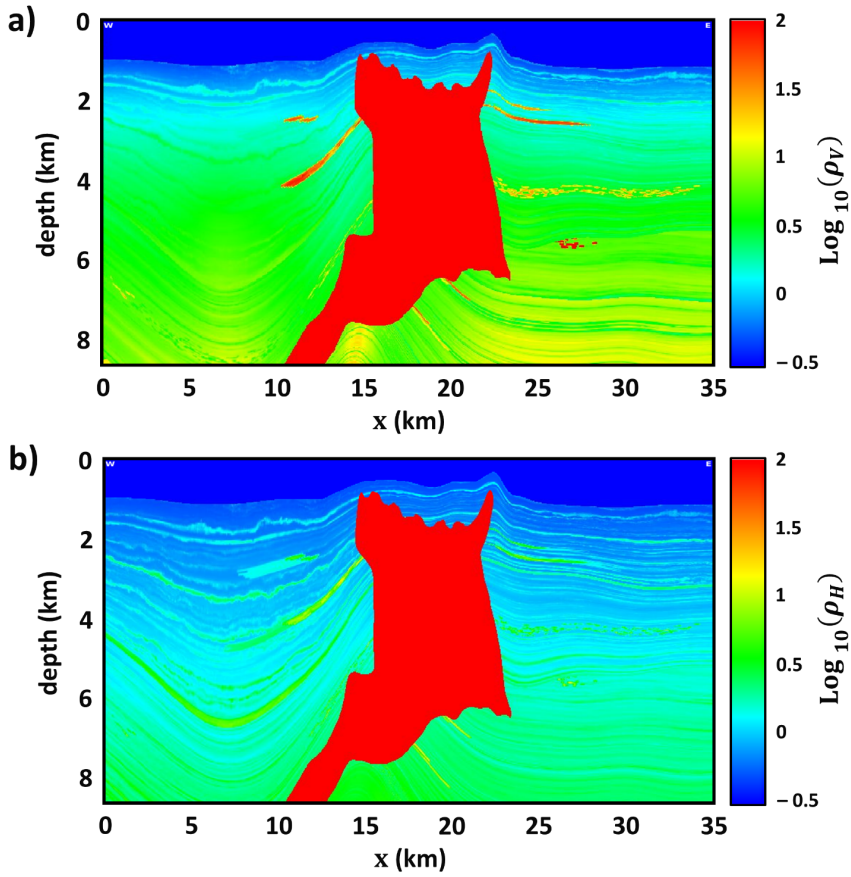


Figure 7.10 – Vertical cross-section for vertical (a) and horizontal (b) resistivities of the SEAM model (S-model) at y = 23.7 km.

sure the combination of strong airwave and zero-field Dirichlet boundary conditions does not lead to edge effects. The current source was an x-oriented horizontal electric dipole (HED) with unit dipole moment at frequency of 0.25 Hz located 30 m above the seabed.

In the padded regions the gridding was severely non-uniform and followed the rules described by Jaysaval et al. (2014), where the air was discretized with 15 cells and the other boundaries with 7 cells. Apart from the padded regions, we used finite-difference grids that were uniform in all three directions. The parameters of five uniform grids used to discretize the H-, D- and S-models are listed in Table 7.6 the cell sizes, number of cells, resulting number of unknowns, and the number of nonzero entries in the system matrix. These discretizations resulted in six different system matrices: H1, H3/D3 and H17 for the H- and D-models; and S3 and S21 for the S-model. The numbers represent the approximate number of unknowns, in millions, in the linear systems associated with each matrix; for example, the linear system corresponding to matrix S21 had about 21 million unknowns. So far, for 3D geophysical EM problems, the largest reported complex-valued linear system that has been solved with a direct solver had 7.8 million unknowns (Puzyrev et al., 2016).

Grid	Matrix	$dx = dy$	dz	N_x	N_y	N_z	$3N$	NNZ
G1	H1	400	200	64	64	74	909,312	11,658,644
G2	H3/D3	200	200	114	114	74	2,885,112	37,148,644
G3	H17	100	100	214	214	127	17,448,276	225,626,874
G4	S3	480	80	98	87	130	3,325,140	42,836,538
G5	S21	240	40	181	160	237	20,590,560	266,361,112

Table 7.6 – Parameters of the uniform grids used to discretize the 3D shallow-water H-model, deep-water D-model and the SEAM S-model. Here dx, dy, and dz are the cell sizes in meters, while N_x, N_y, and N_z are the number of cells in the x-, y-, and z-directions that also include non-uniform cells added to pad the model at the edges. $3N = 3N_x \times N_y \times N_z$ is the total number of unknowns and NNZ is the total number of nonzero entries in the system matrix.

7.2.4 Choice of the low-rank threshold ε

It is necessary to find out which choices of low-rank threshold ε provide acceptable CSEM solution accuracies, and what are the associated reductions in factor size, flops and run time.

We define the relative residual norm δ as the ratio of the residual norm $\|s - Mx^{\varepsilon}\|$ for an approximate BLR solution x^{ε} with a low-rank threshold ε and the zero-solution residual norm $\|s\|$:

$$\delta = \frac{\|s - Mx^{\varepsilon}\|}{\|s\|}. \tag{7.14}$$

The linear systems corresponding to the matrices H1, H3, S3, H17 and S21 are then solved for different values of ε to examine its influence on δ. For all the linear systems, the RHS vector s corresponds to an HED source located 30 m above the seabed in the center of the model. Figure 7.11 shows the value of δ plotted as a function of the low-rank threshold ε. The different curves on each plot correspond to different numbers of iterative refinement steps. Iterative refinement improves the accuracy of the solution of linear systems by the iterative process illustrated in Algorithm 1.10.

A conventional choice for the convergence criterion for iterative solvers used for EM problems, see for example, Newman and Alumbaugh (1995), Smith (1996), and Mulder (2006) is $\delta \leq 10^{-6}$. Figure 7.11 shows that the low-rank threshold ε should be $\leq 10^{-7}$ to fulfill this criterion for all the linear systems. Iterative refinement reduces the relative residual norm δ, but it comes at the cost of an additional forward-backward substitution per refinement step at the solution stage. For the case of thousands of RHSs, which is typical for a CSEM inversion problem, these iterative steps may be too costly. Therefore, the focus of the following discussions is on the BLR solution obtained without any iterative refinement. It follows from Figure 7.11 that the corresponding curves $\delta(\varepsilon)$ look quite similar for all the matrices included in the study. This is a good sign that gives reason to hope that choosing

$\varepsilon \leq 10^{-7}$ will guarantee good accuracy of the solution for most practical CSEM problems. Furthermore, the fact that the accuracy in the solution smoothly decreases when ε increases, also adds confidence in the robustness and usability of the BLR method in a production context.

Let us now investigate the accuracy of the BLR solution x^{ε} for different values of ε and analyze the spatial distribution of the solution error. The error is defined as the relative difference between the BLR solutions x^{ε} and the full-rank solution x :

$$\xi_{m,i,j,k} = \sqrt{\frac{|x^{\varepsilon}_{m,i,j,k} - x_{m,i,j,k}|^2}{\left(|x^{\varepsilon}_{m,i,j,k}|^2 + |x_{m,i,j,k}|^2\right)/2 + \eta^2}}, \tag{7.15}$$

for $m = x$, y and z; $i \in [1; N_x]$, $j \in [1; N_y]$, and $k \in [1; N_z]$. Here, $x_{m,i,j,k}$ represents the m-component of the electric field at the (i,j,k)-th node of the grid, while $\eta = 10^{-16}$ V/m represents the ambient noise level. Figure 7.12 shows 3D maps of the relative difference $\xi_{x,i,j,k}$ between x^{ε} and x for the x-component of the electric field for matrix H3.

In all maps, the relative error in the air is orders of magnitude larger than in the water or formation. Similar observations have been earlier reported by Grayver and Streich (2012). Fortunately, large errors in the air do not create a problem in most practical CSEM applications. For marine CSEM inversion one needs very high accuracy for the computation of the EM fields at the seabed receivers (to compare them to the measured data), as well as reasonably accurate fields in the whole inversion domain (to compute the corresponding Jacobians and/or gradients). However, one never inverts for the air resistivity, hence we can exclude the air from the analysis and focus on the solution errors only in the water and the earth.

One can see from Figure 7.12 that for the smallest low-rank threshold, $\varepsilon = 10^{-10}$, the relative error $\xi_{x,i,j,k}$ in water and formation is negligible ($\approx 10^{-4}$), but it increases for larger ε, and for $\varepsilon = 10^{-8}$ and 10^{-7} reaches $1-2\%$ at depth, though it remains negligible close to the seabed and at shallow depths. For $\varepsilon = 10^{-6}$ the error exceeds 10% in the deeper part of the model, implying that the BLR solution x^{ε} obtained with $\varepsilon = 10^{-6}$ is of poor quality. At the same time, solutions obtained with $\varepsilon \leq 10^{-7}$ are accurate enough and can be considered appropriate for CSEM modeling and inversion. We also computed the error $\xi_{z,i,j,k}$ for the z-component of the electric field and found very similar behavior as in Figure 7.12.

This study on the errors introduced by BLR approximations in the EM modeling has been conducted with the FSCU variant. While it would be interesting to reproduce it with the other more advanced variants (e.g. UFCS+LUAR which is well suited for these matrices), we expect the results would lead to the same conclusions.

Next, we discuss how the complexity of the BLR solver is affected by removing the air layer, which strongly reduces the scale of resistivity variations in the model.

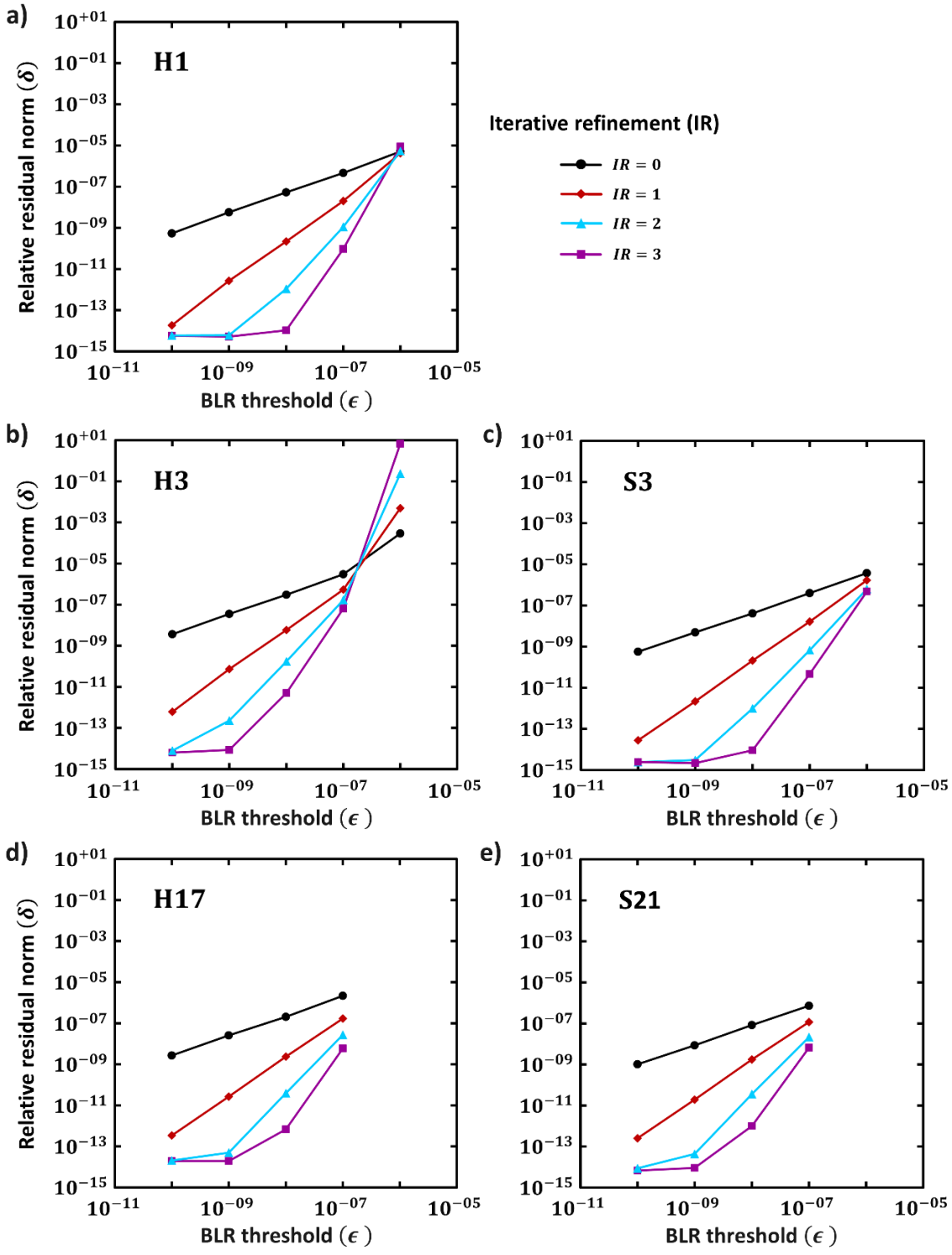


Figure 7.11 – Plots of the relative residual norm δ as a function of the low-rank threshold ε for linear systems corresponding to matrices H1, H3, S3, H17, and S21. The residual δ is always below 10^{-6} if one chooses $\varepsilon \leq 10{-}7$.

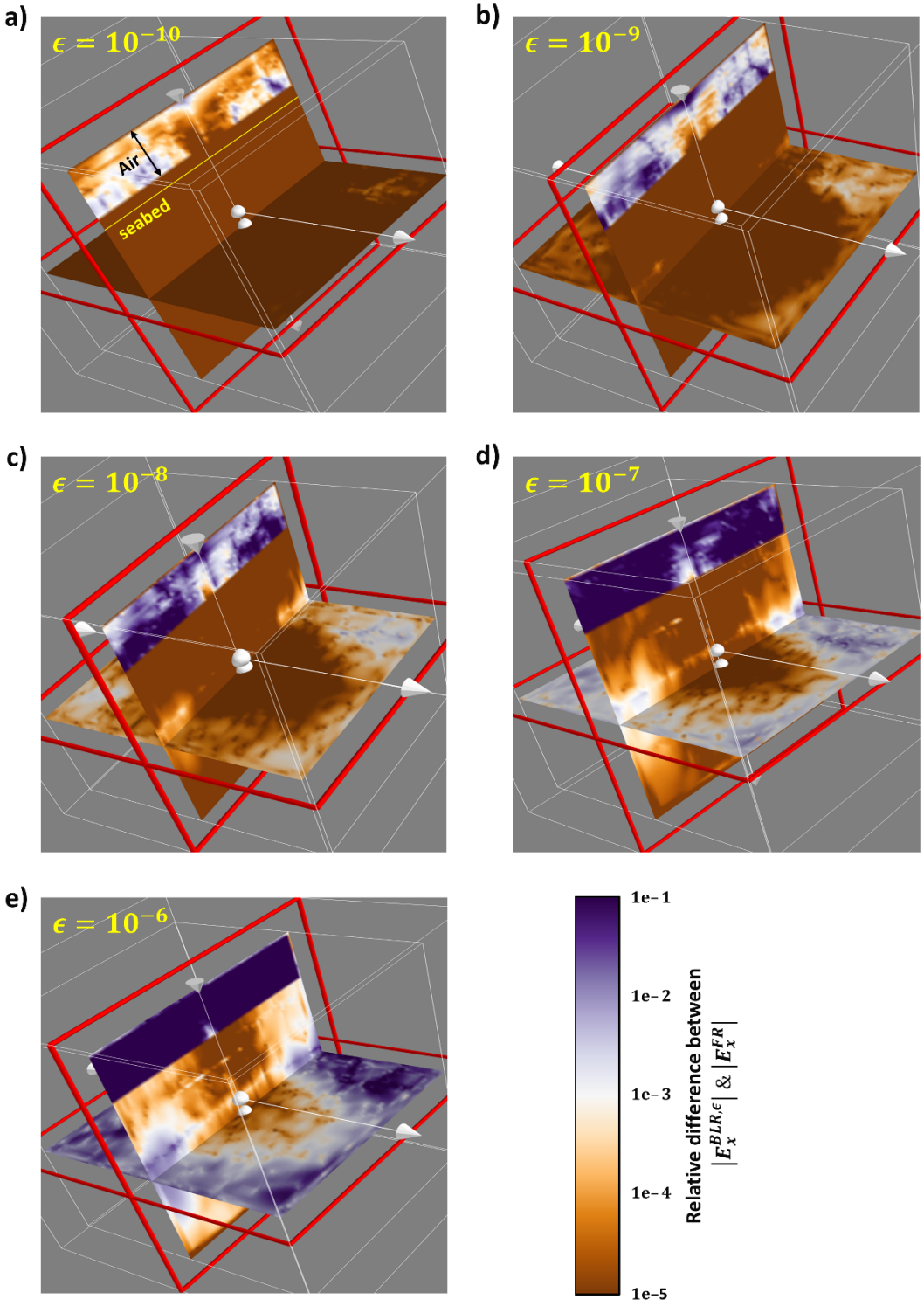


Figure 7.12 – Relative difference between the BLR solution x^{ε} for different low-rank thresholds ε, and the FR solution x for a linear system corresponding to matrix H3. For $\varepsilon = 10^{-7}$, the solution accuracy is acceptable everywhere except in the air layer at the top. The results are for the x-component of the electric field. The air and PML layers are not to scale.

7.2.5 Deep water versus shallow water: effect of the air

The benefits of the BLR solver depend on how efficiently blocks of frontal matrices can be compressed using low-rank approximations. Compression of a block matrix $A_{\sigma\tau}$ is expected to be efficient when the spatial domains corresponding to unknowns σ and τ are far from each other and the two sets of unknowns are thus weakly connected. One complication for CSEM problems is the presence of the insulating air layer whose resistivity is typically many orders of magnitude higher than that in the rest of the computational domain. EM signals propagate through the air almost instantaneously, as compared to relatively slow propagation through conductive water or sediments. Therefore, two regions located close to the air are effectively connected to each other via the so called air-wave even if these regions are geometrically very far apart. It is interesting to know whether this interconnectivity via the air layer can degrade the low-rank properties of corresponding matrices and affect performance of the BLR solver. Therefore, in this section we present additional simulations for earth models that do not include an air layer.

