Scalability of parallel sparse direct solvers: methods, memory and performance

Alfredo Buttari

Institut de Recherche
en Informatique de Toulouse

Centre National de la
Recherche Scientifique

Université
de Toulouse

Scalability of parallel sparse direct solvers: methods, memory and performance

Alfredo Buttari
Chargé de Recherche, CNRS

MÉMOIRE D'HABILITATION À DIRIGER DES RECHERCHES

présenté et soutenu publiquement le 26/09/2018

Rapporteurs		
Michael A. HEROUX	Senior Scientist	Sandia National Lab. (USA)
Xiaoye Sherry LI	Senior Scientist	LBNL (USA)
Yves ROBERT	Professeur des universités	ENS-Lyon, LIP (France)
Examinateurs		
Patrick AMESTOY	Professeur des universités	Toulouse INP, IRIT (France)
Iain DUFF	Senior Scientist	STFC-RAL (UK)
Raymon NAMYST	Professeur des universités	Univ. de Bordeaux, LaBRI(France)
Stéphane OPERTO	Directeur de Recherche	CNRS, Geoazur (France)

Résumé

La solution rapide et précise de systèmes linéaires creux de grande taille est au coeur de nombreuses applications numériques issues d'une gamme très large de domaines incluant la mécanique de structure, la dynamique des fluides, la géophysique, l'imagerie médicale, la chimie. Parmi les techniques les plus couramment utilisées pour la résolution de tels problèmes, les méthodes directes, basées sur la factorisation de la matrice du système, sont généralement appréciées pour leur robustesse numérique et facilité d'utilisation. Cependant, ces méthodes induisent une complexité, en termes d'opérations et de consommation de mémoire, très élevée. Les travaux présentés dans cette thèse se concentrent sur l'amélioration de la scalabilité des méthodes creuses directes, définie comme la capacité de traiter des problèmes de taille de plus en plus importante. Nous introduisons des algorithmes capables d'atteindre un meilleur degré de parallélisme en réduisant les communications et synchronisations afin d'améliorer la scalabilité des performances, à savoir, la capacité de réduire le temps d'exécution lorsque les ressources de calcul disponibles augmentent. Nous nous intéressons à l'utilisation de nouveaux paradigmes et outils de programmation parallèle permettant une implémentation de ces algorithmes efficace et portable sur des supercalculateurs hétérogènes. Nous adressons la scalabilité en mémoire à l'aide de méthodes d'ordonnancement permettant de tirer profit du parallélisme sans augmenter la consommation de mémoire. Finalement, nous démontrons comment il est possible de réduire la complexité des méthodes creuses directes, en termes de nombre d'opérations et taille mémoire, grâce à l'utilisation de techniques d'approximation de rang faible. Les méthodes présentées, dont l'efficacité a été vérifiée sur des problèmes issus d'applications réelles, ont été implantées dans les plateformes logicielles MUMPS et `qr_mumps` distribuées sous licence libre.

Abstract

The fast and accurate solution of large size sparse systems of linear equations is at the heart of numerical applications from a very broad range of domains including structural mechanics, fluid dynamics, geophysics, medical imaging, chemistry. Among the most commonly used techniques, direct methods, based on the factorization of the system matrix, are generally appreciated for their numerical robustness and ease of use. These advantages, however, come at the price of a considerable operations count and memory footprint. The work presented in this thesis is concerned with improving the scalability of sparse direct solvers, intended as the ability to solve problems of larger and larger size. More precisely, our work aims at developing solvers which are scalable in performance, memory consumption and complexity. We address performance scalability, that is the ability to reduce the execution time as more computational resources are available, introducing algorithms that improve parallelism by reducing communications and synchronizations. We discuss the use of novel parallel programming paradigms and tools to achieve their implementation in an efficient and portable way on modern, heterogeneous supercomputers. We present methods that make sparse direct solvers memory-scalable, that is, capable of taking advantage of parallelism without increasing the overall memory footprint. Finally we show how it is possible to use data sparsity to achieve an asymptotic reduction of the cost of such methods. The presented algorithms have been implemented in the freely distributed MUMPS and `qr_mumps` solver packages and their effectiveness assessed on real life problems from academic and industrial applications.

Acknowledgments

I am very grateful to Michael Heroux, Xiaoye Sherry Li and Yves Robert for reporting on the manuscript and to Iain Duff, Raymond Namyst and Stéphane Operto for taking part in my jury; their insightful questions and comments on the manuscript and the presentation have been extremely rewarding and encouraging for the continuation of my career.

During my career I've had the chance to meet and work with very talented researchers. I wish to thank all my previous colleagues from the University or Rome Tor Vergata and from the Innovative Computing Laboratory at UT Knoxville as well as my present colleagues from the APO team of the IRIT laboratory. I am very grateful to the people of the MUMPS team and, especially, to Patrick and Jean-Yves who have been throughout these years, and still are, for me a model and a great source of inspiration. The SOLHAR project has been a fantastic professional experience; I wish to thank all the members and, especially, Abdou and Emmanuel.

Special thanks go to the PhD students I've had the pleasure to supervise: François-Henry, Clément, Florent and Theo. Their work, talent and dedication have been of great contribution to my career.

Thanks to my boys, Roberto and Marco, who did their best to keep me from writing this manuscript but give me, every day, the motivation to improve myself. I owe the deepest gratitude to my wife, Federica, for her constant support and encouragement.

Contents

Chapter 1

Introduction

We are interested in the solution of a linear system of equations

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

where the A matrix

- has m rows and n columns,
- is of large size, i.e., has millions of rows and/or columns,
- is sparse, i.e., most of its coefficients are structurally equal to zero.

In the case where $m = n$ and A is of full rank (which we will assume in the remainder of this document, unless explicitly mentioned) Equation (1.1) admits a solution. If the matrix is overdetermined, i.e., it has more rows than columns, the linear system does not admit a solution in general because the number of constraints is higher than the number of degrees of freedom; in this case we rather seek the x vector which minimizes the 2-norm of the residual $r = Ax - b$:

$$\min_x \|Ax - b\|_2. \tag{1.2}$$

Such a problem is called a *least-squares* problem. If, instead, the matrix is underdetermined, i.e., has more columns than rows, the linear system admits an infinite number of solutions; in this case we seek the solution x whose norm is minimal:

$$\min \|x\|_2, \quad Ax = b. \tag{1.3}$$

Such a problem is called a *least-norm* problem.

One of the most used and well known definitions of a sparse matrix is attributed to James Wilkinson:

> *"A sparse matrix is any matrix with enough zeros that it pays to take advantage of them".*

There are three main ways to take advantage of the fact that a matrix is mostly filled up with zeros. First of all, the memory needed to store the matrix is less than the memory needed for a dense matrix of the same size because the zeros need not be stored explicitly. Second the complexity of most operations on a sparse matrix can be greatly reduced with

respect to the same operation on a dense matrix of the same size because most of the zero coefficients in the original matrix can be skipped during the computation. Finally, parallelism can be much higher than in the dense case because some operations may affect distinct subsets of the matrix nonzeros and can thus be applied concurrently.

Methods for solving Equations (1.1) (1.2) and (1.3) can be roughly grouped into two families: *direct* and *iterative* methods. Direct methods proceed by computing a factorization of the system matrix A such as the LU, LDL^T or QR factorizations; the resulting factors, which are easy and cheap to invert, are then used to compute the solution of the problem. On the other hand, iterative methods start from an initial guess solution, x_0 say,and iteratively improve it until the desired solution accuracy is achieved or when a maximum number of iterations is reached, in which case the method has not converged to a satisfactory solution.

The convergence of an iterative method essentially depends on the numerical properties of the matrix A, more precisely, on its *conditioning* which is defined by the matrix spectrum. Indeed, in some cases the solution of a linear system may require a very high number of iterations or convergence may not be reached at all. For this reason iterative solvers are often used in combination with preconditioning techniques. A preconditioner is usually defined as a matrix $M \approx A^{-1}$ which approximates the inverse of A; this is used to transform the linear system of Equation (1.1) into $MAx = Mb$[1] where the system matrix MA has a more suitable spectrum which leads to a faster convergence. Although general purpose preconditioning methods exist such as Incomplete LU (ILU) or Sparse Approximate Inverse (SPAI), a preconditioner that is effective yet cheap to compute and apply may be hard to find and may, again, rely on the properties of the input problem. As a result, iterative methods can hardly be regarded as "black box" solvers. On the other hand, if the input problem is well conditioned or an appropriate preconditioner is available, iterative methods can achieve converge very quickly and with little memory consumption; moreover, iterative methods generally achieve very good scalability on large scale parallel computing platforms. Despite these good scalability on distributed memory platforms, iterative methods can only achieve a modest fraction of the peak performance of modern processing units. This is because of their poor *arithmetic intensity* (i.e., the ratio between the number of floating-point operations and the number of memory accesses) which prevents the efficient use of cache memories; as a result, iterative methods are *memory bound*, i.e., run at the speed of the memory system which is commonly much slower than the processing unit. Moreover, for the same reason, iterative methods can make very poor use of multicore CPUs or accelerators because in these systems processing units are attached to a shared memory.

In my work I have been mostly interested in the use of direct solvers. These methods are generally regarded as very robust tools for the solution of linear systems as they are capable of computing accurate solutions for a very wide range of problems without the need for the user to have any knowledge of linear systems solvers. Another case where direct methods are often employed is where the same matrix has to be solved with multiple right-hand sides because the matrix factorization, which is the most expensive operation, only has to be computed once and its result reused for multiple, cheap, solve operations[2]. Furthermore, direct methods may have a very broad range of features including the computation of the Schur complement of a matrix, its determinant or its null-space. Finally, because they rely on dense matrix operations which have a very high arithmetic intensity, direct

[1] This is called left preconditioning but other options are possible.

[2] Note that there exist techniques for making iterative methods more effective in the case of multiple right-hand sides such as block methods or subspace recycling.

methods can run very close to the peak performance of modern processing units and can very effectively take advantage of multicore processors and accelerators. Unfortunately, these advantages come at the price of a considerable resource consumption both in terms of memory and time. Moreover, sparse direct solvers usually exhibit a large and irregular workload which is difficult to distribute on parallel computers; this may severely limit the scalability, especially on large scale, distributed memory systems.

For the sake of completeness, it is important to mention that, in recent years, hybrid solvers have known an increasing popularity thanks to their ability to combine the strengths of iterative and direct methods. Among these we can mention *Domain Decomposition* [7, 62, 83] or block-projection [65] methods. The effectiveness of these methods, which heavily rely on direct solvers, is still, to some extent, dependent on the numerical properties of the target problems, though.

This thesis is concerned with the usability of sparse, direct methods for the solution of very large scale sparse linear systems. As such, it deals with the issues related to their scalability, in a broad sense:

1. It addresses the performance scalability of sparse direct solvers on modern, heterogeneous computing systems. These commonly include many CPU cores and , possibly, multiple accelerators such as GPUs. This requires algorithms that can achieve high levels of concurrency in order to feed all the working units. At the same time, it is important to develop data and workload partitioning schemes, as well as scheduling policies, which can make the most out of the computing performance available by taking into account the characteristics of the available processing units. From a strictly technological point of view, the choice of parallel programming models and tools plays an important role for achieving high performance and portability. These issues are the subject of Chapter 3 and are also addressed in Section 5.5.

2. It presents methods that aim at improving the memory scalability of a direct solver in a parallel setting. In order to achieve parallelism in direct solvers, more data is produced and processed in order to feed all the available working units. As a result, when executed in parallel, direct methods consume more memory with respect to a sequential run. In Chapter 4 we discuss techniques that allow the execution of the direct method in parallel within a prescribed memory envelope at the price of little or no loss of concurrency.

3. It investigates methods that improve how the complexity of sparse, direct solvers scales with the size of the problem. These take advantage of a property which is commonly referred to as *data sparsity* through the use of low-rank approximation techniques that allow for discarding redundant or unimportant data; this results in faster computations and lower memory consumption with a loss of accuracy which can be conveniently controlled through a single parameter. This subject is studied in Chapter 5.

Chapter 2 introduces the basic concepts that are necessary for a good understanding of the ensuing chapters; it also attempts at providing a concise introduction to dense and sparse matrix factorization methods and can serve a starting point for students or researchers who are willing to learn the basis of solving linear systems by means of direct methods. Chapter 6 draws some conclusions from the this work and presents some future research developments.

This manuscript describes the research work that I have accomplished since receiving my PhD degree in 2006.

In 2006-2007 I joined the ICL laboratory at the University of Tennessee Knoxville as a post-doc. During that period I worked on the development of "tiled" algorithms [C13, B3, C14, J15, C15, J16, J18, J19] that improve the scalability and performance of dense matrix factorizations by breaking down into multiple steps the operations that are heavy on communications and, therefore, difficult to parallelize efficiently. During this period I also worked on the use of mixed-precision iterative refinement techniques [J7, B4, J12, J13, J18, C18] for the solution of linear systems of equations; under the assumption that the input problem is not too badly conditioned. These methods perform the most expensive operations (e.g., the factorization) in lower-precision arithmetic and then selectively use higher precision to recover the desired accuracy by means of iterative refinement. As a result, the solution can be achieved with high accuracy at the speed of lower-precision operations. These techniques can be equally applied to dense and sparse linear systems and to direct and iterative methods. When applied to dense, direct solvers this performance benefit comes at the price of an increased memory consumption because, although the factors are stored in single precision, a double precision copy of the original matrix must be kept in memory to perform the iterative refinement. On the other hand when applied to sparse, direct solvers they can also reduce the storage because the size of this extra copy is negligible compared to the size of the factors which are stored in single precision. For the sake of conciseness this work is not discussed in this manuscript.

Later, in 2008, I joined the INRIA Graal (currently ROMA) team at the LIP laboratory of Lyon. Here I got acquainted with sparse direct solvers and worked on the development of a parallel symbolic analysis which was integrated into the MUMPS solver.

At the end of 2008 I joined the APO team of the IRIT laboratory of Toulouse as a CNRS Chargé de Recherche (full time researcher). Since then I have devoted most of my research activity to sparse direct solvers, parallel computing and computational linear algebra. This thesis is essentially a résumé of the work I achieved during this period.

Most of this work was achieved in the context of four PhDs that I co-supervised: François-Henry Rouet (INPT-IRIT, 2009-2012) on the memory scalability of sparse direct solvers, Clément Weisbecker (INPT-IRIT, 2010-2013) and Theo Mary (UPS-IRIT, 2014-2017) on low-rank approximation techniques and Florent Lopez (UPS-IRIT, 2012-2015) on parallelism for multicore and heterogeneous systems.

The development of numerical linear algebra software has always been a central part of my research activity. This is mostly due to the fact that the need for fast, scalable and, at the same time, reliable solvers is motivated by the ever evolving and increasing requirements of large-scale numerical simulation applications. Through the development of production quality software packages it is possible to validate the effectiveness of methods and algorithms in working conditions that are hard to model theoretically and to assess their practical interest. Moreover, the free distribution of these packages allows us to reach out for collaborations with experts of large scale simulation applications, opening up opportunities for developing more research. I have contributed to the development of the MUMPS [13] sparse direct solver, and I am the principal developer of the `qr_mumps` [J10] one. I have also contributed, to a minor extent, to the PSBLAS [J17] sparse, iterative solver.

All the achievements of my research activity (including those that are not presented in this manuscript) result from close collaborations and exchanges with the IRIT-APO team as well as with other, external, academic and industrial partners: the ROMA, HiePACS and STORM teams of Inria, Dr Sherry Li's team at the Lawrence Berkeley National Laboratory, the ICL laboratory at UTK, Prof Salvatore Filippone at Cranfield University (formerly at the University of Rome Tor Vergata), the Seiscope consortium, LSTC, EDF and EMGS.

My work received financial support from the following projects and contracts: ANR SOLSTICE (ANR-06-CIS-010, 2007-2010) and SOLHAR (ANR-13-MONU-0007, 2013-2018), French-Israeli “Multicomputing” project (2008-2010), SCEBF (2014-2015), PhD (2016-2017) and SRFCP (2018-2019) TTIL projects, the contracts with Samtech (2008-2010), EDF (2010-2013) and EMGS (2014) and the MUMPS consortium (https://mumps-consortium.org/).

Chapter 2

Background

2.1 Dense matrix factorizations

2.1.1 Unsymmetric matrices: the Gaussian Elimination

One very well known method for computing the solution of Equation (1.1) with A being square (i.e., $m = n$) and non-symmetric is Gaussian Elimination. Through a number of convenient linear combinations of the rows of A and b, this method reduces A to an upper-triangular matrix U such that the solution x of the linear system can be easily computed through a backward substitution. The Gaussian elimination is equivalent to computing $n-1$ linear transformations L_i^{-1} such that

$$L_{n-1}^{-1}...L_2^{-1}L_1^{-1}A = U.$$

Each L_i^{-1} is unit-diagonal and lower-triangular with all the coefficients below the diagonal equal to zero except for the i-th column and eliminates variable x_i from the equations $i+1...m$. Setting $L_1L_2...L_{n-1} = L$ this process yields $A = LU$ which is commonly known under the name of LU factorization. With few algebraic manipulations and a few "Strokes of luck" [154] it is possible to show that L is also unit-diagonal and lower-triangular. The LU factorization can be computed using Algorithm 2.1 where with $a_{i,j}^{(k)}$ we denote the coefficients of the so called *trailing submatrix* $A^{(k)}$ at step k; this algorithm costs $2/3n^3$ floating-point operations and may work in-place, i.e., the L and U factors may be stored in the same memory that originally holds the A matrix (this is possible because the diagonal of L is unitary and need not be stored).

Note that the loops in Algorithm 2.1 can be nested differently which gives rise to different variants on the LU factorization. The one in Algorithm 2.1 is the so-called *right-looking* variant because at step k of the factorization column k is reduced through the transformation L_k^{-1} which is also immediately applied to the trailing submatrix at the right of it. The dual of this approach is the *left-looking* variant: at step k, all the $L_{k-1}^{-1}...L_1^{-1}$ transformations are applied to the k-th column which is then reduced by means of the L_k^{-1} transformation. Other variants exist such as the Crout one [63]. All these variants are numerically equivalent but differ in the data access pattern which may result in better use of the memory hierarchy or better potential for parallelism and vectorization.

Once the L and U factors are computed, the solution x of the linear system can be computed through the following two operations

$$\begin{cases} Ly &= b \\ Ux &= y \end{cases}$$

Algorithm 2.1 *LU* factorization.

1: $a_{ij}^{(0)} = a_{i,j}$ for all $i, j = 1, ..., n$
2: **for** $k = 1, ..., n$ **do**
3: **for** $j = k, ..., n$ **do**
4: $u_{k,j} = a_{k,j}^{(k-1)}$
5: **end for**
6: **for** $i = k + 1, ..., m$ **do**
7: $l_{ik} = a_{i,k}^{(k-1)} / u_{k,k}$
8: **for** $j = k + 1, ..., n$ **do**
9: $a_{i,j}^{(k)} = a_{i,j}^{(k-1)} - l_{i,k} u_{k,j}$
10: **end for**
11: **end for**
12: **end for**

which are called, respectively, *forward elimination* and *backward substitution*. These can be easily and cheaply (n^2 flops each) computed with a doubly-nested loop.

2.1.1.1 Accuracy of the Gaussian Elimination and pivoting

When working in finite-precision arithmetic (as it happens on a computer) we may be concerned about the accuracy of the computed solution $\hat{x}$ of Equation (1.1) because of the roundoff errors that occur during the computation. Classic *backward error analysis* [160] states that the *forward error* $\|x - \hat{x}\|/\|x\|$ can be bounded by the product of *the condition number* and *the backward error*. The condition number measures how the computed solution is sensitive to perturbations in the input data and does not depend on the used method but solely on the input data; for the solution of Equation (1.1) the condition number is $\kappa(A) = \|A^{-1}\| \cdot \|A\|$ (see, for example, the book by Demmel [57] for further details). The backward error, instead, is a measure of the smallest perturbation of the input data such that the computed solution is the exact solution of the perturbed problem. If this quantity is small, the method is deemed *backward stable*. For the solution of Equation (1.1), the backward error, is thus defined by the following quantity

$$\beta_{Ax=b} = \min \quad \{ \quad \epsilon : \ (A + \delta A)\hat{x} = b + \delta b, \\ \|\delta A\| \leq \epsilon \|A\|, \\ \|\delta b\| \leq \epsilon \|b\| \}.$$

A well known result by Rigal and Gaches [138] proves that

$$\beta_{Ax=b} = \frac{\|r\|}{\|A\|\|\hat{x}\| + \|b\|}$$

where $r = b - A\hat{x}$ is the so-called *residual*.

If Gaussian Elimination is used, and assuming $\delta b = 0$, it is possible to prove that [57]

$$\|\delta A\|_\infty \leq 3nu \|L\|_\infty \cdot \|U\|_\infty,$$

where u is the unit roundoff, which implies that the Gaussian Elimination is backward stable if $3nu\|L\|_\infty \cdot \|U\|_\infty = O(u)\|A\|_\infty$. This result basically states that we have to keep the coefficients in L and U low in order to limit the backward error.

Let's take this example extracted from the book of Golub and Van Loan [86]

$$Ax = \begin{bmatrix} 0.001 & 1.00 \\ 1.00 & 2.00 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1.00 \\ 3.00 \end{bmatrix} = b$$

to be solved using Gaussian Elimination on a computer using a three decimal digits floating point representation. The LU factorization yields

$$L = \begin{bmatrix} 1.00 & \\ 1000 & 1.00 \end{bmatrix} \qquad U = \begin{bmatrix} 0.001 & 1.00 \\ & -1000 \end{bmatrix}$$

because $2.00 - 1000 = -1000$ in the three digits representation. As a result

$$L \cdot U = \begin{bmatrix} 0.001 & 1.00 \\ 1.00 & 0.00 \end{bmatrix} \neq A$$

and the computed solution is

$$\tilde{x} = \begin{bmatrix} 0.00 \\ 1.00 \end{bmatrix} \neq \text{ true solution} = \begin{bmatrix} 1.002... \\ 0.998... \end{bmatrix}$$

Note that the condition number $\kappa(A) = 5.84$ and thus the problem is well-conditioned. This poor accuracy can be explained observing that the coefficients of L and U have grown considerably with respect to those of A – a phenomenon that can be measured by the so-called *growth factor*

$$\rho(A) = \frac{|\max_{i,j,k} a_{i,j}^{(k)}|}{|\max_{i,j} a_{i,j}|}.$$

One way to prevent this excessive growth of the coefficients is a technique called *pivoting*. Various flavors of pivoting exist in the literature but they all share the idea of using permutations in the course of the factorization such that at step k the pivotal coefficient $a_{k,k}^{(k-1)}$ is *large enough* with respect to the coefficients in the trailing submatrix. The most commonly used pivoting technique is called *partial pivoting* and consists in scanning the k-th column $a_{i,k}^{(k-1)}$ for $i = k, ..., m$ looking for the coefficient with the largest absolute value and then swapping the corresponding row with row k. Because of the division on line 7 of Algorithm 2.1, partial pivoting ensures that all the coefficients of L are lower than or equal to one. In this case the bound on the backward error becomes [57]

$$\|\delta A\|_\infty \leq 3n^3 u \rho(A) \|A\|_\infty.$$

This bound can be overly pessimistic, not only because of the n^3 term but also because it is not possible to compute a tight bound for $\rho(A)$; it can actually be shown that for some classes of problems the growth factor can be as big as 2^{n-1}. In practice, however, it is very rare to observe large growth factors and the Gaussian elimination with partial pivoting can be used with confidence.

As a result, the Gaussian Elimination with partial pivoting is described by a sequence of linear combinations interleaved with permutations

$$L_{n-1}^{-1} P_{n-1} ... L_2^{-1} P_2 L_1^{-1} P_1 A = U. \tag{2.1}$$

where each P_i only permutes two rows. Through simple algebraic manipulations this can be rewritten as

$$\tilde{L}_{n-1}^{-1} ... \tilde{L}_2^{-1} L_1^{-1} P_{n-1} ... P_2 P_1 A = U. \tag{2.2}$$

where

$$\tilde{L}_i = P_{n-1}...P_{i+1}L_iP_{i+1}^T...P_{n-1}^T.$$

We can observe that the only difference between L_i and $\tilde{L}_i$ is in the fact that the subdiagonal nonzeros in column i are permuted and thus it is still possible to write $\tilde{L}_{n-1}^{-1}...\tilde{L}_2^{-1}L_1^{-1} = L^{-1}$ which yields

$$PA = LU. \tag{2.3}$$

Taking pivoting into account, the LU factorization can be rewritten as in Algorithm 2.2. Note that the P matrix is stored implicitly in the p array of integers, whereas L and U are stored in the memory that originally contained matrix A. This algorithm performs exactly the same number of operations as Algorithm 2.1 because permutations only involve memory copies; nonetheless the use of pivoting degrades performance and limits parallelism. Algorithm 2.2 is implemented in the `_getrf2`[1] of the LAPACK [20] library.

Algorithm 2.2 *LU* factorization with partial pivoting.

1: **for** $k = 1, ..., n$ **do**
2: $p(k) = \underset{i=k,...,m}{\text{argmax}}(|a_{i,k}|)$
3: swap rows $A_{k,1:n}$ and $A_{p(k),1:n}$
4: $A_{k+1:m,k} = A_{k+1:m,k}/a_{k,k}$
5: $A_{k+1:m,k+1:n} = A_{k+1:m,k+1:n} - A_{k+1:m,k} * A_{k,k+1:n}$
6: **end for**

Gaussian Elimination with partial pivoting applied to the matrix of the example above yields

$$P = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \qquad L = \begin{bmatrix} 1.00 & \\ 0.01 & 1.00 \end{bmatrix} \qquad U = \begin{bmatrix} 1.00 & 2.00 \\ & 1.00 \end{bmatrix}$$

resulting in

$$P^T \cdot L \cdot U = \begin{bmatrix} 0.01 & 1.02 \\ 1.00 & 2.00 \end{bmatrix} \qquad x = \begin{bmatrix} 1.00 \\ 0.996 \end{bmatrix}$$

which is a much more accurate solution with respect to the case where pivoting is not used.

2.1.2 Symmetric matrices: the Cholesky and LDL^T factorizations

A matrix is *symmetric positive definite* if and only if $A = A^T$ and $x^TAx > 0$ for all $x \neq 0$. For such matrices the following proposition can be proved:

Proposition 2.1— *A is symmetric positive definite if and only if there is a unique lower triangular nonsingular matrix L with positive diagonal entries such tat* $A = LL^T$.

[1] The underscore has to be replaced be `d`, `s` for computations in real, double and single precision, respectively, and with `z`, `c` for computations in complex, double and single precision, respectively.

We refer the reader to any classic linear algebra textbook (like the one by Demmel [57] or Golub and Van Loan [86]) for a proof. The factorization $A = LL^T$ is called the *Cholesky factorization* and can be computed as in Algorithm 2.3 implemented in the LAPACK `_potf2` routine. This algorithm

- costs $n^3/3$ floating point operations which is half as much as the LU factorization;
- does not require pivoting because it is backward stable due to the positive definiteness property;
- represents the cheapest way to verify whether A is positive definite because otherwise it would break down trying to compute the square root of a negative coefficient or by making a division by zero.

Algorithm 2.3 Cholesky factorization.

```
1: for k = 1, ..., n do
2:     a_{i,i} = sqrt(a_{i,i})
3:     for i = k + 1, ..., n do
4:         a_{i,k} = a_{i,k} / a_{k,k}
5:         for j = k + 1, ..., i do
6:             a_{i,j} = a_{i,j} − a_{i,k} a_{j,k}
7:         end for
8:     end for
9: end for
```

When the matrix A is indefinite, the Cholesky factorization cannot be used because it leads to an unstable computation due to an excessive element growth. Obviously, the partial pivoting technique described above can be used to stabilize (at least in practice) the method but this would destroy the symmetry and thus preclude the possibility to achieve the factorization in $n^3/3$ flops. On the other hand, using symmetric permutations such that $PAP^T = LL^T$ does not always lead to a stable computation (think of the case where all the diagonal elements of A are small). For such matrices, the most commonly used technique consists in using a factorization of the form $PAP^T = LDL^T$ where L is lower-triangular and unit-diagonal and D is a block-diagonal matrix with blocks of size 1×1 or 2×2. One pivoting technique to compute such a factorization was proposed by Bunch and Kaufman [44] and, at each step of the factorization, only involves searching in two columns of the trailing submatrix; this method can be proved to be as stable as the LU factorization with complete pivoting. The LDL^T factorization with the Bunch-Kaufman pivoting is implemented in the LAPACK `_sytf2` routine.

2.1.3 QR factorization

This method decomposes the input matrix $A \in \mathbb{R}^{m \times n}$, assumed to be of full rank, into the product of a square, orthogonal matrix $Q \in \mathbb{R}^{m \times m}$ and an upper triangular matrix $R \in \mathbb{R}^{n \times n}$.

Theorem 2.1 Björck [36, Theorem 1.3.1].— *Let $A \in \mathbb{R}^{m \times n}$, $m \geq n$. Then there is an orthogonal matrix $Q \in \mathbb{R}^{m \times m}$ such that*

$$A = Q \begin{bmatrix} R \\ 0 \end{bmatrix}, \tag{2.4}$$

where R is upper triangular with nonnegative diagonal elements. The decomposition (2.4) is called the QR decomposition of A, and the matrix R will be called the R-factor of A.

The columns of the Q matrix can be split in two groups

$$Q = [Q_1 Q_2] \tag{2.5}$$

where $Q_1 \in \mathbb{R}^{m\times n}$ is an orthogonal basis for the range of A, $\mathcal{R}(A)$ and $Q_2 \in \mathbb{R}^{m\times(m-n)}$ is an orthogonal basis for the kernel of A^T, $\mathcal{N}(A^T)$.

The QR decomposition can be used to solve square linear system of equations

$$Ax = b, \quad \text{with } A \in \mathbb{R}^{n\times n}, \tag{2.6}$$

as the solution x can be computed through the following three steps

$$\begin{cases} A = QR \\ z = Q^T b \\ x = R^{-1} z \end{cases} \tag{2.7}$$

where, first, the QR decomposition is computed (e.g., using one of the methods described below), an intermediate result is computed trough a simple matrix-vector product and, finally, solution x is computed through a triangular system solve. As we will explain the next two sections, the QR decomposition is commonly unattractive in practice for solving square systems mostly due to its excessive cost when compared to other available techniques, despite its desirable numerical properties.

The QR decomposition is instead much more commonly used for solving linear systems where A is overdetemined, i.e. where there are more equations than unknowns. In such cases, unless the right-hand side b is in the range of A, the system admits no solution; it is possible, though, to compute a vector x such that Ax is as close as possible to b, or, equivalently, such that the residual $\|Ax - b\|_2$ is minimized as in Equation (1.2). Such a problem is called a *least-squares* problem and commonly arises in a large variety of applications such as statistics, photogrammetry, geodetics and signal processing. One typical example is given by linear regression where a linear model, say $f(x, y) = \alpha+\beta x+\gamma y$ has to be fit to a number of observations subject to errors (f_i, x_i, y_i), $i = 1, ..., m$. This leads to the overdetermined system

$$\begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ \vdots & \vdots & \vdots \\ 1 & x_m & y_m \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_m \end{bmatrix}.$$

Assuming the QR decomposition of A in Equation (2.4) has been computed and

$$Q^T b = \begin{bmatrix} Q_1^T \\ Q_2^T \end{bmatrix} b = \begin{bmatrix} c \\ d \end{bmatrix}$$

we have

$$\|Ax - b\|_2^2 = \|Q^T Ax - Q^T b\|_2^2 = \|Rx - c\|_2^2 + \|d\|_2^2. \tag{2.8}$$

This quantity is minimized if $Rx = c$ where x can be found with a simple triangular system solve. This is equivalent to saying that $Ax = Q_1 Q_1^T b$ and thus solving Equation (1.2) amounts to finding the vector x such that Ax is the orthogonal projection of b over the range of A, as shown in Figure 2.1. Also note that $r = Q_2 Q_2^T b$ and thus r is the projection of b on the null space of A^T, $\mathcal{N}(A)$.

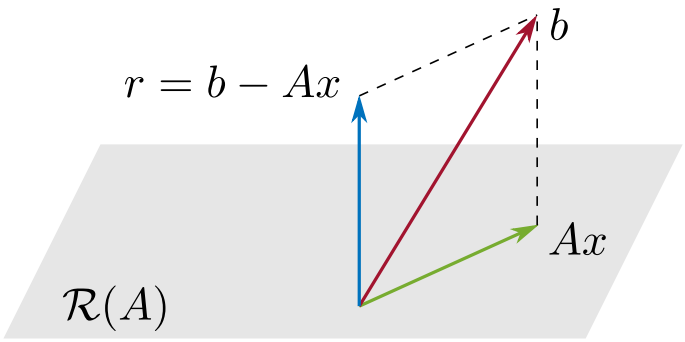


Figure 2.1: Solution of a least-squares problem.

Another commonly used technique for the solution of the least-squares problem is the *Normal Equations* method. Because the residual r is in $\mathcal{N}(A^T)$

$$A^T(Ax - b) = 0$$

and, thus, the solution x to Equation (1.2) can be found solving the linear system $A^T Ax = A^T b$. Because $A^T A$ is Symmetric Positive Definite (assuming A has full rank), this can be achieved through the Cholesky factorization. Nonetheless, the method based on the QR factorization is often preferred because the conditioning of $A^T A$ is equal to the square of the conditioning of A, which may lead to excessive error propagation.

The QR factorization is also commonly used to solve underdetermined systems, i.e., with more unknowns than equations, which admit infinite solutions. In such cases the desired solution is the one with minimum 2-norm:

$$min\|x\|_2, \quad Ax = b. \tag{2.9}$$

The solution of this problem can be achieved by computing the QR factorization of A^T

$$[Q_1 Q_2] \begin{bmatrix} R \\ 0 \end{bmatrix} = A^T$$

where $Q_1 \in \mathbb{R}^{n \times m}$ and $Q_2 \in \mathbb{R}^{n \times (n-m)}$. Then

$$Ax = R^T Q^T x = [R^T 0] \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = b$$

and the minimum 2-norm solution follows by setting $z_2 = 0$. Note that Q_2 is an orthogonal basis for $\mathcal{N}(A)$ and, thus, the minimum 2-norm solution is computed by removing for any admissible solution $\tilde{x}$ all of its components in $\mathcal{N}(A)$.

2.1.3.1 Householder QR decomposition

The QR decomposition of a matrix can be computed in different ways; the use of Givens Rotations [84], Householder reflections [102] or the Gram-Schmidt orthogonalization [142] are among the most commonly used and best known ones. We will not cover here the use of Givens Rotations, Gram-Schmidt orthogonalization and their variants and refer, instead, the reader to classic linear algebra textbooks such as Golub et al. [86] or Björck [36] for an exhaustive discussion of such methods. We focus, instead, on the QR factorization based on Householder reflections which has become the most commonly used method especially because of the availability of algorithms capable of achieving very high performance on processors equipped with memory hierarchies.

For a given a vector u, a *Householder Reflection* is defined as

$$H = I - 2\frac{uu^T}{u^T u} \tag{2.10}$$

and u is commonly referred to as *Householder vector*. It is easy to verify that H is symmetric and orthogonal. Because $P_u = \frac{uu^T}{u^T u}$ is a projector over the space of u, Hx can be regarded as the reflection of a vector x on the hyperplane that has normal vector u. This is depicted in Figure 2.2 (*left*).

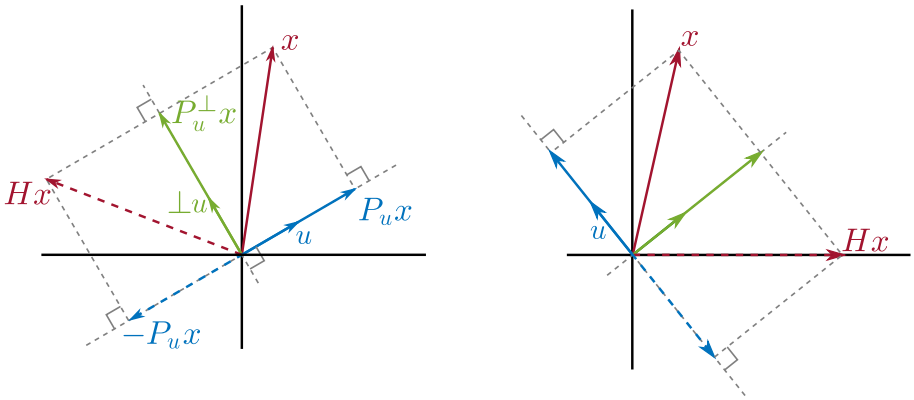


Figure 2.2: Householder reflection

The Householder reflection can be defined in such a way that Hx has all zero coefficients except the first, which means to say that $Hx = \pm\|x\|_2 e_1$ where e_1 is the first unit vector. This can be achieved if Hx is the reflection of x on the hyperplane that bisects the angle between x and $\pm\|x\|_2 e_1$, as illustrated in Figure 2.2 (*right*). This hyperplane is orthogonal to the difference of these two vectors $x \mp \|x\|_2 e_1$ which can thus be used to construct the vector

$$u = x \mp \|x\|_2 e_1. \tag{2.11}$$

Note that, if, for example, x is close to a multiple of e_1, then $\|x\|_2 \approx x(1)$ which may lead to a dangerous cancellation in Equation (2.11); to avoid this problem u is commonly chosen as

$$u = x + sign(x_1)\|x\|_2 e_1. \tag{2.12}$$

In practice, it is very convenient to scale u in such a way that its first coefficient is equal to 1 (more on this will be said later). Assuming $v = u/u_1$, through some simple manipulations, the Householder transformation H is defined as

$$H = I - \tau v v^T, \quad \text{where} \quad \tau = \frac{(sign(x_1)x_1 + \|x\|_2)}{\|x\|_2}. \tag{2.13}$$

Note that the matrix H is never explicitly built neither to store, nor to apply the Householder transformation; the storage is done implicitly by means of v and τ and the transformation can be applied to an entire matrix $A \in \mathbb{R}^{m \times n}$ in $4mn$ flops like this

$$HA = (I - \tau v v^T)A = A - \tau v(v^T A) \tag{2.14}$$

The Householder reflection can be computed and applied as described above using, respectively, the `_larfg` and `_larf` routines in the LAPACK[20] library.

A sequence of Householder transformations can be used to zero-out all the coefficients below the diagonal of a dense matrix to compute its QR factorization:

$$H_n H_{n-1} ... H_2 H_1 A = R, \quad \text{where} \quad H_n H_{n-1} ... H_2 H_1 = Q^T.$$

Each transformation H_k annihilates all the coefficients below the diagonal of column k and modifies all the coefficients in the trailing submatrix $A_{k:m,k+1:n}$. The total cost of this algorithm is $2n^2(m - n/3)$. The Q matrix is implicitly represented by means of the v_k vectors and the τ_k coefficients. One extra array has to be allocated to store the τ_k coefficients, whereas the v_k vectors can be stored inside matrix A in the same memory as the zeroed-out coefficients; this is possible because the v_k have been scaled as described above and thus the 1 coefficients along the diagonal must not be explicitly stored. The LAPACK `_geqr2` routine implements this method which is reported in Algorithm 2.4. In this algorithm, line 2 computes the $r_{k,k}$ coefficient equal to $\|A_{k:m,k}\|_2$ and the v reflector and the τ coefficient as in Equations (2.12)(2.13); these are stored, respectively, in $a_{k,k}$, $A_{k+1:m,k}$ (the first coefficient of v being equal to one is not stored explicitly) and t_k. This transformation is then applied to the trailing submatrix through the operation on line 4 where the $a_{k,k}$ coefficient is assumed to be one.

Algorithm 2.4 Householder QR factorization.

1: **for** $k = 1, ..., n$ **do**
2: $\quad A_{k:m,k}, t_k = \text{house}(A_{k:m,k})$
3: $\quad$ **for** $j = k + 1, ..., n$ **do**
4: $\quad\quad A_{k:m,k+1:n} = (I - t_k A_{k:m,k} A_{k:m,k}^T) A_{k:m,k+1:n}$
5: $\quad$ **end for**
6: **end for**

The following results define the stability of the QR factorization.

Theorem 2.2 Björck [36, Remark 2.4.2].— *Let $\bar{R}$ denote the computed R. It can be shown that there exists an exactly orthogonal matrix $\bar{Q}$ (not the computed Q) such that*

$$A + E = \bar{Q}\bar{R}, \quad \|E\|_F \leq c_1 u \|A\|_F,$$

where the error constant $c_1 = c_1(m, n)$ is a polynomial in m and n, $\|\cdot\|_F$ denotes the Frobenius norm and u the unit roundoff.

In other words, the Householder QR factorization is normwise backward stable.

The use of a QR factorization by means of a sequence of orthogonal transformations to solve least-squares problems was introduced by Golub [85]; this method is also proven to be backward stable:

Theorem 2.3 Björck [36, Remark 2.4.8].— *Golub's method for solving the standard least squares problem is normwise backward stable. The computed solution $\hat{x}$ can be shown to be the exact solution of a slightly perturbed least squares problem*

$$\min_x \|(A + \delta A)x - (b + \delta b)\|_2,$$

where the perturbations satisfy the bounds

$$\|\delta A\|_2 \leq cun^{1/2}\|A\|_2, \quad \|\delta b\|_2 \leq cu\|b\|_2$$

and $c = (6m - 3n + 41)n$.

Despite these very favorable numerical properties, the QR factorization is rarely preferred to the Gaussian Elimination (or LU factorization) with Partial Pivoting (GEPP) for the solution of square systems because its cost is twice the cost of GEPP and because partial pivoting is considered stable in most practical cases.

2.1.4 Blocked variants

The above discussed factorization algorithms, commonly referred to as *point factorizations*, are actually never directly used in practice because they can only achieve a modest fraction of the peak performance of a modern processor. This is due to the fact that most of the computations, which are done in the application of elementary transformations to the trailing submatrix as line 5 of Algorithm 2.2, are based on Level-2 (i.e., matrix-vector) *Basic Linear Algebra Subroutines* (BLAS) operations and thus limited by the speed of the memory rather than the speed of the processor. In order to overcome this limitation and considerably improve the performance of dense matrix factorizations on modern computers equipped with memory hierarchies, *blocking* techniques are commonly employed: these consist in accumulating multiple elementary transformations and applying them at once by means of Level-3 BLAS operations. The resulting factorization algorithms are commonly referred to as *blocked factorizations*.

2.1.4.1 Blocked LU factorization

Assume a square matrix A is partitioned as such

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}$$

where $A_{1,1}$ and $A_{2,2}$ are square submatrices of size $b \ll n$ and $n-b$, respectively. Executing the first b iterations of the outer loop in Algorithm 2.2 leads us to the following situation

$$P_{1..b}A = \begin{bmatrix} L_{1,1} & 0 \\ \hat{L}_{2,1} & I \end{bmatrix} \begin{bmatrix} U_{1,1} & U_{1,2} \\ 0 & \tilde{A}_{2,2} \end{bmatrix} \tag{2.15}$$

where $P_{1..b} = P_b \cdots P_1$ and $\tilde{A}_{2,2}$ is the so-called *Schur complement*. Computing the remaining $n - b$ iterations amounts to computing $P_{b+1..n-1}\tilde{A}_{2,2} = L_{2,2}U_{2,2}$ which yields the complete LU factorization

$$PA = \begin{bmatrix} L_{1,1} & 0 \\ L_{2,1} & L_{2,2} \end{bmatrix} \begin{bmatrix} U_{1,1} & U_{1,2} \\ 0 & U_{2,2} \end{bmatrix}$$

where $P = P_{b+1..n-1}P_{1..b}$ and $L_{2,1} = P_{b+1..n-1}\hat{L}_{2,1}$.

Note that we could get to the same situation as in Equation (2.15) trough the following steps

$$P_{1..b} \begin{bmatrix} A_{1,1} \\ A_{2,1} \end{bmatrix} = \begin{bmatrix} L_{1,1} \\ \hat{L}_{2,1} \end{bmatrix} U_{1,1} \tag{2.16a}$$

$$\begin{bmatrix} \hat{A}_{1,2} \\ \hat{A}_{2,2} \end{bmatrix} = P_{1..b} \begin{bmatrix} A_{1,2} \\ A_{2,2} \end{bmatrix} \tag{2.16b}$$

$$U_{1,2} = L_{1,1}^{-1}\hat{A}_{1,2} \tag{2.16c}$$

$$\tilde{A}_{2,2} = \hat{A}_{2,2} - \hat{L}_{2,1}U_{1,2} \tag{2.16d}$$

Step (2.16a) is referred to as the *panel factorization* or *panel reduction* and can be achieved running Algorithm 2.2 on $[A_{1,1}^T A_{2,1}^T]^T$, step (2.16b) consists in simple row permutations, step (2.16c) is a triangular system solve with multiple right-hand sides and, finally, step (2.16d) is a rank$-b$ update or, simply a matrix-matrix product; these last three steps are commonly called *trailing submatrix update*.

The rest of the factorization can be achieved by applying recursively the same partitioning and steps on $\tilde{A}_{2,2}$. The main advantage of this approach with respect to the point LU factorization of Algorithm 2.2 is that, if $b \ll n$, the vast majority of the floating operations are done in steps (2.16c)(2.16d) which are Level-3 BLAS matrix-matrix operations; these can run very close to the peak performance of a processor thanks to the favorable ratio between computations and memory accesses. The blocked LU factorization is implemented in the `_getrf` LAPACK routine.

This blocking technique can be applied to the Cholesky factorization in a straightforward way and, although more difficult, it can also be applied to the LDL^T factorization; the Cholesky and LDL^T blocked factorizations are implemented, respectively, in the `_potrf` and `_sytrf` LAPACK routines.

2.1.4.2 Blocked QR factorization

Blocking is more complex for the Householder QR factorization because it is not easy to accumulate elementary transformations. Schreiber et al. [143] proposed a way of accumulating multiple Householder transformations and applying them at once by means of Level-3 BLAS operations.

Theorem 2.4 Compact WY representation (Adapted form Schreiber et al. [143]).— *Let $Q = H_1 ... H_{k-1}H_k$, with $H_i \in \mathbb{R}^{m\times m}$ an Householder transformation defined as in Equation (2.13) and $k \leq m$. Then, there exist an upper triangular matrix $T \in \mathbb{R}^{k\times k}$ and a matrix $V \in \mathbb{R}^{m\times k}$ such that*

$$Q = I + VTV^T.$$

Proof. The proof is by induction on k. The case $k = 0$ is straightforward. Now assume that a matrix $Q_k = H_1 ... H_{k-1} H_k$ has a compact WY representation $Q_k = I + V_k T_k V_k^T$ and consider

$$Q_{k+1} = Q_k H_{k+1} = Q_k (I - \tau_{k+1} v_{k+1} v_{k+1}^T).$$

Then a compact WY representation for Q_{k+1} is given by $Q_{k+1} = I + V_{k+1} T_{k+1} V_{k+1}^T$ where

$$T_{k+1} = \begin{bmatrix} T_k & -\tau_{k+1} T_k V_k^T v_{k+1} \\ 0 & -\tau_{k+1} \end{bmatrix}, \quad V_{k+1} = \begin{bmatrix} V_k & v_{k+1} \end{bmatrix} \tag{2.17}$$

which concludes the proof. □

Using this technique, matrix A can be logically split into $\lceil n/b \rceil$ *panels* (block-columns) of size b and the QR factorization achieved in the same number of steps where, at step k, panel k is factorized using the `_geqr2` routine, the corresponding T matrix is built, as in Equation (2.17) using the `_larft` routine and then the set of b transformations is applied at once to the trailing submatrix through the `_larfb` routine. This last operation/routine is responsible for most of the flops in the QR factorization: because it is based on matrix-matrix operations it can achieve a considerable fraction of the processor's peak performance. This method is implemented in the LAPACK `_geqrf` routine which uses an implicitly defined blocking size b and discards the T matrices computed for each panel. More recently, the `_geqrt` routine has been introduced in LAPACK which employs the same algorithm but takes the block size b as an additional argument and returns the computed T matrices.

2.1.4.3 QR factorization with pivoting

In case the A matrix is rank deficient, the R factor resulting from the QR factorization will have zeros on the diagonal on those columns that are linearly dependent on previously eliminated ones. This result, however, cannot be used for solving a linear system because the triangular system solve involving R will fail with divisions by zero. In such cases, a QR factorization with column pivoting [45] can be used: at each step k, the column whose norm is the highest is brought in position k with a column swap. This technique yields the factorization

$$Q^T A \Pi = \begin{matrix} \begin{bmatrix} R_{1,1} & R_{1,2} \\ 0 & R_{2,2} \end{bmatrix} \\ \begin{matrix} r & n-r \end{matrix} \end{matrix} \begin{matrix} r \\ m-r \end{matrix}$$

In exact arithmetic $R_{2,2} = 0$ and r is the rank of A; in finite arithmetic $R_{2,2}$ is considered to be zero if it is small enough, for example, if its 2-norm or it left-topmost coefficient $(R_{2,2})_{1,1}$ are smaller than a prescribed threshold. Assuming x is split in two parts, x_1 containing the first r elements and x_2 containing the remaining $n-r$ ones, the solution of the rank deficient least squares problem can be computed as explained for Equation (2.8) by setting $x_2 = 0$.

Computing the column-pivoted QR factorization requires the norms of all the columns in the trailing submatrix at each step. Instead of computing these values at each step, it is possible to compute them once before the factorization and only update them after each Householder elimination. Let c be a vector of size n containing the square of the 2-norms

of the column of A, $c_i = \|A_{:,i}\|_2^2$, and assume one Householder reflection is applied to A in order to annihilate all the coefficients in the first column except the first. Because this is a unitary transformation, the c vector can be updated to store the square 2-norms of the columns of the trailing submatrix $A^{(1)}$ with the simple operation $c_i = c_i - r_{1,i}^2$. Note that, because all the column norms have to be updated at each step, blocking techniques cannot be used to their full extent and thus, in the column-pivoted QR factorization a large part of operations is done in Level-2 BLAS routines.

2.2 Sparse matrix factorizations

The factorization of a sparse matrix is considerably more complex than its dense counterpart because special care must be taken to the sparsity structure of the matrix in order to avoid useless computations on zero coefficients. In this document we will assume that square matrices are structurally symmetric; this greatly simplifies the handling of dependencies between computations and implies a number of other favorable structural properties. Note that, in the case where the structure of the matrix is not symmetric, it is possible to "symmetrize" it by explicitly storing zero coefficients in order to make sure that if $a_{i,j}$ if present then also is $a_{j,i}$. Of course, if the structure of the original matrix is strongly unsymmetric, many zero coefficients have to be added to the structure in order to symmetrize it which may result in an excessive memory and computational overhead; for these cases dedicated unsymmetric factorization methods can be employed and we will briefly mention them.

2.2.1 The Cholesky multifrontal method

Consider a symmetric, positive definite matrix A with the sparsity structure shown in Figure 2.3 (*left*) and suppose we execute Algorithm 2.3 on it. The structure of the resulting L factor is reported in Figure 2.3 (*right*). Two interesting observations can be made:

1. Eliminating one variable through one iteration of the outer loop in the Cholesky algorithm does not necessarily imply updating all the coefficients in the trailing submatrix. For example, when the first iteration is executed, only the coefficients in columns 1 (obviously), 4 and 9 are modified; this is because for all the other columns j, $l_{j,1}$ is equal to zero and, therefore, the update operation $a_{i,j} = a_{i,j} - l_{j,1}l_{i,1}$ has no effect.

2. Some of the coefficients that were zero in A have been turned into nonzero by the factorization; for example when the second iteration of the Cholesky factorization is executed $a_{4,3}$ becomes nonzero due to the update $a_{4,3} = a_{4,3} - l_{4,2}l_{3,2}$. As a result the structure of $L + L^T$ is denser than that of A; this phenomenon is referred to as *fill-in*. Because of fill-in the cost of a sparse factorization can be very high, both in terms of memory and time, even for very sparse matrices.

This two properties can be more formally stated by Propositions 2.2 and 2.3.

Proposition 2.2— *For $j > k$ the numerical values of column j depend on column k, denoted $k \to j$, if and only if $l_{j,k} \neq 0$.*

By saying that column j depends on column k we mean that eliminating variable k updates the numerical values of column j. This implies that the second term on the

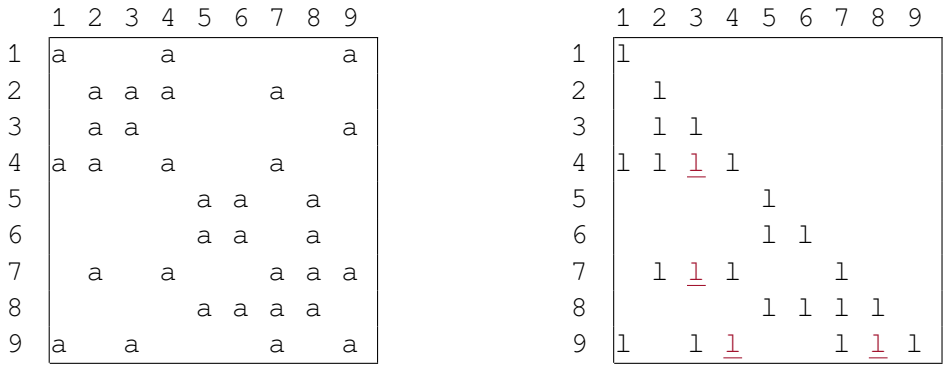


Figure 2.3: The structure of a symmetric sparse matrix A and of the associated Cholesky factor L. The fill-in in L is represented with underlined letters.

right-hand side in line 6 of Algorithm 2.3 is nonzero which is true if and only if $l_{j,k}$ is nonzero.

Proposition 2.3— *Let be $i > j > k$. If column j and column i depend on k then column i depends on column j.*

Proof. The fact that both columns j and i depend on column k implies that $l_{j,k} \neq 0$ and $l_{i,k} \neq 0$. Therefore, $l_{i,j}$ will be nonzero as a result of the update on line 6 of Algorithm 2.3. □

As a consequence of Proposition 2.3 it is easy to see that, a fill-in coefficient $l_{i,j}$ is created if $a_{i,j} = 0$ and $l_{i,k}$ and $l_{j,k}$ are nonzero for some $k < i, j$.

The canonical tools for formalizing the theory of sparse matrix factorizations and characterizing the fill-in are graphs: a sparse matrix A can be represented by the graph $\mathcal{G}(A) = (\mathcal{V}, \mathcal{E})$ whose vertex set $\mathcal{V} = \{1, 2, ..., n\}$ includes the unknowns of A and edge set $\mathcal{E} = \{(i, j) \; \forall \; a_{i,j} \neq 0\}$ includes an edge for each nonzero coefficient of A. In our case, the symmetry of the matrix implies that if (i, j) is in $\mathcal{E}$ then also (j, i) is and therefore we will only represent one of these edges and the graph will be *undirected*. The graph $\mathcal{G}(A)$ is also commonly called the *adjacency graph* of A. Figure 2.4 shows, on the top-left , the adjacency graph for the matrix in Figure 2.3.

It is possible to use the adjacency graph to model the factorization of a sparse matrix. Specifically we can build a sequence of graphs $\mathcal{G}(A) = \mathcal{G}_0(A), \mathcal{G}_1(A), ..., \mathcal{G}_{n-1}(A)$, called *elimination graphs*, such that $\mathcal{G}_k(A)$ is the adjacency graph of the trailing submatrix after elimination of variables $1, 2, ..., k$. Elimination graph $\mathcal{G}_k(A)$ is built from $\mathcal{G}_{k-1}(A)$ by removing node k as well as all its incident edges and by updating the connectivity of the remaining nodes. By Proposition 2.3, for any two nodes i and j neighbors of k we have to add an edge connecting them if it does not exist already; this models the occurrence of a new fill-in coefficient. In other words, upon elimination of node k, a *clique* is added to the graph which includes all the neighbors of k. Algorithm 2.5 shows the elimination graphs process and Figure 2.4 shows the elimination graphs for the matrix in Figure 2.3.

The elimination graphs process suggests that, if in $\mathcal{G}(A)$ there is a path $i, k_1, k_2, ..., k_p, j$ with $k_s < \min(i, j) \; \forall s$, then $l_{i,j}$ must necessarily be nonzero. This is because whenever one of the k_s nodes is eliminated, its two neighbors along the path become connected (if they weren't already); therefore, when the last of these k_s nodes is eliminated i and j become connected. This intuition leads to one very well known result by Rose, Tarjan and Lueker [139]:

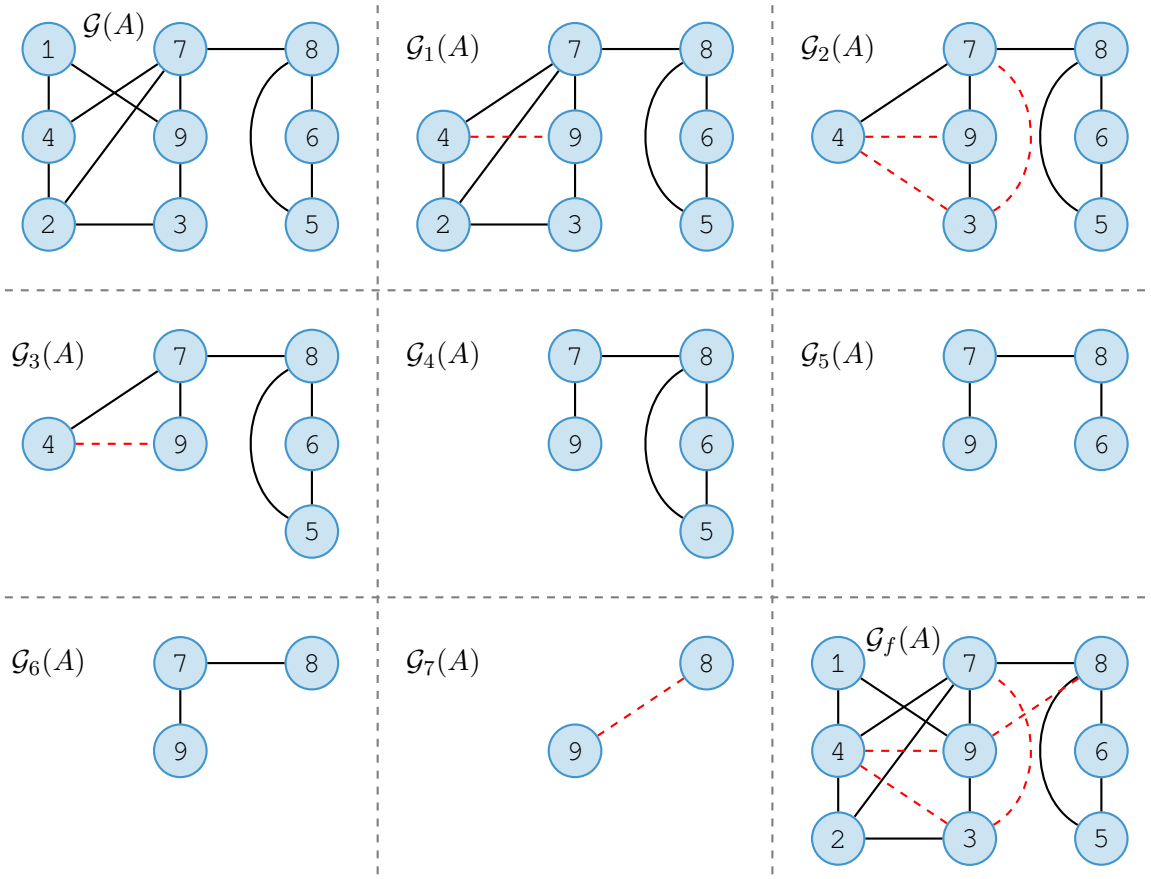


Figure 2.4: Adjacency graph (*top-left*), eliminations graphs and filled graph (*bottom-right*) of matrix A in Figure 2.3.

Theorem 2.5 (Rose et al. [139]).— *Let $i > j$. Then $l_{i,j} \neq 0$ if and only if there exist a path*

$$j, k_1, k_2, ..., k_p, i$$

in $\mathcal{G}(A)$ such that $k_s < \min(i, j)$.

although the "if" part of this theorem can easily be understood following the idea described above, the proof of the "only if" part is more complex and we refer the reader to the original paper for its description.

Algorithm 2.5 Elimination graphs process.

$\mathcal{G}_0(A) \leftarrow \mathcal{G}(A) = (\mathcal{V}, \mathcal{E})$
for $k = 1, ..., n-1$ **do**
  $\mathcal{V} \leftarrow \mathcal{V} - \{k\}$
  $\mathcal{E} \leftarrow \mathcal{E} - \{(k, l) : l \in \text{adj}(k)\} \cup \{(i, j) : i \in \text{adj}(k) \text{ and } j \in \text{adj}(k)\}$
  $\mathcal{G}_k \leftarrow (\mathcal{V}, \mathcal{E})$
end for

The graph $\mathcal{G}_f(A)$ obtained by adding to $\mathcal{G}(A)$ all the edges associated with fill-in coefficients is called *filled graph* and corresponds to the adjacency graph of $L + L^T$. For

the matrix in Figure 2.3 the filled graph is reported in the bottom-right of Figure 2.4. The *directed filled graph* $\overrightarrow{\mathcal{G}}_f(A)$ is obtained from the filled graph assuming that an edge (i, j) is directed from i to j if $j > i$. This graph cannot have cycles, for which reason it is called a *Directed Acyclic Graph* (DAG).

Note that $\overrightarrow{\mathcal{G}}_f(A)$ represents the dependencies among the columns of the matrix as defined in Proposition 2.2 and its *transitive closure* states whether a variable can be eliminated before another. Note that, in order to establish a dependency between columns i and j, we are only interested in knowing whether there exist a path in $\overrightarrow{\mathcal{G}}_f(A)$ connecting nodes i and j; therefore the directed filled graph can be simplified by keeping the longest path connecting i and j and discarding all the others. This amounts to computing the *transitive reduction* of $\overrightarrow{\mathcal{G}}_f(A)$ which we call $\mathcal{T}(A)$. Note that $\mathcal{T}(A)$ is equivalent to the directed graph associated with the matrix obtained by removing from L all the coefficients below the diagonal except the first which can easily be seen using Proposition 2.3. Assume that $l_{i,k} \neq 0$ and $l_{j,k} \neq 0$ with $i > j > k$; because we know that $l_{i,j}$ must necessarily be nonzero, we can suppress $l_{i,k}$ because the dependency $k \to i$ is indirectly represented by the chain of dependencies $k \to j \to i$. Schreiber [144] shows that, if the matrix A is not reducible, this graph is an ordered tree with root n (hence the choice of the letter $\mathcal{T}$ for denoting it). $\mathcal{T}(A)$ is the most compact way of representing all the column dependencies in the factorization of a sparse matrix and is commonly referred to as *elimination tree*.