The results presented in the previous sections are based on a shallow-water H-model (water depth of 100 m) and the S-model with water depth varying from 225 to 1850 m. In both cases, the airwave strongly affects the subsurface response at most source-receiver offsets at the chosen frequency of 0.25 Hz (Andreis and MacGregor, 2008). On the other hand, if the water depth is increased to 3 km, the airwave contribution becomes negligible because EM fields are strongly attenuated in the conductive sea water (cf. e.g. Jaysaval, Shantsev, and Kethulle de Ryhove (2015)). Keeping this in mind, we built a deep-water model (D-model) from the shallow-water H-model by simply removing the air layer and adding 2.9 km of seawater so that the water layer becomes 3 km thick. The source is again an x-oriented HED with a frequency of 0.25 Hz placed 30 m above the seabed. The D-model was discretized using the same grid (Table 1) as the H-model, which resulted in matrix D3 having the same dimensions and number of nonzero entries as H3. We compare the results of simulations with the H- and D-models in Table 7.7.

Matrix, water depth	FR solver		BLR solver ($\varepsilon = 10^{-7}$)	
	Factor storage (GB)	Flops	Factor storage (%)	Flops (%)
H3, shallow	76	5.7×10^{13}	49.5	16.3
D3, deep	76	5.7×10^{13}	45.3	12.0

Table 7.7 – Factor storage and flops for the factorization of shallow-water (H3) and deep-water (D3) matrices. Values for the BLR solver are given as a percentage of the corresponding FR values.

The FR numbers are essentially identical for the shallow-water and deep-water matrices, which is somewhat expected as the matrices have the same number of unknowns and the same structure. On the other hand, it once again demonstrates the robustness of the direct solver whose efficiency is not affected by replacing the conductive water layer with extremely resistive air which changes values of the corresponding matrix elements by six to seven orders of magnitude. In contrast,

many iterative solvers struggle to converge in the presence of an air layer because it makes the system matrix more ill-conditioned due to high resistivities and large aspect ratios of some cells (cf. e.g. Mulder (2006)).

Most importantly, the gains achieved by using the BLR solver are larger for the deep-water matrix D3 than for the shallow-water matrix H3. This is especially evident for the factorization flops that amount to only 12.0% of the FR flops for D3, while for the H3 matrix that number increases to 16.3%. It is interesting to investigate how the observed difference in flops between the deep and shallow-water matrices depends on the matrix size. For that purpose we generated 11 additional grids for discretization of the H- and D-models. We started with a grid resulting in 4.9 million unknowns, which was constructed along the same lines as the other grids in Table 1, but with dx = dy = dz = 167 m. The next grid was obtained by making all its cells proportionally coarser by 5-10 percent, and so on for next grids. The rate at which the cell sizes were increasing, was identical in all parts of the model (air, water, formation, reservoir, non-uniform paddings) and all directions: x, y and z. The number of unknowns for the smallest grid was around 516,000.

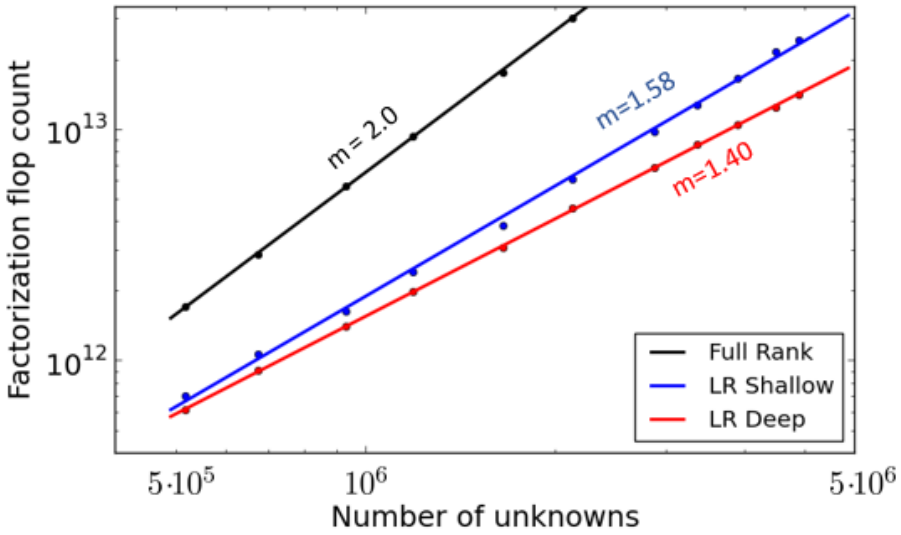


Figure 7.13 – Factorization flop complexity for shallow-water and deep-water matrices with different number of unknowns. The full-rank complexity $\mathcal{O}(N^2)$ is independent of the water-depth. The low-rank method reduces complexity for shallow-water matrices to $\mathcal{O}(N^m)$, with $m = 1.58$. The improvement is even stronger in deep water, where $m = 1.40$, indicating better BLR compression rates in the absence of resistive air.

Figure 7.13 shows how the factorization flops depend on the number N of unknowns for this set of grids. The FR solver has the expected $\mathcal{O}(N^2)$ complexity for both types of matrices. The BLR compression significantly reduces complexity, and the reduction depends on the matrix type. For matrices obtained from the shallow-water H-model, we observe an $\mathcal{O}(N^m)$ behavior with $m = 1.58 \pm 0.02$. At the same time, Figure 7.13 shows that for the deep-water D-model, the complexity is reduced even further, down to $m = 1.40 \pm 0.01$. This confirms that the BLR savings are

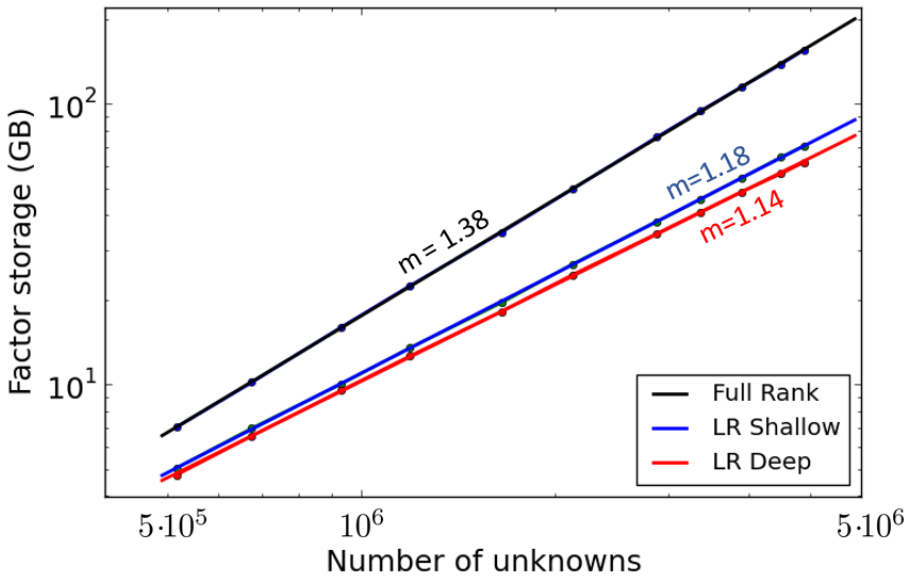


Figure 7.14 – Factor size for shallow-water and deep-water matrices with different number of unknowns N. The memory needed to store factors grows as a power law, $\mathcal{O}(N^m)$, and the use of the BLR solver significantly reduces the value of exponent m. The reduction is slightly stronger for the deep water case.

consistently larger for deep-water matrices and also shows that this effect becomes stronger for larger systems. For example, for the system with 4.9 million unknowns, the factorization of the shallow-water matrix requires 71 percent more flops than factorization of the deep-water matrix. Figure 7.14 shows data for the factor storage computed for the same set of 11 grids with different number of unknowns. One can see that the BLR method also reduces the factor storage complexity. Namely, the FR behavior of $\mathcal{O}(N^m)$ with $m = 1.38 \pm 0.01$ is changed to $m = 1.18 \pm 0.01$ for the shallow-water case, while for the deep-water case the BLR reductions are even stronger, down to $m = 1.14 \pm 0.01$. These values are very close to the exponents reported for 3D seismic problems, and in agreement with the theoretical predictions of Chapter 4.

The deep-water D-model is different from the H-model in two ways: it has a thicker water layer and does not contain air. We made an additional test run for a model with both a thick water layer and an air layer, and found the results to be similar to those for the H-model. It allows us to conclude that the observed improvements in the performance of BLR solver for the deep-water matrices are mainly due to removal of the highly resistive air layer. Presence of the air layer effectively interconnects model domains located close to the air interface, as discussed above. It should lead to higher rank of the corresponding block matrices and make low-rank approximations less efficient. In other words, the air introduces non-locality into the system: in some numerical schemes the air is simply excluded from the computational domain and replaced by a non-local boundary condition at the air-water interface (Wang and Hohmann, 1993). Thus, one may argue that the air effectively increases dimensionality of the system, which in turn, should also increase complexity.

7.2.6 Suitability of BLR solvers for inversion

Direct solvers are very well suited for multisource simulations since once the system matrix M is factorized, the solution for each RHS can be computed using relatively inexpensive forward-backward substitutions. Therefore, they are particularly attractive for applications involving large-scale CSEM inversions where the number of RHSs can reach several thousands. Nevertheless, the computational cost of the factorization phase remains huge and often dominates, tipping the balance in favor of simpler iterative solvers. For example, even for relatively small CSEM matrices with ≈3 million unknowns, an iterative solver is shown to be superior as long as the number of RHSs is kept below 150 (Grayver and Streich, 2012). In this section we benchmark our direct solver with and without BLR functionality against an iterative solver for an application in a realistic CSEM inversion based on the SEAM model.

Let us consider inversion of synthetic CSEM data over the S-model of Figure 7.10. We assume that $n_r = 121$ receivers are used to record simulated responses. For each receiver, HED sources are located along 22 towlines (11 in the x- and 11 in the y-directions) with an interline spacing of 1 km. Each towline has a length of 30 km with independent source positions 200 m apart. This implies 150 source positions per towline, or a total of $n_s = 22 \times 150 = 3300$ source positions. The model is discretized with grid G5 defined in Table 7.6, which results in system matrix S21 with 20.6 million unknowns. The frequency is 0.25 Hz.

To invert the CSEM responses with the above acquisition parameters, we consider two inversion schemes: (1) a quasi-Newton inversion scheme described in Zach, Bjørke, Støren, and Maaø (2008); and (2) a Gauss-Newton inversion scheme described in Amaya (2015). An inversion based on the Gauss-Newton scheme converges faster and is less dependent on the starting model as compared to the quasi-Newton inversion, but this comes at the cost of increased computational complexity. We refer to Habashy and Abubakar (2004) for a detailed discussion of the theoretical differences between the two inversion schemes. One key difference is the number of RHSs that needs to be handled at each inversion iteration. For the quasi-Newton scheme it scales with the number of receivers n_r , while in the Gauss-Newton scheme one should include computations also for all source shot points n_s. In a typical marine CSEM survey one has $n_s \gg n_r$, hence the number of RHSs required by the Gauss-Newton scheme is much larger than that by the quasi-Newton scheme. For the chosen example based on the SEAM model, the quasi-Newton and Gauss-Newton schemes require 968 and 3784 RHSs per inversion iteration for one frequency, respectively.

In Table 7.8, we report the time for the complete resolution (analysis, factorization, and forward and backward substitutions for all RHSs) using the FR and BLR solvers on the **eos** supercomputer using 90 MPI × 10 threads setting and ParMETIS (Karypis and Kumar, 1998) for ordering. For comparison, time estimates for an iterative solver are also presented. This iterative solver was developed following the ideas of Mulder (2006): a complex biconjugate-gradient-type solver, BICGStab(2) (Van der Vorst, 1992; Gutknecht, 1993) is used in combination with a multigrid preconditioner and a block Gauss-Seidel type smoother.

The first two rows of Table 7.8 show the result reported in Shantsev et al. (2017),

Version	Inversion scheme (number of RHSs)	FR solver T_a	T_f	T_s	T_{total}	BLR solver ($\varepsilon = 10^{-7}$) T_a	T_f	T_s	T_{total}	Iterative solver
Old	Quasi-Newton (968)	87	2803	965	**3856**	103	1113	965	**2181**	**803**
	Gauss-Newton (3784)	87	2803	3772	**6663**	103	1113	3772	**4988**	**3141**
New	Quasi-Newton (968)	87	1254	329	**1670**	103	232	301	**636**	**803**
	Gauss-Newton (3784)	87	1254	1287	**2628**	103	232	1177	**1512**	**3141**

Table 7.8 – Suitability of BLR solvers for inversion. We report run times for the FR and BLR direct solvers on the **eos** supercomputer as well as for a multigrid preconditioned iterative solver to perform CSEM simulations for a large number of RHSs. We report two version of the results: the old results, presented in Shantsev et al. (2017) (based on MUMPS 5.0 and using the FSCU BLR variant), and the new ones (MUMPS trunk version, using the UFCS+LUAR BLR variant, and a preliminary version of the BLR solution phase). We consider two different inversion schemes applied to CSEM data over the SEAM model: a quasi-Newton scheme with 968 RHSs and a Gauss-Newton scheme with 3784 RHSs. The simulations are carried out for the system matrix S21 using 900 computational cores. For the direct solvers, T_a is the analysis time, T_f is the factorization time, T_s is the solve time (for forward-backward substitutions for all RHSs), and T_{total} is the total time, all measured in seconds.

which were obtained with MUMPS 5.0; in the case of the BLR solver, the factorization is using the FSCU variant and the solution phase is still performed in FR. While the BLR solver already showed potential to accelerate the factorization, the overall BLR solver remained 2.5 and 1.5 times slower than the iterative one, using the quasi-Newton and Gauss-Newton schemes, respectively.

Since then, many improvements have been made to both the FR and BLR solvers. The last two rows of Table 7.8 show the new results, obtained with the trunk version of MUMPS. Both the factorization and solution phases of the FR solver have been accelerated; moreover, the gains due to the BLR solver are higher due to the use of the improved UFCS+LUAR factorization variant presented in this thesis; a preliminary version of the BLR solution phase is also used to provide a moderate speedup with respect to the FR solver. With these improvements, the BLR solver achieves moderate gains with respect to the iterative solver using the quasi-Newton scheme, and outperforms it by a factor over 2 using the Gauss-Newton scheme.

These new results show the suitability of BLR solvers for CSEM inversion, as they start to become more attractive than iterative solvers for 800 or more RHSs.

7.2.7 Section conclusion

We have demonstrated that the application of BLR multifrontal solvers to solve linear systems arising in finite-difference 3D EM problems leads to significant reductions in matrix factor size, flop count and run time as compared to a FR solver. The savings increase with the number of unknowns N; for example, for the factorization flop count, the $\mathcal{O}(N^2)$ scaling for the FR solver is reduced to $\mathcal{O}(N^m)$ with m

between 1.4 and 1.6 for the BLR solver. This is slightly better than the theoretical complexity computed in Chapter 4. For shallow-water EM problems, we have shown that the reduction due to BLR approach is less than for deep water. This may be related to the highly resistive air layer that increases connectivity between system unknowns and hence decreases low-rank compression rates. The BLR solver runtimes were compared to those of an iterative solver with multigrid preconditioning in an realistic inversion scenario where simulations at multiple source locations are necessary. For a few thousand RHSs, which is typical in Gauss-Newton CSEM inversion schemes today, the BLR solver outperforms the iterative solver by a factor over 2; once the BLR solution phase is optimized, this gain will certainly further increase (cf. Section 9.2).

CHAPTER

8 Comparison with an HSS solver: STRUMPACK

In this chapter, we compare our BLR multifrontal solver described in the previous chapters with STRUMPACK, an HSS multifrontal solver developed at Lawrence Berkeley National Laboratory (Rouet et al., 2016; Ghysels et al., 2016; Ghysels et al., 2017).

The goal of this comparison is to shed light on the differences between these low-rank formats, and how these differences impact the complexity and performance of the solvers.

In all this chapter, the BLR variant considered is the UFCS+LUAR factorization.

8.1 The STRUMPACK solver

STRUMPACK (STRUctured Matrices PACKage) is a fast linear solver and preconditioner for both dense and sparse systems using HSS factorization with randomized sampling. This comparison concerns the sparse component of STRUMPACK, which is based on the multifrontal approach.

A detailed description of the solver can be found in Ghysels et al. (2016). Here, we summarize the main differences with the MUMPS solver.

STRUMPACK currently provides the LU factorization only; therefore, the experiments in this chapter concern unsymmetric matrices (symmetric matrices are unsymmetrized).

Both solvers are fully-algebraic. In this chapter, the METIS 5.1 ordering is used.

In all the experiments of this chapter, the BLR solver refers to the UFCS+LUAR factorization variant. We therefore consider a non fully-structured factorization; the contribution block of the frontal matrices are not compressed. On the other hand, STRUMPACK is a fully-structured solver; the matrix is initially compressed and the assembly is performed in low-rank, based on the randomized sampling algorithm described in Martinsson (2011).

Both solvers use the same compression kernel, a truncated QR factorization with column pivoting (via random sampling in the case of STRUMPACK). However, the low-rank threshold ε is absolute in MUMPS, while it is relative in STRUMPACK (i.e. the stopping criterion is $|r_{k,k}| < \varepsilon$ in MUMPS and $|r_{k,k}| < \varepsilon |r_{1,1}|$ in STRUMPACK).

This intuitively seems reasonable, because the norm of the HSS blocks is expected to be comparable, while that of the BLR blocks can vary significantly, but should be the object of further research.

Finally, both solvers only compress fronts larger than 1000.

8.2 Complexity study

In this section, we compare the experimental complexity of the two solvers.

We use the same experimental setting as in Chapter 4. We use the same two test problems: the Poisson problem with 7-point discretization, and the Helmholtz problem at frequency 4 Hz with 27-point discretization and with 4 grid points per wavelength. We also use the same methodology to compute complexity estimates based on a least-squares fitting.

8.2.1 Theoretical complexity

	operations		factor size	
	$r = \mathcal{O}(1)$	$r = \mathcal{O}(N)$	$r = \mathcal{O}(1)$	$r = \mathcal{O}(N)$
FR	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^{\frac{4}{3}})$	$\mathcal{O}(n^{\frac{4}{3}})$
BLR	$\mathcal{O}(n^{\frac{4}{3}})$	$\mathcal{O}(n^{\frac{5}{3}})$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^{\frac{7}{6}})$
HSS	$\mathcal{O}(n)$	$\mathcal{O}(n^{\frac{4}{3}})$	$\mathcal{O}(n)$	$\mathcal{O}(n^{\frac{7}{6}})$

Table 8.1 – Theoretical complexity of the multifrontal factorization, for a 3D problem of order $n = N^3$ and with a rank bound r.