Definition 2.1 ***Elimination tree*** **[144].**— *Let A be a sparse, symmetric positive definite matrix of size n with Cholesky factor L. The elimination tree is defined to be the structure with n nodes $\{1, 2, ..., n\}$ such that the node p is the parent of node j if and only if*

$$p = \min\{i > j \mid l_{i,j} \neq 0\}.$$

From this definition and the theory discussed above, the two following theorems can be derived.

Theorem 2.6 ([144]).— *If $l_{j,k} \neq 0$ and $k < j$, then the node k is a descendant of j in the elimination tree.*

Theorem 2.7 ([119]).— *If node k is a descendant of node j in the elimination tree, then the structure of the vector $[l_{j,k}, ..., l^T_{n,k}]$ is contained in the structure of $[l_{j,j}, ..., l^T_{n,j}]$.*

Based on the theory discussed in the previous section, various sparse matrix factorization techniques can be defined. Among the most commonly used ones is the *multifrontal* method introduced by Duff et al. [66]. The following discussion of the multifrontal method is extracted from a 1992 paper by Liu [119].

Let A be a sparse, SPD matrix, L its Cholesky factor and $i_0, i_1, ..., i_r$ be the row-subscripts of column j of L with $i_0 = j$. We denote with $\mathcal{T}$ the associated elimination tree and with $\mathcal{T}_j$ the subtree rooted at node j.

The following two definitions lay the foundations of the multifrontal method.

Definition 2.2 ***Subtree update matrix*** **[119].**— *The j-subtree update matrix is*

$$\bar{U}_j = -\sum_{k \in \mathcal{T}_j - \{j\}} \begin{bmatrix} l_{j,k} \\ l_{i_1,k} \\ \vdots \\ l_{i_r,k} \end{bmatrix} \begin{bmatrix} l_{j,k} l_{i_1,k} \cdots l_{i_r,k} \end{bmatrix}.$$

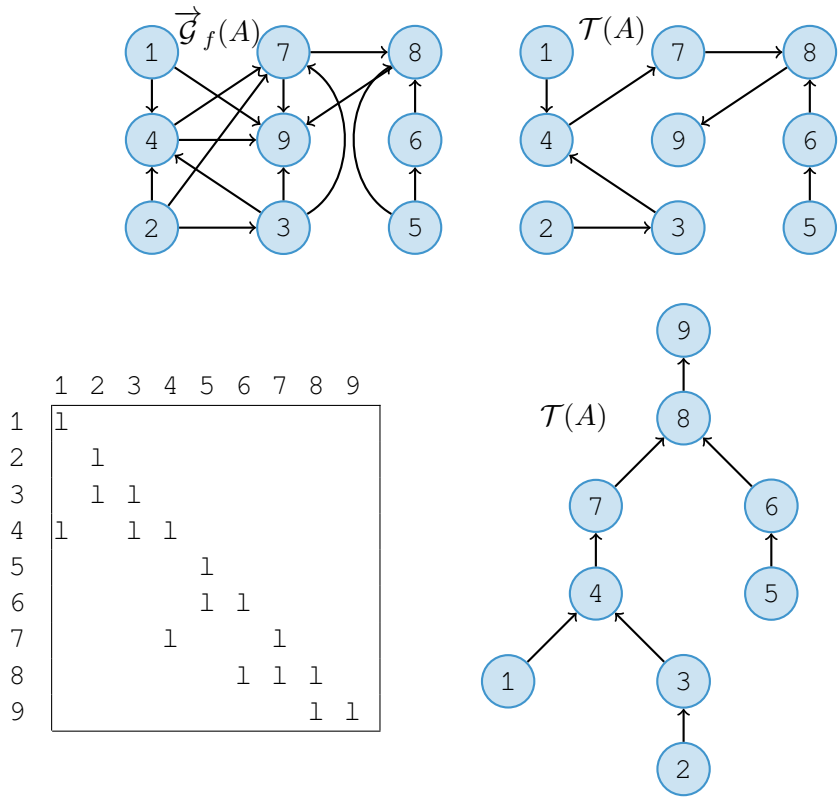


Figure 2.5: Adjacency graph (*top-left*), eliminations graphs and filled graph (*bottom-right*) of matrix A in Figure 2.3.

Definition 2.3 ***Frontal matrix*** **[119].**— *The jth frontal matrix (or front) F_j for A is defined to be*

$$F_j = \begin{bmatrix} a_{j,j} & a_{j,i_1} & \cdots & a_{j,i_r} \\ a_{i_1,j} & & & \\ \vdots & & 0 & \\ a_{i_r,j} & & & \end{bmatrix} + \bar{U}_j.$$

The subtree update matrix contains all the updates related to nodes that are descendants of j in the elimination tree; these are all the updates related to elimination steps k that modify column j either directly, if $l_{j,k} \neq 0$, or indirectly, if $l_{j,k} = 0$. For this reason, when F_j is computed, its first row and column are said to be *fully updated* (or *fully summed* or *fully assembled*) in the sense that they have received all the updates related to previous elimination steps and, therefore, correspond to the row and column j of the trailing submatrix $A^{(k)}$ at step k of the factorization. As a result, a single elimination step on F_j generates column j of the factor L:

$$F_j = \begin{bmatrix} l_{j,j} & 0 & \cdots & 0 \\ l_{i_1,j} & & & \\ \vdots & & I & \\ l_{i_r j} & & & \end{bmatrix} \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & & & \\ \vdots & & U_j & \\ 0 & & & \end{bmatrix} \begin{bmatrix} l_{j,j} & l_{i_1,j} & \cdots & l_{i_r,j} \\ 0 & & & \\ \vdots & & I & \\ 0 & & & \end{bmatrix} \tag{2.18}$$

where U_j is a Schur complement which is commonly referred to as *update matrix* (or *contribution block*). Note that the first row and column of F_j are full, which necessarily

implies that U_j is also full; therefore F_j can be conveniently stored as a dense matrix and the decomposition in Equation (2.18) can be computed with a single iteration of the outer loop of Algorithm 2.3.

The next theorem follows from the definition of F_j and Equation (2.18).

Theorem 2.8 Liu [119].—

$$U_j = -\sum_{k \in \mathcal{T}_j} \begin{bmatrix} l_{i_1,k} \\ \vdots \\ l_{i_r,k} \end{bmatrix} [l_{i_1,k} \cdots l_{i_r,k}]. \tag{2.19}$$

Proof. Equation (2.18) can also be written

$$F_j = \begin{bmatrix} l_{j,j} \\ l_{i_1,j} \\ \vdots \\ l_{i_r,j} \end{bmatrix} [l_{j,j} l_{i_1,j} \cdots l_{i_r,j}] + \begin{bmatrix} 0 & 0 & \cdots & 0 \\ 0 & & & \\ \vdots & & U_j & \\ 0 & & & \end{bmatrix}$$

Note that F_j without the first row and column is equal to $\bar{U}_j$ without the first row and column and therefore we can write

$$-\sum_{k \in \mathcal{T}_j - \{j\}} \begin{bmatrix} l_{i_1,k} \\ \vdots \\ l_{i_r,k} \end{bmatrix} [l_{i_1,k} \cdots l_{i_r,k}] = \begin{bmatrix} l_{i_1,j} \\ \vdots \\ l_{i_r,j} \end{bmatrix} [l_{i_1,j} \cdots l_{i_r,j}] + U_j$$

which concludes the proof. □

One last result is still needed to specify the multifrontal method but this requires the definition of one more tool.

Consider two matrices R and S with indices, respectively, $\mathcal{I}_r = \{i_1, ..., i_r\}$ and $\mathcal{I}_s = \{i_1, ..., i_s\}$; $\mathcal{I}_r$ and $\mathcal{I}_s$ may have a non-empty intersection and their union is $\mathcal{I}_t = \mathcal{I}_r \cup \mathcal{I}_s$. By adding empty rows and columns R ans S can be extended to conform with $\mathcal{I}_t$. We define $T = R \boxplus S$ to be the $t \times t$ matrix obtained by summing the extended R ans S matrices and we refer to the $\boxplus$ matrix operator as the *extend-add* operator.

Theorem 2.9 Liu [119].— *Let nodes $c_1, ..., c_s$ be the children of node j in the elimination tree. Then*

$$F_j = \begin{bmatrix} a_{j,j} & a_{j,i_1} & \cdots & a_{j,i_r} \\ a_{i_1,j} & & & \\ \vdots & & 0 & \\ a_{i_r,j} & & & \end{bmatrix} \boxplus U_{c_1} \boxplus \cdots \boxplus U_{c_s}. \tag{2.20}$$

Proof. Because $c_1, ..., c_s$ are the children of node j, $\mathcal{T}_j - \{j\}$ corresponds to the disjoint union of all the nodes in $\mathcal{T}_{c_1}, ..., \mathcal{T}_{c_s}$. The result follows from the definition of the subtree update matrix $\bar{U}_j$ and Equation (2.19). □

The multifrontal method can finally be defined as in Algorithm 2.6. The first four steps of this algorithm on the matrix of Figure 2.3 are illustrated in Figure 2.6.

The multifrontal method essentially achieves the factorization of a sparse matrix through a sequence of operations on relatively small dense matrices, i.e., the frontal matrices. This is an extremely favorable property because computations can be done using

Algorithm 2.6 The Multifrontal method.

for $j = 1, ..., n$ **do**
  INIT(F_j) : $F_j = 0$
  ASM$(F_j,\ A_{:,j})$: $F_j = F_j \overset{+}{\leftarrow} A_{:,j}$
  for $c = c_1 ... c_s$, children of j **do**
    ASM$(F_j,\ U_c)$: $F_j = F_j \overset{+}{\leftarrow} U_c$
  end for
  FACTO(F_j) :
    $(F_j)_{:,1} = (F_j)_{:,1} / \sqrt{(f_j)_{1,1}}$
    $U_j = (F_j)_{2:,2:} - (F_j)_{2:,1}(F_j)^T_{2:,1}$
end for

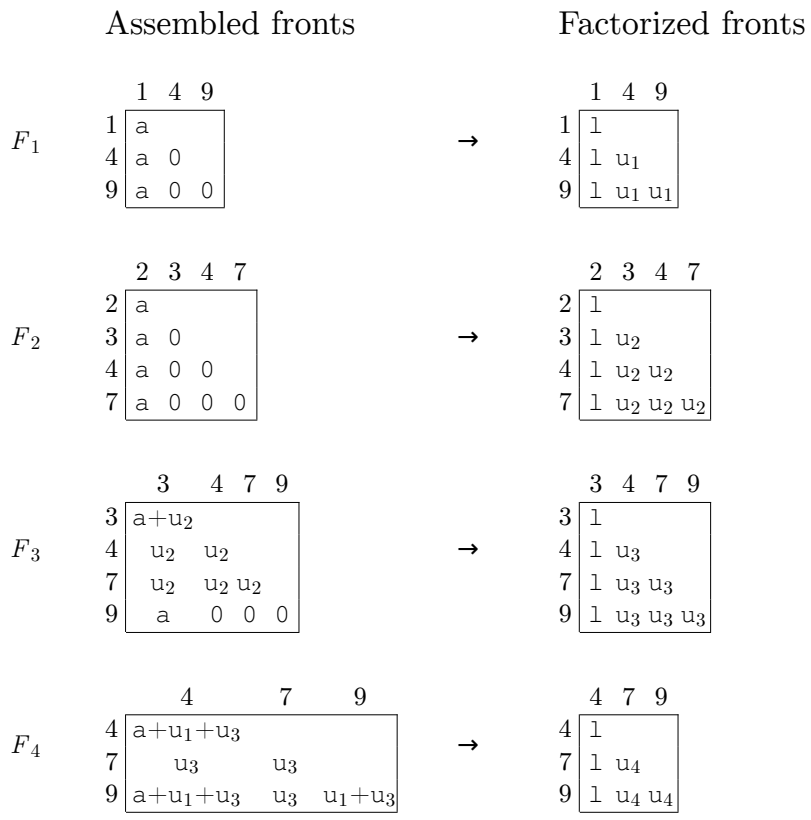


Figure 2.6: The first four steps of the multifrontal method on the matrix of Figure 2.3.

dense linear algebra kernels which are highly efficient and can benefit from techniques such as vectorization. The multifrontal method presented above, however, cannot take advantage of Level-3 BLAS routines because only one elimination step is applied to each front and thus most of the computations are done at line 9 of Algorithm 2.6 which is a Level-2 BLAS rank-1 update. This issue can be overcome by leveraging the concept of *supernodes*. A supernode is defined as a set of consecutive columns of L which have the same structure, apart from the related triangular block on the diagonal. Rather than having a frontal matrix for each one of these columns, they can be assembled and eliminated together in a single front; because these columns have the same structure, this does not imply any additional fill in or computations. The advantage of using supernodes is that multiple eliminations can be done on each front and, thus, the blocking techniques

describes in Section 2.1.4 can be employed; this also has the advantage that the size of the elimination tree can be considerably reduced leading to more compact data structures and more efficient handling. Various techniques can be used to detect supernodes but they are generally referred to as *amalgamation* since multiple nodes of the elimination tree are merged together in a single supernode; the resulting amalgamated elimination tree is commonly referred to as *assembly tree*. Note that it is also possible to amalgamate nodes related to columns of L that do not have the same structure; this necessarily implies additional fill-in but it may be worth paying this extra cost if computations can be sped up considerably. This is commonly referred to as *relaxed amalgamation* because it is guided by a relaxation parameter which defines what amount of extra fill-in is deemed acceptable. In the matrix of Figure 2.3, for example, nodes 3 and 4 can be amalgamated without additional fill-in because the corresponding columns have the same structure; nodes 2 can also be amalgamated to this supernode but with an overhead represented by the fill-in coefficient $l_{9,2}$.

2.2.2 The QR multifrontal method

Although the first methods conceived for computing the QR factorization of a sparse matrix are based on the use of Givens rotations [75], techniques based on Householder reflections have successively become more popular due to their higher efficiency.

Let A be a sparse matrix of size $m \times n$ with $m \geq n$ and suppose we apply the first step of a (unblocked) QR factorization to this matrix, that is, we want to compute a Householder reflection H that annihilates all the nonzero coefficients except one in the first column. Note that the corresponding Householder reflector, computed as in Equations (2.12)(2.13), will have the same structure as the first column of A and therefore, when the reflection is applied to the trailing submatrix, only part of its coefficients will be updated. Specifically, only coefficients along the rows that have a nonzero in the pivotal column (the first column in this case) and in the columns that have at least one nonzero in the same position as the pivotal column. This is formalized in Theorem 2.10; here, and in the remainder of this section we denote $\mathcal{S}(x)$ the structure of a vector, i.e., $\mathcal{S}(x) = \{i | x_i \neq 0\}$.

Theorem 2.10 George et al. [76].— *Consider $HA = A - \tau v(v^T A)$. Then $(HA)_{i,:}$ where $i \notin \mathcal{S}(v)$ is equal to row i of A. For any row $i \in \mathcal{S}(v)$, the nonzero pattern of $(HA)_{i,:}$, is*

$$\mathcal{S}((HA)_{i,:}) = \bigcup_{j \in \mathcal{S}(v)} \mathcal{S}(A_{j,:})$$

That is, in HA, the nonzero pattern of any modified row $i \in \mathcal{S}(v)$ is replaced with the set union of all rows that are modified by the Householder reflection H.

The first step of the Householder QR on a sparse matrix is illustrated in Figure 2.7 where all the coefficients in the first column are annihilated except the first. This operations updates only rows 1, 6, 10 and 18 and, upon its execution, these rows will have nonzeros in columns $\{1,\ 4,\ 9\} = \mathcal{S}(A_{1,:}) \cup \mathcal{S}(A_{6,:}) \cup \mathcal{S}(A_{10,:}) \cup \mathcal{S}(A_{18,:})$. Note that we have the freedom to choose which nonzero coefficient, among those in the pivotal column, to keep. Actually, any row permutation of A will always lead to the same QR factorization; the only possible difference may be in intermediate fill-in, depending on how the factorization is computed. On the right part of Figure 2.7 we report the final structure of the R factor and the V matrix containing the computed Householder reflectors.

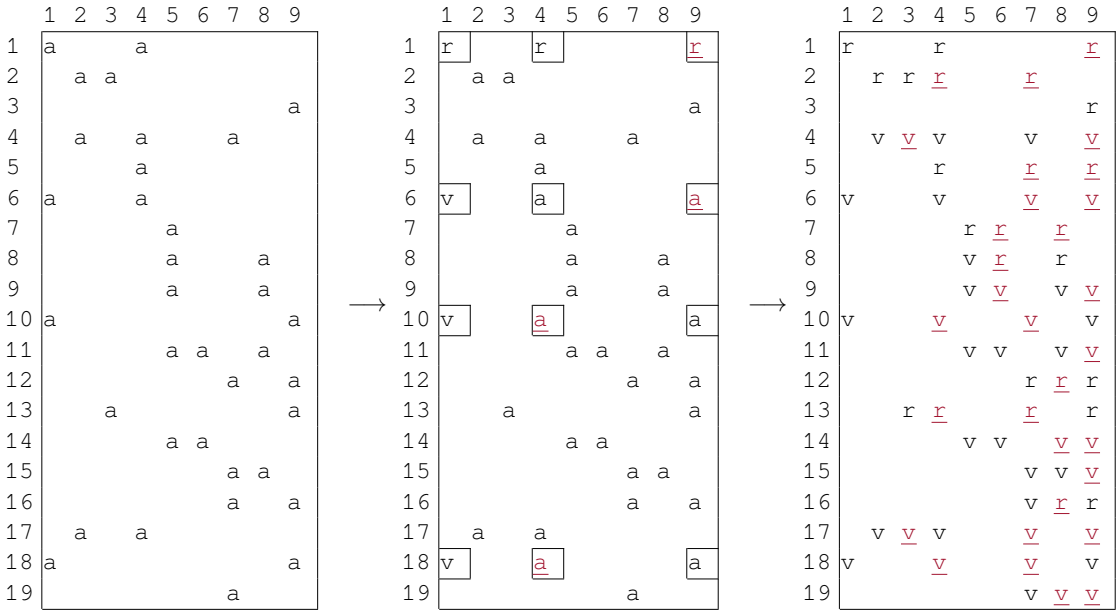


Figure 2.7: A sparse, overdetermined matrix on the left side; the nonzero coefficients of the matrix are denoted with the letter `a`. In the middle part, the first step of the Householder QR factorization. This modifies rows 1, 6, 10 and 18. After this step, the computed coefficients of the R factor and of the V matrix are denoted, respectively, with the letters `r`and `v`; fill-in coefficients are underlined. On the right size, the result of the complete QR factorization.

Most of the theory behind sparse QR factorizations relies on the so-called *Strong Hall* property which we report in Definition 2.4.

Definition 2.4 ***Strong Hall property*** **[43].**— *A matrix A of size $m \times n$, $n \geq 2$ is Strong Hall if, for any $1 \leq k < n$, every set of k columns has nonzeros in at least $k+1$ different rows.*

Note that any matrix that is not strong Hall can be permuted to a block upper triangular form called the Dulmage-Mendelson decomposition (see, for example the book by Brualdi et al. [42]) with diagonal blocks having this property.

We report below a list of results that lay the foundation of the multifrontal QR method. For the sake of readability and without loss of generality, we will assume that $a_{k,k}^{(k-1)} \neq 0$ structurally, that is to say, at elimination step k the diagonal coefficient is structurally nonzero. Note that whenever this condition does not hold, the matrix A can be permuted to have a nonzero diagonal if its structural rank (i.e., the maximum rank of all the matrices with the same sparsity pattern) is full; otherwise explicit zeros can be added to enforce it. Therefore we will assume that each Householder elimination step annihilates all the nonzero coefficients in the corresponding column except the diagonal one.

Theorem 2.11— *Let A be a Strong Hall matrix. After one Householder elimination step, the trailing submatrix will also be Strong Hall.*

Proof. Assume that there are s nonzeros in the first column of A, i.e., $s = |\mathcal{S}(A_{:,1})|$. Now take, in A, any group of $k+1$ columns including the first one. Because A is Strong Hall,

these $k+1$ columns will have nonzeros in at least $k+2$ different rows and because the first column has s nonzeros, the remaining k columns have at least $k+2-s$ nonzeros in rows other than $\mathcal{S}(A_{:,1})$. We want to show that upon the elimination of the first column, the remaining k columns have nonzeros in $k+1$ different rows. If these columns do not have nonzeros in $\mathcal{S}(A_{:,1})$, then they will be unaffected by the Householder elimination and thus they will keep all of their (at least) $k+1$ nonzero rows. If, instead, they have nonzeros in some rows of $\mathcal{S}(A_{:,1})$, they will certainly lose one nonzero row; however, because of Theorem 2.10, after the elimination they will have nonzeros in all the remaining $s-1$ rows of $\mathcal{S}(A_{:,1})$. Therefore the will have nonzeros in at least $(s-1)+(k+2-s)=k+1$ row, which concludes the proof. □

Theorem 2.12— *Let A be a Strong Hall matrix and $QR = A$ its Householder QR factorization. Then, for $k > j$, column elimination k depends on column elimination j if and only if $r_{j,k} \neq 0$.*

Proof. Suppose one Householder elimination step is applied to A in order to annihilate all the nonzeros in $A_{:,1}$ except $a_{1,1}$. This operation will compute $R_{1,:}$ and $V_{:,1}$ but, because of Theorem 2.10, will also update all the coefficients with row index $\mathcal{S}(A_{:,1}) \setminus \{1\}$ and column index $\mathcal{S}(R_{1,:}) \setminus \{1\}$; because A is Strong Hall, $|\mathcal{S}(A_{:,1})| > 1$ which implies that this elimination step will update all columns and only those for which $r_{1,k} \neq 0$. The result follows because the trailing submatrix after this elimination step is still Strong Hall as stated by Theorem 2.11. □

Theorem 2.13— *Let A be a Strong Hall matrix, $QR = A$ its Householder QR factorization and $i > j > k$. If column eliminations i and j depend on column elimination k, the column elimination i depends on column elimination j.*

Proof. Because $A^{(k-1)}$ is Strong Hall, upon elimination of column k, the trailing submatrix $A^{(k)}$ will have nonzeros in columns j and i and rows $\mathcal{S}(A^{(k-1)}_{:,k}) \setminus \{k\}$. This implies that upon elimination of column j, $r_{j,i} \neq 0$ which concludes the proof. □

Note that the R factor of the QR factorization of a matrix A is mathematically equivalent to the Cholesky factor of the *normal equations* A^TA (which exists if A is full-rank) because $A^TA = R^TQ^TQR = R^TR$. This may suggest that the structure of R may be computed through a symbolic analysis of the Cholesky factorization of A^TA, for example, by using the elimination graphs procedure explained in Section 2.2.1. This unfortunately does not hold in the general case because in the numerical Cholesky factorization of A^TA *cancellations* may occur: a nonzero coefficient in the trailing submatrix is turned into zero in the course of the factorization. Cancellations may be divided in two types: *lucky* and *essential* cancellations. The first type depends on the actual values of the coefficients of A and, therefore, are impossible to predict just by looking at its structure. The second type, however, depends solely on the structure of A and will always occur regardless of the actual numerical values. This is illustrated in Figure 2.8. Consider the sparse matrix illustrated on the left side of the figure: because A is an upper triangular matrix, its QR factorization is trivially given by $Q = I$ and $R = A$ and, therefore, R has the same structure as A as shown in the middle part of the figure. Now, the (i,j) coefficient of A^TA is nonzero if in A there is a row k with $a_{k,i}$ and $a_{k,j}$ nonzero. Therefore, A^TA is a dense matrix with all nonzero coefficients because the first row of A is full; as a result, the symbolic Cholesky factorization will predict that R is a full, triangular matrix as shown on the right side of the figure. The numerical Cholesky factorization of A^TA, instead, will

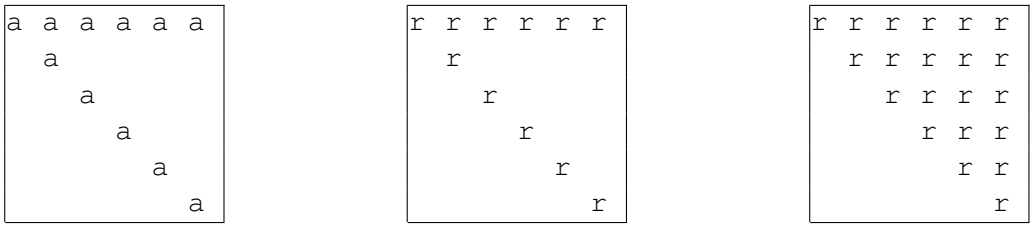


Figure 2.8: A sparse matrix A on the left, the actual R structure in the middle and the structure of R predicted by the symbolic Cholesky factorization of A^TA on the right.

have essential cancellations on all the coefficients that correspond to the zero coefficients in the upper triangular part of A.

Coleman et al. [50] showed that if A is a Strong Hall matrix, no essential cancellation occurs in the Cholesky factorization of A^TA; as a result, for such class of matrices, the structure of R can be correctly predicted by the symbolic Cholesky factorization of A^TA. George et al. [75] showed that for matrices not belonging to this category, the actual R factor will always fit in the structure computed by the symbolic Cholesky factorization of A^TA. Finally, as said above, non Strong Hall matrices can always be permuted to block-triangular form with diagonal blocks having this property. It is, therefore, safe to assume that the structure of R can be computed through a symbolic Cholesky factorization of A^TA.

As a consequence of all the results presented above, we can conclude that the most compact way of representing the dependencies between Householder elimination steps in the sparse QR factorization of a matrix A is a so-called *column elimination tree* which corresponds to the elimination tree for the Cholesky factorization of A^TA. The structure of R and the column elimination tree can be computed without explicitly forming A^TA as shown by Gilbert et al. [80]. Methods have also been proposed in the literature to compute the structure of the V matrix that holds the Householder reflectors [76, 80].

Based on the results reported above, it is possible to develop the multifrontal QR factorization: it consists in a traversal of the column elimination tree and, when a node is visited, a dense frontal matrix is formed through assembly operations that combine coefficients from A and from the child nodes and it is reduced through a single elimination step. The assembly and elimination operations are clearly different with respect to the Cholesky factorization described in the previous section. The assembly of a frontal matrix F_j is achieved by means of an *extend stack* operator $\overleftrightarrow{\ }$; the update matrices (or contribution blocks) from child nodes are extended to make their column indices conform to those of the front and are stacked to those rows of A whose first, leftmost nonzero is in column j, denoted $\mathcal{F}_j$

$$F_j = A_{\mathcal{F}_j,*} \overleftrightarrow{\ } U_{c_1} \overleftrightarrow{\ } \cdots \overleftrightarrow{\ } U_{c_s} = \begin{bmatrix} \overleftrightarrow{A}_{\mathcal{F}_j,*} \\ \overleftrightarrow{U}_{c_1} \\ \vdots \\ \overleftrightarrow{U}_{c_s} \end{bmatrix}$$

where by $\overleftrightarrow{X}$ we denoted the column-extended matrix X. As for the elimination step, this corresponds to one single step of the dense Householder QR factorization of Algorithm 2.4:

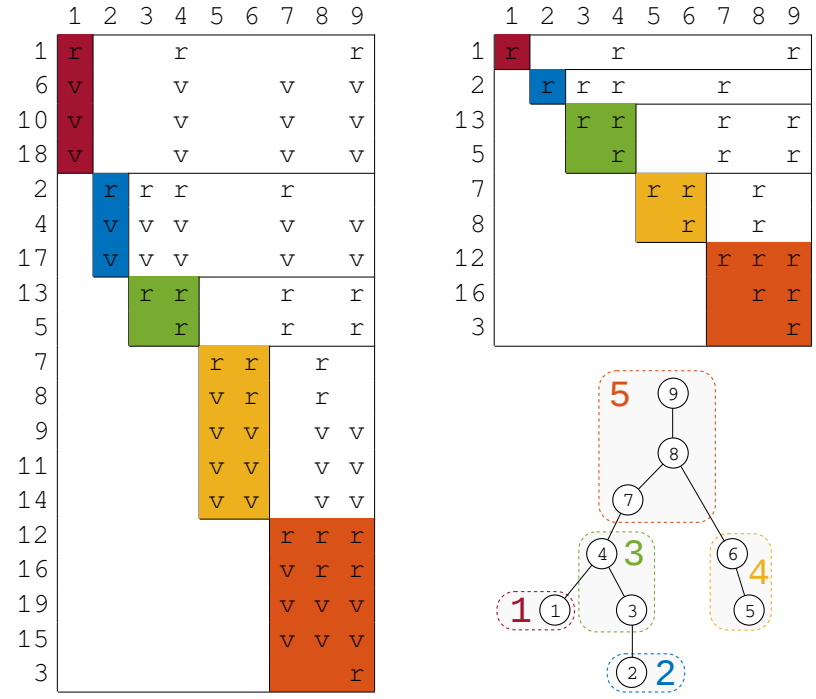


Figure 2.9: Example of multifrontal QR factorization. On the left side the factorized matrix (the same as in Figure 2.7 with a row permutation). On the upper-right part, the structure of the resulting R factor. On the right-bottom part the elimination tree; the dashed boxes show how the nodes are amalgamated into supernodes.

$$(I - \tau v_j v_j^T) F_j = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & & & \\ \vdots & & U_j & \\ 0 & & & \end{bmatrix} \begin{bmatrix} r_{j,j} & r_{j,i_1} & \cdots & r_{j,i_r} \\ 0 & & & \\ \vdots & & I & \\ 0 & & & \end{bmatrix}.$$

Note that here, instead of annihilating all the nonzero coefficients in column j except the diagonal one as assumed above, we are keeping the coefficient in the first row of $\mathcal{F}_j$.

As for the Cholesky case, in practical implementations of the multifrontal QR factorization, nodes of the elimination tree are amalgamated to form supernodes. The amalgamated pivots correspond to rows of R that have the same structure and can be eliminated at once within the same frontal matrix without producing any additional fill-in in the R factor. The elimination of amalgamated pivots and the consequent update of the trailing frontal submatrix can thus be performed by means of efficient Level-3 BLAS routines through the WY representation [143] described in Section 2.1.4.2. Moreover, amalgamation reduces the number of assembly operations increasing the computations-to-communications ratio which results in better performance. Figure 2.9 shows some details of a sparse QR factorization. The factorized matrix is shown on the left part of the figure. Note that this is the same matrix as in Figure 2.7 where the rows of A are sorted in order of increasing index of the leftmost nonzero in order to show more clearly the computational pattern of the method on the input data. On the top-right part of the figure, the structure of the resulting R factor is shown. The elimination/assembly tree is, instead reported in the bottom-right part: dashed boxes show how the nodes can be amalgamated into supernodes with the corresponding indices denoted by bigger size numbers. The amalgamated

nodes have the same row structure in R modulo a full, diagonal block. It has to be noted that in practical implementations the amalgamation procedure is based only on information related to the R factor and, as such, it does not take into account fill-in that may eventually appear in the V matrix. The supernode indices are reported on the left part of the figure in order to show the pivotal columns eliminated within the corresponding supernode and on the top-right part to show the rows of R produced by the corresponding supernode factorization. Note that this matrix is such that the structure of A^TA is the same as the one of the matrix in Figure 2.3 and thus the structure of R is the same as that of the L factor in Figure 2.5 and the elimination and assembly tree are the same as those in the same figure.

In order to reduce the operation count of the multifrontal QR factorization, two optimizations are commonly applied:

1. once a frontal matrix is assembled, its rows are sorted in order of increasing index of the leftmost nonzero. The number of operations can thus be reduced, as well as the fill-in in the V matrix, by ignoring the zeros in the bottom-left part of the frontal matrix;

2. the frontal matrix is completely factorized. Despite the fact that more Householder vectors have to be computed for each frontal matrix, the overall number of floating point operations is lower since frontal matrices are smaller. This is due to the fact that contribution blocks resulting from the complete factorization of frontal matrices are smaller. Note that this implies that contribution blocks are triangular for overdetermined fronts and trapezoidal for overdetermined ones. This makes the other optimization above necessary.

Figure 2.10 shows the assembly and factorization operations for the supernodes 1, 2 and 3 in Figure 2.9 when these optimization techniques (referred to as *Strategy 3* in Amestoy et al. [15]) are applied. Note that, because supernodes 1 and 2 are leaves of the assembly tree, the corresponding assembled frontal matrices only include coefficients from the matrix A. The contribution blocks resulting from the factorization of supernodes 1 and 2 are appended to the rows of the input A matrix associated with supernode 3 in such a way that the resulting, assembled, frontal matrix has the *staircase* structure shown in Figure 2.10 (*bottom*).

2.2.3 Additional topics

2.2.3.1 Fill-reducing orderings

In the previous sections we have assumed that all the unknowns of the matrix are eliminated in the natural order but this does not necessarily have to be the case. Eliminating the unknowns in a different order basically amounts to applying a permutation to the matrix; this is a row and column permutation for Gaussian Elimination of structure-symmetric matrix and a column permutation for the QR factorization. This, however, changes the fill-in and the dependencies among columns; an extreme case is illustrated in Figure 2.11 where eliminating variables in the natural order implies filling-in the whole structure whereas using the reverse order does not introduce any fill-in.

The problem of finding a matrix permutation that minimizes the fill-in was proved to be NP-Complete by Yannakakis [165]. Later, Luce et al. [121] showed that finding a permutation that minimizes the number of floating point operations is a distinct, NP-hard problem. In the literature many heuristic techniques have been proposed to tackle these problems (mostly the first one). These are generally divided into two groups.

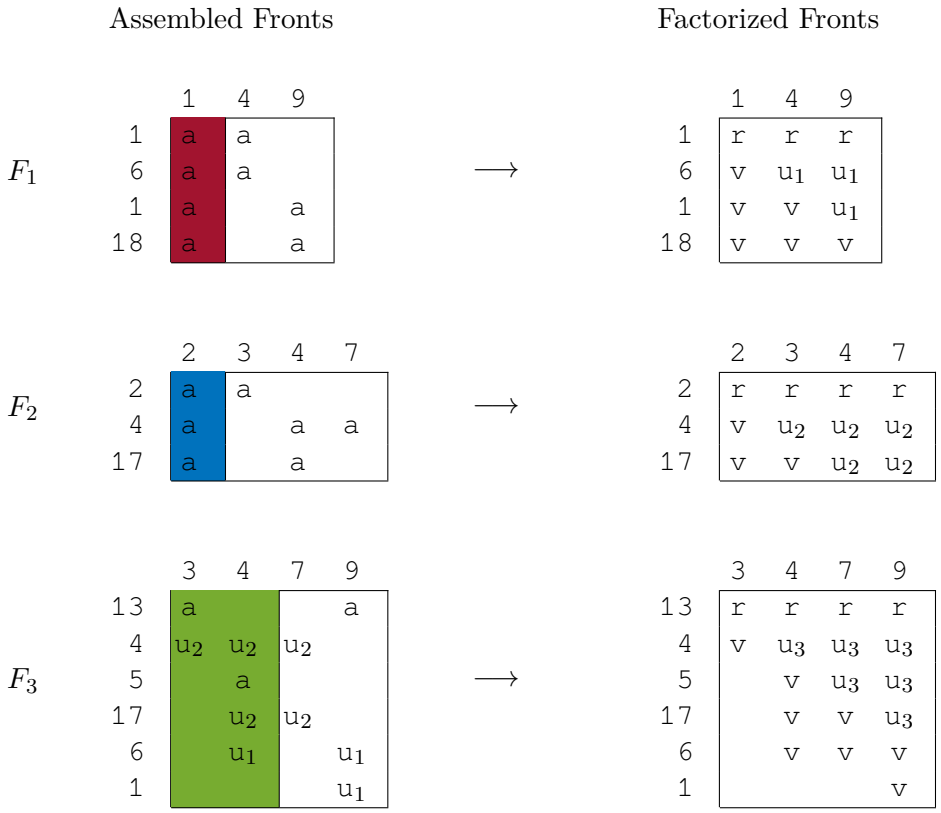


Figure 2.10: The first three steps of the multifrontal QR factorization for the matrix of Figure 2.7.

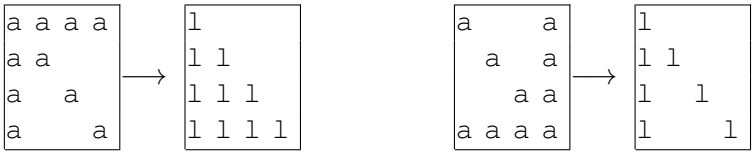


Figure 2.11: Example of matrix permutation. When factorizing the matrix on the left side, the whole structure is filled, whereas factorizing the permuted matrix on the right does not introduce any fill-in.

The first group is the one of *local methods*. The name stems from the fact that these methods proceed by selecting pivots successively but the selection of each pivot is based on a limited information which does not allow to foresee what the effects of these selection will be in the remainder of the process. One such method, probably the most well known, is the *Minimum Degree* one. Remember the elimination graphs process described in Section 2.2.1: when a variable is eliminated, the corresponding node is retired from the adjacency graph and edges are added to connect its neighbors that weren't already connected. These newly added edges model he apparition of fill-in coefficients. Therefore, one might expect that the fill-in can be reduced by selecting, at each elimination step, the node whose degree (i.e., number of neighbors) is smallest. This is the idea at the base of the Minimum Degree method. The Minimum Fill is a similar method where the selection of the pivots is based on a criterion that more accurately models the amount of fill-in created by an elimination step. Many variants of the Minimum Degree method have been proposed in the literature that aim at reducing its execution time or memory

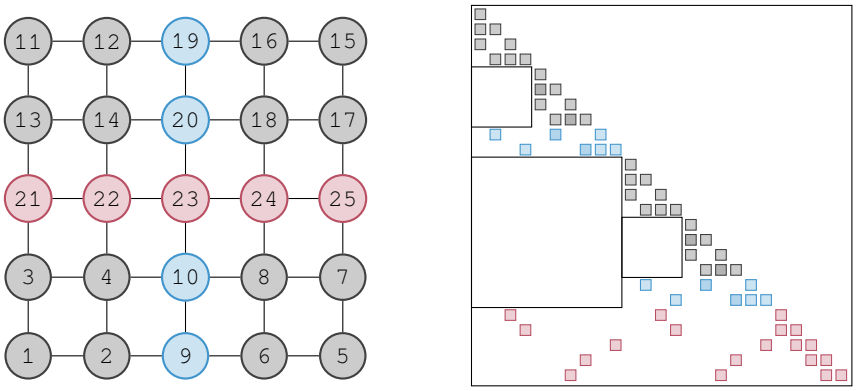


Figure 2.12: An example of nested dissection on a 5×5 grid. The topmost separator cuts the domain horizontally in two halves which are then cut vertically by other separators. The resulting permuted matrix is shown on the right side.

consumption; Approximated Minimum Degree by Amestoy et al. [11] (AMD) and Multiple Minimum Degree by Liu [117] (MMD) are among the most commonly known and used. Also, note that a variant of AMD, called COLAMD, has been developed by Davis et al. [53] to compute an AMD ordering of $A^T A$ without explicitly building it.

The second group is those of the *global methods* which rely on the knowledge of the whole adjacency graph to compute a fill-reducing matrix permutation. To this category belong the well known and widely used *nested dissection* method which was conceived by George [74]. This technique is based on the concept of *separator*: a subset of nodes such that, if removed, the domain is split in two parts or subdomains. Once a separator is identified, the matrix is permuted in such a way that all the variables in one subdomain are eliminated first, all the variables in the other second and all the variables in the separator are eliminated last. Because all the paths that connect the two resulting subdomains necessarily include nodes in the separator which are eliminated last, by Theorem 2.5, there cannot be fill-in in the block that connects the two subdomains. The process is then applied recursively in the two subdomains. Figure 2.12 shows two steps of nested dissection on a 5×5 grid and the resulting permuted matrix; the zero blocks that connect the subdomains are shown by the empty squares. This process produces a *separators tree* which can also be used as an assembly tree because all the nodes in a separator are associated to columns of L that necessarily have the same structure and, therefore, form a supernode. George [74] showed that, when nested dissection is used, the complexity of the factorization is of $O(N^3)$ and $O(N^6)$ for a square and cubic domain of size N, respectively; the number of nonzeros in the factors are $O(N^2 log(N))$ and $O(N^4)$, respectively. We sketch the proof of this result assuming that the input sparse matrix has been permuted using a nested dissection method with *cross-shaped* separators, as in George's paper, shown in Figure 2.13 for the 2D case. This choice allows for easier computations because the size of a separator is divided by 2^{d-1} and the number of nodes multiplied by 2^d moving from one level of the tree to the one below; here d is the dimension of the domain, e.g., 2 for a bi-dimensional domain (as in Figure 2.13) and 3 for a three-dimensional one. Assembly operations only account for a low-order term and thus will be ignored below. At each level ℓ of the separators tree we have $(2^d)^\ell$ fronts of order $O((\frac{N}{2^\ell})^{d-1})$, for ℓ ranging from 0 to $L = \log_2(N)$. Therefore, the flop complexity $\mathcal{C}(N)$ to factorize a sparse matrix of order

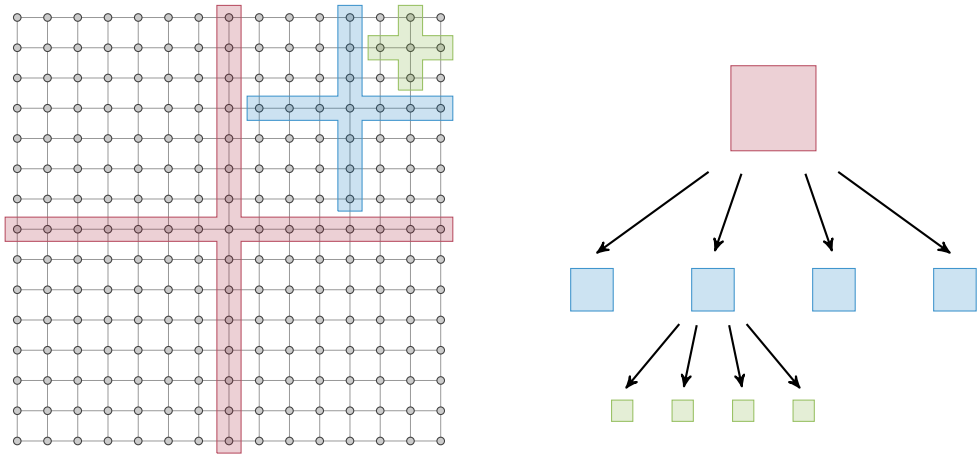


Figure 2.13: On the left, three level of nested dissection with cross-shaped separators on a 2D square domain. On the right, the corresponding separators/assembly tree.

N^d is

$$\mathcal{C}(N) = \sum_{\ell=0}^{L} (2^d)^\ell ((\frac{N}{2^\ell})^{d-1})^3 = N^{3d-3} = \begin{cases} N^3 & \text{if } d = 2 \\ N^6 & \text{if } d = 3 \end{cases} \tag{2.21}$$

We similarly compute the factor size complexity:

$$\mathcal{M}(N) = \sum_{\ell=0}^{L} (2^d)^\ell ((\frac{N}{2^\ell})^{d-1})^2 = \begin{cases} N^2 log(N) & \text{if } d = 2 \\ N^4 & \text{if } d = 3 \end{cases} \tag{2.22}$$

George et al. [77] showed that the same asymptotic complexity can be achieved using a QR factorization although experimental results show that this is more expensive than Cholesky or LU by a rather substantial constant factor.

Global and local methods can be combined to reduce the execution time or improve the quality of the final ordering; for instance, nested dissection can be applied until a prescribed subdomain size is reached and then an ordering such as AMD is used locally on each subdomain. This is the choice of modern ordering tools such as Metis [105] or Scotch [133] which are probably the most commonly used ordering tools especially on large size problems.

2.2.3.2 Pivoting

Pivoting in the multifrontal method, or in sparse factorizations in general, is much more complicated than in dense factorizations. Within a frontal matrix, pivots can only be selected inside the block of fully summed variables, i.e., the top-left block because the remaining rows are not fully summed. However, eligible pivots may be unsatisfactory because too small with respect to other coefficients in the unassembled rows. When this is the case, the bad pivots can be moved at the end of the fully summed block and their elimination postponed; if, when they are reached, these pivots are still unsatisfactory, their elimination can be delayed to the parent node. This technique, commonly referred to as *delayed pivoting*, makes the multifrontal method practically stable but may severely harm performance. It must be noted that, when pivots are delayed, the contribution block of the front they belong to and the fully-summed block of the parent are augmented by as many rows and columns as the delayed pivots. This has a twofold effect. First, it adds fill-in

because the delayed rows and columns have to be padded with zeros in order to conform to the parent indices and be assembled therein. Second, it changes the computational pattern of the multifrontal method in a way which is not possible to predict just by looking at the structure of the problem; this makes it difficult to model the workload and arrange the computations consequently (see more details in Section 2.2.3.5) and make the use of dynamic data structures necessary in order to accommodate for the occurrence of delayed pivots. For this reason the pivot selection criterion is often relaxed by means of a technique called *threshold pivoting*: at step k, a pivot is accepted if it satisfies the condition

$$a_{k,k} \geq u \max_{i=k:n} |a_{i,k}|.$$

This implies that growth in a single step of Gaussian Elimination is limited to $1 + 1/u$. This technique allows for limiting the number of delayed pivots and thus for having a workload that better matches what can be predicted through a symbolic analysis.

In other practical approaches, pivoting is completely avoided. One such method is the so called *static pivoting* [112] where pivot $a_{k,k}$ is replaced with $\epsilon\|A\|$, ϵ being the machine precision, if it falls below a certain threshold. This corresponds to solving a slightly and selectively perturbed system. The static pivoting technique is not as robust as the delayed pivoting technique but works well in many practical cases.

The same issue can be encountered in the multifrontal QR factorization of rank-deficient matrices. A column-pivoting technique, such as the one discussed in Section 2.1.4.3, can be applied within a frontal matrix; the pivoting has to be restricted to the fully summed columns and, if needed, columns whose norm falls below a given threshold can be postponed to the parent front. Unfortunately, in the multifrontal QR factorization, pivoting is even more harmful for performance than in Gaussian Elimination because it can completely destroy the staircase structure of fronts which may result in an excessive fill-in growth. Pierce et al. [135] proposed a sparse, multifrontal QR method with column pivoting where deficiencies are identified based on an incremental condition number estimation.

Heath [98], instead, proposed a method for computing the QR factorization of sparse matrices which does not require pivoting: when a diagonal coefficient of R falls below a certain threshold, the whole row of R is zeroed out using Givens rotations. This technique, originally conceived for a QR factorization based on row-wise Givens rotations, was later extended to the Householder multifrontal QR method by Davis [55].

Another technique which does not require pivoting is the Tikhonov regularization [152] which consists in appending a matrix τD, with τ being a scalar and D a diagonal matrix, to the original matrix A. If $\tau > 0$, the least squares problem is thus transformed in the full-rank problem

$$\min_x \left\| \begin{bmatrix} A \\ \tau D \end{bmatrix} x - \begin{bmatrix} b \\ 0 \end{bmatrix} \right\|_2 = \min_x \|Ax - b\|_2^2 + \tau^2 \|Dx\|_2^2$$

whose solution can be made close to that of the original rank-deficient problem by conveniently choosing τ and D. The regularized problem can be solved by means of a standard QR factorization although this introduces additional fill-in and computations. For this reason Avron et al. [27] proposed a technique where rows are appended to regularize only linearly dependent columns.

$PA = LU$	$A = LL^T$	$A = QR$	$A = R^TQ^T$	Order
$Ly = Pb$	$Ly = b$	$y = Q^Tb$	$R^Ty = b$	Bottom-up
$Ux = y$	$L^Tx = y$	$Rx = y$	$x = Qy$	Top-down

Table 2.1: Matrix factorizations with related solve operations and tree traversal order.

2.2.3.3 Unsymmetric methods

In the previous sections and in the rest of this document we have assumed that, for the sparse Gaussian Elimination, the A matrix is structure-symmetric. Note that, when this is not the case, the structure of A can be padded with explicit zeros and the symbolic analysis can be performed on the structure of $A + A^T$. This approach, however, can have serious limitations on problems that are strongly unsymmetric.

Various approaches have been proposed to comply with unsymmetry.

Gilbert et al. [81] proposed a method where the structure of the factors is characterized by a couple of DAGs, called *elimination DAGs* or *edags* (as opposed to the elimination tree or *etree* of the symmetric case) which are obtained from the transitive reductions of the directed graphs of L and U, $\overrightarrow{\mathcal{G}}(L)$ and $\overrightarrow{\mathcal{G}}(U)$.

In a more recently proposed method, Eisenstat et al. [68] replace these edags with a tree: $j > i$ is the parent of i if it is the node with smallest index such that there is a path from i to j in $\overrightarrow{\mathcal{G}}(L)$ and back from j to i in $\overrightarrow{\mathcal{G}}(U)$. In a separate paper [69] they describe methods to compute this tree starting from the structure of A and how it can be used to compute the structure of the factors.

Amestoy et al. [17] proposed a method which relies on the symbolic analysis of the $A + A^T$ matrix and the associated elimination tree. However, when frontal matrices are assembled, the method can identify entire zero-rows or zero-columns which result from the symmetrization of the matrix and skip them both in the front assembly and factorization.

Finally, it must be noted that there is a very close relation between the LU and QR factorizations. It can be proved (see, for example, the work by Gilbert et al. [79]) that the structure of the R factor and V matrix (holding the Householder reflections) resulting from the QR factorization of a square matrix A, provide an upper bound for the U and L factors, respectively. This upper bound can accommodate for any possible row permutation resulting from pivoting and, for this reason, can be very loose. Nonetheless, according to this result, it is possible to use a symbolic QR factorization and the resulting column-elimination tree to arrange the computations and storage in an unsymmetric LU factorization.

2.2.3.4 Multifrontal solve

Once the matrix factorization has been computed, the factors can be used to solve the systems against one or more right-hand sides as explained in Section 2.1. In the case of sparse linear systems, this can be achieved in a multifrontal fashion by traversing the assembly tree either in a bottom up or a top-down order as described in Table 2.1.

We describe the multifrontal solve for the forward elimination $Ly = b$; the other operations proceed in a similar way. Recall that, upon factorization of frontal matrix F_j, a set of columns of the L factor is produced

$$F_j \longrightarrow \begin{bmatrix} L_{j,j} \\ L_{j+1,j} \end{bmatrix}$$

where $L_{j,j}$ denotes triangular block associated with the fully summed variables and, with a slight abuse of notation, $L_{j+1,j}$ denotes the sub-diagonal rectangular block associated with the non fully assembled variables. The multifrontal forward elimination proceeds by assembling, at each node of the tree, a local right-hand side f which we also suppose logically split in two parts f_j and f_{j+1} conforming with the splitting of L defined above. The operations done at each node of the tree are

$$f = b_j \overset{\updownarrow}{\leftrightarrow} u_{c_1} \overset{\updownarrow}{\leftrightarrow} \cdots \overset{\updownarrow}{\leftrightarrow} u_{c_s}; \qquad L_{j,j}y_j = f_j; \qquad u_j = f_{j+1} - L_{j+1,j}y_j$$

where b_j and y_j denote all the coefficients of the right-hand side b and the solution y corresponding to the front's fully summed variables and we assume that node j has children $c_1, ..., c_s$. The first operation assembles the local right-hand side by means of extend-add operations that combine b_j with update vectors resulting from the processing of the child nodes. The second operation computes y_j by means of a triangular system solve and the third computes the update vector u_j through a simple matrix-vector product. Note that in case of multiple right-hand sides, f, b_j and all the update vectors become matrices with as many columns as the number of right-hand sides but the "extend" part of the $\overset{\updownarrow}{\leftrightarrow}$ operation only concerns rows.

The first three steps of the multifrontal forward elimination for the matrix of Figure 2.3, assuming nodes 3 and 4 of the elimination tree have been amalgamated are

$$f = \begin{bmatrix} b_1 \\ 0 \\ 0 \end{bmatrix} \longrightarrow y_1 = f_1/l_{1,1}, \quad u_1 = \begin{bmatrix} -l_{4,1}y_1 \\ -l_{9,1}y_1 \end{bmatrix}$$

$$f = \begin{bmatrix} b_2 \\ 0 \\ 0 \\ 0 \end{bmatrix} \longrightarrow y_2 = f_2/l_{2,2}, \quad u_2 = \begin{bmatrix} -l_{3,1}y_2 \\ -l_{4,1}y_2 \\ -l_{7,1}y_2 \end{bmatrix}$$

$$f = \begin{bmatrix} b_3 \\ b_4 \end{bmatrix} \overset{\updownarrow}{\leftrightarrow} u_1 \overset{\updownarrow}{\leftrightarrow} u_2 = \begin{bmatrix} b_3 - l_{3,1}y_2 \\ b_4 - l_{4,1}y_1 - l_{4,1}y_2 \\ -l_{7,1}y_2 \\ -l_{9,1}y_1 \end{bmatrix} \longrightarrow \begin{bmatrix} l_{3,3} & \\ l_{4,3} & l_{4,4} \end{bmatrix} \begin{bmatrix} y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} f_3 \\ f_4 \end{bmatrix}$$

$$\begin{bmatrix} u_7 \\ u_9 \end{bmatrix} = \begin{bmatrix} f_7 \\ f_9 \end{bmatrix} - \begin{bmatrix} l_{7,3} & l_{7,4} \\ l_{9,3} & l_{9,4} \end{bmatrix} \begin{bmatrix} y_3 \\ y_4 \end{bmatrix}$$

An interesting case is where the right-hand side is sparse. Assume, for example, that all the entries of the right-hand side are null except one. In this case, the forward elimination operation need not access the whole L factor but only part of it; specifically it only needs to do computations related to the nodes of the elimination tree that lie along the path connecting the node associated with the right-hand side nonzero and the root. This allows for considerable savings in the forward elimination. Clearly, if the right-hand side contains multiple nonzeros, then the nodes of the tree concerned by the forward elimination are given by the merge of the paths related to each one of them. By the same token, if only one nonzero of the solution vector is needed, the backward substitution only involves nodes that lie along the path that goes from the root to the node associated with the nonzero

element of the solution. Note that computing the inverse of a matrix can be achieved by solving the linear system $AM = I$ where I is the identity matrix and, therefore, $M = A^{-1}$; this implies solving the linear systems against many right-hand sides, each having only one nonzero. Moreover, if only selected entries of the inverse are needed then also the solution is sparse. Taking advantage of the sparsity of right-hand sides and/or solution (also for computing selected entries of the inverse) is a subject that has been addressed in the literature, for example by Slavova [149], Amestoy et al. [14] or Amestoy et al. [18].

2.2.3.5 A three-phases approach

Modern sparse, direct solvers combine all the theory and methods described above. Typically, they achieve the solution of a sparse linear system in three distinct steps:

1. **Analysis.** First, a fill-reducing matrix ordering is computed. The structure of the resulting permuted matrix is then analyzed in order to arrange the computations of the subsequent Factorization and Solve steps. Namely, a symbolic factorization is done in order to characterize the structure of the factors (and, consequently, of the frontal matrices) and compute the elimination and/or assembly trees. This phase may include computing an estimate of the memory consumption and of the workload which is of particular importance in a parallel setting where memory and workload distribution have to be carefully done in order to achieve a good balance. No floating-point operations are involved in this phase whose complexity and execution time is, supposedly, much smaller than the two, subsequent steps.

2. **Factorization.** This is where the actual matrix factorization takes place. In this document we have focused on the multifrontal method but other approaches exist such as the left-looking or right-looking supernodal methods (also referred to as *fan-in* and *fan-out*, respectively, especially in a parallel setting).

3. **Solve.** The solve phase is where the factors are used to solve the linear system against one or more right-hand sides. Again, in this document we have focused on the multifrontal solve technique but left or right-looking solve techniques are also commonly used.

In some applications it is required to solve the same matrix against multiple right-hand sides which are not all available at the same time. This is, for example, the case of iterative processes where one right-hand side (or a block of) is computed at each iteration based on the result of the previous one. In these cases, clearly, the analysis and factorization steps need be done only once and only the solve operation has to be repeated. Similarly, because the analysis phase only involves structural computations, if multiple problems have to be solved where only the coefficients of the matrices differ but not their position, the analysis operation can be done once for all the problems. This is shown schematically in the blocks diagram of Figure 2.14.

2.2.3.6 Sparse, direct solver packages

Most of the methods and algorithms described in the upcoming sections have been implemented within the MUMPS [13] and `qr_mumps` [J2] software packages for the purpose of evaluating experimentally their effectiveness. Those methods whose implementation has reached a satisfactory maturity and reliability have also been made freely available in recent public releases of these packages. Both these solvers rely on the multifrontal method. The first implements LU and LDL^T factorizations, is originally designed for

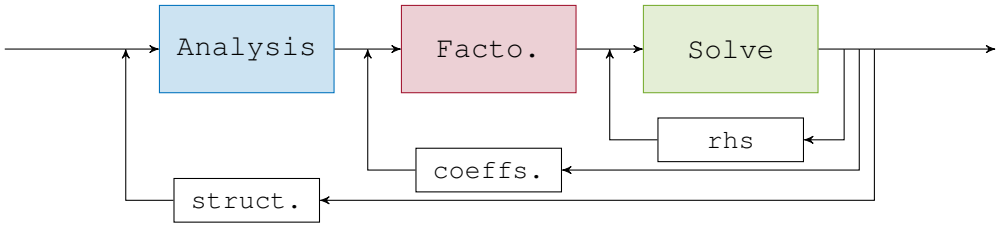


Figure 2.14: A block diagram showing the three phases of a typical sparse direct solver.

distributed memory parallel systems but can efficiently handle shared memory or hybrid ones, uses delayed pivoting which makes it very roust and reliable and provides a very wide set of features. The `qr_mumps` package, instead, is based on the QR factorization and, thus, it aims primarily at linear least-squares problems; it currently runs only on parallel shared memory systems although a development version that can also use GPUs exists (presented in Section 3.5).

Many other sparse, direct solvers are available although a comprehensive list is out of the scope of this document.

Other popular sparse direct solvers relying on the multifrontal method include those of the HSL collection such as MA41 [12], MA49 [15] or MA86 [101], UMFPACK [54], SuiteSparseQR [55] and WSMP [90].

Well known packages that, instead, rely on left or right-looking supernodal methods are SuperLU [113] and its variants SuperLU_MT and SuperLU_DIST designed for sequential, shared memory and distributed memory systems, respectively, PaStiX [99], SPOOLES [25], CHOLMOD [48] or PARDISO [141].

2.3 Supercomputer architectures

The world of computing, and particularly the world of High Performance Computing (HPC), have witnessed a substantial change at the beginning of the last decade as all the classic techniques used to improve the performance of microprocessors reached the point of diminishing returns [23]. These techniques, such as deep pipelining, speculative execution or superscalar execution, were mostly based on the use of Instruction Level Parallelism (ILP) and required higher and higher clock frequencies to the point where the processors power consumption, which grows as the cube of the clock frequency, became (or was about to become) unsustainable. This was not only true for large data or supercomputing centers but also, and even more so, for portable devices such as laptops, tablets and smartphones which have recently become very widespread. In order to address this issue, the microprocessor industry sharply turned towards a new design based on the use of Thread Level Parallelism (TLP) achieved by accommodating multiple processing units or *cores* on the same die. This led to the production of *multicore* processors that are nowadays ubiquitous. The main advantage over the previous design principles lies in the fact that the multicore design does not require an increase in the clock frequency but only implies an augmentation of the chip capacitance (i.e., the number of transistors) on which the power consumption only depends linearly. As a result, the multicore technology not only enables improvement in performance, but also reduces the power consumption: assuming a single-core processor with frequency f, a dual-core with frequency $0.75 * f$ is 50% faster and consumes 15% less energy.

Since their introduction, multicore processors have become increasingly popular and

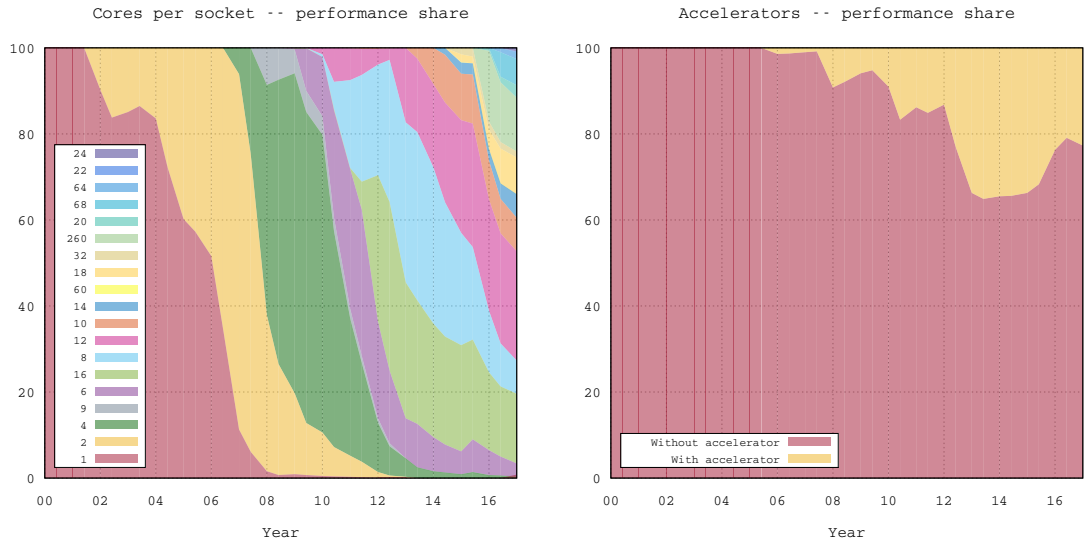


Figure 2.15: Performance share of multicore processors (on the left) and accelerators (on the right) in the Top500 list.

can be found, nowadays, in almost any device that requires some computing power. Because they allow for running multiple processes simultaneously, the introduction of multicore in high throughput computing or in desktop computing was transparent and immediately beneficial. In HPC, though, the switch to the multicore technology lead to a sharp discontinuity with the past as methods and algorithms had to be rethought and codes rewritten in order to take advantage of the added computing power through the use of TLP. Nonetheless, multicore processors have quickly become dominant and are currently used in basically all supercomputers. Figure 2.15 (*left*) shows the performance share of multicore processors in the Top500[2] list (a list of the 500 most powerful supercomputers in the world, which is updated twice per year); the figure shows that after their first appearance in the list (May 2005) multicore has quickly become the predominant technology in the Top500 list and ramped up to nearly 100% of the share in only 5 years. The figure also shows that the number of cores per socket has grown steadily over the years: a typical modern processor features between 8 and 16 cores.