In Table 8.1, we report the theoretical complexity of the full-rank (FR), BLR, and HSS multifrontal factorizations in the 3D case (a similar analysis for the 2D case is possible). We consider two types of rank bounds, $r = \mathcal{O}(1)$ and $r = \mathcal{O}(N)$.

For the Poisson problem, the rank bound is $\mathcal{O}(1)$ for BLR (Bebendorf and Hackbusch, 2003), but $\mathcal{O}(N)$ for HSS (Chandrasekaran, Dewilde, Gu, and Somasunderam, 2010). This is due to the weak admissibility condition of the HSS format.

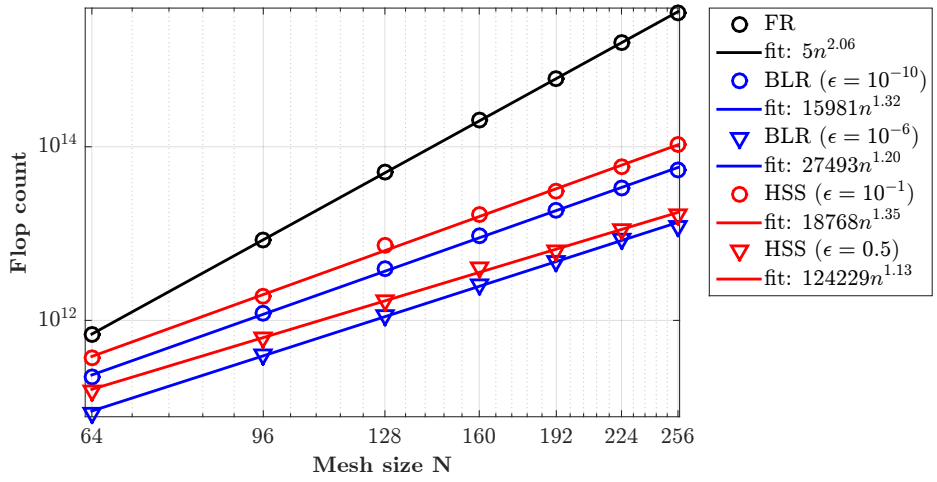
For the Helmholtz problem, although there is no rigorous proof of it, we assume a rank bound $\mathcal{O}(N)$, both for BLR and HSS, as is done in the related literature (Xia, 2013a; Wang et al., 2016; Engquist and Ying, 2011).

8.2.2 Flop complexity

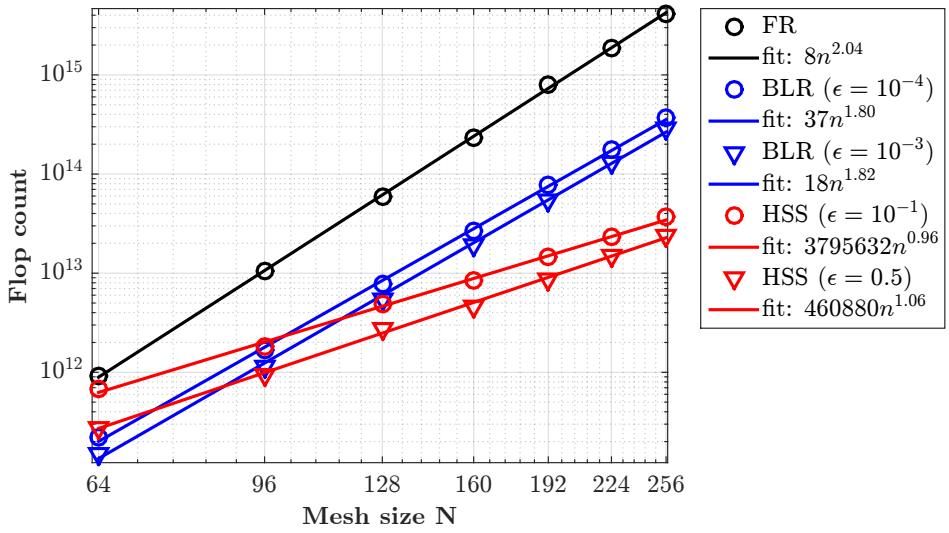
We first analyze the flop complexity results reported in Figure 8.1.

For the Poisson problem, the BLR and HSS complexity exhibit the same asymptotic behavior, as expected since the theoretical complexity is $\mathcal{O}(n^{\frac{4}{3}})$ in both cases (because $r = \mathcal{O}(1)$ in BLR and $r = \mathcal{O}(N)$ in HSS). BLR has a lower prefactor and therefore outperforms HSS by a significant factor.

For the Helmholtz problem, the BLR complexity has a higher exponent but a lower prefactor compared to the HSS one; thus, while BLR outperforms HSS for



(a) Poisson problem



(b) Helmholtz problem

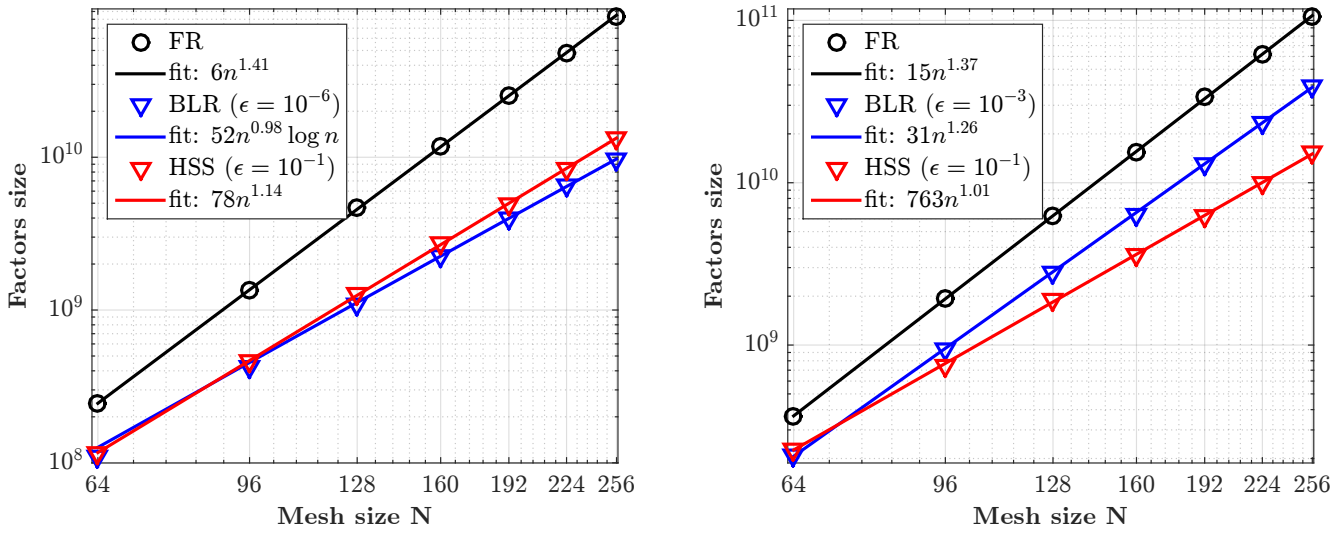
Figure 8.1 – BLR and HSS flop complexity comparison.

the smaller problems, there is a cutoff point after which HSS gains the upper hand. The exact value of this cutoff point depends on several parameters including the tolerance choice.

Thus, the experimental complexity results are in good agreement with the theory. Note that, in the BLR case, the complexity results of this chapter may be slightly different from those of Chapter 4 due to a different experimental setting (METIS reordering, block size chosen to optimized performance rather than flops, etc.).

8.2.3 Factor size complexity

Next, we analyze the factor size complexity results reported in Figure 8.2.



(a) Poisson problem

(b) Helmholtz problem

Figure 8.2 – BLR and HSS factor size complexity comparison.

Overall, the factor size results lead to the same conclusions as the flop results. For the Poisson problem, the asymptotic cost of BLR is slightly lower than that of HSS due to the higher HSS rank bound ($\mathcal{O}(n \log n)$ with BLR, $\mathcal{O}(n^{1.17})$ with HSS). This gives the upper hand to BLR on this problem and range of sizes. Conversely, for the Helmholtz problem, HSS achieves a better asymptotic experimental complexity (whereas the theory predicts the same bound). This means that HSS leads to much better compression rates for the larger mesh sizes, although this observation should be slightly attenuated by the fact that the low-rank threshold is set to a higher value ($\varepsilon = 10^{-1}$ in HSS compared to $\varepsilon = 10^{-3}$ with BLR).

8.2.4 Influence of the low-rank threshold on the complexity

Finally, we analyze the influence of the low-rank threshold ε.

For both BLR and HSS, the theory states the threshold ε should only play a role in the constant factor of the complexity, not the exponent. However, that is not exactly what the numerical experiments show. For Helmholtz, the threshold does seem to play a role only in the constant factor. However, for Poisson, the exponent lowers as the threshold increases, both for BLR and HSS.

In the BLR case, this trend has been explained in Chapter 4 by the presence of zero-rank blocks, which become asymptotically dominant.

In the HSS case, the large off-diagonal blocks are rarely zero-rank; however, the ranks of the intermediary blocks appearing at the different levels in the HSS tree are often lower than the maximal rank bound: in fact, they often follow a rank pattern, that may depend on the threshold. By considering such a rank pattern, the theoretical bound can be relaxed to match the experimental observation (Xia, 2013a).

8.3 Low-rank factor size and flops study

In Figures 8.3 and 8.4, we compare the size of low-rank factors and flops for the low-rank factorization, respectively, with respect to the full-rank ones both the BLR and HSS solvers.

For the bigger thresholds BLR and HSS achieve comparable compression rates, although BLR is consistently slightly better than HSS on this set of problems. The difference in factor size is greater than in flops. For these medium-sized problems, the constant in the HSS complexity is too big to outperform the smaller BLR constant.

For the smaller thresholds, the difference between BLR and HSS becomes much more important: while the BLR compression rate slowly decreases, that of HSS degrades much more rapidly; in terms of flops, HSS is not even beneficial compared to FR due to the compression overhead.

This result could be explained by the different admissibility conditions of BLR and HSS. Indeed, with the flat BLR format, it is simple to select which blocks are approximated by a low-rank structure and which are kept dense, depending on their rank. Thus, when the threshold decreases, the rank growth can be easily controlled by reverting to FR the blocks that correspond to strong interactions. This leads to a relatively slow, smooth increase of the low-rank factor size as the threshold decreases.

On the contrary, the weak admissibility condition of the HSS format means that any off-diagonal block has to be represented by a low-rank structure, even if that rank is quite big. This could also lead to a higher factor size, as illustrated in Figure 8.5.

8.4 Sequential performance comparison

In this section, we compare the performance of the two solvers in a sequential setting. Runs were performed on one node of the **cori** supercomputer, using one thread only.

8.4.1 Comparison of the full-rank direct solvers

We first compare the full-rank solvers in Figure 8.6. It can be observed that the factor nonzeros, and the flops and time for the factorization are all very similar for both solvers, with a variation always under 20% and often much smaller than that. However, regarding the solve, STRUMPACK is consistently and significantly faster than MUMPS. Given the factor nonzeros are comparable, this probably means there is some performance optimization to be done in MUMPS.

8.4.2 Comparison of the low-rank solvers used as preconditioners

For the low-rank comparison, each solver is tested with several low-rank thresholds ε, from 0.9 to 10^{-6}. We then apply a GMRES iterative algorithm until the

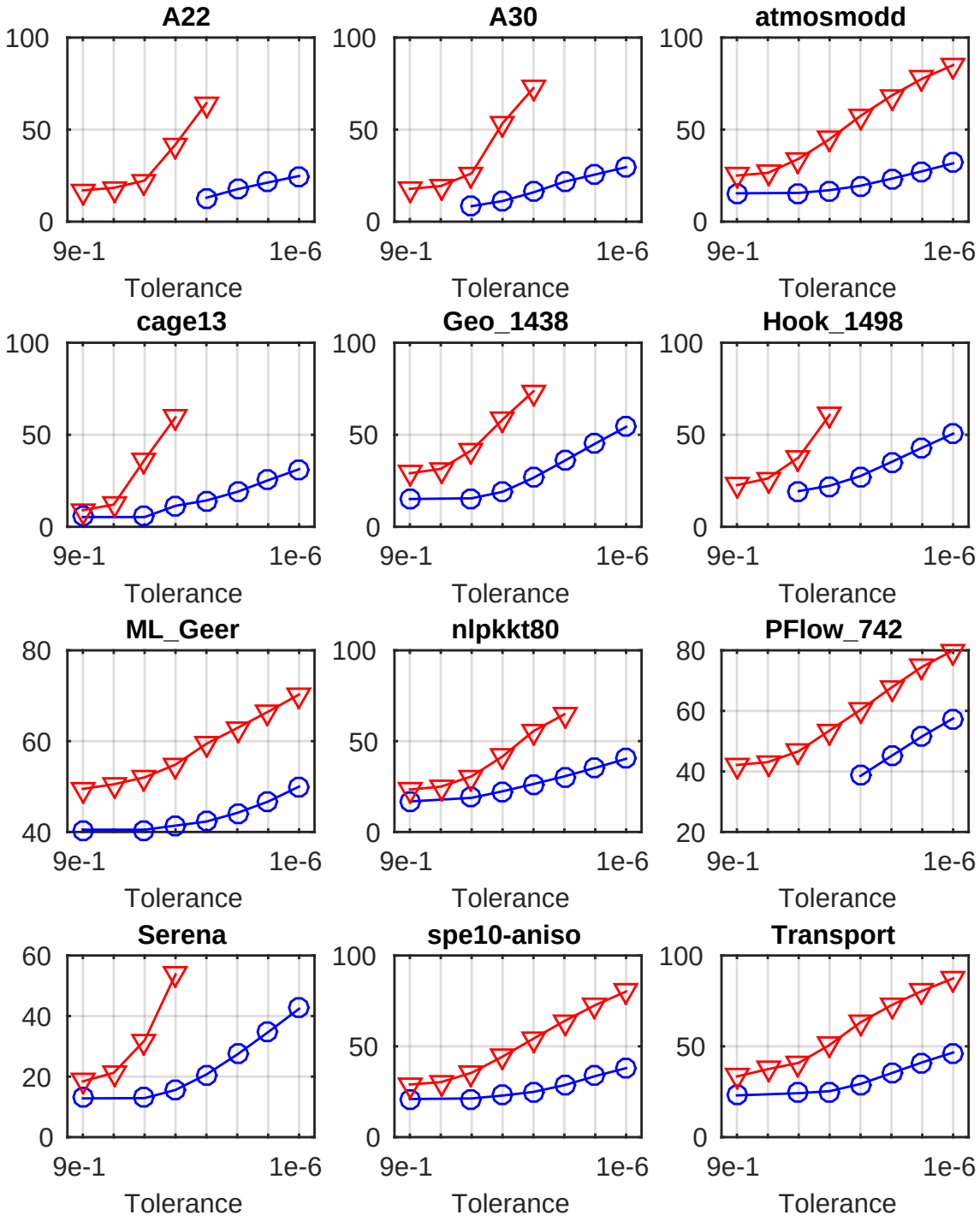


Figure 8.3 – Size of BLR and HSS low-rank factors (in % of FR one). Blue circle plot is BLR and red triangle one is HSS. Tolerance varies from 0.9 to 10^{-6}.

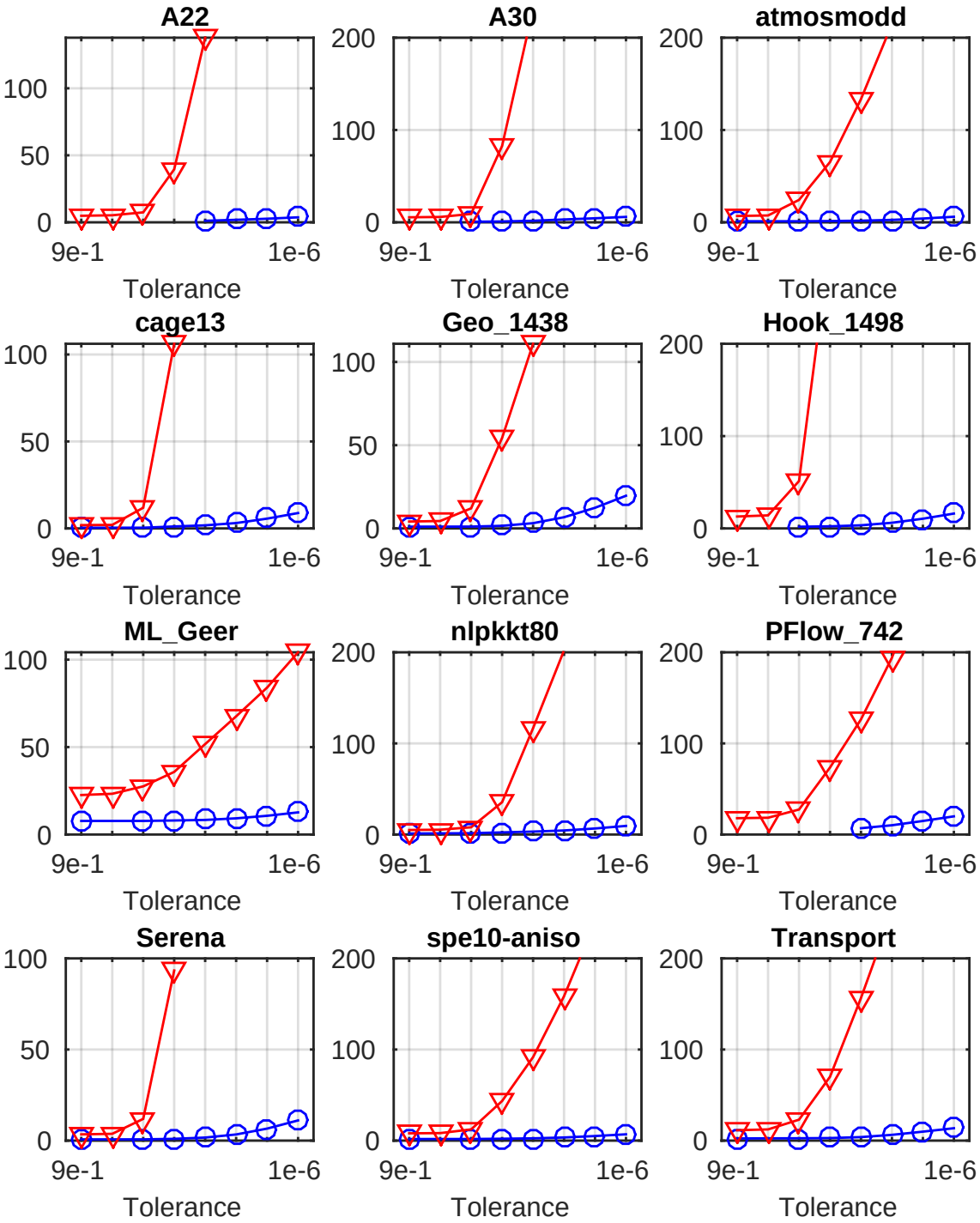
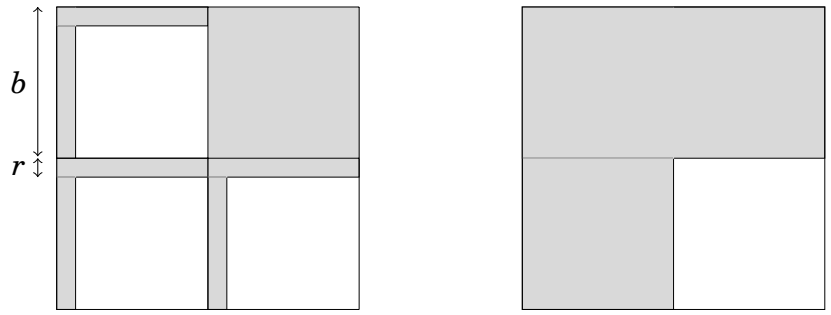


Figure 8.4 – Flops for BLR and HSS factorizations (in % of FR one). Blue circle plot is BLR and red triangle one is HSS. Tolerance varies from 0.9 to 10^{-6}.