In addition to an ever-increasing number of cores per socket, modern processors rely more and more on SIMD (Single Instruction Multiple Data) parallelism achieved by means of vector units. At the beginning of the years 2000, processors were equipped with moderately sized vector extensions such as SSE (Streaming SIMD Extensions, in the Intel and AMD x86 architecture) or AltiVec (in the IBM POWER architecture). These were made of a set of 128-wide vector registers that could host two double precision, real coefficients (or four single-precision, real ones) and a set of instructions to perform the same operation on all the coefficients in a vector register at once. In 2011 Intel introduced the AVX (Advanced Vector eXtensions) with their Sandy Bridge family of processors which are based on 256-bit vector units. This was later extended by AVX-2.0 which added the Fused Multiply-Add operation. The latest evolution of this trend is the AVX-512 extensions which brought the length of vector units to 512 bits. As a result, a modern processor can perform up to 32 double-precision, real operations per clock cycle:

[2] http://top500.org

8(# data in a SIMD vector) × 2(FMA) × 2(dual-issue).

All this technological advances have allowed for increasing the peak performance while keeping under control the power consumption as shown in Figure 2.17 (*left*) that plots the evolution over the last few year of the CPU speed over TDP (Thermal Design Power) ratio (measured in Gflop/s divided by Watts) for some of the Intel flagship processors. The latest point in the curve is related to the Skyline 8176 processor released in Q3 2017; this processor has 28 cores clocked at 2.1 GHz with AVX-512 extensions for a massive peak performance of 1.88 Tflop/s (for double-precision, real computations), has a TDP of 165 W and a memory bandwidth of 119.21 GB/s. As a reference, the first supercomputer capable of passing the 1 Tflop/s mark (1.3, exactly) was the ASCI Red which was ranked number one on the Top500 list of June 2000. This computer was equipped with 9298 processors, consumed 850 KW and occupied a surface of 150 m^2.

Intel has pushed the envelope of multicore technologies even further with the Xeon Phi processors. The latest addition to this family is the KNL (Knights Landing) processor. Although it shares some features with accelerating boards (see below), the KNL has a x86 architecture and therefore can run any application that was developed for conventional processors, including the operating system, without the need for dedicated programming languages or paradigms. It can have as many as 72 cores clocked at a relatively low frequency (1.3 to 1.5 GHz) with AVX-512 extensions and can reach a peak performance of 3.45 Tflop/s. The bandwidth towards main memory is of 115 GB/s. One distinctive feature of the KNL is that the cores can be arranged according to different pre-defined schemes to resemble more or less to a SMP (Symmetric Multi-Processor) or a NUMA (Non-Uniform Memory Access) system.

Around the same period as the introduction of multicore processors, the use of *accelerators* or *coprocessors* started gaining the interest of the HPC community. Although this idea was not new (for example FPGA boards were previously used as coprocessors), it was revamped thanks to the possibility of using extremely efficient commodity hardware for accelerating scientific applications. One such example is the Cell processor [100] produced by the STI consortium (formed by IBM, Toshiba and Sony) from 2005 to 2009. The Cell was used as an accelerator board on the Roadrunner supercomputer installed at the Los Alamos National Labrador (USA) which was ranked #1 in the Top500 list of November 2008. The use of accelerators for HPC scientific computing, however, gained a very widespread popularity with the advent of *General Purpose GPU* computing. Specifically designed for image processing operations, Graphical Processing Units (GPUs) offer a massive computing power which is easily accessible for highly data parallel applications (image processing often consists in repeating the same operation on a large number of pixels). This led researchers to think that these devices could be employed to accelerate scientific computing applications, especially those based on the use of operations with a very regular behaviour and data access pattern, such as dense linear algebra ones. In the last few years, GPGPU has become extremely popular in scientific computing and is employed in a very wide range of applications, not only dense linear algebra. This widespread use of GPU accelerators was also eased by the fact that GPUs, which were very limited in computing capabilities and difficult to program, have become, over the years, suited to a much wider range of applications and much easier to program thanks to the development of specific high-level programming languages and development kits. Figure 2.15 (*right*) shows the performance share of supercomputers equipped with accelerators. Although some GPU accelerators are also produced by AMD, currently the most widely used ones are produced by Nvidia. Figure 2.16 shows a block diagram of the architecture of a recent Nvidia GPU device, the P100 of the Tesla family. This board is equipped with 56 Streaming Multiprocessors (SMX), each containing 64 single precision cores and 32 double

precision ones for a peak performance of 5.3 (10.6) Tflop/s for double (single) precision computations.

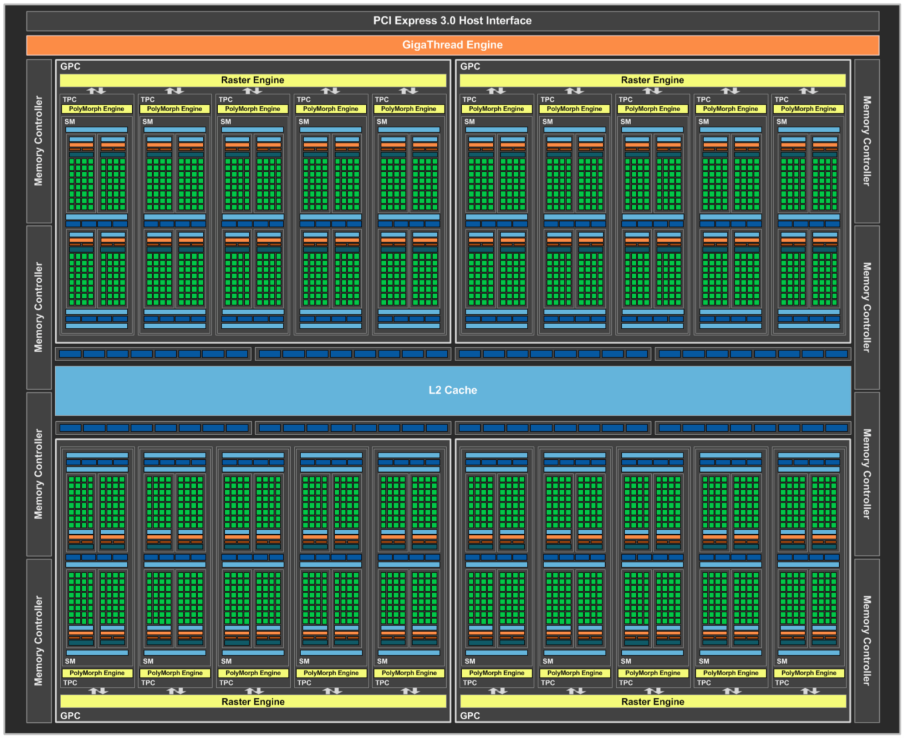


Figure 2.16: Block diagram of the Nvidia Tesla P100 GPU.

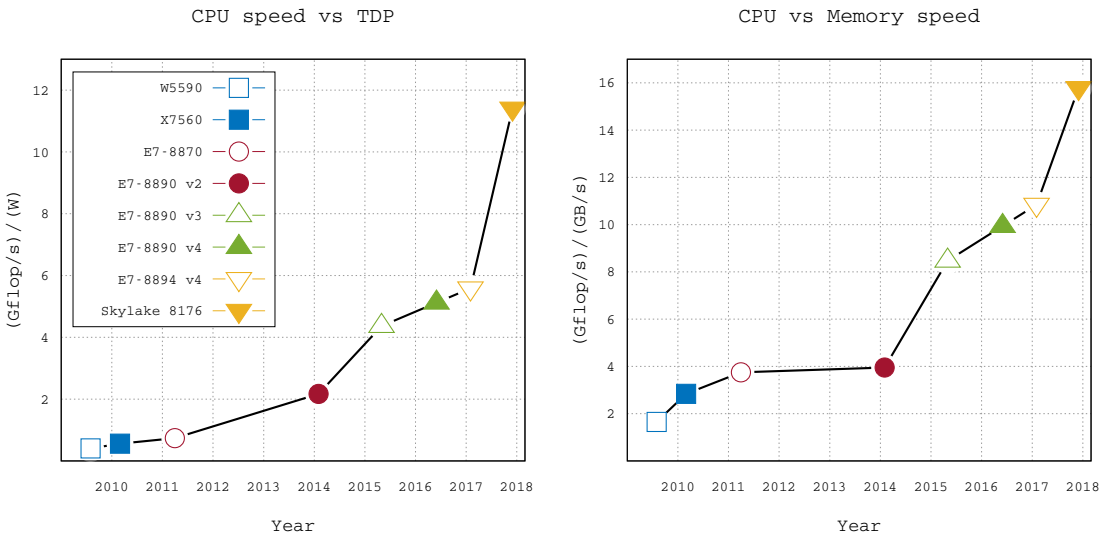


Figure 2.17: Evolution of the ratio between CPU speed (in double-precision Gflop/s) Thermal Design Power (in W) and memory speed (in GB/s). Data is taken from Intel's website.

Although processors performance has continued to increase considerably and steadily

over the last years, memories have not managed to keep up with these improvements. This is a long lasting problem, much older than multicore technologies; these, however, have made it even worse because all the cores of a processor share the same memory and the same bus to move data in and out of it. Figure 2.17 (*right*) shows the evolution of the CPU over memory performance (measured in Gflop/s divided by GB/s) for the same processors and time frame as for the plot on the left side. In the case of operations with a high arithmetic intensity, such as Level-3 BLAS operations, this problem can be circumvented through a careful reuse of data in cache memories but for others (think about a sparse matrix-vector product which cannot make any use of caches) it makes it impossible to take advantage of the full potential of modern multicore processors. For this reason, memory producers have turned their attention towards *3D stacked* technologies. Tree-dimensional integration allows for stacking the memory directly on top of the processor; as a result the wire delay between the two is considerably reduced which translates into higher bandwidths and lower latencies. HBM (High Bandwidth Memory) is one among the various standards that have been proposed for 3D stacked memories. The Tesla P100 board described above is equipped with a local HBM2 (an evolution of the HBM standard) memory of 16 GB that has a bandwidth of 732 GB/s. The Intel KNL processor is also equipped with a local 16GB fast MCDRAM (a variant of the HBM standard) memory with a bandwidth of more then 400 GB/s, as reported by the constructor. Another distinctive feature of the KNL processor is that this memory can also be configured according to different schemes: it can be configured as an explicitly addressed memory, in which case is up to the programmer to move data in and out of it, or as an additional cache level, in which case its use is completely transparent to the programmer, or as an hybrid of these two configurations.

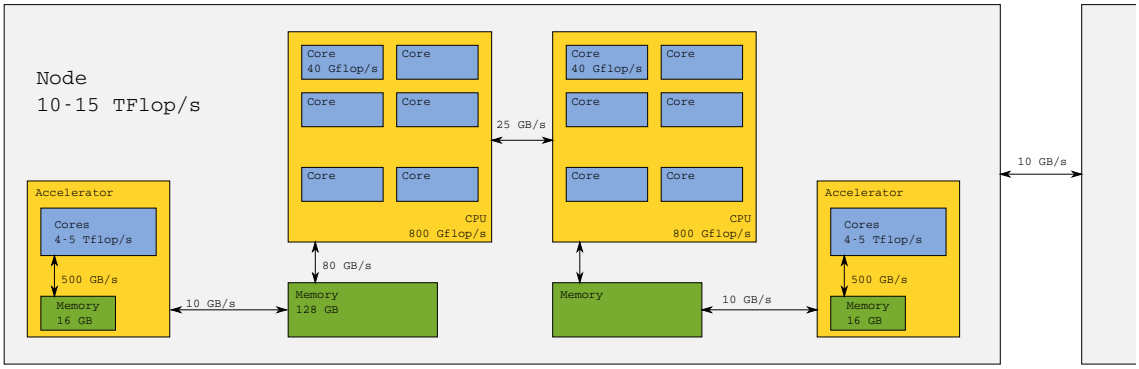


Figure 2.18: Illustration of a typical HPC computing platform architecture.

Figure 2.18 shows a typical configuration of a modern HPC computing platform. This is formed by multiple (up to thousands) nodes connected through a high-speed network; each node may include multiple processors, each connected with a NUMA memory module. A node may also be equipped with one or more accelerators. Please note that the figure reports indicative values for performance, bandwidths and memory capacities and do not refer to any specific device. The figure shows that modern HPC platforms are based on extremely heterogeneous architectures as they employ processing units with different performance, memories with different capacities and interconnects with different latencies and bandwidths. In this context, achieving performance and performance portability is an extremely challenging task as different applications may benefit differently from all the involved technologies.

2.4 Runtime systems and the STF model

In the diverse landscape of supercomputing architectures described above, and because of the increasing complexity of algorithms for scientific computing, traditional methods for implementing parallel applications based on a blend of different technologies (for instance, MPI, OpenMP and CUDA), may fall short in achieving high performance, scalability, performance portability and code maintainability. As a result, the need has recently emerged for novel parallel programming models and tools capable of

- addressing in a consistent way the diversity and heterogeneity of modern platforms,
- relieving the programmer from the burden of dealing with the low-level details of the architecture
- achieving code and performance portability, i.e., making a code portable across a wide range of architectures (with, possibly, minor modifications) while keeping a satisfactory level of performance.

Modern *runtime systems* (or, simply, *runtimes*) are designed to address this demand. A runtime is conceived as a software layer that makes an abstraction of the underlying computing platform by hiding most of the low-level architectural details and provide the programmer with a unified programming interface, which is portable across a wide range of architectures, to represent his workload. The runtime is then in charge of deploying and executing the workload on the computing platform. Generally speaking, the following four components can be found in a modern runtime system:

1. A programming model. The commonly used approach to program supercomputing platforms relies on a mixture of different technologies; this can include, for example, MPI for communications among different nodes, OpenMP for taking advantage of multicores and CUDA for GPUs. Modern runtimes, instead, provide a programming model which allows the programmer to express his workload in an abstract way, independently of the details of the underlying architecture. Obviously, one such programming model may have more or less expressivity or may be more or less suited to a specific class of algorithms.

2. A scheduler. When multiple execution units are available, the scheduler is in charge of deciding when and where a specific *task* (a unit of work) has to be executed. This has to account for the different speeds and capabilities of the available working units as well as their distance in order to cope with data transfers. When possible, the characteristics of the workload can also be considered such as performance models for tasks; this information can either be provided by the programmer or automatically built by the runtime. The set of rules that is used to take this decision is what we call a *scheduling policy*. Modern runtime systems usually come with a set of pre-defined scheduling policies (such as work-stealing [22], Minimum Completion Time [153]) suited for the most common architectures but may also provide the necessary API for the programmer to implement his own scheduling policy.

3. Drivers. Each of these modules is in charge of triggering the execution of a task on an execution unit. Therefore, when a new type of execution unit appears, the runtime developer has to write a new driver to pilot it.

4. Memory manager. A modern computing platform may have multiple memories each with its own address space; this is, for example, the case of clusters of nodes or of systems equipped with GPU boards. Therefore, when the scheduler decides to execute a task on a unit which is far away from the memory that holds the corresponding data, this has to be transferred for the task to execute. This is the role of the memory manager. Varying degrees of optimization can be implemented in a memory manager. For example, if the same data is only read by multiple tasks running on units associated with different memories, multiple copies of it may be generated by the memory manager in order to avoid unnecessary communications; in this case the memory manager has to handle the coherency among these copies.

In recent years, the use of runtime systems has become increasingly popular. Most of these runtimes rely on an interface where the workload has to be expressed as a Directed Acyclic Graph (DAG) where nodes represent tasks and edges the dependencies among them. The runtime programming model is thus just a mean of expressing this DAG. Among the most well known and used models provided by recent runtime systems is the *Sequential Task Flow (STF)* one. The STF model simply consists of submitting a sequence of tasks through a non blocking function call that delegates the execution of the task to the runtime system; this submission describes how the tasks accesses the data, i.e., whether in read, write mode or both. Upon submission, the runtime system adds the task to the current DAG along with its dependencies which are automatically computed through data dependency analysis [10]:

- for all data accessed in read mode: the submitted task will be dependent on all previously submitted tasks that access the same data in write mode;
- for all data accessed in write mode: the submitted task will be dependent on all previously submitted tasks that access the same data in read or write mode.

The actual execution of the task is then postponed to the moment when its dependencies are satisfied. This paradigm is also sometimes referred to as *Superscalar* since it mimics the functioning of superscalar processors where instructions are issued sequentially from a single stream but can actually be executed in a different order and, possibly, in parallel depending on their mutual dependencies. Figure 2.19 shows a dummy sequential algorithm

```
1 call f(x,y)
  call g(x,u)
3 call h(y,z)
```

```
1 call submit(f,x:RW,y:RW)
  call submit(g,x:R,u:RW)
3 call submit(h,y:R,z:RW)
  call wait_tasks_completion()
```

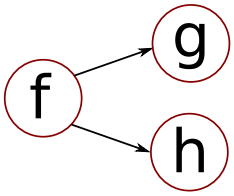


Figure 2.19: Pseudo-code for a dummy sequential algorithm (*left*), corresponding STF version (*center*) and subsequent DAG (*right*).

and its corresponding STF version. Instead of making three function calls (`f`, g, h), the equivalent STF submits the three corresponding tasks. The data onto which these functions operate as well as their access mode (Read, Write or Read/Write) are also specified. Because task `g` accesses data `x` after task `f` has accessed it in Write mode, the runtime infers a dependency between tasks `f` and `g`. Similarly a dependency is inferred between tasks `f` and `h` due to data `y`. Figure 2.19 (*right*) shows the DAG corresponding to this STF dummy code. In the STF model, one thread is in charge of submitting the tasks;

we refer to this thread as the *master thread*. The execution of tasks is instead achieved by *worker threads*. The function called at the end of the STF pseudo-code is simply a barrier that prevents the master thread from continuing until all of the submitted tasks are executed. Note that, in principle, the master thread can also act as a worker; whether this is possible or not depends on the design choices of the runtime system developers.

Many runtime systems [26, 28, 108] support the STF paradigm such as OpenMP (since standard 4.0) which provides the `task` directive to submit tasks with the `depend` clause to declare their data access mode. For the work presented in Chapter 3 we chose to rely on StarPU [26] as it provides a very wide set of features that allows, most importantly, for a better control of the tasks scheduling policy and because it supports accelerators and distributed memory parallelism which will be addressed in future work.

Other popular runtimes, such as Parsec by Bosilca et al. [38], rely on a programming model called *Parametrized Task Graph*. In this approach the dataflow and, thus, the DAG, is described by means of a dedicated language which allows for defining tasks and rules that are used to pass data from one task to the others. As a consequence, the DAG is never explicitly and entirely built but progressively unrolled by means of these rules; this provides a better scalability of the runtime which, however, comes at the cost of a much higher programming effort.

Chapter 3

Parallelism and performance scalability

Sparse computations are well known for being hard to parallelize on shared-memory, multicore systems. This is due to the fact that the efficiency of many sparse operations, such as the sparse matrix-vector product, is limited by the speed of the memory system. This is not the case for the multifrontal method; since computations are performed as operations on dense matrices, a favorable ratio between memory accesses and computations can be achieved which reduces the utilization of the memory system and opens opportunities for multithreaded, parallel execution. Moreover, sparse direct methods lend themselves naturally to parallelism because they benefit from two sources of concurrency. The first stems from the matrix sparsity and is commonly referred to as **tree parallelism**: fronts lying on distinct branches of the assembly tree are independent and can be processed concurrently. The second is inherent to the dense linear algebra operations performed on each frontal matrix and is commonly referred to as **node parallelism**: if a frontal matrix is big enough, multiple processes can be used to assemble and factorize it.

The use of both these types of parallelism is essential to achieving good scalability with parallel implementations of sparse, direct solvers. This is because they are complementary: at the bottom of the tree, node parallelism is scarce because fronts are small and tree parallelism is abundant because there are many branches; inversely, at the top of the tree fronts are large and provide much node parallelism but the tree is narrow.

Let us assume that the execution time of a sparse matrix factorization is proportional to the number of performed floating point operations. For the case of a problem from a regular square (2D) or cubic (3D) domain of size N where nested dissection is used, the sequential execution time is, therefore, $O(N^3)$ and $O(N^6)$, respectively; this is derived from the result of Section 2.2.3.1. Note that here, not only we have assumed that the assemblies account for a lower-order term in the flop count, but also that their execution time is negligible which is not necessarily the case in practice because these operations are extremely inefficient and involve many communications and indirect addressing.

For a parallel execution, a lower-bound for the running time can be computed by modifying Equation (2.21) in order to take node and tree parallelism into account:

$$\mathcal{T} = \sum_{\ell=0}^{L} \left(\left(2^d\right)^\ell \right)^\alpha \left(\left(\frac{N}{2^\ell} \right)^{d-1} \right)^\beta = N^{\alpha d - \alpha} \sum_{\ell=0}^{L} 2^{\beta l d - \alpha l d + \alpha l} \tag{3.1}$$

The α and β exponents define how tree and node parallelism, respectively, affect the execution time. When tree parallelism is used, the best possible execution time is lower-

	Params	2D	3D
$\mathcal{T}_{seq}$	$\alpha = 1, \quad \beta = 3$	N^3	N^6
$\mathcal{T}_{tree}$	$\alpha = 0, \quad \beta = 3$	N^3	N^6
$\mathcal{T}_{node}$	$\alpha = 1, \quad \beta = 1$	N^2	N^3
	$\alpha = 1, \quad \beta = 2$	$N^2 log(N)$	N^4
$\mathcal{T}_{tree+node}$	$\alpha = 0, \quad \beta = 1$	N	N^2
	$\alpha = 0, \quad \beta = 2$	N^2	N^4

Table 3.1: Execution time lower bounds for the sequential and parallel factorization of sparse matrices from a square (2D) or cubic (3D) domain of size N.

bounded by the time it takes to traverse the longest branch of the assembly tree. Because in our case all the branches are equal, this lower-bound is obtained by setting $\alpha = 0$ which amounts to traversing any branch of the tree. When, instead, node parallelism is used, each single assembly tree node will be factorized by multiple processes and, thus, faster. How faster is defined by the value of β and depends on the type of parallel front factorization method being used: it can easily be proved that the execution time for the parallel Cholesky factorization of a dense front of size m is $\Omega(m)$, whereas for the classical blocked LU factorization with partial pivoting (see Section 2.1.4.1) or the Householder QR factorization (see Section 2.1.4.2) is $\Omega(m^2)$ (more details on this will be provided below).

Table 3.1 reports lower bounds for the execution time of the factorization of sparse matrices from a square (2D) or cubic (3D) domain of size N derived from Equation (3.1); these are computed for a sequential execution ($\alpha = 1, \quad \beta = 3$) as well as for parallel executions where only tree ($\alpha = 0$), only node ($\beta = 1, \ 2$) and both tree and node parallelism are used. Several interesting observations can be made on these results. First, tree parallelism alone does not bring any asymptotic improvement to the execution time. This can be easily understood considering that the sequential execution time in both the 2D and 3D cases is dominated by the time for factorizing the topmost front, where no tree parallelism is available. For the same reason, node parallelism alone can actually reduce the execution time by a factor that depends on how parallel the dense front factorization is (i.e. $\beta = 1$ or 2). In the case of Cholesky factorization ($\beta = 1$) node parallelism reduces the relative weight of the topmost nodes by a factor such that adding tree parallelism on top of it can further reduce the execution time. This also holds for LU with partial pivoting and QR ($\beta = 2$) but only in the 2D case whereas in the 3D case where tree parallelism does not bring any actual improvement even when node parallelism is used.

In view of the above analysis, it is clear that any improvement on the parallelization of dense LU and QR factorizations will considerably benefit the scalability of sparse direct solvers. Section 3.4.1 presents dense LU and QR factorization algorithms whose parallel execution time lower bound is $O(m)$ for a dense matrix of size m. The use of these techniques, referred to as *tiled* or *Communication Avoiding* methods, within sparse direct solvers is discussed in Section 3.4. This leads to complex parallel algorithms that are hard to implement in an efficient, scalable and portable way. This is especially the case considered that modern supercomputing architectures heavily rely on thread-level parallelism to achieve high performance and scalability; this demands for parallel programming models that can handle high levels of concurrency in order to feed all the available processing units. Sections 3.1 and 3.3 discuss the use of *task-based parallelism* and runtime systems for achieving efficient parallel implementations of these methods on multicore platforms.

Section 3.5 discusses how this implementation can be extended to use heterogeneous platforms equipped with GPUs by means of appropriate data and tasks partitioning as well as scheduling methods. Finally, in Section 3.6 we present a performance analysis approach for evaluating the efficiency of parallel codes on heterogeneous platforms.

3.1 Task-based multifrontal QR for manycore systems

Because of the heavy use of Thread Level Parallelism, which is at the base of the multicore technology, and the availability of increasing levels of concurrency, the task-based parallel programming model has known a renewed popularity. In such a model, a workload is expressed as a Directed Acyclic Graph (DAG) where nodes represent *tasks* (i.e., units of work) and edges the dependencies among them; this representation can be explicit and thus the DAG is entirely generated and stored in memory or implicit, if possible, by means of a set of rules that define the tasks and their mutual dependencies. Based on this representation, an *asynchronous* execution approach can be employed where a ready task (i.e., one for which all the dependencies are satisfied) is executed as soon as possible, depending on the availability of a suitable processing unit. This allows for making a more effective use of all the available processing units and for easily reacting to dynamic variations of the workload or inaccuracies of performance models, which are quite likely given the complex and heterogeneous nature of modern computing platforms.

The classical approach to shared-memory parallelization of QR multifrontal solvers [15, 55, 126] is based on a complete separation of the two sources of concurrency described above. A task-based approach is commonly employed to express tree parallelism: the assembly tree is actually a DAG of tasks, where each task corresponds to the assembly and factorization of a frontal matrix. Node parallelism, instead, is commonly delegated to multithreaded BLAS or LAPACK libraries. Although this approach works reasonably well for a limited number of cores or processors, it suffers scalability problems mostly due to two factors:

- separation of tree and node parallelism: the degree of concurrency in both types of parallelism changes during the bottom-up traversal of the tree; fronts are relatively small at leaf nodes of the assembly tree and grow bigger towards the root node. On the other hand, tree parallelism provides a high level of concurrency at the bottom of the tree and only a little at the top part where the tree shrinks towards the root node. Since the node parallelism is delegated to an external multithreaded BLAS library, the number of threads dedicated to node parallelism and to tree parallelism has to be fixed before the execution of the factorization. Thus, a thread configuration that may be optimal for the bottom part of the tree will result in a poor parallelization of the top part and vice versa. Although some recent parallel BLAS libraries (for example, Intel MKL) allow for changing the numbers of threads dynamically at runtime, it would require an accurate performance modeling and a rigid thread-to-front mapping in order to keep all the cores working at any time. Relying on some specific BLAS library could, moreover, limit the portability of the code.
- synchronizations: the assembly of a front is an atomic operation. This inevitably introduces synchronizations that limit the concurrency level in the multifrontal factorization; most importantly, it is not possible to start working on a front until all of its children have been fully factorized.

The limitations of the classical approach discussed above can be overcome by employing a task-based approach for exploiting both tree and node parallelism, which is achieved

through a fine-grained partitioning of data and operations. A block-column partitioning of the fronts is applied as shown in Figure 3.1 (*left*) and five elementary operations defined:

1. `activate`: the activation of a frontal matrix corresponds to computing its structure (row/column indices, staircase structure, etc.), allocating its memory, filling it with zeros and assembling the nonzero coefficients from the corresponding rows of the input sparse matrix;

2. `_geqrt`: this operation, also referred to as *panel*, amounts to computing the QR factorization of a block-column; Figure 3.1 (*middle*) shows the data modified when the panel operation is executed on the first block-column;

3. `_gemqrt`: this operation, also referred to as update, updates a block-column with respect to a panel and corresponds to applying to the block-column the Householder reflectors resulting from the panel reduction; Figure 3.1 (*right*) shows the coefficients read and modified when the third block-column is `update`'d with respect to the first panel;

4. `assemble`: for a block-column, assembles the corresponding part of the contribution block into the parent node (if it exists);

5. `deactivate`: stores the coefficients of the R and H factors aside and deallocates the memory needed for the frontal matrix storage;

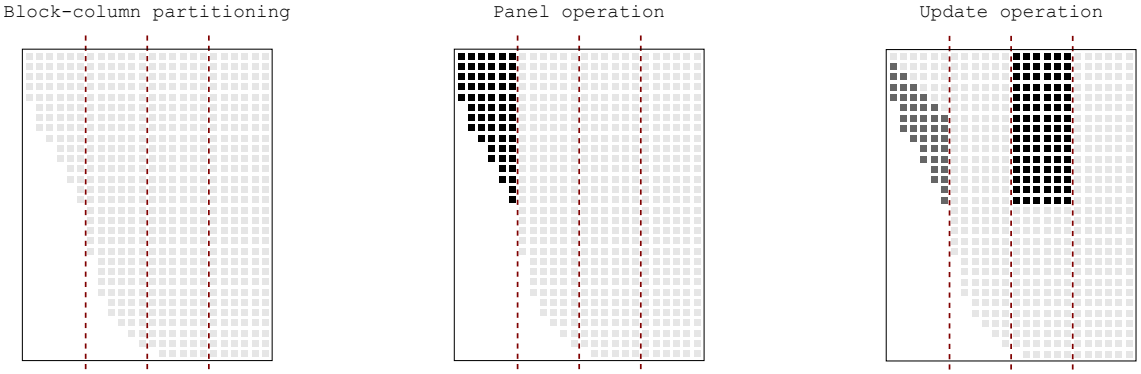


Figure 3.1: Block-column partitioning of a frontal matrix (*left*) and panel and update operations pattern (*middle* and *right*, respectively); dark gray coefficients represent data read by an operation while black coefficients represent written data.

Based on this decomposition, a sequential pseudo-code illustrating the multifrontal QR factorization is reported in Figure 3.2. This pseudo-code loops over all the fronts in the elimination tree in a bottom-up order. When a front `f` is visited, first of all its structure (rows and columns indices, staircase etc.) is computed and the corresponding memory allocated by the `activate` routine which also initializes the front by first filling it up with zeros and then assembling the nonzero coefficients from the related rows of the input sparse matrix. Then a loop over all the front children follows where, for each child `c`, its columns are assembled into `f`; note that only those columns that overlap with the contribution block have to be assembled but we omit this detail for the sake of readability. The `assemble` routine assembles one column of `c` into multiple columns of `f` depending

```
forall fronts f in topological order
   ! compute structure,  allocate and initialize
   call activate(f)

   forall children c of f
      forall blockcolumns j=1...n in c
         ! assemble column j of c into f
         call assemble(c(j), f)
      end do
      ! Deactivate child
      call deactivate(c)
   end do

   forall panels k=1...n in f
      ! panel reduction of column k
      call _geqrt(f(k))
      forall blockcolumns j=k+1...n in f
         ! update of column j with panel k
         call _gemqrt(f(k), f(j))
      end do
   end do
end do
```

Figure 3.2: Pseudo-code for the sequential multifrontal QR factorization with 1D partitioned frontal matrices.

on the `c`-to-`f` columns mapping. When all the columns of a child have been assembled, the child can be freed by means of the `deactivate` routine. When the front is assembled, it can be factorized; this is achieved by the the double, nested loop starting at line 14 that uses the `_geqrt` and `_gemqrt` routines. Note that, although for these routines we keep the original LAPACK name, these have been modified to take advantage of the fronts staircase structure by means of an internal blocking `ib`, as explained below. Note that the doubly-nested loop starting at line 14 essentially corresponds to the blocked QR factorization described in Section 2.1.4.2.

The multifrontal factorization of a sparse matrix can thus be represented as a DAG of tasks where each task represents the execution of one of the function calls in the pseudo-code of Figure 3.2. Figure 3.3 shows the DAG associated with the subtree defined by supernodes one, two and three for the problem in Figure 2.9 for the case where the block-columns have size one[1]; the dashed boxes surround all the tasks that are related to a single front.

The dependencies in the DAG are defined according to the following rules (an example of each of these rules is presented in Figure 3.3 with labels on the edges):

- `d1`: no other elementary operation can be executed on a front or on one of its block-columns until the front is not activated;
- `d2`: a block column can be updated with respect to a panel only if the corresponding panel factorization is completed;

[1]Figure 3.3 actually shows the transitive reduction of the DAG, i.e., the direct dependency between two nodes is not shown in the case where it can be represented implicitly by a path of length greater than one connecting them.

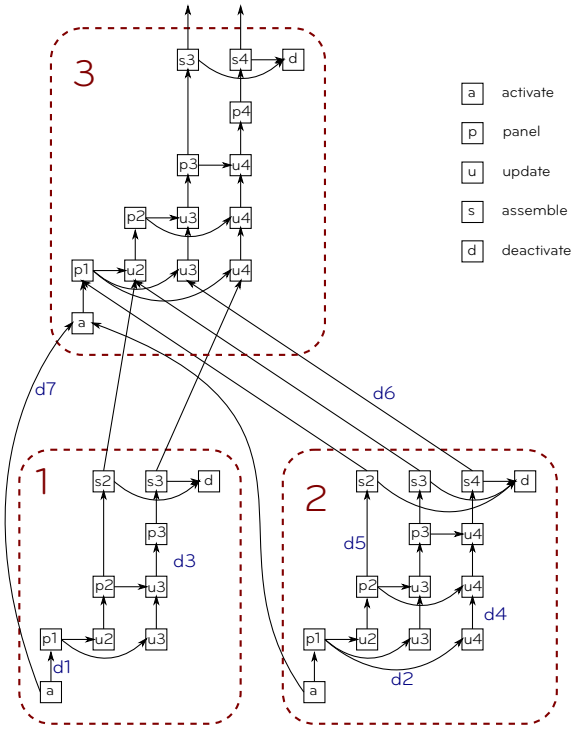


Figure 3.3: The DAG associated with supernodes 1, 2 and 3 of the problem in Figure 2.9; for the `panel`, `update` and `assemble` operations, the corresponding block-column index is specified. For this example, the block-column size is chosen to be one.

- `d3`: the panel operation can be executed on block-column k only if it is up-to-date with respect to panel $k-1$;
- `d4`: a block-column can be updated with respect to a panel k in its front only if it is up-to-date with respect to the previous panel $k-1$ in the same front;
- `d5`: a block-column can be assembled into the parent (if it exists) when it is up-to-date with respect to the last panel factorization to be performed on the front it belongs to (in this case it is assumed that block-column j is up-to-date with respect to panel k when the corresponding panel operation is executed);
- `d6`: no other elementary operation can be executed on a block-column until all the corresponding portions of the contribution blocks from the child nodes have been assembled into it, in which case the block-column is said to be *assembled*;
- `d7`: since the structure of a frontal matrix depends on the structure of its children, a front can be activated only if all of its children are already active;

This DAG globally retains the structure of the assembly tree but expresses a higher degree of concurrency because tasks are defined on a block-column basis instead of a front basis. Moreover, it implicitly represents both tree and node parallelism which allows for exploiting both of them in a consistent way. Finally, it removes unnecessary dependencies making it possible, for example, to start working on the assembled block-columns of a front even if the rest of the front is not yet assembled and, most importantly, even if the children of the front have not yet been completely factorized; we refer to this additional source of concurrency as *inter-level parallelism*.

The next two sections discuss two different ways of implementing this task-based parallelization approach. In the first case, described in Section 3.2, this is achieved by hand coding a complex task-queueing system where ready tasks are added through a costly search along the assembly tree. This tasking mechanism relies on some peculiar properties of the multifrontal method and, thus, can hardly be used in other applications. Efficiency is achieved thanks to a number of low-level optimizations that make the code hard to maintain and extend. In the second case, described in Section 3.3, this is achieved, in a more modular, efficient and portable way, through the use of a STF runtime system (see Section 2.4), namely, StarPU.

3.2 A hand coded task-based parallel implementation

In this approach [J10] the scheduling and execution of tasks is implemented through a hand-coded system of task queues containing executable tasks (i.e., tasks whose dependencies are already satisfied).

Upon execution of the factorization, threads enter in a loop where, at every iteration a thread:

1. checks whether the number of tasks globally available for execution has fallen below a certain value (which depends, e.g., on the number of threads) and, if it is the case, it calls the `fill_queues` routine, described below, which searches for ready tasks and pushes them into the queueing system;

2. picks a task. This operation consists in popping a task from the queueing system and is executed by the `pop_task` routine;

3. executes the selected task if the previous step has succeeded.

The tasks are pushed into the queueing system by the `fill_queues` routine. At every moment, during the factorization there exists a list of active fronts; the `fill_queues` routine goes through this list looking for ready tasks on each front. Whenever one such task is found, it is pushed in the queueing system. If no task is found related to any of the active fronts, a new ready front (if any) is scheduled for activation; the search for an activable front follows a postorder traversal of the assembly tree, which provides a good memory consumption (as explained below) and temporal locality of data. Simultaneous access to the same front in the `fill_queues` routine is prevented through the use of locks.

The size of the search space for the `fill_queues` routine is, thus, proportional to the number of fronts that are active, at a given moment, during the factorization. The size of this search space may become excessively large and, consequently, the relative cost of the `fill_queues` routine intolerable, in some cases like, for example, when the assembly tree is very large and/or when it has many nodes of small size. Two techniques are employed in order to keep the number of active nodes limited during the factorization:

- **Logical pruning**. Commonly, large assembly trees provide much more tree-level parallelism than what is really needed. A logical pruning can thus be applied to simplify the tree: all the subtrees whose relative computational weight is smaller than a certain threshold are made invisible to the `fill_queues` routine. When one of the remaining nodes is activated, all the small subtrees attached to it are processed sequentially by the same thread that performs the activation. This is a variant of the Geist-Ng algorithm [73] and, as shown in Figure 3.4, this corresponds to identifying a layer in the assembly tree such that all the subtrees below it will be

processed sequentially. This layer has to be as high as possible in order to reduce the number of potentially active nodes but low enough to provide a sufficient amount of tree-level parallelism on the top part of the tree. Note that this has the further advantage of increasing the locality of access to data because the same thread works on all the nodes within an entire subtree.

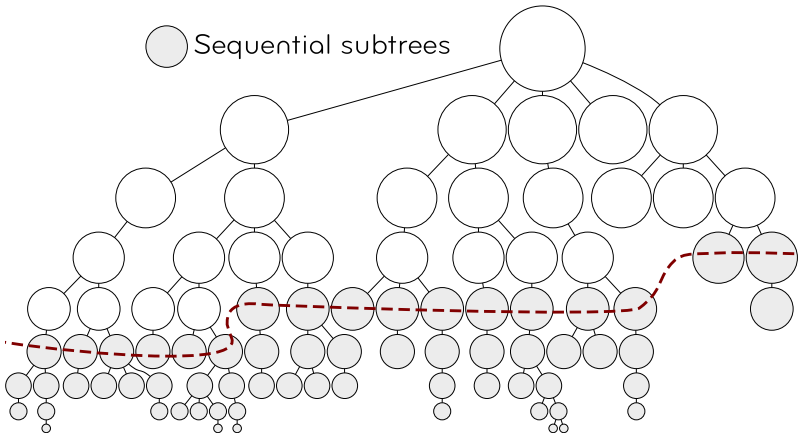


Figure 3.4: A graphical representation of how the logical amalgamation and logical pruning may be applied to an assembly tree.

- **Tree reordering**. Assuming that the assembly tree is processed sequentially following a postorder traversal, the maximum number of fronts active at any time in the subtree rooted at node i, P_i is defined as the maximum of two quantities:

  1. $nc_i + 1$, where nc_i is the number of children of node i. This is the number of active nodes at the moment when i is activated;
  2. $\max_{j=1,\ldots,nc_i}(j-1+P_j)$. This quantity captures the maximum number of active fronts when the children of node i are being processed. In fact, at the moment when the peak P_j is reached in the subtree rooted at the j-th child, all the previous $j-1$ children of i are still active.

  In order to minimize the maximum number of active nodes during the traversal of the assembly tree it is, thus, necessary to minimize, for each node i, the second quantity, which is achieved by sorting all of its children j in decreasing order of P_j [118]. The effect of this reordering on an example tree is illustrated in Figure 3.5. If the tree is traversed in the order shown in the left part of the figure, $P_{19} = 10$ (the nodes active at the moment when the peak is reached are highlighted with a thick border); instead, if the tree is reordered as in the right part of the figure following the method described above P_{19} is equal to four. Although no guarantee is given that a postorder is followed in a parallel factorization, this tree reordering technique still provides excellent results on every problem that has been tested so far. Besides, by reducing the number of active nodes, this reordering also helps in reducing the consumed memory although it will not be optimal in this sense as the actual size of the frontal matrices is not taken into account (the reordering technique for memory consumption minimization is described in the paper by Guermouche et al. [89] and Liu [118]).

Both the tree pruning and reordering are executed during the solver's analysis phase.

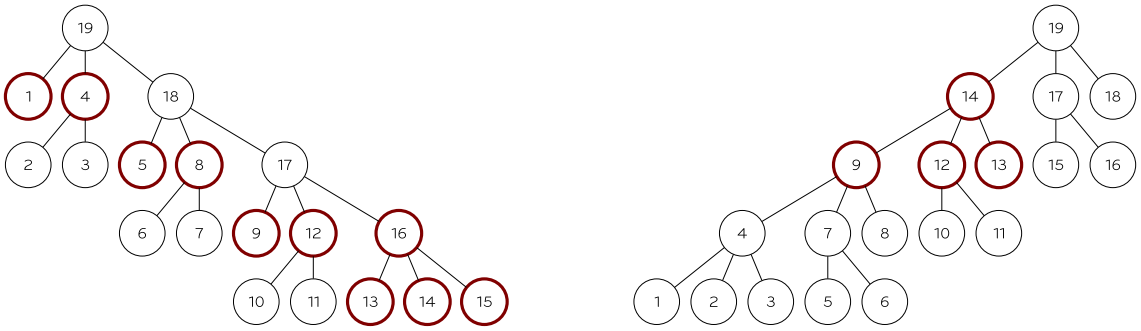


Figure 3.5: The effect of leaves reordering on the number of active nodes.

In order to achieve better performance and memory consumption, a dedicated task scheduling policy was implemented that, by deciding how tasks are pushed and popped to and from the queueing system, defines the behavior of the solver.

This scheduling technique aims at optimizing the reuse of local data in a NUMA system while still prioritizing tasks that have a high number of outgoing edges (fan-out). Although the multifrontal method is rich in Level-3 BLAS operations (mostly in the `_gemqrt` tasks), there is still a considerable amount of Level-2 BLAS operations (within the `_geqrt` tasks) and symbolic or memory ones (the `activate`, `assembly` and `deactivate` tasks) whose efficiency is limited by the speed of the memory system. As the number of threads participating in the factorization increases, the relative cost of these memory-bound operations grows too high and there are fewer opportunities to hide this cost by overlapping these slow operations with faster ones. In addition, some frontal matrices, especially at the bottom of the tree, may be too small to achieve the surface-to-volume effect in Level-3 BLAS operations. Therefore, in order to improve the scalability, it is important to perform these memory-bound operations as efficiently as possible and this can be achieved by executing each of them on the core which is closest to the data it manipulates. The proposed method is based on a concept of ownership of a front: the thread that performs the `activate` operation on a front becomes its owner and, therefore, becomes the privileged thread to perform all the subsequent tasks related to that front. By using methods like the "first touch rule" (memory is placed on the NUMA node which generates the first reference) or allocation routines which are specific for NUMA architectures [40], the memory needed for a front can be allocated in the NUMA node which is closest to its owner thread. No front-to-thread mapping is performed and thus the ownership of a front is dynamically set at the moment when the front is activated. Each thread is associated with a local task queue, filled by the `fill_queues` routine, which contains the tasks related to the fronts it owns. The `pop_task` routine, when executed by a thread, first tries to pop a ready task from the local queue; in case no task is available on the local queue, an architecture aware work-stealing technique is employed, i.e., the thread will try to steal a task from queues associated with threads with which it shares some level of memory (caches or DRAM module on a NUMA machine) and if still no task is found it will attempt to steal a task from any other queue. The computer's architecture is detected using the `hwloc` [40] tool.

Although tasks are always popped from the head of each queue, they can be pushed either on the head or on the tail which allows for prioritizing certain tasks. In our implementation, the `_geqrf` operations are always pushed on the head because the corresponding nodes in the execution DAG have higher fan-out and, thus, their execution satisfies

more dependencies.

Finally, it must be noted that an efficient use of tree-level parallelism makes it hard, if not impossible, to follow a postorder traversal of the tree which results in an increased memory consumption with respect to the sequential case [89, 118]. Therefore the proposed scheduling method tries to exploit node-level parallelism as much as possible and dynamically resorts to tree-level parallelism by activating a new node only when no more tasks are found on already active fronts. This keeps the tree traversal as close as possible to the one followed in the sequential execution and avoids the memory consumption to grow out of control.

3.2.1 Blocking of dense matrix operations

It is obviously desirable to use blocked operations that rely on Level-3 BLAS routines in order to achieve a better use of the memory hierarchy and, thus, better performance. The use of blocked operations, however, introduces additional fill-in in the Householder vectors due to the fact that the staircase structure of the frontal matrices cannot be fully exploited. It can be safely said that it is always worth paying the extra cost of this additional fill-in because the overall performance will be drastically improved by the high efficiency of Level-3 BLAS routines; nonetheless it is important to choose the blocking value that gives the best compromise between number of operations and efficiency of the BLAS. This blocking size, which defines the granularity of computations, has to be chosen with respect to the block-columns size used for partitioning the frontal matrices, which defines the granularity of parallel tasks.

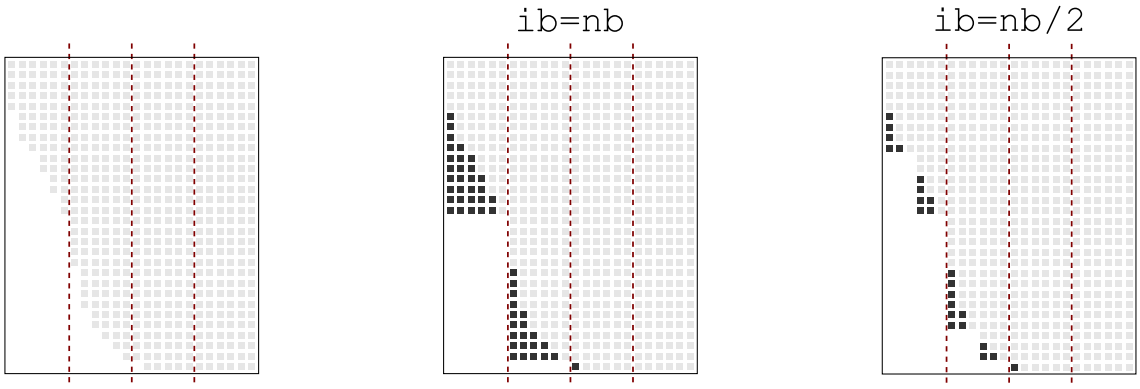


Figure 3.6: The effect of internal blocking on the generated fill-in. The light gray dots show the frontal matrix structure if no blocking of operations is applied whereas the dark gray dots show the additional fill-in introduced by blocked operations.

Figure 3.6 shows as dark gray dots the extra fill-in introduced by the blocking of operations (denoted as `ib` for internal blocking) with respect to the partitioning size `nb` on an example frontal matrix.

3.2.2 Experimental results

The method discussed in this section was implemented in version 1.0 of the `qr_mumps`[2] (or `qrm` for the sake of brevity) package, released in November 2012. The code is written in Fortran2003 and OpenMP is the technology chosen to implement the multithreading.

[2] http://buttari.perso.enseeiht.fr/qr_mumps

The experiments were run on a set of fifty matrices from the UF Sparse Matrix Collection [56]; they were chosen from the complete set of over and under-determined problems by excluding those that are rank-deficient (because `qr_mumps` cannot currently handle them), those that are too small to evaluate the scalability or that are too big for being factorized on the computer platforms described below. More detailed experiments were performed on a subset of matrices whose details are reported in Appendix A.1. In the case of under-determined systems, the transposed matrix is factorized, as it is commonly done to find the least-norm solution of a problem. All of the results presented below were produced without storing the H matrix in order to extend the test set to very large matrices that couldn't otherwise be factorized on the available computers.

For all the 51 matrices in the test set the blocking values were chosen among three different combinations, i.e., `(nb,ib)`=(120, 120), (120, 60) or (60, 60) as those that delivered the shortest factorization time using all the cores available on the system. All the experimental results presented in the rest of this section are related to that choice.

3.2.2.1 Understanding the memory utilization

As described above, the scheduling of tasks in `qrm` is based on a method that aims at maximizing the locality of data in a NUMA environment. The purpose of this section is to provide an analysis of the effectiveness of this approach. This analysis was conducted on the `dude` system, always using all of the 24 cores available, with the PAPI [128] tool. Recalling the architectural characteristics of the `dude` system (see Appendix A.2), the efficiency of the scheduling technique has been evaluated based on three metrics:

- the completion time;
- the amount of data transferred on the HyperTransport links[3];
- the number of conflicts on the DRAM controllers[4]

Experiments were run on all the matrices in the test set and with three different settings for the memory policy:

- no locality: this is a code variant where the multiple task queues are replaced with a single one shared by all threads. Tasks are, thus, pushed and popped from this queue regardless of their affinity with the data placement in the memory system;
- locality: this corresponds to the scheduling strategy described in Section 3.2, i.e., each task is pushed on the queue attached to the thread which owns the related front;
- round robin: this setting uses the same code variant of the "locality" one but, in this case, allocated memory pages are interleaved in a round robin fashion over all the DRAM modules. This is achieved using the `numactl` [107] tool with the "`-i all`" option. Since the code does not control the placement of data in the memory system, the ownership of fronts does not make sense anymore and, consequently, the data locality is completely destroyed.

[3] This quantity was measured as the number of occurrences of the PAPI `HYPERTRANSPORT_LINKx:DATA_DWORD_SENT` event (where `x` is 0, 1, 2, 3 since each processor has 4 HyperTransport links) which counts the number of double-words transferred over the HyperTransport links.

[4] This quantity was measured as the number of occurrences of the PAPI `DRAM_ACCESSES_PAGE:DCTx_PAGE_CONFLICT` event (where `x` is 0, 1 since each processor has two DRAM controllers) which counts the number of conflicts occurring on the DRAM controllers.

Matrix	3	6	7	8	9	10
	Time (sec.)					
no. loc.	8.97	28.48	53.38	53.07	157.20	371.78
loc.	7.57	25.77	48.43	48.25	143.48	345.75
r. r.	6.54	22.38	28.90	42.50	113.70	328.40
	Dwords on HT ($\times 10^9$)					
no. loc.	23.59	65.61	81.40	109.45	371.73	803.27
loc.	11.92	47.30	76.94	90.52	259.89	656.50
r. r.	25.00	72.60	86.13	125.66	378.09	869.87
	Conflicts on DCT ($\times 10^9$)					
no. loc.	0.26	0.68	0.98	1.07	3.29	7.36
loc.	0.29	0.70	0.92	1.21	4.07	8.14
r. r.	0.24	0.52	0.62	0.79	3.17	6.31

Table 3.2: Analysis of the memory usage for the matrix factorization on the `dude` system with 24 threads.

Table 3.2 shows the results of these experiments for a subset of the matrices in Table A.1. It can be seen that the locality aware scheduling strategy described above provides better execution times with respect to a naive dynamic scheduling policy: the improvement may be as high as almost 20% (for the EternityII_E matrix) and quite often around 10%. This can be explained by the reduced amount of data transferred over the HyperTransport links as shown in the middle of the table. However, the best execution times are achieved with the round robin distribution of memory allocations. In this case, the amount of data transfers is higher than the other two cases since the frontal matrices are completely scattered over the NUMA nodes; nonetheless, this more even distribution of data reduces the number of conflicts on the DRAM controllers (see Table 3.2 (*bottom*)) and provides a better use of the memory bandwidth. This behavior is coherent to what was observed on the `HSL_MA87` [101] code which uses a similar approach for the Cholesky factorization of sparse linear systems.

The considerable performance improvement resulting from the interleaved memory allocation suggests that reducing the memory conflicts may be more important than minimizing the amount of data transferred over the HyperTransport links. A closer look to the behavior of matrices e18 and flower_7_4 (columns three and four in Table 3.2) shows, instead, that both these objectives are very important for the efficiency of the code. The factorization of these two matrices takes roughly the same time in the "locality" and "no locality" cases but the memory interleaving provides only a small improvement to the second matrix due to a bigger increase of the data traffic on the HyperTransport links.

Maximizing the data locality and minimizing the memory conflicts are not conflicting objectives although it may be rather complicated to achieve both of them at the same time on a very heterogeneous workload such as a sparse factorization. It has to be noted that in the proposed locality aware scheduling policy, the placement of data and the ownership of the associated tasks is defined on a front basis. Because the number of fronts becomes smaller than the number of working threads when the factorization approaches the root front, a lot of work stealing and memory contention take place on the top part of the assembly tree where most of the work is done. Further improvements can be obtained by defining the data placement and the tasks affinity on a block-column basis; despite this would make some operations such as the front assembly much more complex, it is

Matrix	0	1	2	4	5	10
none	3.52	385.10	151.40	7.12	9.53	9239.00
reord.	2.89	113.80	140.90	7.01	8.75	1651.00
prune	0.94	17.27	5.52	6.99	9.38	328.50
both	0.84	14.44	5.16	6.83	9.20	326.40

Table 3.3: The effect of tree pruning and tree reordering on the matrix factorization time on the `dude` system with 24 threads.

reasonable to expect that it will allow for more evenly and efficiently distributing the data and thus improving the locality of reference and reduce the memory contention at the same time.

3.2.2.2 The effect of tree pruning and reordering

The tree reordering and pruning techniques have been evaluated on the test matrices. For the tree pruning, the initial threshold was set to 0.01 which means that all the subtrees whose weight is smaller than 1% of the total factorization workload are pruned off. If the remaining tree does not provide enough tree-level parallelism, the threshold is divided by 2 and a new pruning is done on the original assembly tree. More precisely this procedure is iterated until the number of leaves in the pruned tree is bigger than twice the number of working threads. Clearly, the optimal values for both the starting threshold and the stopping criterion depend on the structure of the tree and, therefore, on the specific input matrix; the values described above were determined experimentally and were found to work well on the large set of test matrices previously described.

Table 3.3 shows the experimental results related to a subset of matrices for which these techniques have proved to be particularly effective. The experiments show that, when applied separately, these two methods provide considerable benefits in some specific cases. The tree reordering yields good improvements on very unbalanced and irregular trees: this is the case of the LargeRegFile and sls matrices. The tree pruning, proved to be very effective on all the problems but particularly on those with extremely large trees and extremely small frontal matrices such as the cont11_l and the sls matrices.

It can be observed that the pruning clearly reduces the need for sorting as well as its effectiveness. Nonetheless on some matrices (see, particularly, the first three columns in Table 3.3) the best execution time is achieved when both techniques are applied. It is important to note that the size of the pruned tree increases with the number of working threads and, therefore, the improvements provided by the sorting, when both techniques are applied, are likely to be more important for higher degrees of parallelism.

3.2.2.3 Absolute performance and Scaling

The `qrm` code was compared to the SuiteSparseQR [55] (referred to as `spqr`) released by Tim Davis in 2009; this comparison was made on the `dude` system described in Appendix A.2. For both packages, the COLAMD matrix permutation was applied in the analysis phase to reduce the fill-in and equivalent nodes amalgamation methods were used so that the differences between the produced assembly trees can be considered negligible. Both packages are based on the same variant of the multifrontal method (that includes the two optimization techniques discussed in Section 2.2.2) and, thus, the number of floating point operations done in the factorization and the number of entries in

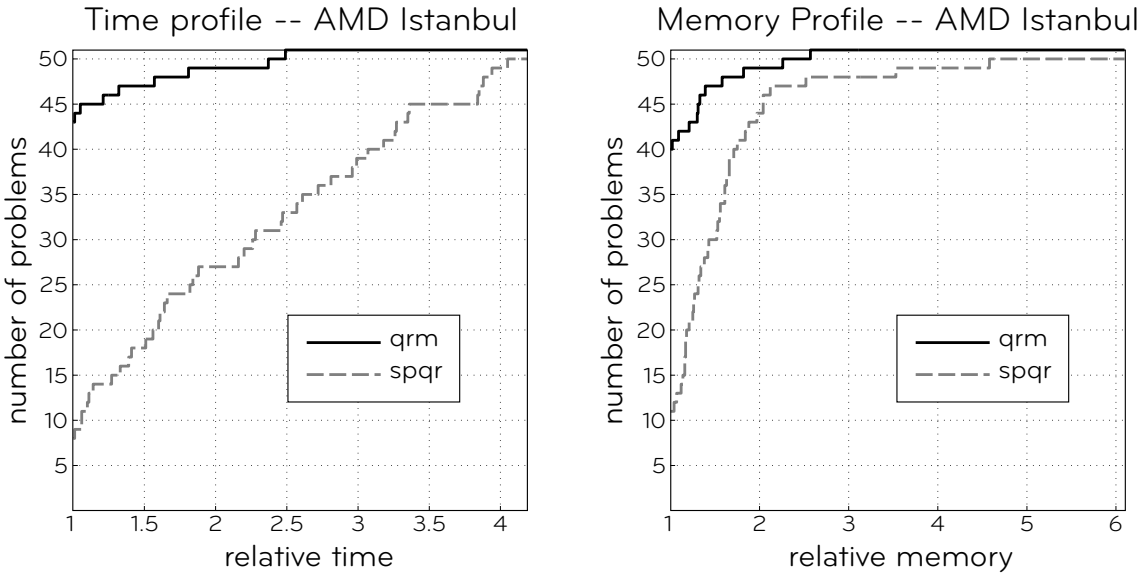


Figure 3.7: Time and memory profiles comparing the `qrm` and `spqr` factorizations for the 51 test matrices on the `dude` system.

the resulting factors are comparable. For both codes, runs were executed with `numactl` memory interleaving.

Figure 3.7 shows time and memory profiles [61] for both codes using 24 threads. For any given code, a data point (x, y) on the profile means that the code is no worse than x times the best of the two codes for y problems. This means that the $(1, y)$ point gives the number y of problems on which the associated code was found to be the best. The time profile shows that `qrm` is faster than `spqr` on 43 problems out of 51; the remaining eight problems are very small, i.e., their factorization takes less than one second. On basically half the problems in the test set `qrm` was more than twice as fast as `spqr`. The memory profile is more difficult to interpret. Because `qrm` is based on a much more eager execution model it is reasonable to expect that it consumes more memory. Indeed, our scheduling method exploits as much as possible the node parallelism and resorts on tree parallelism, by activating new nodes, only when needed; this keeps the tree traversal close to the one followed in the sequential case and limits the memory consumption growth in parallel. Moreover, the tree reordering technique, as explained in the same section, helps reducing the memory footprint. As a result, `qrm` achieves, in general, a smaller memory consumption than `spqr` on the 51 test matrices as shown in the memory profile of Figure 3.7. Experimental data show that most of the matrices where `qrm` has a better memory consumption are relatively small, which may probably be explained with some overhead in `spqr` that becomes negligible for bigger size problems. Note that, in both codes, the memory footprint is defined as the peak memory consumption reached during the factorization, including the input matrix and all the allocated memory areas of any type. On bigger problems, the two codes have a similar memory consumption and no clear winner can be identified. If the H matrix was kept in memory, which is not the case for the results reported in Figure 3.7, the difference between the two codes would be even smaller because the relative weight of the contribution blocks, responsible for the increased memory consumption in parallel, would be lower.

Table 3.4 shows execution times for some test problems on the `dude` and `vargas`

dude										
Matrix	2	3	4	5	6	7	8	9	10	11
th.										
1	48.8	88.5	103.2	134.9	392.0	474.5	774.7	1786	3301	5185
2	33.8	49.2	52.2	65.5	209.9	250.3	357.4	961	1802	2770
4	14.8	24.8	26.5	33.4	106.4	126.3	181.6	495	932	1372
6	12.4	17.6	18.4	22.9	71.9	86.6	124.7	341	655	923
12	6.2	9.8	10.4	12.8	37.8	46.2	65.8	181	421	477
18	5.2	7.9	7.7	9.1	27.3	32.9	48.6	132	341	325
24	4.8	6.5	6.8	8.4	22.4	28.9	42.5	114	327	260
su	10.1	13.6	15.1	16.0	17.5	16.4	18.2	15.6	10.0	19.9
vargas										
Matrix	2	3	4	5	6	7	8	9	10	11
th.										
1	36.2	51.9	60.5	75.2	227.3	290.2	426.1	1156	2142	3121
2	24.5	27.6	32.0	38.6	117.2	151.7	192.9	600	1259	1656
4	10.6	14.0	16.0	19.4	57.1	76.5	98.0	304	672	850
8	5.9	7.3	8.2	9.9	28.6	39.3	49.5	155	348	458
16	3.9	4.1	4.7	5.7	15.4	21.3	27.6	84	197	235
24	4.5	3.1	3.7	4.3	11.3	15.5	22.8	62	183	174
32	6.4	2.6	3.3	4.0	9.5	12.9	18.9	52	182	141
su	5.6	19.9	18.3	18.8	23.9	22.4	22.5	22.2	11.7	22.1

Table 3.4: Factorization times, in seconds, on the `dude` and `vargas` systems for `qrm`.

systems. The reported results show and overall good performance and scalability of the code with speedups that reach around 20 and 23 for the two machines, respectively. It must be noted that a lower scaling is reached or smaller problems and on the sls matrix whose fronts are extremely overdetermined. These classes of problems will be addressed in the Section 3.4.

3.3 Runtime-based multifrontal QR for manycore systems

The multifrontal QR method described in the previous section fully conforms to task-based parallel programming paradigm that is supported by some of the most popular, modern runtime systems. In essence, the actual implementation relies on our own, hand-written runtime system whose functioning and features are described above. This, however, has a number of drawbacks and limitations. First of all, this runtime system is deeply embedded with the multifrontal QR method and far from being usable for other applications. Second, it has a very limited set of features and, for example, cannot handle systems other simple multicore nodes. Third, it is inefficient because the search for ready tasks implies searching in a rather large space (which required the use of techniques such as the tree reordering described in Section 3.2). It is thus a natural choice to port this method on a proper, modern runtime in order to take advantage of better efficiency portability and a wider set of feature. As Explained in Section 2.4, among all the available runtimes, we have chosen to use StarPU mostly because it relies on the Sequential Task Flow programming

model, which is extremely convenient for implementing complex algorithms such as the multifrontal one, and because it has a very wide set of features, some of which we will describe below.