(a) BLR partitioning of a big off-diagonal block

(b) Hierarchical partitioning: the block is not refined

Figure 8.5 – Illustration of how a weak-admissibility condition can result in higher storage. The BLR storage is equal to $b^2 + 6br = b^2 + \mathcal{O}(b)$; the hierarchical storage is equal to $4b^2$.

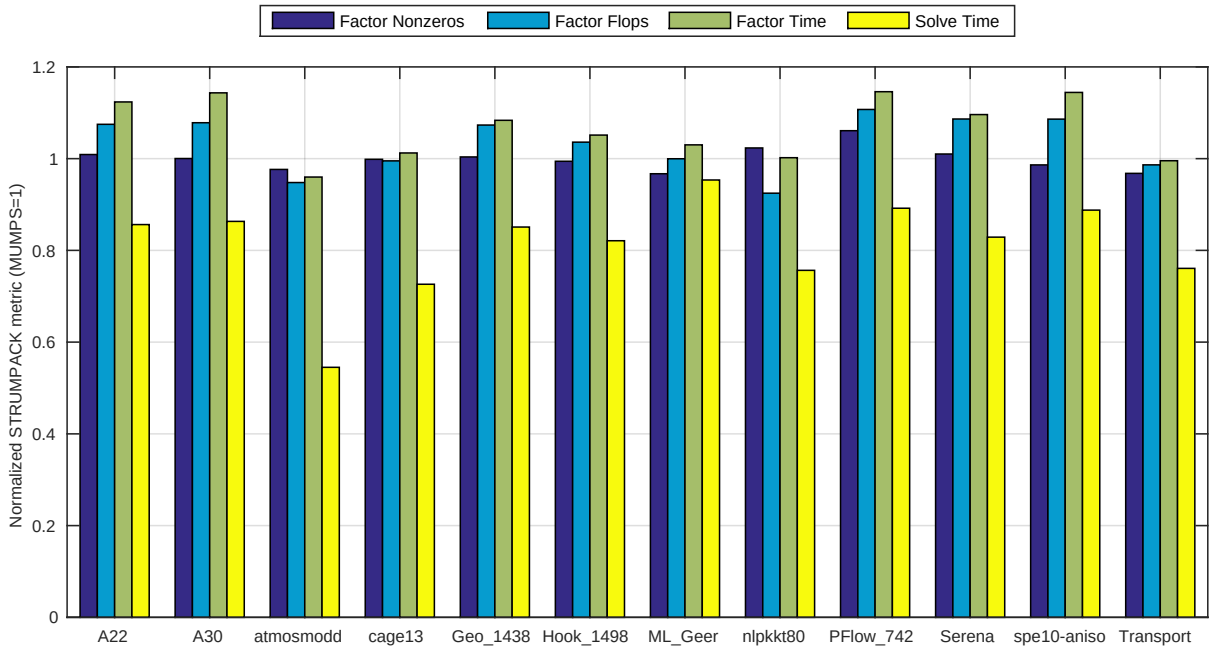


Figure 8.6 – MUMPS and STRUMPACK sequential full-rank comparison.

required accuracy of 10^{-6} is achieved. The stopping criterion is relative (i.e. convergence is achieved when $\|r_{k+1}\|/\|r_0\| < 10^{-6}$).

In Table 8.2, we report the best choice of threshold, i.e., the one that minimizes the total time for factorization and solution.

In the BLR case, that optimal threshold is often quite small, close to the required accuracy of 10^{-6}. We refer to this as *direct solver mode*, because the time spent in the iterations (solution phase) is small compared to the setup time (time spent in the factorization).

	BLR		HSS	
	ε	time	ε	time
A22	1e-5	136.7	FR	983.4
A30	1e-4	141.3	FR	906.1
atmosmodd	1e-4	81.0	9e-1	129.7
cage13	1e-1	174.7	9e-1	109.8
Geo_1438	1e-4	283.5	FR	1001.2
Hook_1498	1e-5	162.4	FR	471.0
ML_Geer	1e-6	65.2	FR	137.4
nlpkkt80	1e-5	195.3	5e-1	234.8
PFlow_742	1e-6	56.1	FR	96.5
Serena	1e-4	272.3	1e-1	712.4
spe10-aniso	1e-5	87.7	FR	360.5
Transport	1e-5	118.7	FR	301.3

Table 8.2 – Best tolerance choice for BLR and HSS solvers and associated time for factorization+solve, using 1 thread. The shaded rows correspond to the three representative problems discussed in Section 8.5 (cf. Figure 8.7).

Conversely, in the HSS case, the optimal threshold tends to be much bigger. We refer to this as *preconditioner mode*, because the bulk of the time is spent in the iterations while the cost of the factorization is cheaper.

This difference in the behavior of the two formats could again be explained by the rank growth due to the weak admissibility of the HSS format, as suggested in the previous section.

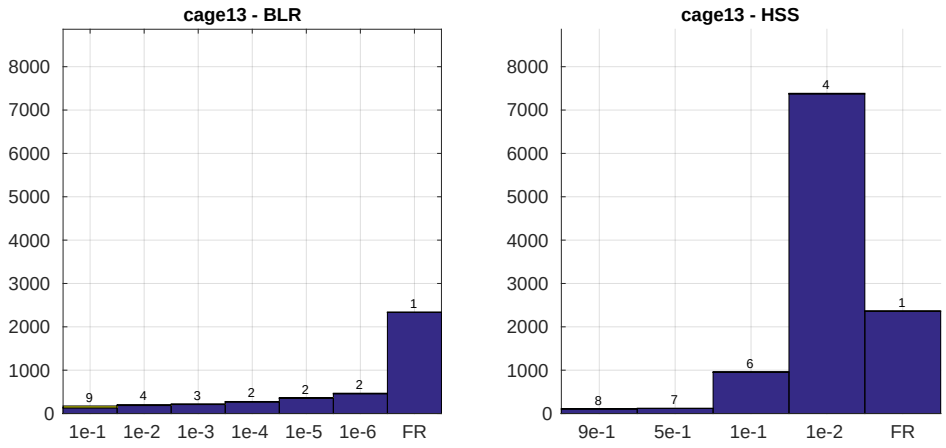
8.5 Discussion

We provide in Figure 8.7 detailed results for three test problems (shaded rows of Table 8.2: cage13, spe10-aniso, and atmosmodd) which are representative of different types of situation.

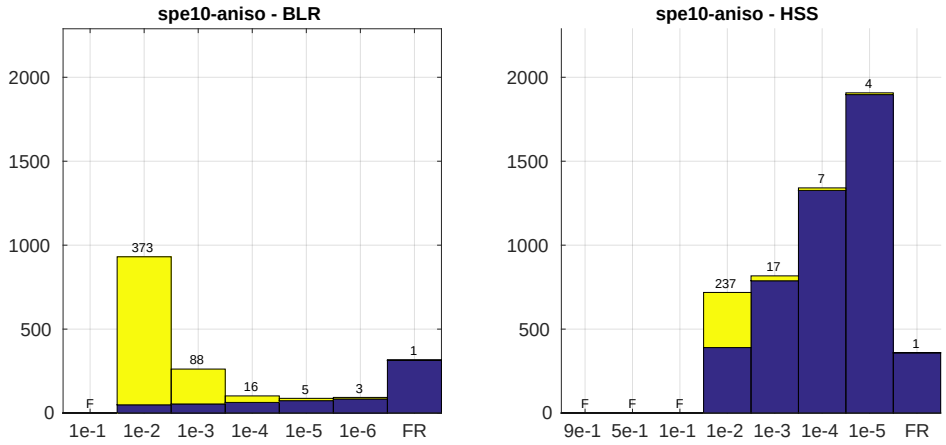
- On one hand, cage13 is a numerically easy problem, for which preconditioning techniques work very well. Indeed, even for aggressive compression strategies (with a very large threshold such as 0.9), the solver converges after just a few iterations. In this case both BLR and HSS are best when used in "preconditioner mode"; as the problem size increases, HSS will gain the upper hand.

- On the other hand, spe10-aniso is a much more difficult problem. Both the BLR and HSS solvers fail to converge when used in preconditioner mode. In this case, the solvers must be used in "direct mode", with a smaller low-rank threshold. However, due to the rank growth, HSS is less suited for this type of problem than BLR.

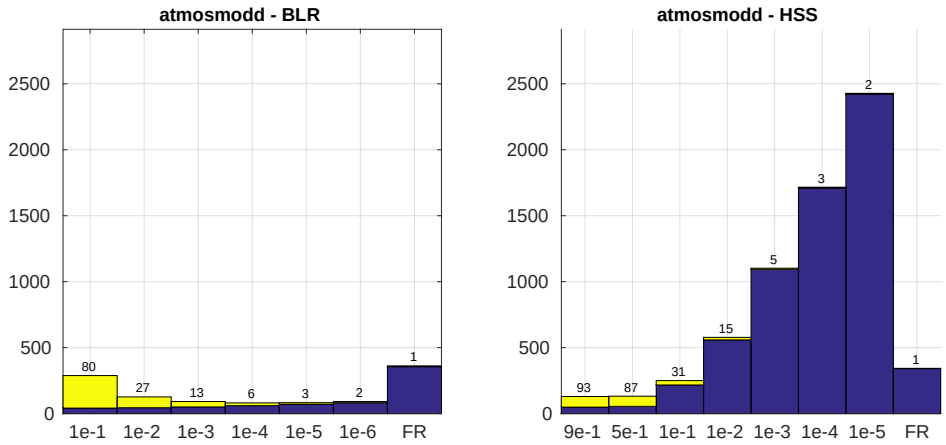
- The atmosmodd problem lies in the middle ground: there is a compromise to be found between accuracy and compression. As analyzed, BLR favors being



(a) cage13 problem.



(b) spe10-aniso problem.



(c) atmosmodd problem.

Figure 8.7 – Sequential performance comparison. Blue and yellow represent the time spent in the factorization and solution phases, respectively. The numbers above the bars indicate the number of iterations; 'F' stands for 'Failed to converge'.

used in direct mode, while HSS favors being used in preconditioner mode. The relative performance of the two solvers will depend on the size of the problem and its numerical difficulty (i.e. convergence rate to the required accuracy).

Therefore, these results seem to suggest the trend depicted in Figure 8.8. This is an intuitive, informal trend based on preliminary results. In particular, the concept of "difficulty" (which here means difficulty to converge to the required accuracy) merits to be properly defined. Furthermore, other constraints such as the memory consumption would influence this trend: indeed, the FR solver will run out of memory before the BLR one, which in turn might consume more memory than the HSS one.

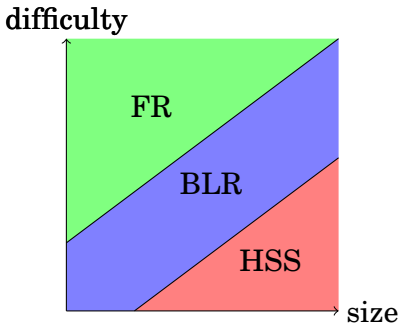


Figure 8.8 – Which solver is the most suited in which situation? Trend suggested by Figure 8.7 and Table 8.2.

These preliminary results should be completed with further experiments. First, this study has focused on 3D problems. It is not clear whether a study on 2D problems would lead to the same results and conclusions. Second, the performance comparison should be extended to multicore and distributed-memory environments. Finally, to truly understand the differences between these low-rank formats, this comparison should be extended to the non-nested formats (such as HODLR) as well as the strongly admissible formats (such as $\mathscr{H}$).

CHAPTER

9 Future challenges for large-scale BLR solvers

In this chapter, we discuss some of the future challenges that BLR solvers are likely to encounter when solving increasingly large problems.

We discuss three aspects of BLR solvers that we have scarcely mentioned so far in this thesis, but that will be of growing importance when targeting large-scale systems and applications: the BLR analysis and solution phases, and the memory consumption. Indeed, because the factorization time has been greatly reduced in BLR, the time for the other two phases can become critical. Moreover, while BLR helps reducing the overall memory consumption, it paradoxically lowers the memory efficiency of the solver, as we will show. Finally, we illustrate these challenges by reporting experiments on a set of very large problems (of order up to 90 million unknowns).

9.1 BLR analysis phase

The time for computing the BLR clustering (cf. Section 1.4.3.1) represents an overhead. For some problems, we have observed that this overhead can represent an important part of the total analysis time, and therefore of the overall solver time. The BLR clustering time should thus be reduced.

A key parameter is the halo depth d_h, which controls how the halo subgraph $\mathcal{G}_H$ (on which we perform the clustering) is built from the separator subgraph $\mathcal{G}_S$, as explained in Section 1.4.3.1. In the original BLR solver presented in Amestoy et al. (2015a) and Weisbecker (2013), d_h was set to 2. This was partly due to the observation that on regular grids, a halo depth of two is enough to reconnect the separators. However, as reported in Table 9.1 for several problems, the BLR clustering time with $d_h = 2$ can be very large with respect to the FR analysis time. For example, for the perf00{8d,8ar,8cr,9d,9ar} matrices coming from our EDF application (cf. Section 1.5.2.2), the BLR clustering represents an overhead between 25 and 50% of the FR analysis time.

Therefore, we consider using a lower halo depth. We must check whether this can lead to a significant degradation of the compression rate. With $d_h = 0$, the BLR clustering is computed with the lowest times, but it is not robust. Indeed, as reported in Table 9.1, the compression rate for the atmosmodd and kkt_power ma-

trices drops to much lower values than those with $d_h \geq 1$. On the other hand, with $d_h = 1$, the compression rate of atmosmodd is satisfying, while that of kkt_power remains significantly higher than with $d_h = 2$. We have not observed a degradation of the compression rate for any other of the matrices in our test suite (which includes, in addition to those reported in Table 9.1, all the matrices from the Janna group of the UFSMC). In fact, the compression rate is often slightly better than that with $d_h = 2$, which may illustrate the fact that the clustering is computed on a graph that matches more closely the actual separator that we want to cluster.

We conclude from these experiments that a halo depth of 1 seems to be a good compromise between compression rate and cost of the BLR clustering, and that is the value that we have used for the experiments presented throughout this thesis.

matrix	FR analysis time (s)	BLR clustering time (s)			compression rate (%)		
		$d_h = 0$	$d_h = 1$	$d_h = 2$	$d_h = 0$	$d_h = 1$	$d_h = 2$
A22	8.0	3.9	10.8	24.6	12.4	13.2	13.8
atmosmodd	19.7	0.2	0.7	0.9	56.9	9.2	9.1
audikw_1	18.8	2.4	6.3	13.5	32.1	32.0	32.5
cage13	26.5	0.7	3.0	7.2	11.0	10.2	11.8
Cube_Coup_dt0	27.4	4.2	7.8	14.0	15.5	15.4	15.3
Geo_1438	19.7	1.9	3.4	5.7	5.5	5.5	5.5
Hook_1498	20.1	1.8	3.6	6.3	23.7	23.6	23.6
HV15R	135.3	17.7	38.8	76.9	11.5	11.3	12.1
kkt_power	33.8	0.5	1.0	2.2	90.6	51.2	30.9
ML_Geer	27.7	1.8	2.7	3.9	18.2	18.1	18.1
nlpkkt80	15.2	1.8	1.8	1.8	12.3	12.3	12.3
perf008d	33.9	4.3	8.8	16.8	21.8	21.6	21.6
perf008ar	89.8	10.5	21.0	39.5	14.8	14.8	14.7
perf008cr	196.6	20.6	40.7	76.3	10.0	10.0	10.0
perf009d	15.1	1.3	2.3	3.7	50.6	51.2	51.1
perf009ar	129.8	10.9	22.1	39.7	28.1	27.8	28.0
Queen_4147	73.6	9.7	20.1	39.3	8.3	8.3	8.3
Serena	22.0	2.0	4.1	7.7	16.2	16.3	16.4
spe10-aniso	24.8	0.9	1.4	2.2	10.7	10.7	10.6
StocF_1495	28.5	0.9	1.5	2.4	10.2	10.1	10.2
Transport	28.4	1.0	1.4	2.1	17.4	17.7	17.7

Table 9.1 – Influence of the halo depth parameter d_h on the BLR clustering time and on the compression rate. The total BLR analysis time is equal to the sum of the FR analysis time and the BLR clustering time. Experiments are performed on 1 core on **brunch**.

Nevertheless, with $d_h = 1$, the BLR clustering time remains important for some problems. For example, for the perf00* matrices, it still represents an overhead between 15 and 25% of the FR analysis time.