A first attempt to port the multifrontal QR method of Section 3.1 on StarPU [C3] was mainly meant to asses the usability of a modern runtime system for implementing a sparse, direct solver. For this reason we tried to reproduce as accurately as possible the behavior of the code described in the previous section. This lead to satisfactory results. Nonetheless, the resulting code had a number of shortcomings [C3], including a rather intricate tasks submission scheme which did not fully adhere with the STF model; this prevents the runtime from properly handling the data (which is necessary when multiple memories are available) because the flow of data between tasks was not correctly expressed.

In the remainder of this section we describe an implementation that is fully compliant with and takes full advantage of the expressiveness of the STF programming model.

As explained in Section 2.4, parallelizing a code with the STF model is conceptually as simple as replacing operations (or, function calls) with tasks submissions. This can easily be achieved by replacing the function calls in Figure 3.2 with the submission of the corresponding tasks. In our case there is one caveat though: the assembly tasks can only be submitted after the structure of the current front is computed by the `activate`. As a consequence, this routine cannot be made into a task whose execution is deferred but, instead, has to be executed synchronously by the master thread prior to submitting all the other tasks related to the corresponding front. It is thus very important to keep this routine as lightweight as possible in order not to delay the submission of tasks in the main loop. For this reason the front initialization was moved into a separate `init` routine and only the structure computation and the memory allocation were left in the `activate` one. This last also registers the data to StarPU; once this is done, StarPU has full control of the data and the master process can only reference it by means of a *handle* returned by StarPU upon registration. The pseudo-code in Figure 3.8 shows the resulting STF parallel multifrontal QR factorization.

Using the data access modes and the order of task submission, the runtime system automatically infers dependencies between tasks, which are described in Section 3.1 and thus builds the DAG.

This code also includes the logical tree pruning optimization described in Section 3.2 but we did not take it into account in the pseudo-code for the sake of readability. Each leaf-subtree whose weight is smaller than a certain threshold is processed sequentially by a single task of type `do_subtree`; clearly, no partitioning is applied to fronts in a sequential subtree and a standard LAPACK-style factorization (which can take advantage of the staircase structure) is used on them. This has a threefold advantage. First of all, it drastically reduces the number of tasks and thus reduces the runtime overhead. Second, subtree tasks, which are of relatively large size, keep the worker threads busy at the very beginning of the factorization; this gives the master thread enough time to progress in the submission of other tasks which reduces the risk of tasks starvation during the execution. Finally, it improves the efficiency of operations on those parts of the elimination tree that are mostly populated with small size fronts and, thus, less performance effective.

A minor, but profitable improvement over the original `qr_mumps` solver presented in Section 3.1, is the use of a blocked storage format. In the previous version the frontal matrices are allocated as a whole memory area and therefore the partitioning is logical. In this implementation, instead, each block column is allocated individually; although this does not bring any improvement to the performance (because Fortran uses column-major storage), it saves some memory due to the staircase structure of the fronts, as shown in Figure 3.16 (*left*).

```
forall fronts f in topological order
   ! compute structure, allocate memory and register handles
   call activate(f)

   ! initialize front
   call submit(init, f:RW, children(f):R)

   forall children c of f
      forall blockcolumns j=1...n in c
         ! assemble column j of c into f
         call submit(assemble, c(j):R, f:RW|C)
      end do
      ! Deactivate child
      call submit(deactivate, c:RW)
   end do

   forall panels k=1...n in f
      ! panel reduction of column k
      call submit(_geqrt, f(k):RW)
      forall blockcolumns j=k+1...n in f
         ! update of column j with panel k
         call submit(_gemqrt, f(k):R, f(j):RW)
      end do
   end do
end do

call wait_tasks_completion()
```

Figure 3.8: Pseudo-code for the STF-parallel multifrontal QR factorization with 1D partitioned frontal matrices.

The resulting implementation, which we refer to as 1D, because of the front partitioning into block-columns, was tested on a subset of problems front the SuiteSparse Matrix Collection [56] listed in Table A.1. Experiments were done on one node of the `ada` supercomputer described in Appendix A.2. In the experiments discussed below, local and interleaved memory allocation policies were used, respectively, for sequential and parallel runs through the use of the `numactl` tool; these are the policies that deliver the best performance in their respective cases.

The reference sequential execution times are obtained with a purely sequential code (no potential runtime overhead) with no frontal matrix partitioning which ensures that all the LAPACK and BLAS routines execute at the maximum possible speed (no granularity trade-off).

The performance of the presented approach depends on the choice of the values for a number of different parameters. These are the block-column `nb` on which depends the amount of concurrency and the internal block size `ib` on which depend the efficiency of elementary BLAS operations and the global amount of flop (this parameter defines how well the staircase structure of each front is exploited). The choice of these values depends on a number of factors, such as the number of working threads, the size and structure of the matrix, the shape of the elimination tree and of frontal matrices and the features of the underlying architecture. It has to be noted that these parameters may be set to different values for each frontal matrix; moreover, it would be possible to let the software automatically choose values for these parameters. Both these tasks are very difficult

and challenging and have not been investigated yet. Therefore, for our experiments we performed a large number of runs with varying values for all these parameters, using the same values for all the fronts in the elimination tree, and selected the best results (shortest running time) among those. For the sequential runs internal block sizes `ib={32, 40, 64, 80, 128}` were used for a total of five runs per matrix. For the 1D parallel STF case, the used values were `(nb,ib)={(128,32), (128,64), (128,128), (160,40), (160,80)}` for a total of five runs per matrix.

All of the results presented in this section were produced without storing the factors in order to extend the tests to the largest matrices in our experimental set that could not otherwise be factorized (even in sequential) on the target platform. This was achieved by simply deallocating the tiles containing the factors coefficients at each `deactivate` task rather than keeping them in memory. As confirmed by experiments that we do not report here for the sake of space and readability, this does not have a relevant impact on the following performance analysis.

	Sequential reference			Parallel 1D STF			
Mat.	ib	Time (s.)	Gflop/s	nb	ib	Time (s.)	Gflop/s
12	40	1.00E+02	14.4	128	64	5.337E+00	272.4
13	32	1.73E+02	17.0	128	128	9.809E+00	312.0
14	80	3.40E+02	17.1	128	128	1.922E+01	309.8
15	128	5.76E+02	19.0	128	128	3.116E+01	352.0
16	80	8.71E+02	19.2	128	128	4.646E+01	362.0
17	80	1.18E+03	17.7	128	32	4.945E+01	407.8
18	128	1.58E+03	19.3	128	128	8.383E+01	365.9
19	128	3.25E+03	19.5	128	128	1.501E+02	422.4
20	128	3.99E+03	16.7	128	64	6.432E+02	102.7
21	128	9.93E+03	19.7	128	128	4.402E+02	446.8
22	128	1.13E+04	19.7	128	128	5.207E+02	430.6
23	128	1.37E+04	19.2	128	128	6.233E+02	422.7

Table 3.5: Sequential reference execution time and optimum performance for the STF 1D factorization on `ada` (32 cores).

Table 3.5 shows for the STF 1D algorithm the parameter values delivering the shortest execution time along with the corresponding attained factorization time and Gflop rate. The Gflop rates reported in the table are related to the operation count achieved with the internal block size `ib`. The smaller block-column size of 128 always delivers better performance because it offers a better compromise between concurrency and efficiency of BLAS operations, whereas a large internal block size is more desirable because it leads to better BLAS speed despite a worse exploitation of the fronts staircase structure.

Figure 3.9, generated with the timing data in Table 3.5, shows the speedup achieved by the 1D parallel code with respect to the sequential one when using all the 32 cores available on the system. This figure shows that the speedup increases with the problem size and may be extremely low on some problems such as for matrix #20 whose speedup is less than 7. Although these results are satisfactory on larger size problems, on smaller ones and on matrix #20 performance and scalability are relatively poor. This is due to a lack of concurrency, especially for matrix #20 where most of the flops are done in one front which has 1.3 M rows and only 7 K columns; the 1D partitioning into block-columns is clearly not suitable for the case of strongly over-determined frontal matrices. This

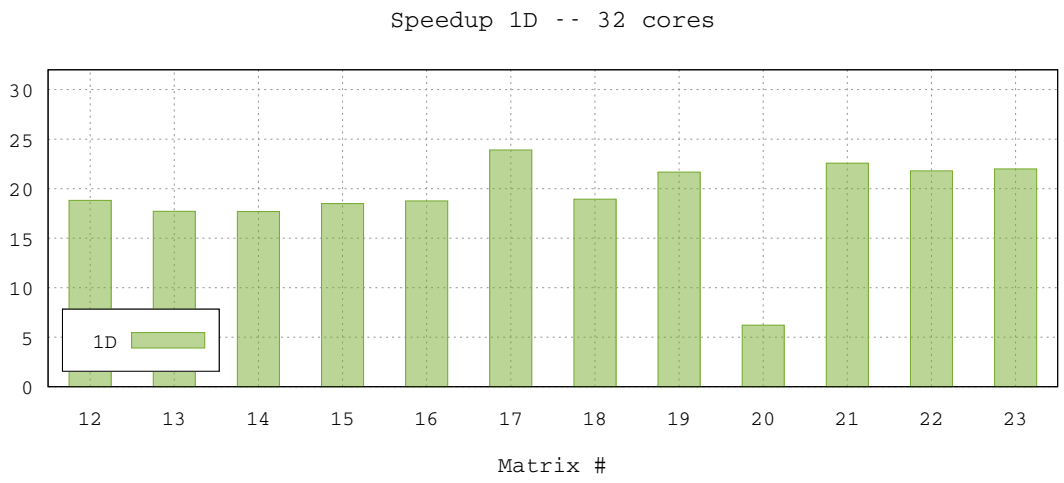


Figure 3.9: Speedup of the STF 1D algorithm with respect to the sequential case on `ada` (32 cores).

problem will be addressed in the next section.

3.4 Communication Avoiding QR fronts factorizations

The parallel factorization algorithm presented in the two previous sections relies on the use of the blocked algorithms presented in Section 2.1.4 combined with a 1D partitioning of frontal matrices into block-columns. These can be roughly described as a sequence of panel reduction and trailing submatrix update. The first operation factorizes a panel, i.e., a block of columns, by means of a point factorization method and thus it is based on Level-2 BLAS operations; the second applies all the transformations computed in a panel reduction to the trailing submatrix using Level-3 BLAS operations. In order to maximize the amount of operations in Level-3 BLAS routines, the size of the panel has to be much smaller than the size of the matrix; however, for these Level-3 BLAS routines to be efficient, the size of the panel need not be too small. In these methods, parallelism mostly comes from the update step which is where most of the floating point operations occur. The panel step, instead, cannot be efficiently parallelized for two main reasons. First, in the blocked factorizations described in Section 2.1.4, the work in the panel reduction cannot be shared without incurring a high amount of communications; note that here by communications we intend either message exchanges in a distributed memory systems or data transfers from memory and synchronizations in a shared memory one. In the case of the LU factorization this is due to partial pivoting which implies a search along a whole column and in the case of the QR factorization this is due to the norm computation in Equation (2.11) required to construct the Householder reflection which also requires access to a whole column. Second, because the panel reduction is made of Level-2 BLAS routines, the cost of the communications becomes predominant and cannot be hidden behind computations. As a result, in these blocked factorization, parallelization is achieved mainly through a *fork-join* approach where sequential operations (panel reductions) are alternated with parallel ones (trailing submatrix updates). *Lookahead* techniques can improve the efficiency by overlapping panel reductions with updates from previous steps. Nonetheless this approach

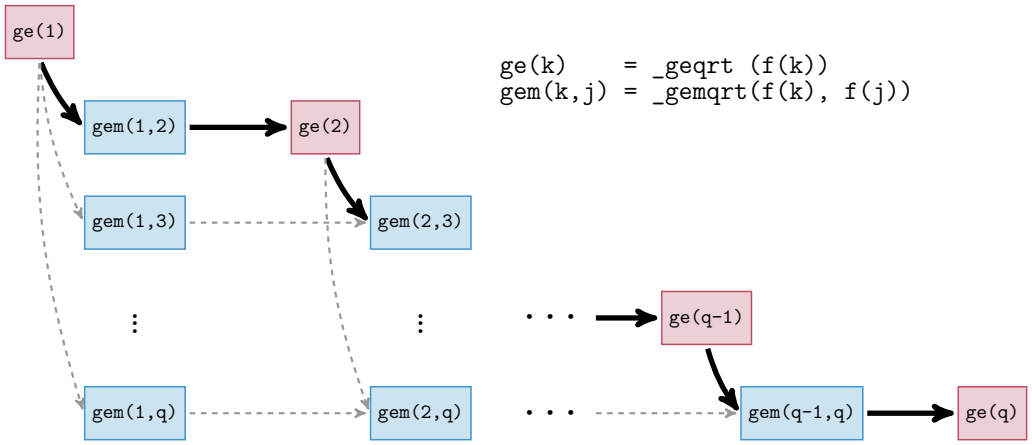


Figure 3.10: Task graph for the block-column QR factorization of a dense matrix with q block columns. Arrows show tasks dependencies; the critical path is shown with thick solid edges.

presents serious scalability limits especially for overdetermined matrices where the relative weight of the panel reduction becomes dominant.

Consider a block size b and a dense matrix of size $m \times n$ with $m = pb$ and $n = qb$; for this matrix, the DAG of the dense QR factorization by block-column in line 14 of Figure 3.2 is illustrated in Figure 3.10. All the tasks that lie on a given path of this DAG must be executed sequentially because of the dependencies expressed by the edges of the path; as a result, the *makespan* (i.e., the total execution time) cannot be smaller than the duration of the longest path in the DAG, which we refer to as *critical path*. Assuming that the cost of `_geqrt` and `_gemqrt` on a block-column of height pb are, respectively, $2(3p-1)b^3/3$ and $3(4p-2)b^3/3$, the length of the critical path, shown in Figure 3.10 with thick solid edges, is given by

$$\sum_{k=0}^{q-1}(2(3(p-k)-1)+3(4(p-k)-1)) = O(2pq-q^2) = O(2mn-n^2)$$

where we have dropped the $b^3/3$ term assuming that b is constant (i.e., independent of the size of the problem) and $b \ll min(m,n)$. It is thus clear why, as mentioned in Section 3, for a square frontal matrix of size m, this approach can, at best, reduce the execution time from $O(m^3)$ to $O(m^2)$. Remember that the sequential execution time for the QR factorization is $O(2n^2(m-n/3))$; therefore, this parallelization approach suffers from a severe lack of concurrency in the case of (strongly) overdetermined matrices because both the parallel and sequential execution times grow linearly with m for a fixed n. This is often the case in the multifrontal QR method where fronts commonly have (many) more rows than columns. In the multifrontal method this problem is mitigated by the fact that multiple frontal matrices are factorized at the same time. Nonetheless, considering that in the multifrontal factorization most of the computational weight is related to the topmost nodes where tree parallelism is scarce, a 1D front factorization approach can still seriously limit the scalability, as shown in the analysis of Section 3.

The objective of tiled matrix factorizations is to overcome these difficulties by breaking down the panel reduction and the corresponding updates into fine grained tasks that can be efficiently parallelized or pipelined. The case of the Cholesky factorization is trivial and will not be discussed here; we refer the reader to the original paper [J15] on tiled

factorizations for the details. In the next section we will present the tiled QR and LU factorizations.

3.4.1 Communication Avoiding dense factorizations

Algorithm 3.1 Tiled QR and LU (with block pairwise pivoting) factorizations.

for $k = 1, 2..., min(p, q)$ **do**	**for** $k = 1, 2..., min(p, q)$ **do**
`_geqrt`$(A_{k,k},\ V_{k,k},\ R_{k,k},\ T_{k,k})$	`_getrf`$(A_{k,k},\ L_{k,k},\ U_{k,k},\ P_{k,k})$
for $j = k+1, k+2, ..., q$ **do**	**for** $j = k+1, k+2, ..., q$ **do**
`_gemqrt`$(A_{k,j},\ V_{k,k},\ T_{k,k},\ R_{k,j})$	`_gessm`$(A_{k,j},\ L_{k,k},\ P_{k,k},\ U_k, j)$
end for	**end for**
for $i = k+1, k+2, ..., p$ **do**	**for** $i = k+1, k+2, ..., p$ **do**
`_tpqrt`$(R_{k,k},\ A_{i,k},\ V_{i,k},\ T_{i,k})$	`_tstrf`$(U_{k,k},\ A_{i,k},\ P_{i,k})$
for $j = k+1, k+2, ..., q$ **do**	**for** $j = k+1, k+2, ..., q$ **do**
`_tpmqrt`$(R_{k,j},\ A_{i,j},\ V_{i,k},\ T_{i,k})$	`_ssssm`$(U_{k,j},\ A_{i,j},\ L_{i,k},\ P_{i,k})$
end for	**end for**
end for	**end for**
end for	**end for**

These methods work by partitioning the input matrix into blocks, or *tiles* (we will use the term tile rather than block to avoid confusion with blocked factorization methods); for the sake of simplicity we assume that these tiles are square (of size b) but they need not be. At each step of the factorization the diagonal tile is first factorized and then it is used to annihilate all the subdiagonal tiles one after the other. Algorithm 3.1 shows these two factorization techniques; here we have assumed that the input matrix A of size $m \times n$ is partitioned into square tiles of size nb such that $p = m/b$ and $q = n/b$. The used kernels are:

`_geqrt,_getrf` : these two kernels perform the QR and LU factorization of a square tile, respectively:

$$\begin{array}{lcl} \texttt{\_geqrt} & : & A_{k,k} \longrightarrow (V_{k,k}, R_{k,k}, T_{k,k}) = QR(A_{k,k}) \\ \texttt{\_getrf} & : & A_{k,k} \longrightarrow L_{k,k}, U_{k,k}, P_{k,k} = LU(A_{k,k}) \end{array}$$

They both use an internal blocking for better efficiency. The `_geqrt` is equivalent to the LAPACK `_geqrf` described in Section 2.1.4 except that it does not discard the computed T matrices but it keeps them for later use in the `_gemqrt` kernel described below. The `_getrf` is the standard blocked LU factorization with partial pivoting described in Section 2.1.4.

`_gemqrt,_gessm` : these two kernels apply the transformations computed by the two corresponding ones above to a square tile:

$$\begin{array}{lcl} \texttt{\_gemqrt} & : & A_{k,j}, V_{k,k}, T_{k,k} \longrightarrow R_{k,j} = (I - V_{k,k} T_{k,k}^T V_{k,k}^T) A_{k,j} \\ \texttt{\_gessm} & : & A_{k,j}, L_{k,k}, P_{k,k} \longrightarrow U_{k,j} = L_{k,k}^{-1} P_{k,k} A_{k,j} \end{array}$$

_tpqrt,_tstrf : these two kernels are used to factorize a matrix formed by a triangular tile on top of a square one:

$$\begin{aligned} \texttt{\_tpqrt} &: \begin{pmatrix} R_{k,k} \\ A_{i,k} \end{pmatrix} \longrightarrow (V_{i,k}, T_{i,k}, R_{k,k}) = QR \begin{pmatrix} R_{k,k} \\ A_{i,k} \end{pmatrix} \\ \texttt{\_tstrf} &: \begin{pmatrix} U_{k,k} \\ A_{i,k} \end{pmatrix} \longrightarrow U_{k,k}, L_{i,k}, P_{i,k} = LU \begin{pmatrix} U_{k,k} \\ A_{i,k} \end{pmatrix} \end{aligned}$$

Both these kernels use an internal blocking; this allows for skipping the zeros below the diagonal of the top tile. The **_tstrf** does partial pivoting.

_tpmqrt,_ssssm : these two kernels apply the transformations computed by the two corresponding ones above to a couple of square tiles:

$$\begin{aligned} \texttt{\_tpmqrt} &: \begin{pmatrix} R_{k,j} \\ A_{i,j} \end{pmatrix}, V_{i,k}, T_{i,k} \longrightarrow \begin{pmatrix} R_{k,j} \\ A_{i,j} \end{pmatrix} = (I - V_{i,k} T_{i,k}^T V_{i,k}^T) \begin{pmatrix} R_{k,j} \\ A_{i,j} \end{pmatrix} \\ \texttt{\_tpmqrt} &: \begin{pmatrix} U_{k,j} \\ A_{i,j} \end{pmatrix}, L_{i,k}, P_{i,k} \longrightarrow \begin{pmatrix} U_{k,j} \\ A_{i,j} \end{pmatrix} = L_{i,k}^{-1} P_{i,k} \begin{pmatrix} U_{k,j} \\ A_{i,j} \end{pmatrix}. \end{aligned}$$

The main advantage of these tiled factorization comes from the fact that the two problematic operations in the standard, blocked QR and LU factorizations, i.e., the column norm-2 computation and the pivot search along a column, respectively, are limited within the diagonal tile (for the **_geqrt** and **_getrf** kernels) or within a couple of tiles (for the **_tpqrt** and **_tstrf** kernels). For this reason, these methods are also referred to as *Communication Avoiding* [29, 58]. Although this does not have any drawback for the QR factorization, the pivoting technique used in the tiled LU factorization, called *pairwise block pivoting*, is less effective than the partial pivoting used in the standard blocked LU factorization. This technique can, however, be "stable enough" in many practical cases [J15]. In both methods, thanks to the use of an internal blocking within each kernel, the overall cost of the factorization is higher than the classical methods only by a negligible amount [J15].

Figure 3.11 shows the DAG for the tiled QR factorization of a dense matrix of size $m \times n = pb \times 4b$. For a tile-size b, the cost of the **_geqrt**, **_gemqrt**, **_tpqrt** and **_tpmqrt** if of 4, 6, 6 and 12 times $b^3/3$ flops, respectively. The critical path of this DAG and its length can be computed using the method proposed in the work by Bouwmeester et al. [39] which deals with a slightly different method where all the sub-diagonal tiles in a column are triangularized before being annihilated. The critical path for the DAG of Figure 3.11 is illustrated with thick, solid edges; in the general case of a dense matrix of size $m \times n = pb \times qb$ its length can be computed as

4	+		_geqrt	$k = 1$
6	+		_gemqrt	$k = 1,\ j = 2$
$12(p-2)$	+		_tpmqrt	$k = 1,\ j = 2,\ i = 2..p-1$
$12(q-1)$	+		_tpmqrt	$k = 1..q-1, i = p,\ j = k+1$
$6(q-1)$	=		_tpqrt	$k = 2..q,\ i = p$
$O(2p+3q)$	=	$O(2m+3n)$		

where we reported, on the right, the operations related to each term. This result shows how, for a square front of size m, tiled algorithms can reduce the lower bound of the

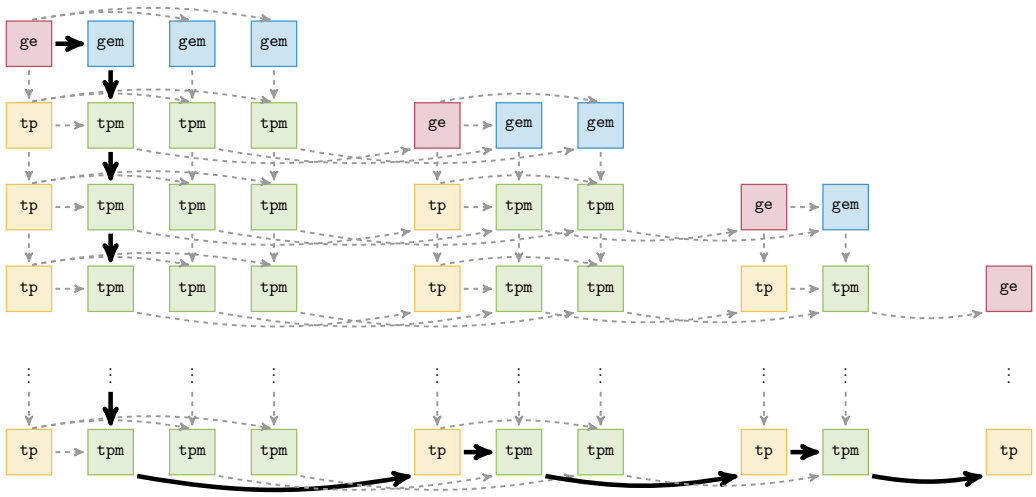


Figure 3.11: The DAG for the tiled QR factorization of a dense matrix of size $m \times n = pb \times 4b$ using Algorithm 3.1. The critical path is illustrated with thick, solid edges.

execution time to $O(m)$ which brings a considerable benefit not only to the scalability of dense matrix factorizations but also to the scalability of sparse matrix factorizations, as discussed in Section 3.

It must still be noted, however, that even for these tiled factorizations, the length of the critical path still grows linearly with the number of rows m as much as the overall cost (i.e., the sequential execution time). This is due to the fact that all the subdiagonal tiles in a panel $A_{k+1,k}...A_{p,k}$ are annihilated sequentially, i.e., one after the other; as a result the reduction of a block-column is not parallelizable. The advantage with respect to the block-column methods comes from a better pipelining between successive stages of the factorization: in order to begin reducing the $k+1$-th column, it is not necessary to wait that all of its tiles are up to date with respect to stage k but, for example, operation `_geqrt`$(A_{k+1,k+1}, V_{k+1,k+1}, R_{k+1,k+1}, T_{k+1,k+1})$ can be executed as soon as the `_tpmqrt`$(R_{k,k+1}, A_{k+1,k+1}, V_{k+1,k}, T_{k+1,k})$ one is done. Although this better pipelining yields a much higher level of concurrency and a much better scalability for square or moderately overdetermined matrices, scalability can still be poor in the case of strongly overdetermined ones. For this reason the tiled QR factorization was later extended to achieve even better scalability on strongly over-determined matrices [6, 58]. As explained, in the tiled factorization presented above, at step k the subdiagonal tiles $A_{i,k}$, $i = k+1, ..., p$ are annihilated sequentially. This process can be described as a reduction operation based on a flat reduction tree, as shown in Figure 3.12 (*left*). Different reduction trees can be used to achieve concurrency in the panel reduction. For example, if a binary tree is used, first all the tiles $A_{i,k}$, $i = k, ..., p$ are reduced to a triangle by means of the `_qeqrt` kernel then $\lceil \log_2(p-k+1) \rceil$ steps follow where the remaining triangular tiles are treated in couples and, for each couple, one tile is annihilated by means of the other as shown in Figure 3.12 (*middle*). This is done using the `_tpqrt` kernel which limits the computations only to the nonzero coefficients of both triangles. A binary reduction tree clearly delivers much higher concurrency that a flat one on strongly overdetermined matrices although it must be noted that it leads to a worse pipelining of successive factorization stages [39] and is based on less efficient operations (the triangle-triangle reduction and the associated updates). As a consequence, hybrid trees are often preferred in practice where the panel tiles are divided in groups of size `bh`: within each group a flat tree is used and then

the remaining triangular tiles (one per group) are reduced using a binary tree as shown in Figure 3.12 (*right*). This approach is implemented in the PLASMA [5] library where multithreading is achieved using the QUARK [166] runtime engine which is based on the STF programming model.

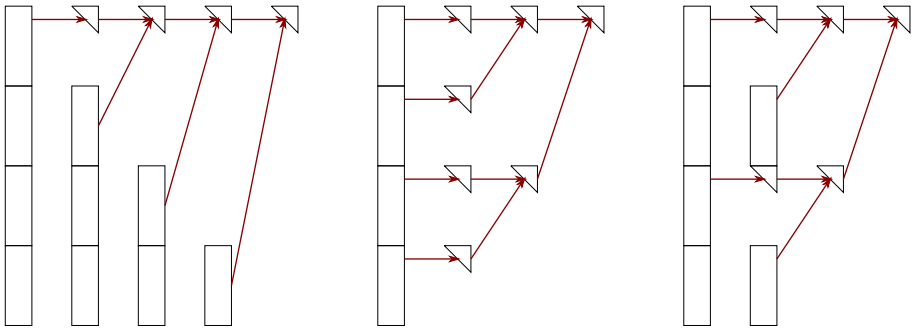


Figure 3.12: Possible panel reduction trees for the 2D front factorization. On the (*left*), the case of a flat tree, i.e., with `bh`$= \infty$. In the (*middle*), the case of a binary tree, i.e., with (`bh`$= 1$). On the (*right*) the case of an hybrid tree with `bh`$= 2$.

The pipelining between successive stages of the tiled QR and LU factorizations is rather difficult to achieve and take advantage of in practice with a static execution approach. In order to make the best use of the available concurrency and maximize the occupancy of computing resources, the dynamic and asynchronous execution model of task-based runtime systems is better suited. Figure 3.13 shows the implementation of the tiled QR factorization of a $m \times n = pb \times qb$ dense matrix using the STF programming model; this method uses hybrid panel reduction trees with subdomains of size `bh`.

Figure 3.14 plots the performance achieved by the block-column and tiled parallel QR factorization methods using 24 threads on the `sirocco` system (see Appendix A.2). Each point of the curves corresponds to the best performance achieved for a matrix size using different combinations of the `b`, `ib` and `bh` parameters. For the 1D method we tested `b`$= \{32, 64, 128, 256\}$ and `ib`$= min(b, \{32, 64, 128, 256\})$; for the 2D method we tested `b`$= \{128, 256, 512\}$, `ib`$= \{32, 64\}$ (higher values lead to an excessive flops overhead) and `bh`$= min(m/b, \{1, 2, 4, 8, 1620\})$.

These results demonstrate the superior performance and scalability of the tiled methods over the block-column ones. This is due to the higher amount of concurrency that tiled methods can deliver. In the case of square matrices all the best results were achieved with flat panel reduction trees (i.e. `bh`$=m/b$) because this variant produces enough parallelism, achieves a good pipelining of successive panels and relies on relatively efficient kernels. In the case of overdetermined matrices, a hybrid panel reduction tree always leads to better performance; nonetheless, absolute performance is not on par with the case of square matrices because when hybrid trees are used, operations are less efficient.

3.4.2 Using Communication Avoiding factorizations within the multifrontal method

The tiled QR factorization method presented in the previous section, namely the flat/binary hybrid approach, was integrated in our multifrontal STF parallel code described in Section 3.3. Lines 17-24 in Figure 3.8 were replaced by the pseudocode in Figure 3.13 which implements the 2D factorization algorithm with hybrid panel reduction tree; this code had to be adapted in order to ignore the tiles that lie entirely below the staircase

```
do k=1, q
   ! for all the block-columns in the front
   do i = k, p, bh
      call submit(_geqrt, f(k,i):RW)
      do j=k+1, q
         call submit(_gemqrt, f(k,i):R, f(i,j):RW)
      end do
      ! intra-subdomain flat-tree reduction
      do l=i+1, min(i+bh-1,p)
         call submit(_tpqrt, f(i,k):RW, f(l,k):RW)
         do j=k+1, q
            call submit(_tpmqrt, f(l,k):R, f(i,j):RW, f(l,j):RW)
         end do
      end do
   end do
   do while (bh.le.p-k+1)
      ! inter-subdomains binary-tree reduction
      do i = k, p-bh, 2*bh
         l = i+bh
         if(l.le.p)) then
            call submit(_tpqrt, f(i,k):RW, f(l,k):RW)
            do j=k+1, q
               call submit(_tpmqrt, f(l,k):R, f(i,j):RW, f(l,j):RW)
            end do
         end if
      end do
      bh = bh*2
   end do
end do
```

Figure 3.13: Pseudo-code showing the implementation of the tiled QR.