Therefore, to further accelerate the BLR clustering, we have parallelized it with OpenMP directives to exploit multiple threads. One difficulty is that to build the

halo subgraph $\mathcal{G}_H$, the threads must access the global adjacency graph of the matrix $\mathcal{G}_A$ (in order to mark already visited nodes), which thus leads to conflicts between different threads. To avoid these conflicts, a simple solution is to hold a private copy of the graph on each thread. With this strategy, the BLR clustering time using 8 threads on **brunch** is reported in Table 9.2 (column "mt. private") and we compare it to the sequential case (column "seq."). We do not include in the table matrices for which the sequential clustering time was inferior to 2 seconds; for these matrices, no gains are achieved by multithreading the clustering. For the rest of the matrices, important gains are achieved, even though the potential speedup of 8 is far from captured, due to the operations being very memory-bound.

To solve very large problems, holding a copy of the graph on each thread is not acceptable in terms of storage cost. Therefore, we implemented a multithreaded strategy where the graph is shared by all threads. To avoid conflicts, the halo building phase is protected by critical regions; the other phases (mainly, the clustering of the halo subgraph) can be executed in parallel. The performance of this strategy is evaluated in Table 9.2 (column "mt. shared"). It is very close to that of the "mt. private" strategy, which means the addition of critical regions has only slightly impacted the performance. With this strategy, the cost of the BLR clustering is kept under 20% of the FR analysis time and often much less than that. Moreover, because the speedup increases with the size of the problem, the relative cost of the BLR clustering actually decreases as the problem size increases.

matrix	seq.	mt. private	mt. shared
A22	10.8	3.7	4.5
audikw_1	6.3	3.6	3.9
cage13	3.0	2.1	2.2
Cube_Coup_dt0	7.8	6.3	6.0
Geo_1438	3.4	2.7	2.4
Hook_1498	3.6	2.7	2.8
HV15R	38.8	17.0	19.7
ML_Geer	2.7	1.9	1.8
perf008d	8.8	5.6	5.9
perf008ar	21.0	13.9	14.5
perf008cr	40.7	25.2	26.4
perf009d	2.3	1.5	1.5
perf009ar	22.1	12.9	12.9
Queen_4147	20.1	13.2	12.9
Serena	4.1	3.2	3.3

Table 9.2 – Acceleration of the BLR clustering with multithreading. We report the time for BLR clustering in seconds for three versions: sequential (seq.), multithreaded with a private copy of the graph on each thread (mt. private), and multithreaded with one graph shared by all threads and protected by critical regions (mt. shared). The experiments are performed on **brunch**; the multithreaded versions use 8 cores.

Therefore, we conclude that, with a halo depth of 1 and the use of multiple

threads to speed up the computations, the cost of the BLR clustering can be kept limited with respect of the FR analysis time, without sacrificing the quality of the computed clustering.

The overall analysis time will then be dominated again by the computation of the matrix reordering rather than the BLR clustering. For very large problems, technologies to reduce the cost of the matrix reordering will thus become critical. For example, we could consider the use of multithreaded or parallel graph partitioners, such as MT-METIS, ParMETIS (Karypis and Kumar, 1998), and PT-SCOTCH (Chevalier and Pellegrini, 2006). In the case of parallel partitioners, the impact on the quality of the ordering has been studied (Amestoy, Buttari, and L'Excellent, 2008). Their impact on the low-rank compression rate should also be carefully analyzed.

9.2 BLR solution phase

In this work, we have mostly focused on reducing the cost of the factorization phase, which is usually considered to be the most computationally demanding phase.

However, as we have illustrated in our two applicative case-studies in Chapter 7, the solution phase can represent a significant part of the total cost, or even be the bottleneck in some large-scale applications with many right-hand sides.

This observation becomes even more relevant in BLR, which leads to a lower factorization complexity. Indeed, even though the complexity of the solution phase is also reduced by BLR approximations, it follows the reduction of the factor size complexity, which is lower than that of the flops. Therefore, the BLR solution phase is likely to become the bottleneck when dealing with many right-hand sides.

We mention several challenges and opportunities to accelerate the BLR solution phase that should be the object of further work.

First, as we have explained in Section 2.2.3, it is not straightforward to use the LAPACK style of pivoting with the BLR factorization, because it may require to perform column swaps between different low-rank blocks. One could of course switch to the LINPACK style, but as mentioned in Section 1.3.2.6, the cost of swapping the right-hand sides may severely hinder the performance of the solver if there are many of them. Therefore, the different strategies we have described to use the LAPACK style in conjunction with the BLR factorization may become critical. Their performance should be assessed and compared.

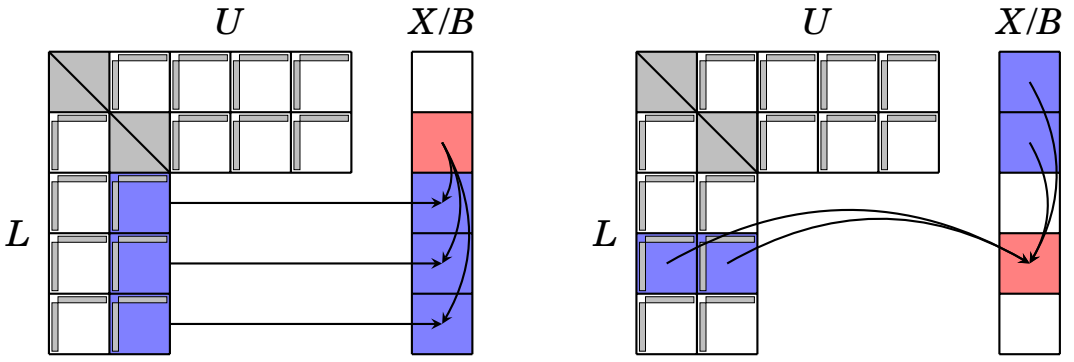
Second, in the case of multiple right-hand sides, the computations taking place during the BLR solution phase are very similar to those of the factorization phase. Indeed, both the forward elimination and backward substitution require low-rank products to be summed together. This is illustrated in Figure 9.1 in the case of the forward elimination; the same holds for the backward substitution. As a consequence, three key algorithmic properties analyzed in the context of the factorization also apply to the solution phase:

- The left-looking pattern minimizes memory transfers: we have explained in Section 5.3.3 that the left-looking factorization leads to a lower volume of

memory transfers due to the fact that it tends to access more often low-rank blocks than full-rank ones, whereas the right-looking factorization has the opposite behavior. In Figure 9.1, a similar effect takes place due to the fact that the LU factors are compressed but the right-hand side is not. The analysis is somewhat more complex because part of the RHS is still accessed at each step regardless of the pattern. It is likely that the left-looking pattern also reduces the volume of memory transfers, although this should be formally proved. Note that this observation is even more critical than for the factorization, because the solution phase is often more memory-bound.

- In left-looking, the low-rank updates can be accumulated to increase the granularity of the outer products: all inner products take the form of a block from the LU factors multiplied by a RHS block; if the factors block is low-rank, then the result is also low-rank and can thus be accumulated with low-rank blocks before being decompressed into the RHS. Again, due to the solution phase being very memory-bound, the accumulation is likely to be even more critical than for the factorization.

- In left-looking, the low-rank updates can be recompressed to reduce the cost of the outer products: since the low-rank updates can be accumulated, they can also be recompressed. The strategies discussed in Chapter 3 also apply. Contrarily to the factorization (cf. Section 4.5), the recompression does not asymptotically reduce the number of operations for the solution phase. Indeed, the inner products, whose cost is left unchanged by the recompression, already represent half of the total computations. Thus, even though the complexity of the solution phase remains asymptotically the same, its cost can potentially be divided by at most a factor 2.



(a) BLR forward elimination (right-looking).

(b) BLR forward elimination (left-looking).

Figure 9.1 – BLR right- and left-looking frontal forward elimination.

Finally, the complexity of the solution phase deserves to be further studied. In particular, consider the ratio between the total number of operations and the number of operations performed in any given branch of the assembly tree. It has been proved (Amestoy, L'Excellent, and Moreau, 2017g) that this ratio is asymptotically larger for the solution phase than for the factorization, due to the surface to volume

effect. More interestingly, it is also asymptotically larger in BLR than in FR; this is because the fronts at the top of the tree compress more than those at lower levels. This observation has in particular the following two crucial consequences. First, the theoretical speedup resulting from tree parallelism (which can be computed as the total number of flops over the flops performed on the critical path) is higher in BLR than in FR, and is also higher for the solution phase with respect to the factorization phase. This means that tree parallelism will be of the utmost importance in the BLR solution phase, even more so than for the factorization (as it was shown in Chapter 5). Second, for several applications, including the two presented in Chapter 7, the right-hand sides are sparse. Recent work based on the approach described by Gilbert and Liu (1993) aims at exploiting the sparsity of the RHS to reduce the cost of the solution phase (Amestoy, Duff, L'Excellent, and Rouet, 2015f; Amestoy, L'Excellent, and Moreau, 2017f). This often amounts to pruning the assembly tree to only traverse a constant number of branches. Therefore, one may expect the gains due to the exploitation of sparse RHS to be asymptotically larger in BLR than in FR.

9.3 Memory consumption of the BLR solver

In this section, we briefly comment on the memory consumption and scalability of the BLR solver with respect to the FR one.

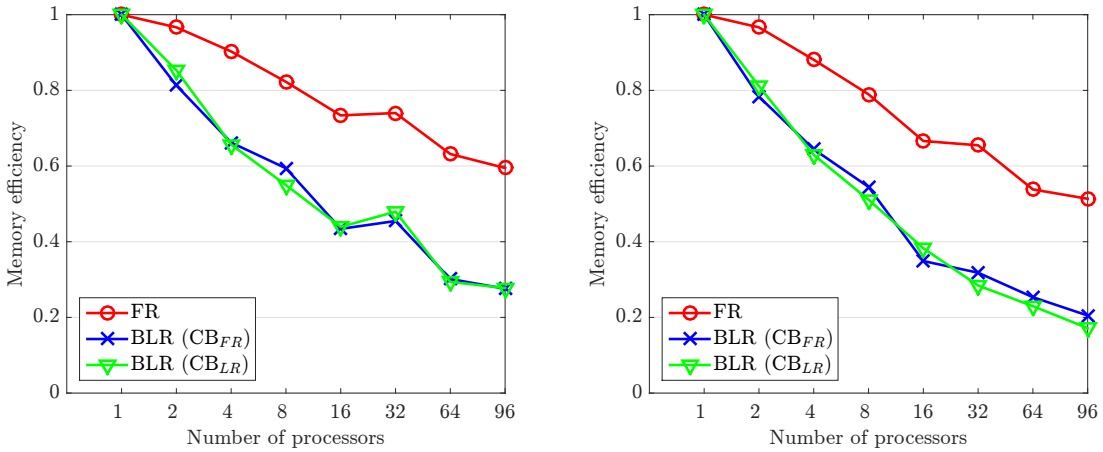
To illustrate the behavior of the BLR solver, we consider the factorization of matrix 7Hz (matrix ID 2, cf. Table 1.3) on an increasing number of MPI processes, from 1 to 96. We analyze in Table 9.3 the total memory consumption (average per process and maximum over all processes) of the FR and BLR factorizations; in the BLR case, we compare the full-rank CB (CB_{FR}) and low-rank CB (CB_{LR}) strategies described in Section 2.5, which control whether the contribution block is also compressed.

number of processes	Total memory consumption						Active front
	FR		BLR (CB_{FR})		BLR (CB_{LR})		
	avg.	max.	avg.	max.	avg.	max.	
1	216.8	216.8	85.5	85.5	72.9	72.9	10.7
2	112.2	112.2	52.6	54.6	42.7	44.9	10.7
4	60.0	61.5	32.3	33.2	27.7	28.9	10.7
8	33.0	34.3	18.0	19.6	16.6	17.8	6.2
16	18.5	20.3	12.3	15.3	10.4	11.9	5.1
32	9.2	10.3	5.9	8.4	4.8	8.0	1.8
64	5.4	6.3	4.4	5.3	3.9	5.0	1.5
96	3.8	4.4	3.2	4.3	2.7	4.4	0.9

Table 9.3 – Memory consumption analysis of the FR and BLR factorizations. All values are provided in GB. Experiments are performed on **brunch**.

We analyze the average total memory consumption per process (fourth to sixth columns of Table 9.3). The total storage for the LU factors is equal to 205.0 GB in

FR and 57.3 GB in BLR. The factors are distributed over the processes and their storage per process scales very well, both in FR and BLR. Unlike the factors, the scalability of the total memory is not ideal. This comes from the fact that the total memory consists of two parts, the factors and the active memory, as described in Section 1.3.2.5. The active memory is known to scale much less than the factors, and therefore limits the total memory efficiency.



(a) Average memory per process.

(b) Maximal memory.

Figure 9.2 – Memory efficiency of the FR, BLR CB$_{FR}$, and BLR CB$_{LR}$ factorizations.

We illustrate this by plotting the total memory efficiency of the FR, BLR CB$_{FR}$, and BLR CB$_{LR}$ factorizations in Figure 9.2. The FR solver achieves a satisfying memory efficiency, which gradually lowers as the number of processes increases, achieving around 59% on 96 processes. However, in BLR CB$_{FR}$, the factors are compressed and therefore the active memory has a much higher relative weight with respect to the total. The BLR CB$_{FR}$ factorization thus inherits the poor memory scalability of the active memory, achieving a significantly lower memory efficiency compared to the FR solver (e.g., 28% on 96 processes).

To overcome this issue, we must thus also reduce the active memory. The active memory is itself composed of two parts: the contribution blocks, and the active front. Therefore, the memory consumption can be reduced by compressing the CB (CB$_{LR}$ strategy). As reported in Table 9.3, the absolute memory gain obtained by compressing the CB is significant. Unfortunately, compressing the CB is not enough to improve the memory efficiency, as shown in Figure 9.2. This is because on large number of processes, the active and thus total memory is actually dominated by the storage for the active front. This comes from the fact that, in the context of the non fully-structured BLR factorization, the active front is allocated in full-rank before being compressed. To overcome this issue, three solutions can be considered. First, we could simply switch to a fully-structured factorization, although we would have to perform slow and relatively complex low-rank assembly operations. Second, memory-aware strategies (Agullo et al., 2016) could be used to map the critical active fronts on all available processes, at the expense of some serializations. Third, and perhaps most promisingly, we could interlace the allocation, assembly,

and compression of the active front panel by panel. This can be hard to achieve especially in a distributed-memory parallel code with a very asynchronous execution behavior. Therefore, novel algorithms must likely be developed and studied to make sure parallelism is not hampered. As we illustrate in next section, the active front storage is the next memory bottleneck to be addressed.

9.4 Results on very large problems

We conclude this chapter by illustrating the future challenges for large-scale BLR solvers by reporting results on our largest problems coming from the three applications described in Section 1.5.2.2: matrices 15Hz and 20Hz (seismic modeling, SEISCOPE), D5 (electromagnetic modeling, EMGS), and perf008ar2 (structural mechanics, EDF). The full-rank statistics of these matrices are reported in Table 9.4.

matrix	arith.	fact. type	n	nnz	flops	factor size
15Hz	c	LU	58.0M	1523M	29.6 PF	3.7 TB
20Hz	c	LU	129.9M	3432M	150.0 PF	11.0 TB
D5	z	LDL^T	90.1M	1168M	29.2 PF	6.1 TB
perf008ar2	d	LDL^T	31.1M	1267M	24.1 PF	2.1 TB

Table 9.4 – Set of very large problems: full-rank statistics.

In Table 9.5, we provide the low-rank statistics obtained by running the BLR factorization with (CB_{LR}) and without (CB_{FR}) compression of the CB. On these very large problems, both the factors and CB achieve a very high compression rate; the flops for the factorization are also greatly reduced with respect to the full-rank flop count. In particular, it is interesting to compare this compression to that of the smaller matrices of the same problem class that were studied in the previous chapters. For example, on matrix 10Hz, the flops for the BLR factorization were 7.5% of the FR flops (cf. Table 7.3); on matrices 15Hz and 20Hz, that value drops to 3.7% and 2.4%, respectively.

matrix	ε	factor size	CB compr. rate	flops ($\times 10^{15}$) CB_{FR}	flops ($\times 10^{15}$) CB_{LR}
15Hz	10^{-3}	719 GB (19.5%)	16.9%	1.11 (3.7%)	1.29 (4.4%)
20Hz	10^{-3}	1831 GB (16.7%)	4.7%	3.57 (2.4%)	3.87 (2.6%)
D5	10^{-7}	1008 GB (16.4%)	14.6%	0.33 (1.1%)	0.38 (1.3%)
perf008ar2	10^{-9}	571 GB (26.8%)	27.7%	0.94 (3.9%)	1.01 (4.2%)

Table 9.5 – Set of very large problems: low-rank statistics. Percentage of the full-rank metric is provided between parenthesis.

The high compression rate allows for considerable savings in memory, as analyzed in Table 9.6. According to the predictions of the analysis phase, the sequential

shared-memory FR factorization would require 4.7 TB, 8.8 TB, and 3.2 TB for matrices 15Hz, D5, and perf008ar2, respectively. Matrix 15Hz would also require a maximum of 91 GB per process on 90×10 cores, while matrix 20Hz would require a maximum of 151 GB per process on 200×12 cores. These values largely surpass the available memory on **brunch** and **eosmesca** (shared-memory, 1.5 and 2 TB, respectively), and **eos** and **occigen** (distributed-memory, 64 GB and 128 GB per node, respectively), and therefore the FR factorization runs out of memory (OOM) in all cases.

		FR		BLR (CB_{FR})	BLR (CB_{LR})		Active front
1 process	15Hz	4660* (OOM)		1208	971		185
	D5	7663* (OOM)		OOM**	1628		549
	perf008ar2	3187* (OOM)		1156	975		525
		avg.	max.		avg.	max.	
90 processes	15Hz	79*	91*	OOM**	37	53	18
200 processes	20Hz	125*	151*	OOM**	54	81	27

Table 9.6 – Very large problems: total memory consumption (average per process and maximum). All values are provided in GB. In shared-memory, matrices 15Hz and perf008ar2 were run on **brunch** while matrix D5 was run on **eosmesca**. In distributed-memory, matrices 15Hz and 20Hz were run on **eos** and **occigen**, respectively. * estimated memory at the analysis; the experiment ran out of memory (OOM) during the factorization. ** ran out of memory; estimated memory required cannot be predicted at the analysis due to low-rank compression.

Thanks to the low-rank compression, the memory requirements for the problems to fit in-core are greatly reduced. Compressing the CB also significantly reduces the total memory, and is even necessary in the case of matrices 15Hz and 20Hz on 900 and 2400 cores, respectively. Matrix 15Hz on 90 processes achieves an average and maximum memory efficiency of 29% and 20%, respectively. These relatively low values illustrate the issue discussed in Section 9.3: the active front storage becomes dominant on large number of processes, requiring 18 GB out of 53 GB (compared to 185 out of 971 GB on 1 process).

Finally, in Table 9.7, we analyze the computational cost of the BLR CB_{LR} solver. We measure the elapsed time in the analysis (FR analysis + overhead for computing the BLR clustering), factorization, and solution phases.

On these very large problems, the analysis time is significant. For matrix 15Hz, it remains limited in shared-memory (less than 8% of the factorization time), but becomes important in distributed-memory (51% of the factorization time). For matrices D5 and perf008ar2, the analysis time is also significant even in shared-memory (42% and 53% of the factorization time, respectively). This illustrates the necessity to investigate strategies to accelerate the analysis phase, as mentioned in Section 9.1.

The solution phase time is given for one RHS. For matrix 15Hz, it has been computed based on a run with 116 sparse RHS, while for matrix perf008ar2 it has

#processes ×#threads	matrix	FR analysis	BLR clustering	factorization	solution
1×48	15Hz	337	82	5392	1.2/RHS*
	D5	3328	107	8083	N/A
	perf008ar2	1777	190	3699	71.4
90×10	15Hz	337	100	856	0.2/RHS*
200×12	20Hz	921	216	2641	N/A

Table 9.7 – Very large problems: elapsed time in seconds for the BLR CB_{LR} solver. In shared-memory, the matrices were run on **brunch** using 48 threads. In distributed-memory, matrices 15Hz and 20Hz were run on **eos** and **occigen**, respectively, using 900 and 2400 cores, respectively. * time per RHS computed with a run with 116 RHS.

been computed with 1 dense RHS, to fit the typical use in these applications. This explains the performance difference because BLAS-3 operations can be used in the former case. The time for the solution with one RHS is relatively small with respect to the factorization phase. However, in the case of matrix 15Hz, the application (seismic modeling) requires to solve simultaneously a large number of RHS (several thousands); the total solution time can therefore be significant or even dominant compared to the factorization time, even on very large problems. This remark also applies to matrices 20Hz and D5. This has been illustrated on smaller size problems in Chapter 7.

Thanks to the BLR feature, we have thus been able to solve in-core very large problems at the accuracy required by the application with a very significant reduction of the memory footprint and of the number of operations required to factor the matrices.

Conclusion

Research contributions

In this thesis, we have investigated the suitability of Block Low-Rank solvers for reducing the cost of sparse multifrontal solvers without sacrificing their robustness and ease of use.

We first surveyed in Chapter 1 the existing low-rank formats that have been proposed in the literature to reduce the complexity of sparse direct solvers. We then focused on the BLR format, whose simplicity and flexibility make it easy to use in a general purpose, algebraic multifrontal solver allowing for numerical pivoting.

In Chapter 2, we have described the standard BLR factorization from Amestoy et al. (2015a), and introduced several variants to improve it. First, we proposed a variant which performs the compression earlier to achieve a higher reduction of the number of operations (Section 2.3.1). However, it is not straightforward to perform numerical pivoting with this variant; we designed an algorithm to make it possible by taking into account the low-rank blocks (Section 2.3.2). Then, we proposed an algorithm to reduce the cost of applying the low-rank updates by accumulating and recompressing them (Section 2.6). We presented in Chapter 3 several strategies to recompress these low-rank updates and performed a detailed analysis and comparison.

We have investigated the theoretical complexity of the BLR factorization, that was previously unknown, in Chapter 4. Simply applying the work done on hierarchical matrices does not lead to a satisfying result, so we extended it to prove that the complexity of the BLR factorization is asymptotically lower than that of the full-rank solver. For a three-dimensional problem, we computed that the factor size complexity of the multifrontal factorization is reduced from $\mathcal{O}(n^{4/3})$ to $\mathcal{O}(n \log n)$, while the number of operations is reduced from $\mathcal{O}(n^2)$ to $\mathcal{O}(n^{5/3})$ for the standard BLR variant. Moreover, the improved BLR variants can further reduce the flop complexity to $\mathcal{O}(n^{4/3})$. Theoretical complexity reduction results were also obtained for two-dimensional problems. We provided an experimental study with numerical results to support these complexity bounds.

After giving the theoretical justification that BLR solvers deserve to be considered because they can achieve low complexity, we turned to the problem of translating this low complexity into actual performance gains on modern architectures.

We first presented in Chapter 5 a multithreaded BLR factorization, and analyzed its performance in shared-memory multicore environments on a large set of real-life problems. We showed that the standard BLR variant does not fully translate the complexity reduction into time gains; tree-based multithreading, a left-looking factorization, and the algorithmic properties of the BLR variants are all critical to efficiently exploit multicore systems.

We then presented the distributed-memory BLR factorization in Chapter 6. We analyzed that it suffers from a high relative weight of communications and load unbalance. We showed both theoretically and experimentally that compressing the contribution block of the frontal matrices can greatly reduce the volume of communications. Moreover, even though compressing the CB represents an overhead cost, it can potentially also improve the time for factorization. We also revisited our mapping and splitting strategies to improve the load balance of the BLR factorization.

Throughout this thesis, we have illustrated the use of BLR solvers in three industrial applications coming from geosciences (seismic and electromagnetic modeling) and structural mechanics, as well as on numerous other matrices coming from a variety of real-life applications. In Chapter 7, we have provided a detailed case-study of the seismic and electromagnetic applications, which both rely on frequency-domain inversion. We have shown that sparse direct solvers, coupled with BLR acceleration, can be competitive with iterative solvers or time-domain approaches, for frequencies up to 10Hz (problems of order few tens of millions).

In Chapter 8, we have compared our BLR solver with the STRUMPACK (Ghysels et al., 2017) solver, based on the HSS format. Our results suggest that, among low-rank approximated direct solvers, BLR solvers work best as accurate, high-precision solvers, while HSS solvers tend to favor more aggressive approximations to build fast preconditioners. We hope that this comparison can pave the way towards a better understanding of the practical behavior and the differences between low-rank formats.

Finally, in Chapter 9, we have discussed the future challenges that await BLR solvers for large-scale systems and applications. Due to the reduction of the cost of the factorization, the analysis and solution phases will become of growing importance; moreover, to solve increasingly large problems, the memory efficiency of BLR solvers will be critical. We have proposed some ways to tackle these challenges, and illustrated them by reporting results on very large problems (up to 130 million unknowns).

Software contributions

The algorithms presented throughout this thesis have been implemented within the MUMPS solver. The standard BLR factorization algorithm was released to the public in MUMPS 5.1. We hope that the improved BLR variants will follow soon once they are mature and robust enough to be publicly exposed.

The simplicity of the BLR format has made it possible to integrate it within a general purpose solver such as MUMPS. In particular, the numerous features of MUMPS, such as out-of-core (Agullo et al., 2010) and memory-aware (Agullo et al., 2016) factorizations, numerical pivoting, dynamic scheduling, asynchronism (Sid-

Lakhdar, 2014), or the exploitation of sparse right-hand sides (Amestoy et al., 2015f; Amestoy et al., 2017f), can all be coupled with BLR approximations.

We emphasize that the ideas presented in this work do not rely on a particular implementation of the multifrontal method, and could therefore be applied to any multifrontal (or even supernodal, see next section) solver.

Perspectives

We briefly discuss remaining challenges and open questions that could be the object of further research.

We have already discussed in Chapter 9 some aspects of increasing importance for large-scale systems and applications: the performance of the analysis and solution phases, and the memory efficiency of BLR solvers.

Out of all the BLR variants presented in this thesis, two would need to be further studied: the offline compression FSUC variant (Section 2.1), and the CUFS variant (Section 2.4). We believe that the FSUC variant, as simple as it is, may be of interest in some applications with many right-hand sides. Moreover, as mentioned in Section 6.3.2, the CUFS variant is likely to greatly improve the performance of the factorization in parallel settings where the contribution block of the frontal matrices is compressed.

A rigorous error analysis of BLR solvers should be performed to better understand the effect of BLR approximations on the accuracy and stability of the solver and, in particular, answer the following four open questions. First, we have experimentally observed that the BLR solver generally yields a scaled residual of the same order as the low-rank threshold ε. It is unclear whether this can be proved theoretically. Second, because the error analysis in the hierarchical literature (Bebendorf, 2008) is based on block-wise norm estimates, the accuracy of the low-rank approximation depends on the sparsity constant, which in turn, in the BLR case, depends on the size of the problem (cf. Chapter 4). BLR solvers could thus become unstable as problems get larger and larger. Third, the BLR variants achieve a higher compression rate, which can potentially degrade the solution accuracy; this effect should be formally quantified. Finally, the use of an absolute low-rank threshold implies that the matrix is scaled before compressing it; the effect of the scaling choice should be studied.

The next bottleneck of the BLR factorization to be tackled is the compression cost. Indeed, as mentioned in Section 5.2, the cost of the Compress step is not negligible in terms of time, especially after all the improvements proposed in this thesis which have reduced the cost of the other steps. We have focused on the truncated QR factorization with column pivoting, but other kernels should be investigated. In particular, randomized approaches (Halko et al., 2011) attract growing interest. One difficulty for BLR solvers will be that the blocks are relatively small and of unknown rank; in this case, fixed-accuracy randomized sampling on small blocks tends to be less efficient.

A possible way to overcome this latter issue, and more generally to improve the performance of the low-rank operations, would be to use batched BLAS operations (Haidar, Dong, Luszczek, Tomov, and Dongarra, 2015), or specialized li-

braries for low granularity computations, such as LIBXSMM (Heinecke, Henry, Hutchinson, and Pabst, 2016) or BLASFEO (Frison, Kouzoupis, Zanelli, and Diehl, 2017). These technologies are especially relevant when hardware accelerators such as GPUs or MICs are used. In this context, the algorithms we have designed to improve the arithmetic intensity of BLR solvers will be even more critical.

A task-based multithreading could further improve the performance of the BLR factorization on multicores (Chapter 5). This approach, which has been described in the dense case in, for example, Anton et al. (2016) and Sergent, Goudin, Thibault, and Aumage (2016), would allow for a pipelining of the successive steps of the factorization as opposed to the fork-join approach hereby presented. However, the taskification of the BLR factorization is not straightforward as it raises two questions: how to control the memory consumption; and how much of the gain due to the left-looking factorization, which also makes the accumulation and recompression of low-rank updates possible, can be preserved?

Distributed-memory BLR solvers pose many challenges yet to be addressed. The ideas we have proposed in Section 6.4 to improve the load balance of the BLR factorization should be formalized and generalized. Robust mapping and scheduling strategies should be specially designed to account for the low-rank compression. Moreover, the influence of the splitting on the quality of the BLR clustering should also be further investigated. Finally, as the number of processes increases, synchronizations will become more expensive. Recent work by Sid-Lakhdar (2014) aiming to avoid such synchronizations in the full-rank case should be extended to the BLR case, for which it will certainly be even more critical.

As mentioned in the previous section, our BLR solver can benefit from a number of recent advances in sparse direct solvers that have been implemented in MUMPS. Some of them would deserve a dedicated study in conjunction with BLR approximations. For example, among the different strategies to reduce the memory consumption of sparse direct solvers, we could investigate how to find the best compromise between out-of-core, memory-aware, and low-rank approximation approaches.

Many of the algorithms proposed in this thesis can also be applied to supernodal solvers; however, some challenges specific to the supernodal approach arise. For example, Pichon, Darve, Faverge, Ramet, and Roman (2017) propose two methods to accelerate the right-looking supernodal solver PaStiX, which are based on the non fully-structured FCSU and fully-structured C;FSU variants. Unlike multifrontal solvers, right-looking supernodal solvers based on a non fully-structured factorization do not achieve memory gains (because all supernodes must be stored in full-rank, rather than just the active fronts); on the other hand, a fully-structured factorization suffers from the inefficient low-rank assembly operations, which require padding and recompressing low-rank matrices (cf. Section 1.4.3.2). The strategies we developed in Chapter 3 to recompress low-rank updates could be extended to this context.

Even though this thesis has focused on the BLR format, we believe that some of the ideas proposed in this thesis could be applied to hierarchical formats. For example, the algorithm to factorize a low-rank panel with threshold partial pivoting (Section 2.3.2); the accumulation and recompression of low-rank matrices (Chapter 3); and the communication and load balance analysis in distributed-memory settings (Sections 6.3 and 6.4) are all independent from the low-rank format used.

Moreover, we believe that it is currently not clear which low-rank format is best suited for which kind of system and application; we intend to pursue the comparison of the MUMPS and STRUMPACK solvers to improve the understanding of the differences between low-rank formats.

More generally, we believe that, when considering the factorization of a dense matrix of order m, there is a compromise to be found between the $\mathcal{O}(m^2)$ monolevel BLR complexity and the optimal $\mathcal{O}(m)$ hierarchical complexity. In particular, we are working on a strategy to set the number of levels in the block hierarchy to some constant value in order to achieve a desired $\mathcal{O}(m^\alpha)$ complexity, with $1 \leq \alpha \leq 2$. This can be especially relevant for 3D sparse direct solvers, for which a dense complexity lower than $\mathcal{O}(m^{1.5})$ already leads to an optimal sparse $\mathcal{O}(n)$ complexity. By striking a balance between the simplicity of the BLR format and the low complexity of the hierarchical ones, this multilevel format will bridge the gap between BLR and hierarchical matrices.

Publications related to the thesis

Some of the work presented in this thesis has been the object of communications to the scientific community, as reported below. We have published the content of Chapter 4 in the SIAM *Journal of Scientific Computing* (Amestoy, Buttari, L'Excellent, and Mary, 2017d). The applicative case-studies presented in Sections 7.1 and 7.2 are the object of a collaboration with the SEISCOPE consortium and EMGS, respectively, and have been published in *Geophysics* (Amestoy et al., 2016b), and *Geophysical Journal International* (Shantsev et al., 2017), respectively. Finally, we have submitted the work of Chapter 5 to the ACM *Transactions on Mathematical Software* (Amestoy, Buttari, L'Excellent, and Mary, 2017e).

Submitted articles

Amestoy, P. R., A. Buttari, J.-Y. L'Excellent, and T. Mary (2017e). "Performance and Scalability of the Block Low-Rank Multifrontal Factorization on Multicore Architectures". In: submitted to *ACM Transactions on Mathematical Software*.

Articles

Amestoy, P. R., R. Brossier, A. Buttari, J.-Y. L'Excellent, T. Mary, L. Métivier, A. Miniussi, and S. Operto (2016b). "Fast 3D frequency-domain full waveform inversion with a parallel Block Low-Rank multifrontal direct solver: application to OBC data from the North Sea". In: *Geophysics* 81.6, R363–R383.

Amestoy, P. R., A. Buttari, J.-Y. L'Excellent, and T. Mary (2017d). "On the complexity of the Block Low-Rank multifrontal factorization". In: *SIAM Journal on Scientific Computing* 39.4, A1710–A1740.

Shantsev, D. V., P. Jaysaval, S. de la Kethulle de Ryhove, P. R. Amestoy, A. Buttari, J.-Y. L'Excellent, and T. Mary (2017). "Large-scale 3D EM modeling with a Block Low-Rank multifrontal direct solver". In: *Geophysical Journal International* 209.3, pp. 1558–1571.

Proceedings

Amestoy, P. R., R. Brossier, A. Buttari, J.-Y. L'Excellent, T. Mary, L. Métivier, A. Miniussi, S. Operto, A. Ribodetti, J. Virieux, and C. Weisbecker (2015c). "Efficient 3D frequency-domain full-waveform inversion of ocean-bottom cable data with sparse block low-rank direct solver: a real data case study from the North Sea". In: *SEG Technical Program Expanded Abstracts 2015*. Chap. 251, pp. 1303–1308.

Amestoy, P. R., R. Brossier, A. Buttari, J.-Y. L'Excellent, T. Mary, L. Métivier, A. Miniussi, S. Operto, J. Virieux, and C. Weisbecker (2015d). "3D frequency-domain seismic modeling with a Parallel BLR multifrontal direct solver". In: *SEG Technical Program Expanded Abstracts 2015*. Chap. 692, pp. 3606–3611.

Conference presentations

Amestoy, P. R., J. Anton, C. Ashcraft, A. Buttari, P. Ghysels, J.-Y. L'Excellent, X. S. Li, T. Mary, F.-H. Rouet, and C. Weisbecker (2016a). "A comparison of parallel rank-structured solvers". In: *SIAM Conference on Parallel Processing (SIAM PP16)*. Paris, France.

Amestoy, P. R., C. Ashcraft, A. Buttari, P. Ghysels, J.-Y. L'Excellent, X. S. Li, T. Mary, F.-H. Rouet, and C. Weisbecker (2015b). "A comparison of different low-rank approximation techniques". In: *SIAM Conference on Linear Algebra (SIAM LA15)*. Atlanta, USA.

Amestoy, P. R., A. Buttari, P. Ghysels, J.-Y. L'Excellent, X. S. Li, T. Mary, and F.-H. Rouet (2017a). "Comparison of BLR and HSS low-rank formats in multifrontal solvers: theory and practice". In: *SIAM Conference on Computational Science and Engineering (SIAM CSE17)*. Atlanta, USA.

Amestoy, P. R., A. Buttari, J.-Y. L'Excellent, and T. Mary (2015e). "Improving multifrontal solvers by means of Block Low-Rank approximations". In: *CIMI HPC Semester: workshop on fast direct solvers*. Toulouse, France.

Amestoy, P. R., A. Buttari, J.-Y. L'Excellent, and T. Mary (2016c). "Complexity and performance of the Block Low-Rank multifrontal factorization". In: *SIAM Conference on Parallel Processing (SIAM PP16)*. Paris, France.

Amestoy, P. R., A. Buttari, J.-Y. L'Excellent, and T. Mary (2016d). "On the complexity of the Block Low-Rank multifrontal factorization". In: *Sparse Days*. Toulouse, France.

Amestoy, P. R., A. Buttari, J.-Y. L'Excellent, and T. Mary (2016e). "Performance and scalability of the Block Low-Rank multifrontal factorization". In: *Parallel Matrix Algorithms and Applications (PMAA'16)*. Bordeaux, France.

Amestoy, P. R., A. Buttari, J.-Y. L'Excellent, and T. Mary (2016f). "Sparse direct solvers towards seismic imaging of large 3D domains". In: *78th EAGE Conference, workshop methods and challenges of seismic wave modelling for seismic imaging*. Vienna, Austria.

Amestoy, P. R., A. Buttari, J.-Y. L'Excellent, and T. Mary (2017b). "Block Low-Rank multifrontal solvers: complexity, performance, and scalability". In: *Sparse Days*. Toulouse, France.

Amestoy, P. R., A. Buttari, J.-Y. L'Excellent, and T. Mary (2017c). "Block Low-Rank multifrontal sparse direct solvers". In: *Mathias 2017*. Paris, France.

References

Aasen, J. O. (1971). "On the Reduction of a Symmetric Matrix to Tridiagonal Form". In: *BIT* 11, pp. 233–242.

Agullo, E., P. R. Amestoy, A. Buttari, A. Guermouche, J.-Y. L'Excellent, and F.-H. Rouet (2016). "Robust memory-aware mappings for parallel multifrontal factorizations". In: *SIAM Journal on Scientific Computing* 38.3, pp. C256–C279.

Agullo, E., A. Guermouche, and J.-Y. L'Excellent (2010). "Reducing the I/O Volume in Sparse Out-of-core Multifrontal Methods". In: *SIAM Journal on Scientific Computing* 31.6, pp. 4774–4794.

Aho, A. V., M. R. Garey, and J. D. Ullman (1972). "The transitive reduction of a directed graph". In: *SIAM Journal on Computing* 1, pp. 131–137.

Aho, A. V., J. E. Hopcroft, and J. D. Ullman (1983). *Data Structures and Algorithms*. Reading, MA.: Addison-Wesley.

Akbudak, K., H. Ltaief, A. Mikhalev, and D. Keyes (2017). "Tile Low Rank Cholesky Factorization for Climate/Weather Modeling Applications on Manycore Architectures". In: *Proceedings of ISC'17*.

Amaya, M. (2015). "High-order optimization methods for large-scale 3D CSEM data inversion". PhD thesis. NTNU.

Amestoy, P. R., C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L'Excellent, and C. Weisbecker (2015a). "Improving multifrontal methods by means of block low-rank representations". In: *SIAM Journal on Scientific Computing* 37.3, A1451–A1474.

Amestoy, P. R., A. Buttari, I. S. Duff, A. Guermouche, J.-Y. L'Excellent, and B. Uçar (2011). "MUMPS". In: *Encyclopedia of Parallel Computing*. Ed. by D. Padua. Springer, pp. 1232–1238.

Amestoy, P. R., A. Buttari, and J.-Y. L'Excellent (2008). *Towards a parallel analysis phase for a multifrontal sparse solver*. Presentation at the 5th International workshop on Parallel Matrix Algorithms and Applications (PMAA'08).

Amestoy, P. R., T. A. Davis, and I. S. Duff (1996). "An approximate minimum degree ordering algorithm". In: *SIAM Journal on Matrix Analysis and Applications* 17.4, pp. 886–905.

Amestoy, P. R., I. S. Duff, J. Koster, and J.-Y. L'Excellent (2001). "A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling". In: *SIAM Journal on Matrix Analysis and Applications* 23.1, pp. 15–41.

Amestoy, P. R., I. S. Duff, J.-Y. L'Excellent, and F.-H. Rouet (2015f). "Parallel computation of entries of A^{-1}". In: *SIAM Journal on Scientific Computing* 37.2, pp. C268–C284.

Amestoy, P. R., A. Guermouche, J.-Y. L'Excellent, and S. Pralet (2006). "Hybrid scheduling for the parallel solution of linear systems". In: *Parallel Computing* 32.2, pp. 136–156.

Amestoy, P. R., J.-Y. L'Excellent, and G. Moreau (2017f). *On Exploiting Sparsity of Multiple Right-Hand Sides in Sparse Direct Solvers*. Research report RR-9122. INRIA.

Amestoy, P. R., J.-Y. L'Excellent, and G. Moreau (2017g). "Performance of the Solution Phase: Recent Work and Perspectives". In: *Internal communication*. Grenoble, France.

Amestoy, P. R. and C. Puglisi (2002). "An unsymmetrized multifrontal LU factorization". In: *SIAM Journal on Matrix Analysis and Applications* 24, pp. 553–569.

Aminfar, A., S. Ambikasaran, and E. Darve (2016). "A fast block low-rank dense solver with applications to finite-element matrices". In: *Journal of Computational Physics* 304, pp. 170–188.

Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen (1995). *LAPACK Users' Guide*. Third. Philadelphia, PA: SIAM Press.

Andreis, D. and L. MacGregor (2008). "Controlled-Source Electro-magnetic Sounding in Shallow Water: Principles and Applications". In: *Geophysics* 73.1, F21–F32.

Anton, J., C. Ashcraft, and C. Weisbecker (2016). "A Block Low-Rank multithreaded factorization for dense BEM operators". In: *SIAM Conference on Parallel Processing (SIAM PP16)*. Paris, France.

Ashcraft, C., R. G. Grimes, and J. G. Lewis (1998). "Accurate symmetric indefinite linear equation solver". In: *SIAM Journal on Matrix Analysis and Applications* 20, pp. 513–561.

Avdeev, D. B. (2005). "Three-dimensional electromagnetic modelling and inversion from theory to application". In: *Surveys in Geophysics* 26.6, pp. 767–799.

Barkved, O., U. Albertin, P. Heavey, J. Kommedal, J. van Gestel, R. Synnove, H. Pettersen, and C. Kent (2010). "Business Impact of Full Waveform Inversion at Valhall". In: *Expanded Abstracts, 91 Annual SEG Meeting and Exposition (October 17-22, Denver)*. Society of Exploration Geophysics, pp. 925–929.

Barnes, C. and M. Charara (2009). "The domain of applicability of acoustic full-waveform inversion for marine seismic data". In: *Geophysics* 74.6, WCC91–WCC103.

Bebendorf, M. (2000). "Approximation of boundary element matrices". In: *Numerische Mathematik* 86.4, pp. 565–589.

Bebendorf, M. (2005). "Efficient inversion of Galerkin matrices of general second-order elliptic differential operators with nonsmooth coefficients". In: *Mathematics of Computation* 74, pp. 1179–1199.

Bebendorf, M. (2007). "Why finite element discretizations can be factored by triangular hierarchical matrices". In: *SIAM Journal on Numerical Analysis* 45, p. 1472.

Bebendorf, M. (2008). *Hierarchical Matrices: A Means to Efficiently Solve Elliptic Boundary Value Problems*. Vol. 63. Lecture Notes in Computational Science and Engineering (LNCSE). Springer-Verlag.

Bebendorf, M. and W. Hackbusch (2003). "Existence of $\mathscr{H}$-matrix approximants to the inverse FE-matrix of elliptic operators with L^∞-coefficients". In: *Numerische Mathematik* 95.1, pp. 1–28.

Ben Hadj Ali, H., S. Operto, and J. Virieux (2008). "Velocity model building by 3D frequency-domain, full-waveform inversion of wide-aperture seismic data". In: *Geophysics* 73.5 (5), VE101–VE117.

Bérenger, J. P. (1996). "Three-dimensional perfectly matched layer for the absorption of electromagnetic waves". In: *Journal of Computational Physics* 127.2, pp. 363–379.

Blome, M., H. Maurer, and K. Schmidt (2009). "Advances in three-dimensional geoelectric forward solver techniques". In: *Geophysical Journal International* 176.3, pp. 740–752.

Börm, S., L. Grasedyck, and W. Hackbusch (2003). "Introduction to hierarchical matrices with applications". In: *Engineering analysis with boundary elements* 27.5, pp. 405–422.

Börner, R.-U. (2010). "Numerical modelling in geo-electromagnetics: advances and challenges". In: *Surveys in Geophysics* 31.2, pp. 225–245.

Brossier, R., V. Etienne, S. Operto, and J. Virieux (2010). "Frequency-domain numerical modelling of visco-acoustic waves based on finite-difference and finite-element discontinuous Galerkin methods". In: *Acoustic Waves*. Ed. by D. W. Dissanayake. SCIYO, pp. 125–158.

Bunch, J. R. and L. Kaufman (1977). "Some Stable Methods for Calculating Inertia and Solving Symmetric Linear Systems". In: *Mathematics of Computation* 31, pp. 162–179.

Bunch, J. R. and B. N. Parlett (1971). "Direct Methods for Solving Symmetric Indefinite Systems of Linear Systems". In: *SIAM Journal on Numerical Analysis* 8, pp. 639–655.

Businger, P. and G. H. Golub (1965). "Linear Least Squares Solutions by Householder Transformations". In: *Numerische Mathematik* 7, pp. 269–276.

Buttari, A., J. Langou, J. Kurzak, and J. Dongarra (2009). "A class of parallel tiled linear algebra algorithms for multicore architectures". In: *Parallel Computing* 35.1, pp. 38–53.

Castagna, J. P. (1993). "AVO analysis - tutorial and review". In: *Offset-dependent reflectivity - Theory and practice of AVO analysis*. Vol. 8. Eds Castagna J. P. and Backus M. M., Investigations in geophysics, Soc. Expl. Geophys., Tulsa, Oklahoma, pp. 3–36.

Chandrasekaran, S., P. Dewilde, M. Gu, and N. Somasunderam (2010). "On the Numerical Rank of the Off-Diagonal Blocks of Schur Complements of Discretized Elliptic PDEs". In: *SIAM Journal on Matrix Analysis and Applications* 31.5, pp. 2261–2290.

Chandrasekaran, S., M. Gu, and T. Pals (2006). "A fast ULV decomposition solver for hierarchically semiseparable representations". In: *SIAM Journal on Matrix Analysis and Applications* 28.3, pp. 603–622.

Cheng, H., Z. Gimbutas, P. G. Martinsson, and V. Rokhlin (2005). “On the Compression of Low Rank Matrices”. In: *SIAM Journal on Scientific Computing* 26.4, pp. 1389–1404.

Chevalier, C. and F. Pellegrini (2006). “PT-Scotch: A tool for efficient parallel graph ordering”. In: *Proceedings of PMAA2006, Rennes, France*.

Constable, S. (2010). “Ten years of marine CSEM for hydrocarbon exploration”. In: *Geophysics* 75.5, 75A67–75A81.

Davis, T. A. and I. S. Duff (1997). “An unsymmetric-pattern multifrontal method for sparse LU factorization”. In: *SIAM Journal on Matrix Analysis and Applications* 18, pp. 140–158.

Davis, T. A. and Y. Hu (2011). “The university of Florida sparse matrix collection”. In: *ACM Transactions on Mathematical Software* 38.1, 1:1–1:25.

Davydycheva, S. (2010). “3D modeling of new-generation (1999–2010) resistivity logging tools”. In: *The Leading Edge* 29.7, pp. 780–789.

Demmel, J. W., S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu (1999). “A supernodal approach to sparse partial pivoting”. In: *SIAM Journal on Matrix Analysis and Applications* 20.3, pp. 720–755.

Dongarra, J. J., J. Du Croz, S. Hammarling, and I. S. Duff (1990). “A set of level 3 basic linear algebra subprograms”. In: *ACM Trans. Math. Softw.* 16.1, pp. 1–17.

Dongarra, J., J. R. Bunch, C. B. Moler, and G. W. Stewart (1979). *LINPACK Users Guide*. Philadelphia: SIAM.

Dongarra, J. J., I. S. Duff, D. C. Sorensen, and H. A. van der Vorst (1998). *Numerical Linear Algebra for High-Performance Computers*. Philadelphia: SIAM Press.

Duff, I. S., A. M. Erisman, and J. K. Reid (1986). *Direct Methods for Sparse Matrices*. London: Oxford University Press.

Duff, I. S. and S. Pralet (2005). “Strategies for scaling and pivoting for sparse symmetric indefinite problems”. In: *SIAM Journal on Matrix Analysis and Applications* 27.2, pp. 313–340.

Duff, I. S. and S. Pralet (2007). “Towards Stable Mixed Pivoting Strategies for the Sequential and Parallel Solution of Sparse Symmetric Indefinite Systems”. In: *SIAM Journal on Matrix Anaysis and Applications* 29.3, pp. 1007–1024.

Duff, I. S. and J. K. Reid (1983). “The multifrontal solution of indefinite sparse symmetric linear systems”. In: *ACM Transactions on Mathematical Software* 9, pp. 302–325.

Duff, I. S. and J. K. Reid (1984). “The multifrontal solution of unsymmetric sets of linear systems”. In: *SIAM Journal on Scientific and Statistical Computing* 5, pp. 633–641.

Duff, I. S. and J. K. Reid (1996). “Exploiting zeros on the diagonal in the direct solution of indefinite sparse symmetric linear systems”. In: *ACM Transactions on Mathematical Software* 22.2, pp. 227–257.

Eckard, C. and G. Young (1936). “The Approximation of One Matrix by Another of Lower Rank”. In: *Psychometrica* 1, pp. 211–218.

Eisenstat, S. C. and J. W. H. Liu (2005). “The theory of elimination trees for sparse unsymmetric matrices”. In: *SIAM Journal on Matrix Analysis and Applications* 26, pp. 686–705.

Ellingsrud, S., T. Eidesmo, S. Johansen, M. C. Sinha, L. M. MacGregor, and S. Constable (2002). “Remote sensing of hydrocarbon layers by seabed logging (SBL):

Results from a cruise offshore Angola". In: *The Leading Edge* 21.10, pp. 972–982.

Engquist, B. and L. Ying (2011). "Sweeping preconditioner for the Helmholtz equation: Hierarchical matrix representation". In: *Communications on Pure and Applied Mathematics* 64.5, pp. 697–735.

Erlangga, Y. A. and R. Nabben (2008). "On a multilevel Krylov method for the Helmholtz equation preconditioned by shifted Laplacian". In: *Electronic Transactions on Numerical Analysis* 31, pp. 403–424.

Frison, G., D. Kouzoupis, A. Zanelli, and M. Diehl (2017). "BLASFEO: Basic linear algebra subroutines for embedded optimization". In: *CoRR* abs/1704.02457.

George, J. A. (1973). "Nested dissection of a regular finite-element mesh". In: *SIAM Journal on Numerical Analysis* 10.2, pp. 345–363.

Ghysels, P., X. S. Li, C. Gorman, and F. H. Rouet (2017). "A Robust Parallel Preconditioner for Indefinite Systems Using Hierarchical Matrices and Randomized Sampling". In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 897–906.

Ghysels, P., X. S. Li, F.-H. Rouet, S. Williams, and A. Napov (2016). "An Efficient Multicore Implementation of a Novel HSS-Structured Multifrontal Solver Using Randomized Sampling". In: *SIAM Journal on Scientific Computing* 38.5, S358–S384.

Gilbert, J. R. and J. W. H. Liu (1993). "Elimination structures for unsymmetric sparse LU factors". In: *SIAM Journal on Matrix Analysis and Applications* 14, pp. 334–352.

Gillman, A. (2011). "Fast direct solvers for elliptic partial differential equations". PhD thesis. University of Colorado.

Gillman, A., P. Young, and P.-G. Martinsson (2012). "A direct solver with $\mathcal{O}$(N) complexity for integral equations on one-dimensional domains". In: *Frontiers of Mathematics in China* 7 (2), pp. 217–247.

Golub, G. H. and C. F. Van Loan (2012). *Matrix Computations. 4th ed.* Baltimore, MD.: Johns Hopkins Press.

Gosselin-Cliche, B. and B. Giroux (2014). "3D frequency-domain finite-difference viscoelastic-wave modeling using weighted average 27-point operators with optimal coefficients". In: *Geophysics* 79.3, T169–T188.

Grasedyck, L. and W. Hackbusch (2003). "Construction and Arithmetics of H-Matrices". English. In: *Computing* 70.4, pp. 295–334.

Grasedyck, L., R. Kriemann, and S. Le Borne (2008). "Parallel black box $\mathcal{H}$-LU preconditioning for elliptic boundary value problems". In: *Computing and Visualization in Science* 11.4, pp. 273–291.

Grayver, A. and R. Streich (2012). "Comparison of iterative and direct solvers for 3D CSEM modeling". In: *SEG Technical Program Expanded Abstracts 2012*. Society of Exploration Geophysicists, pp. 1–6.

Gupta, A. (2002). "Recent advances in direct methods for solving unsymmetric sparse systems of linear equations". In: *ACM Transactions on Mathematical Software* 28.3, pp. 301–324.

Gustavson, F. (1997). "Recursion leads to automatic variable blocking for dense linear-algebra algorithms". In: *IBM Journal of Research and Development* 41.6, pp. 737–755.

Gutknecht, M. H. (1993). “Variants of BICGSTAB for matrices with complex spectrum”. In: *SIAM Journal on Scientific Computing* 14.5, pp. 1020–1033.

Habashy, T. M. and A. Abubakar (2004). “A general framework for constraint minimization for the inversion of electromagnetic measurements”. In: *Progress in electromagnetics Research* 46, pp. 265–312.

Hackbusch, W., B. N. Khoromskij, and R. Kriemann (2004). “Hierarchical Matrices Based on a Weak Admissibility Criterion”. English. In: *Computing* 73.3, pp. 207–243.

Hackbusch, W. (1999). “A sparse matrix arithmetic based on H-matrices. Part I: introduction to H-matrices”. In: *Computing* 62.2, pp. 89–108.

Haidar, A., T. Dong, P. Luszczek, S. Tomov, and J. Dongarra (2015). “Towards batched linear solvers on accelerated hardware platforms”. In: *ACM SIGPLAN Notices*. Vol. 50. 8. ACM, pp. 261–262.

Halko, N., P.-G. Martinsson, and J. A. Tropp (2011). “Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions”. In: *SIAM Review* 53.2, pp. 217–288.

Heinecke, A., G. Henry, M. Hutchinson, and H. Pabst (2016). “LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation”. In: *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 981–991.

Hogg, J. D. and J. A. Scott (2014). “Compressed threshold pivoting for sparse symmetric indefinite systems”. In: *SIAM Journal on Matrix Analysis and Applications* 35.2, pp. 783–817.

Hustedt, B., S. Operto, and J. Virieux (2004). “Mixed-grid and staggered-grid finite difference methods for frequency domain acoustic wave modelling”. In: *Geophysical Journal International* 157, pp. 1269–1296.

Jaysaval, P., D. V. Shantsev, and S. de la Kethulle de Ryhove (2015). “Efficient 3-D controlled-source electromagnetic modelling using an exponential finite-difference method”. In: *Geophysical Journal International* 203.3, pp. 1541–1574.

Jaysaval, P., D. V. Shantsev, S. de la Kethulle de Ryhove, and T. Bratteland (2016). “Fully anisotropic 3-D EM modelling on a Lebedev grid with a multigrid preconditioner”. In: *Geophysical Journal International* 207.3, pp. 1554–1572.

Jaysaval, P., D. Shantsev, and S. de la Kethulle de Ryhove (2014). “Fast multimodel finite-difference controlled-source electromagnetic simulations based on a Schur complement approach”. In: *Geophysics* 79.6, E315–E327.

Jo, C. H., C. Shin, and J. H. Suh (1996). “An optimal 9-point, finite-difference, frequency-space 2D scalar extrapolator”. In: *Geophysics* 61, pp. 529–537.

Karypis, G. and V. Kumar (1998). METIS – *A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices – Version 4.0*. University of Minnesota.

Key, K. (2012). “Marine Electromagnetic Studies of Seafloor Resources and Tectonics”. In: *Surveys in Geophysics* 33.1, pp. 135–167.

Knight, P. A., D. Ruiz, and B. Uçar (2014). “A Symmetry Preserving Algorithm for Matrix Scaling”. In: *SIAM Journal on Matrix Analysis and Applications* 35.3, pp. 931–955.

Krebs, J., J. Anderson, D. Hinkley, R. Neelamani, S. Lee, A. Baumstein, and M. D. Lacasse (2009). “Fast Full-Wavefield Seismic Inversion Using Encoded Sources”. In: *Geophysics* 74(6), WCC105–WCC116.

L’Excellent, J.-Y. and M. W. Sid-Lakhdar (2014). “A study of shared-memory parallelism in a multifrontal solver”. In: *Parallel Computing* 40.3-4, pp. 34–46.

Li, X. S. and J. W. Demmel (1998). “Making sparse Gaussian elimination scalable by static pivoting”. In: *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, pp. 1–17.

Li, X. S. and J. W. Demmel (2003). “SuperLU_DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems”. In: *ACM Transactions on Mathematical Software* 29.2, pp. 110–140.

Li, Y., L. Métivier, R. Brossier, B. Han, and J. Virieux (2015). “2D and 3D frequency-domain elastic wave modeling in complex media with a parallel iterative solver”. In: *Geophysics* 80(3), T101–T118.

Liberty, E., F. Woolfe, P.-G. Martinsson, V. Rokhlin, and M. Tygert (2007). “Randomized algorithms for the low-rank approximation of matrices”. In: *Proceedings of the National Academy of Sciences* 104.51, pp. 20167–20172.

Liu, J. W. H. (1985). “Modification of the Minimum Degree Algorithm by Multiple Elimination”. In: *ACM Transactions on Mathematical Software* 11.2, pp. 141–153.

Liu, J. W. H. (1992). “The multifrontal method for sparse matrix solution: Theory and Practice”. In: *SIAM Review* 34, pp. 82–109.

Liu, J. W. (1987). “A partial pivoting strategy for sparse symmetric matrix decomposition”. In: *ACM Transactions on Mathematical Software* 13.2, pp. 173–182.

Mahoney, M. W. and P. Drineas (2009). “CUR matrix decompositions for improved data analysis”. In: *Proceedings of the National Academy of Sciences* 106.3, pp. 697–702.

Marfurt, K. (1984). “Accuracy of finite-difference and finite-element modeling of the scalar and elastic wave equations”. In: *Geophysics* 49, pp. 533–549.

Martinsson, P. G. (2011). “A Fast Randomized Algorithm for Computing a Hierarchically Semiseparable Representation of a Matrix”. In: *SIAM Journal on Matrix Analysis and Applications* 32.4, pp. 1251–1274.

Martinsson, P.-G. (2016). “Compressing Rank-Structured Matrices via Randomized Sampling”. In: *SIAM Journal on Scientific Computing* 38.4, A1959–A1986.

Métivier, L. and R. Brossier (2016). “The SEISCOPE optimization toolbox: A large-scale nonlinear optimization library based on reverse communication”. In: *GEOPHYSICS* 81.2, F1–F15.

Min, D. J., C. Sin, B.-D. Kwon, and S. Chung (2000). “Improved frequency-domain elastic wave modeling using weighted-averaging difference operators”. In: *Geophysics* 65, pp. 884–895.

Mulder, W. (2006). “A multigrid solver for 3D electromagnetic diffusion”. In: *Geophysical prospecting* 54.5, pp. 633–649.

Newman, G. A. and D. L. Alumbaugh (1995). “Frequency-domain modelling of airborne electromagnetic responses using staggered finite differences”. In: *Geophysical Prospecting* 43.8, pp. 1021–1042.

Ng, E. G. and P. Raghavan (1999). “Performance of greedy heuristics for sparse Cholesky factorization”. In: *SIAM Journal on Matrix Analysis and Applications* 20, pp. 902–914.

Nihei, K. T. and X. Li (2007). “Frequency response modelling of seismic waves using finite difference time domain with phase sensitive detection (TD-PSD)”. In: *Geophysical Journal International* 169, pp. 1069–1078.

Oettli, W. and W. Prager (1964). “Compatibility of Approximate Solution of Linear Equations with Given Error Bounds for Coefficients and Right-Hand Sides”. In: *Numerische Mathematik* 6, pp. 405–409.

Oldenburg, D. W., E. Haber, and R. Shekhtman (2013). “Three dimensional inversion of multisource time domain electromagnetic data”. In: *Geophysics* 78.1, E47–E57.

Operto, S., R. Brossier, L. Combe, L. Métivier, A. Ribodetti, and J. Virieux (2014). “Computationally-efficient three-dimensional visco-acoustic finite-difference frequency-domain seismic modeling in vertical transversely isotropic media with sparse direct solver”. In: *Geophysics* 79(5), T257–T275.

Operto, S., A. Miniussi, R. Brossier, L. Combe, L. Métivier, V. Monteiller, A. Ribodetti, and J. Virieux (2015). “Efficient 3-D frequency-domain mono-parameter full-waveform inversion of ocean-bottom cable data: application to Valhall in the visco-acoustic vertical transverse isotropic approximation”. In: *Geophysical Journal International* 202.2, pp. 1362–1391.

Operto, S., J. Virieux, P. R. Amestoy, J.-Y. L’Excellent, L. Giraud, and H. Ben Hadj Ali (2007). “3D finite-difference frequency-domain modeling of visco-acoustic wave propagation using a massively parallel direct solver: A feasibility study”. In: *Geophysics* 72.5, SM195–SM211.

Parlett, B. N. and J. K. Reid (1970). “On the Solution of a System of Linear Equations whose Matrix is Symmetric but not Definite”. In: *BIT* 10, pp. 386–397.

Peiró, J. and S. Sherwin (2005). “Finite difference, finite element and finite volume methods for partial differential equations”. In: *Handbook of materials modeling*. Springer, pp. 2415–2446.

Pellegrini, F. (2007). SCOTCH *and* LIBSCOTCH 5.0 *User’s guide*. Technical Report. LaBRI.

Petrov, P. V. and G. A. Newman (2012). “3D finite-difference modeling of elastic wave propagation in the Laplace-Fourier domain”. In: *Geophysics* 77(4).4, T137–T155.

Petrov, P. V. and G. A. Newman (2014). “Three-dimensional inverse modelling of damped elastic wave propagation in the Fourier domain”. In: *Geophysical Journal International* 198, pp. 1599–1617.

Pichon, G., E. Darve, M. Faverge, P. Ramet, and J. Roman (2017). “Sparse Supernodal Solver Using Block Low-Rank Compression”. In: *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 1138–1147.

Plessix, R.-E. (2007). “A Helmholtz iterative solver for 3D seismic-imaging problems”. In: *Geophysics* 72.5, SM185–SM194.

Plessix, R.-E. (2009). “Three-dimensional frequency-domain full-waveform inversion with an iterative solver”. In: *Geophysics* 74.6, WCC53–WCC61.

Plessix, R.-E. and C. Perez Solano (2015). “Modified surface boundary conditions for elastic waveform inversion of low-frequency wide-angle active land seismic data”. In: *Geophysical Journal International* 201, pp. 1324–1334.

Pothen, A. and C. Sun (1993). “A Mapping Algorithm for Parallel Sparse Cholesky Factorization”. In: *SIAM Journal on Scientific Computing* 14(5), pp. 1253–1257.

Pratt, R. G. (1999). “Seismic waveform inversion in the frequency domain, part I : theory and verification in a physic scale model”. In: *Geophysics* 64, pp. 888–901.

Pratt, R. G. and M. H. Worthington (1990). “Inverse theory applied to multi-source cross-hole tomography. Part I: acoustic wave-equation method”. In: *Geophysical Prospecting* 38, pp. 287–310.

Prieux, V., R. Brossier, Y. Gholami, S. Operto, J. Virieux, O. Barkved, and J. Kommedal (2011). “On the footprint of anisotropy on isotropic full waveform inversion: the Valhall case study”. In: *Geophysical Journal International* 187, pp. 1495–1515.

Prieux, V., R. Brossier, S. Operto, and J. Virieux (2013a). “Multiparameter full waveform inversion of multicomponent OBC data from Valhall. Part 1: imaging compressional wavespeed, density and attenuation”. In: *Geophysical Journal International* 194.3, pp. 1640–1664.

Prieux, V., R. Brossier, S. Operto, and J. Virieux (2013b). “Multiparameter full waveform inversion of multicomponent OBC data from Valhall. Part 2: imaging compressional and shear-wave velocities”. In: *Geophysical Journal International* 194.3, pp. 1665–1681.

Puzyrev, V., J. Koldan, J. de la Puente, G. Houzeaux, M. Vázquez, and J. M. Cela (2013). “A parallel finite-element method for three-dimensional controlled-source electromagnetic forward modelling”. In: *Geophysical Journal International* 193.2, pp. 678–693.

Puzyrev, V., S. Koric, and S. Wilkin (2016). “Evaluation of parallel direct sparse linear solvers in electromagnetic geophysical problems”. In: *Computers & Geosciences* 89, pp. 79–87.

Quintana-Ortí, G., E. S. Quintana-Ortí, R. A. V. D. Geijn, F. G. V. Zee, and E. Chan (2009). “Programming Matrix Algorithms-by-blocks for Thread-level Parallelism”. In: *ACM Transactions on Mathematical Software* 36.3, 14:1–14:26.

Rigal, J. and J. Gaches (1967). “On the compatibility of a given solution with the data of a linear system”. In: *Journal of the ACM* 14, pp. 526–543.

Riyanti, C. D., Y. A. Erlangga, R. .-.-E. Plessix, W. A. Mulder, C. Vuik, and C. Oosterlee (2006). “A new iterative solver for the time-harmonic wave equation”. In: *Geophysics* 71.E, pp. 57–63.

Rose, D. J., R. E. Tarjan, and G. S. Lueker (1976). “Algorithmic aspects of vertex elimination on graphs”. In: *SIAM Journal on Computing* 5.2, pp. 266–283.

Rothberg, E. and S. C. Eisenstat (1998). “Node Selection Strategies for Bottom-Up Sparse Matrix Ordering”. In: *SIAM Journal on Matrix Analysis and Applications* 19.3, pp. 682–695.

Rouet, F.-H. (2012). “Memory and performance issues in parallel multifrontal factorizations and triangular solutions with sparse right-hand sides”. PhD Thesis. Institut National Polytechnique de Toulouse.

Rouet, F.-H., X. S. Li, P. Ghysels, and A. Napov (2016). “A Distributed-Memory Package for Dense Hierarchically Semi-Separable Matrix Computations Using

Randomization". In: *ACM Transactions on Mathematical Software* 42.4, 27:1–27:35.

Ruiz, D. (2001). *A scaling algorithm to equilibrate both rows and columns norms in matrices*. Tech. rep. RT/APO/01/4. Also appeared as RAL report RAL-TR-2001-034. ENSEEIHT-IRIT.

Schreiber, R. (1982). "A new implementation of sparse Gaussian elimination". In: *ACM Transactions on Mathematical Software* 8, pp. 256–276.

Sergent, M., D. Goudin, S. Thibault, and O. Aumage (2016). "Controlling the Memory Subscription of Distributed Applications with a Task-Based Runtime System". In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 318–327.

Shin, C., S. Jang, and D. J. Min (2001). "Improved amplitude preservation for prestack depth migration by inverse scattering theory". In: *Geophysical Prospecting* 49, pp. 592–606.

Sid-Lakhdar, W. M. (2014). "Scaling multifrontal methods for the solution of large sparse linear systems on hybrid shared-distributed memory architectures". Ph.D. dissertation. ENS Lyon.

Silva, N. V. da, J. V. Morgan, L. MacGregor, and M. Warner (2012). "A finite element multifrontal method for 3D CSEM modeling in the frequency domain". In: *Geophysics* 77.2, E101–E115.

Sirgue, L., O. I. Barkved, J. Dellinger, J. Etgen, U. Albertin, and J. H. Kommedal (2010). "Full waveform inversion: the next leap forward in imaging at Valhall". In: *First Break* 28, pp. 65–70.

Sirgue, L., J. T. Etgen, and U. Albertin (2008). "3D Frequency Domain Waveform Inversion using Time Domain Finite Difference Methods". In: *Proceedings 70th EAGE, Conference and Exhibition, Roma, Italy*, F022.

Sirgue, L. and R. G. Pratt (2004). "Efficient waveform inversion and imaging : a strategy for selecting temporal frequencies". In: *Geophysics* 69.1, pp. 231–248.

Skeel, R. D. (1979). "Scaling for Numerical Stability in Gaussian Elimination". In: *J. of the ACM* 26, pp. 494–526.

Smith, J. T. (1996). "Conservative modeling of 3-D electromagnetic fields, Part II: Biconjugate gradient solution and an accelerator". In: *Geophysics* 61.5, pp. 1319–1324.

Sourbier, F., A. Haiddar, L. Giraud, H. Ben-Hadj-Ali, S. Operto, and J. Virieux (2011). "Three-dimensional parallel frequency-domain visco-acoustic wave modelling based on a hybrid direct/iterative solver". In: *Geophysical Prospecting* 59.5, pp. 834–856.

Stefani, J., M. Frenkel, N. Bundalo, R. Day, and M. Fehler (2010). "SEAM update: Models for EM and gravity simulations". In: *The Leading Edge* 29.2, pp. 132–135.

Stekl, I. and R. G. Pratt (1998). "Accurate viscoelastic modeling by frequency-domain finite difference using rotated operators". In: *Geophysics* 63, pp. 1779–1794.

Streich, R. (2009). "3D finite-difference frequency-domain modeling of controlled-source electromagnetic data: Direct solution and optimization for high accuracy". In: *Geophysics* 74.5, F95–F105.

Tarantola, A. (1984). "Inversion of seismic reflection data in the acoustic approximation". In: *Geophysics* 49.8, pp. 1259–1266.

Van der Vorst, H. A. (1992). “Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems”. In: *SIAM Journal on scientific and Statistical Computing* 13.2, pp. 631–644.

Vigh, D. and E. W. Starr (2008a). “3D prestack plane-wave, full waveform inversion”. In: *Geophysics* 73, VE135–VE144.

Vigh, D. and E. W. Starr (2008b). “Comparisons for Waveform Inversion, Time domain or Frequency domain?” In: *Extended Abstracts*, pp. 1890–1894.

Virieux, J. and S. Operto (2009). “An overview of full waveform inversion in exploration geophysics”. In: *Geophysics* 74.6, WCC1–WCC26.

Wang, S., M. V. de Hoop, and J. Xia (2011). “On 3D modeling of seismic wave propagation via a structured parallel multifrontal direct Helmholtz solver”. In: *Geophysical Prospecting* 59.5, pp. 857–873.

Wang, S., M. V. de Hoop, J. Xia, and X. S. Li (2012a). “Massively parallel structured multifrontal solver for time-harmonic elastic waves in 3D anisotropic media”. In: *Geophysical Journal International* 191, pp. 346–366.

Wang, S., J. Xia, M. V. de Hoop, and X. S. Li (2012b). “Massively parallel structured direct solver for equations describing time-harmonic qP-polarized waves in TTI media”. In: *Geophysics* 77.3, T69–T82.

Wang, S., X. S. Li, F.-H. Rouet, J. Xia, and M. V. De Hoop (2016). “A Parallel Geometric Multifrontal Solver Using Hierarchically Semiseparable Structure”. In: *ACM Transactions on Mathematical Software* 42.3, 21:1–21:21.

Wang, T. and G. W. Hohmann (1993). “A finite-difference, time-domain solution for three-dimensional electromagnetic modeling”. In: *Geophysics* 58.6, pp. 797–809.

Warner, M., A. Ratcliffe, T. Nangoo, J. Morgan, A. Umpleby, N. Shah, V. Vinje, I. Stekl, L. Guasch, C. Win, G. Conroy, and A. Bertrand (2013). “Anisotropic 3D full-waveform inversion”. In: *Geophysics* 78.2, R59–R80.

Weisbecker, C. (2013). “Improving multifrontal solvers by means of algebraic block low-rank representations”. PhD Thesis. Institut National Polytechnique de Toulouse.

Wilkinson, J. H. (1961). “Error Analysis of Direct Methods of Matrix Inversion”. In: *Journal of the ACM* 8.3, pp. 281–330.

Wilkinson, J. H. (1963). *Rounding Errors in Algebraic Processes*. Englewood Cliffs, New Jersey: Prentice-Hall.

Williams, S., A. Waterman, and D. Patterson (2009). “Roofline: An Insightful Visual Performance Model for Multicore Architectures”. In: *Communications of the ACM* 52.4, pp. 65–76.

Xia, J. (2013a). “Efficient Structured Multifrontal Factorization for General Large Sparse Matrices”. In: *SIAM Journal on Scientific Computing* 35.2, A832–A860.

Xia, J. (2013b). “Randomized Sparse Direct Solvers”. In: *SIAM Journal on Matrix Analysis and Applications* 34.1, pp. 197–227.

Xia, J., S. Chandrasekaran, M. Gu, and X. S. Li (2010). “Fast Algorithms for hierarchically semiseparable matrices”. In: *Numerical Linear Algebra with Applications* 17.6, pp. 953–976.

Yannakakis, M. (1981). “Computing the Minimum Fill-In is NP-Complete”. In: *SIAM Journal on Algebraic and Discrete Methods* 2, pp. 77–79.

Yee, K. (1966). “Numerical solution of initial boundary value problems involving Maxwell’s equations in isotropic media”. In: *IEEE Transactions on antennas and propagation* 14.3, pp. 302–307.

Zach, J., A. Bjørke, T. Støren, and F. Maaø (2008). “3D inversion of marine CSEM data using a fast finite-difference time-domain forward code and approximate Hessian-based optimization”. In: *SEG Technical Program Expanded Abstracts 2008*. Society of Exploration Geophysicists, pp. 614–618.