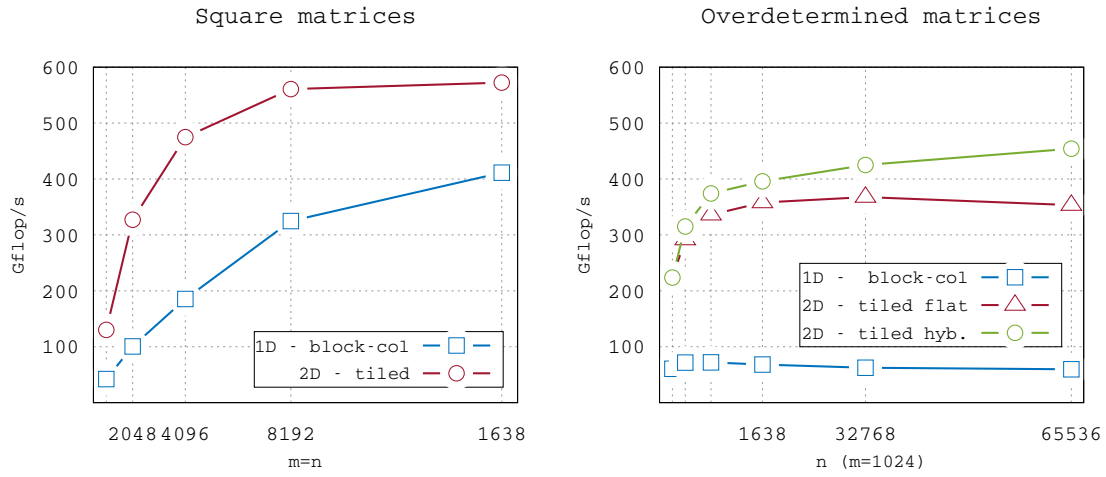


Figure 3.14: Performance of the block-column and tiled parallel QR factorizations on square (*left*) and overdetermined (*right*) matrices using 24 threads on the `sirocco` system.

structure of the front. The assembly operations have also been parallelized according to the 2D frontal matrix blocking: lines 9-11 in Figure 3.8 were replaced with a double, nested loop to span all the tiles lying in the contribution block: each assembly operation reads one tile of a node `c` and assembles its coefficients into a subset of the tiles of its parent `f`. As a consequence, some tiles of a front can be fully assembled and ready to be processed before others and before the child nodes are completely factorized. This finer granularity (with respect to the 1D approach presented in Section 3.1) leads to more concurrency since a better pipelining between a front and its children is now enabled.

The benefit brought by the use of tiled fronts factorizations to the multifrontal QR method is depicted in Figure 3.15. This figure shows the maximum achievable speedup, measured as the ratio between the overall workload cost and the cost of the critical path, on our experimental test set for four different methods. The first relies on a 1D partitioning of frontal matrices with no inter-level parallelism; this means that a node is processed only when all of its children are finished and is equivalent to the approach used in `spqr` [55] or MA49 [15]. The second uses a 1D partitioning with inter-level parallelism and corresponds to the approach used in Sections 3.1 and 3.3. The third and fourth, instead, use the 2D partitioning with Communication Avoiding fronts factorization described in this section with and without inter-level parallelism, respectively. Clearly, the concurrency provided by the 1D methods is unsatisfactory, especially on small size matrices or problems which include strongly over-determined fronts; this explains the poor performance reported, for some problems, in Figure 3.9. The 2D methods, instead, provide much more parallelism and lead to good performance and scalability as shown below.

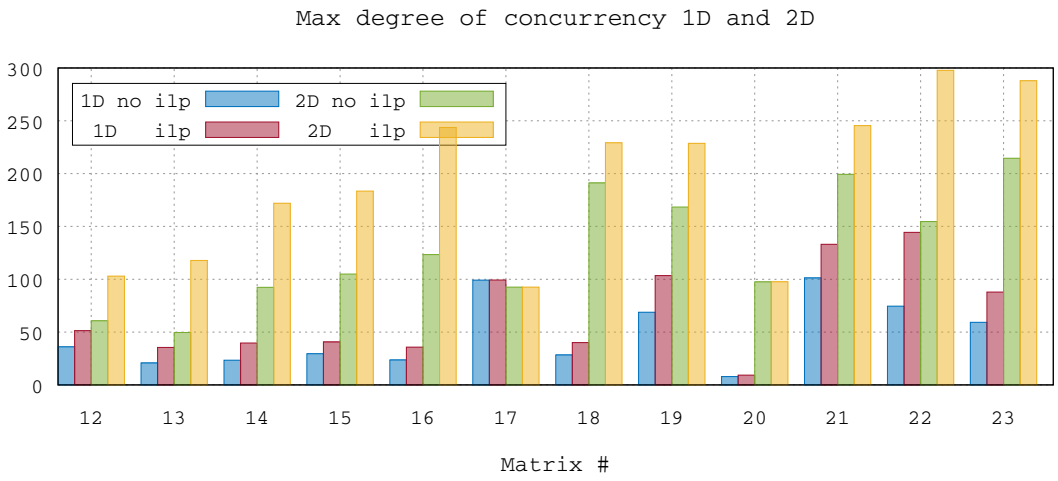


Figure 3.15: Maximum degree of concurrency on `ada` (32 cores) coming from the DAG with and without inter-level parallelism for both 1D and 2D partitioning.

The development of this version also included a number of other, minor improvements:

- In our implementation tiles do not have to be square but can be rectangular with more rows than columns. This is only a minor detail from an algorithmic point of view but, as far as we know, it has never been discussed in the literature and, as shown by the experimental results below, provides considerable performance benefits for our case;

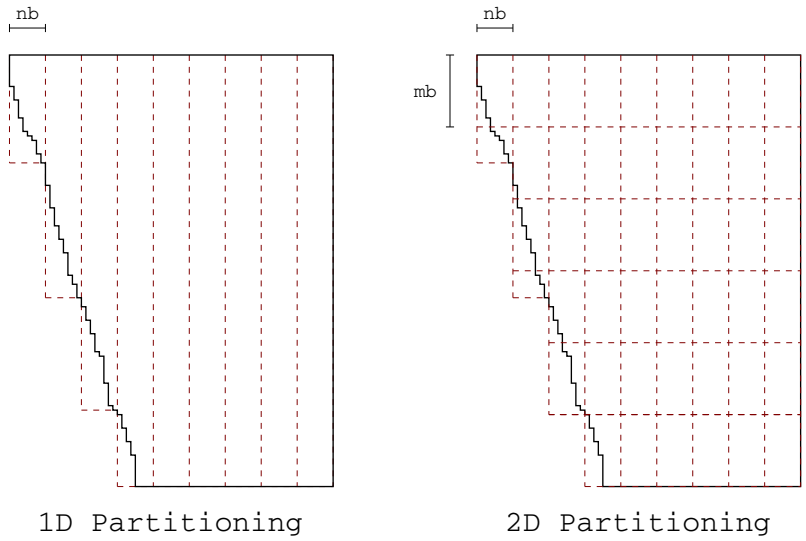


Figure 3.16: 1D partitioning of a frontal matrix into block-columns (*left*), 2D partitioning into tiles (*right*) with blocked storage.

- As in the 1D case presented in the previous section, block storage is also used in this case, as shown in Figure 3.16 (`right`). In addition to the memory savings, which are the same as in the 1D case, here the block storage also benefits the performance because of the lower leading dimension of the blocks [J15];
- As for `_geqrt` and `_gemqrt`, the `_tpqrt` and `_tpmqrt` LAPACK routines were modified in order to cope as efficiently as possible with the fronts staircase structure by means of an internal blocking `ib`.

This parallelization leads to very large DAGs with tasks that are very heterogeneous, both in nature and granularity; moreover, not only are intra-fronts task dependencies more complex because of the 2D front factorization, but also inter-fronts task dependencies due to the parallelization of the assembly operations. The use of an STF-based runtime system relieves the developer from the burden of explicitly representing the DAG and achieving the execution of the included tasks on a parallel machine.

The resulting implementation was evaluated and compared to the 1D approach; all the experiments were run in the same environment and setting as in the previous section.

Table 3.6 shows the parameter values delivering the shortest execution time along with the corresponding attained factorization time and Gflop/s rate. As explained in the previous section for the 1D case, the performance of the factorization depends on a combination of several parameters. For the parallel 2D STF case these parameters are, the size of the tiles `(mb,nb)`, the type of panel reduction algorithm set by the `bh` parameter described in Section 3.4.1 and the internal block size `ib`. Since the optimum values for parameters depends on a large number of factors it is extremely difficult to choose automatically the best values for every given problems. Moreover we increased the complexity of this problem compared to the 1D case because the number of parameters is greater for 2D algorithms. Therefore, using the same experimental protocol as for the previous strategy we performed a large number of runs for each problem with varying values for all the input parameters, using the same values for all the fronts in the elimination tree, and selected the best results (shortest running time) among those. We tested the following values `(nb,ib)={(160,32), (160,40), (192, 32), (192,64)}`, `mb={nb,`

	Parallel 2D STF					
Mat.	mb	nb	ib	bh	Time (s.)	Gflop/s
12	576	192	32	4	4.303E+00	321.6
13	480	160	40	8	7.217E+00	397.5
14	480	160	40	12	1.426E+01	399.6
15	480	160	40	20	2.427E+01	442.1
16	480	160	32	16	3.781E+01	435.4
17	640	160	40	4	4.784E+01	424.4
18	480	160	40	24	6.922E+01	439.0
19	480	160	32	∞	1.408E+02	441.9
20	576	192	32	24	1.728E+02	379.5
21	576	192	32	∞	4.286E+02	453.7
22	576	192	64	∞	4.807E+02	462.8
23	576	192	64	20	5.642E+02	462.6

Table 3.6: Optimum performance for the STF 2D factorization on `ada` (32 cores).

`nb*2, nb*3, nb*4}` and `bh={4, 8, 12, 16, 20, 24,` ∞`}` for a total of 112 runs per matrix ($bh = \infty$ means that a flat reduction tree was used). Note that compared to the 1D case, concurrency is abundant when using the 2D partitioning. For this reason we choose bigger values for `nb` for two reasons: it increases the granularity of tasks in order to achieve a better BLAS efficiency and it limits the number of tasks in the DAG which contributes to keeping the runtime overhead small. The internal block size `ib`, however, has to be relatively small to keep the flop overhead (see Section 3.4.1) under control.

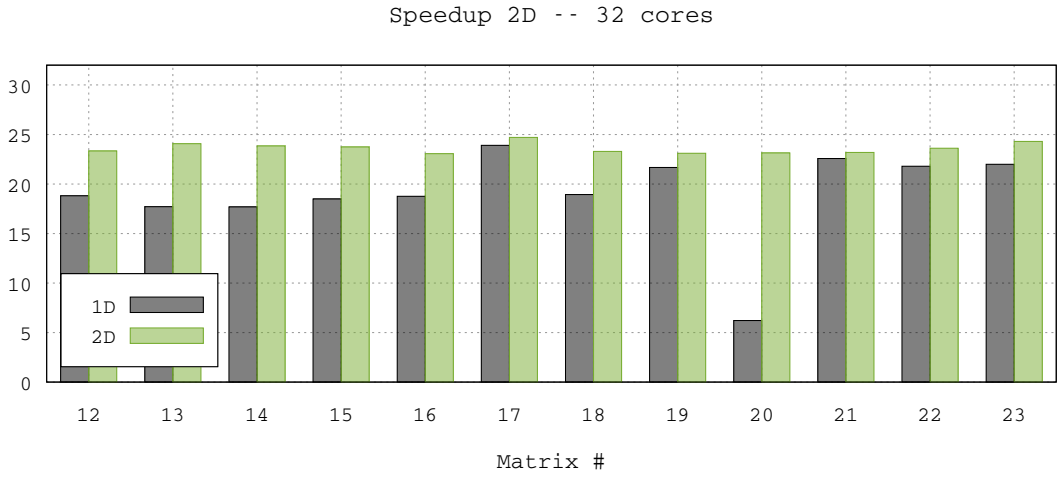


Figure 3.17: Speedup of the 1D and 2D algorithms with respect to the sequential case on `ada` (32 cores).

The speedup achieved by the STF 2D implementation is shown in Figure 3.17 along with the speedup obtained with the STF 1D implementation previously presented in Section 3.3. Results show that the 2D algorithm provides better efficiency on all the tested matrices especially for the smaller ones and those where frontal matrices are extremely

overdetermined, such as matrix #20, where the 1D method does not provide enough concurrency (as explained in Section 3.3, in this problem most of the operations are done on a frontal matrix which has over a million rows and only a few thousands columns). The average speedup achieved by the 2D code is 23.61 with a standard deviation of 0.53, reaching a maximum of 24.71 for matrix #17. For the 1D case, instead, the average is 19.04 with a standard deviation of 4.55. In conclusion, the 2D code achieves better and more consistent scalability over our set of matrices.

3.5 Multifrontal QR for heterogeneous systems

From a technical point of view, porting the StarPU based method described in Sections 3.3 and 3.4 on systems equipped with GPU boards is as simple as providing the StarPU runtime with a GPU-executable version of the tasks kernels. For example, assuming the 1D approach described in Section 3.3, it would suffice to provide StarPU with a GPU implementation of the `_gemqrt` routine, which is where the large majority of the flops is done, to use the accelerator; such a kernel, for instance, can be found in the MAGMA [5] library. The runtime will take care of offloading some of these tasks on the GPU board and move there all the data that are needed for their execution. This, however, does not necessarily means that the GPU will be used to its full potential. For example, because of the way they are created and submitted, tasks may be unsuited for the GPU; this may not only result in a poor acceleration of the tasks, but may actually reduce the overall performance because of the overhead imposed by data transfers between CPU and GPU memories. In essence, the runtime provides a mean of using the accelerator but we still have to make sure that our algorithm complies with its features and capabilities. In addition, we are not only interested in efficiently using the GPU but we want to take advantage of all the computing resources of the underlying machine, that is, both the CPUs and the GPUs whose performance and capabilities are extremely different. We are thus faced with a problem of *heterogeneity*. We have formulated this heterogeneity problem along three main issues:

Granularity : it is well known that GPUs achieve their full speed when the size of computations is relatively large; we would thus be temped to choose a large grain partitioning for our data but this could severely reduce concurrency and lead the CPUs (which may be numerous) to starvation.

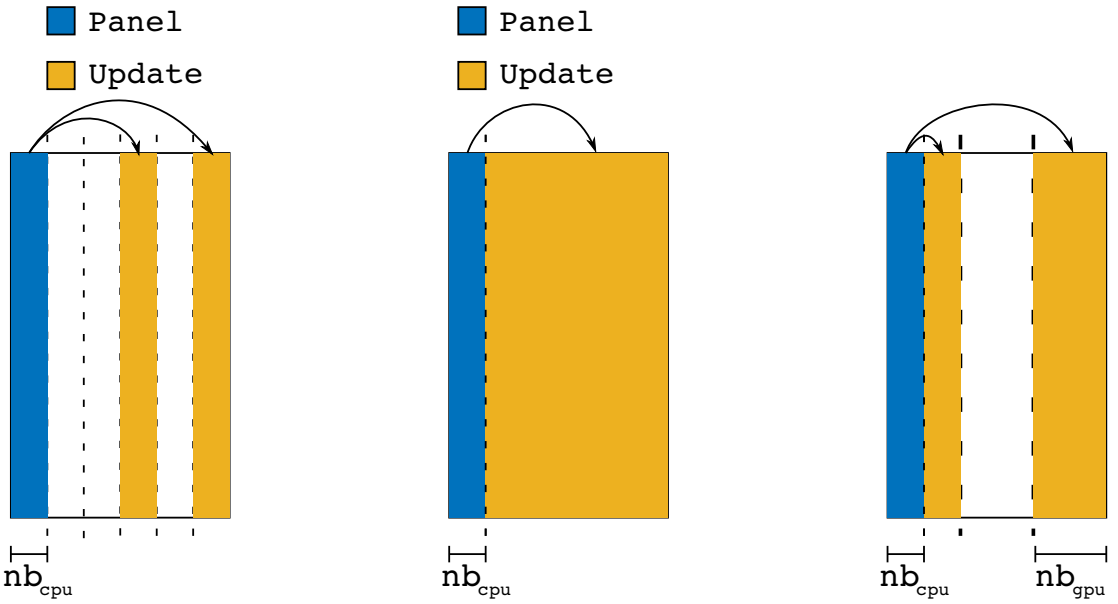
Scheduling : at any time during the DAG traversal, multiple tasks may be ready for execution and multiple processing units may be idle and ready to execute one of them. Because the tasks may have radically different characteristics and the processing units different speeds and capabilities, it is not easy to choose which unit executes which task.

Communications : data has to be moved back and forth from the main memory to the GPUs memory which implies an overhead that can heavily harm performance. This problem can be overcome either by reducing the number and volume of communications, which can be formulated as a scheduling problem, or by making sure that these communications are not (or less) harmful, for example by hiding them behind useful computations.

In the remainder of this section we will discuss methods to address these issues.

3.5.1 Frontal matrices partitioning schemes

Finding frontal matrix partitioning strategies that allow for an efficient use of both CPU and GPU resources represents one of the main challenges in implementing a multifrontal method for GPU-accelerated multicore systems. This results from the fact that GPUs, which are potentially able to deliver much higher performance than CPUs, require coarse granularity operations to achieve high performance while a CPU core reaches its peak with relatively small granularity tasks. The approach presented in Section 3.3, where frontal matrices are uniformly partitioned into small size block-columns, could be readily ported to GPU-accelerated platforms by simply providing GPU implementations for the various tasks to the StarPU runtime system. This, however, would result in an unsatisfactory performance because, due to the fine granularity of tasks, only a small fraction of the GPU performance could be used. This front partitioning strategy, which we refer to as *fine-grain* partitioning, shown in Figure 3.18(a), is not suited to heterogeneous architectures despite being able to deliver sufficient concurrency to feed both the CPU cores and the GPU and reduce idle times on all the resources. A radically different approach is what we refer to as *coarse-grain* partitioning (Figure 3.18(b)), where fine-grained panel tasks are executed on CPU and large-grain (as large as possible) update tasks are performed on GPU. This corresponds to the method used in the MAGMA package [5] and aims at obtaining the best acceleration of computationally intensive tasks on the GPU. In order to keep the GPU constantly busy, static scheduling is used that allows for overlapping GPU and CPU computations thanks to a depth-1 lookahead technique; this is achieved by splitting the trailing submatrix update into two separate tasks of, respectively, fine and coarse granularity. This second approach clearly incurs the opposite problem than the one we face with the fine-grain partitioning: despite being able to maximize the efficiency of GPU operations with respect to the problem size, it severely limits the amount of node parallelism as well as of inter-level parallelism and therefore leads to resource (especially CPUs) starvation.



(a) Fine-grain partitioning.

(b) Coarse-grain partitioning.

(c) Hierarchical-grain partitioning.

Figure 3.18: Partitioning schemes.

```
forall outer panels o_p=1...o_n in f
   ! partition (outer) block column f(o_p) into
   ! i_n inner block columns f(o_p,1) .. f(o_p,i_n)
   call submit(partition, f(o_p):R, f(o_p,1):W... f(o_p,i_n):W)

   forall inner panels i_p=1..i_n
      ! panel reduction of inner block column i_p
      call submit(_geqrt, f(i_p):RW)
      forall inner blockcolumns i_u=i_p+1..i_n in f(o_p)
         ! update (inner) column in_u with panel i_p
         call submit(_gemqrt, f(i_p):R, f(i_u):RW)
      end do

      forall outer blockcolumns o_u=o_p+1..o_n
         ! update outer block column o_u with panel i_p
         call submit(_gemqrt, f(i_p):R,f(o_u):RW)
      end do
   end do

   ! unpartition (outer) block column
   call submit(unpartition, f(o_p,1):R...f(o_p,i_n):R, f(o_p):W)
end do
```

Figure 3.19: STF code for the hierarchical QR factorization of frontal matrices.

In order to take advantage of the fine and coarse-grain approaches and to overcome the limitations of both, we developed a *hierarchical* and dynamic partitioning of fronts (Figure 3.18(c)) which is similar to the approach proposed by Wu et al. [162] and corresponds to a trade-off between parallelism and GPU kernel efficiency with task granularity suited for both types of resources. Frontal matrices are first partitioned into coarse grain block-columns, referred to as outer block-columns, of width $\texttt{nb}_{\texttt{GPU}}$ suitable for GPU computation (this happens at the moment when the front is activated) and then each outer block-column is dynamically re-partitioned into inner block-columns of width $\texttt{nb}_{\texttt{CPU}}$ appropriate for the CPU only immediately before being factorized. This is achieved through dedicated partitioning tasks which are subject to dependencies with respect to the other, previously submitted, tasks that operate on the same data. When these dependencies are satisfied, StarPU ensures that the block being re-partitioned is in a consistent state, in case there are multiple copies of it. Furthermore, StarPU ensures that the partitioning is done in a logical fashion: no actual copy is performed and there is no extra data allocated. The partitioning is done using two tasks: `partition` and `unpartition`. In order to partition a data i represented by the handle `f(i)` into n pieces, it is necessary to declare the handles associated with the sub-data `f(i,1)..f(i,n)`. The `partition` task takes as input the data to be partitioned with a `Read` access mode and the resulting sub-data with a `Write`. The `unpartition` tasks take as input the sub-data with a `Read` access mode and the original data with a `Write` access mode. In the STF code, as long as all tasks working on sub-data are submitted between the `partition` and `unpartition` tasks and no tasks working on the partitioned data are submitted, the data consistency between data and sub-data is ensured. It should be noted that in order to avoid memory copy, both `partition` and `unpartition` tasks should be executed on the node where the data is allocated and in this case these tasks are associated with an empty function.

In order to use the hierarchical-grain partitioning in the multifrontal factorization, the initial STF code corresponding to the QR factorization of a front using a fine-grain

partitioning (lines 14-21 in Figure 3.8) is turned into the one proposed in Figure 3.19 for hierarchically partitioned fronts. We define inner and outer tasks depending on whether these tasks are executed on inner or outer block-columns. In order to ease the understanding we use different names for inner and outer updates although both types of tasks perform exactly the same operation and thus employ the same code.

The work described above has prompted the StarPU developers to implement the dynamic partition and unpartition capability in the runtime system. As a result, the StarPU API now includes the `starpu_data_partition_plan`, `starpu_data_partition_submit` and `starpu_data_unpartition_submit` which allow for, associating a partitioning scheme with some data handle and submitting a partition and an unpartition task respectively.

3.5.2 Scheduling strategies

Along with the fronts partitioning strategies discussed in the previous section, task scheduling is a key factor for archiving reasonable performance on heterogeneous systems. One strategy to schedule the tasks resulting from the partitioning is to statically assign the coarse granularity tasks to GPUs and fine granularity tasks to CPU cores. This is the strategy adopted in the work by Lacoste [111] where the GPU kernels are statically mapped onto the devices. However, in our problem, the variety of front shapes and staircase structures combined with this hierarchical partitioning induces an important workload heterogeneity, making load balancing extremely hard to anticipate. For this reason, we chose to rely on a dynamic scheduling strategy.

In the context of a heterogeneous architecture, the scheduler should be able to handle the workload heterogeneity and distribute the tasks taking into account a number of factors including resource capabilities or memory transfers while ensuring a good load balance between the workers. Dynamic scheduling allows for dealing with the complexity of the workload and limits load imbalance between resources.

Algorithms based on the Heterogeneous Earliest Finish Time (HEFT) scheduling strategy by Topcuouglu et al. [153] represent a commonly used and well known solution to scheduling task graphs on heterogeneous systems. These methods consist in first ranking tasks (typically according to their position with respect to the critical path) and then assigning them to resources using a minimum completion time criterion. Despite the fact that GPUs can accelerate the execution of most (if not all) tasks, not all tasks are accelerated by the same amount, depending on their type (e.g., compute or memory bound, regular or irregular memory access pattern) or granularity: therefore we say that some tasks have a better acceleration factor on the GPU than others. The main drawback of HEFT-like methods lies in the fact that the acceleration factor of tasks is ignored during the worker selection phase, i.e., these methods do not attempt to schedule a task on the unit which is best suited for its execution. In addition, the centralized decision during the worker selection potentially imposes a significant runtime overhead during the execution. A performance analysis conducted with the so-called `dmdas` StarPU built-in implementation of HEFT showed that these drawbacks are too severe for designing a high-performance multifrontal method.

Instead, we implemented and extended a scheduling technique known as HeteroPrio, first introduced by Agullo et al. [4] in the context of Fast Multipole Methods (FMM). This technique is inspired by the observation that a DAG of tasks may be extremely irregular with some part where concurrency is abundant and others where it is scarce as shown in Figure 3.20. In the first case we can perform tasks on the units where they are executed the most effectively without any risk of incurring resource starvation because parallelism is plentiful. In the second case, however, what counts most is to prioritize tasks which

lie along the critical path because delaying their execution would result in penalizing stalls in the execution pipeline. As a result, in the HeteroPrio scheduler the execution is characterized by two states: a *steady-state* when the number of tasks is large compared to the number of resources and a *critical-state* in the opposite case. The scheduler can automatically switch from one state to another depending on a configurable criterion which mostly depends on the amount of ready tasks and of computational resources.

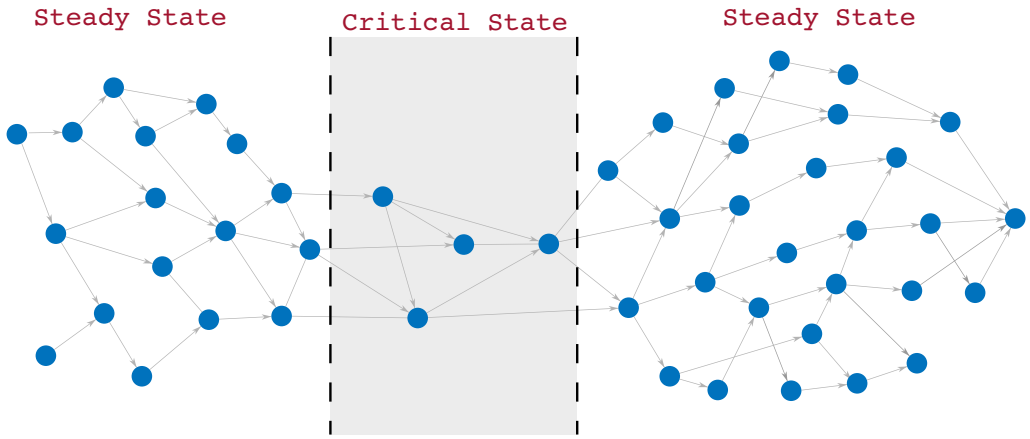


Figure 3.20: Example of a DAG with a succession of steady and critical states corresponding to rich and poor concurrency regions respectively as defined in the HETEROPRIO scheduler.

A complex, irregular workload, such as a sparse factorization, is typically a succession of steady and critical state phases, where the steady-state corresponds to rich concurrency regions in the DAG whereas the critical-state corresponds to scarce concurrency regions. During a steady-state phase, tasks are pushed to different scheduling queues depending on their expected acceleration factor (see Figure 3.21). In our current implementation, we have defined one scheduling queue per type of tasks (eight in total as listed in column 1 of Table 3.7). When they pop tasks, CPU and GPU workers poll the scheduling queues in different orders. The GPU worker first polls scheduling queues corresponding to coarse-grain tasks such as outer updates (priority 0 on GPU in Table 3.7) because their acceleration factor is higher. On the contrary, CPU workers first poll scheduling queues of small granularity such as subtree factorizations or inner panels (as well as tasks performing symbolic work such as activation that are critical to ensure progress). Consequently, during a steady-state, workers process tasks that are best suited for their capabilities. The detailed polling orders are provided in Table 3.7. Furthermore, to ensure fairness in the progress of the different paths of the elimination tree, tasks within each scheduling queue are sorted according to the distance (in terms of flop) between the corresponding front and root node of the elimination tree.

In the original HeteroPrio scheduler [4], the worker selection is performed right before popping the task in a scheduling queue following the previously presented rules. If data associated with the task are not present on the memory node corresponding to the selected worker then the task completion time is increased by the memory transfers. While the associated penalty is usually limited in the FMM case [4], preliminary experiments (not reported here for a matter of conciseness) showed that it may be a severe drawback for the multifrontal method. For the purpose of the present study, we have therefore extended the original scheduler by adding worker queues (one queue per worker) along with the scheduling queues as shown in Figure 3.21. When it becomes idle, a worker pops a task

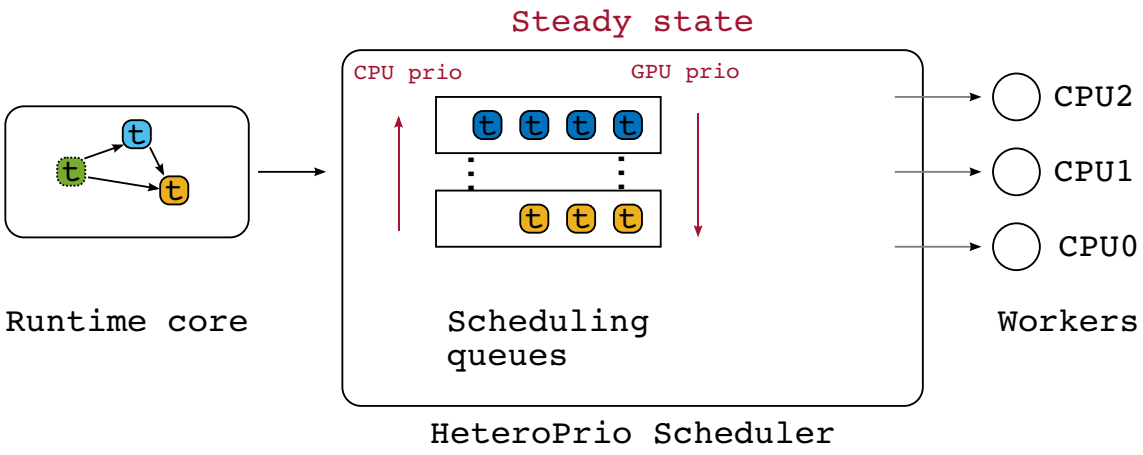


Figure 3.21: HeteroPrio steady-state policy.

from its worker queue and then fills it up again by picking a new task from the scheduling queues through the polling procedure described above. The data associated with tasks in a worker queue can be automatically prefetched on the corresponding memory node while the corresponding worker is executing other tasks. If the size of the worker queues is too high, a task may be assigned to a worker much earlier than its actual execution, which may result in a sub-optimal choice. Therefore, because no additional benefit was observed beyond this value, we set this size to two in our experiments.

Scheduling queues	Steady-state		Critical-state	
	CPU	GPU	CPU	GPU
activate	0	-	0	-
assemble	7	-	5	-
deactivate	1	-	1	-
do_subtree	2	2	2	0
part./unpart.	3	-	3	-
inner_panel	4	-	4	-
inner_update	5	1	6	1
outer_update	6	0	7	2

Table 3.7: Scheduling queues and polling orders in HeteroPrio.

When the number of tasks becomes low (with respect to a fixed threshold which is set depending on the amount of computational power of the platform), the scheduling algorithm switches to critical-state. CPU and GPU workers cooperate to process critical tasks as early as possible in order to produce new ready tasks quickly. For instance, because outer updates are less likely to be on the critical path, the GPU worker will select them last in spite of their high acceleration factor. The last two columns of Table 3.7 provide the corresponding polling order. Additionally, in this state, CPU workers are allowed to select a task only if its expected completion time does not exceed the total completion time of the tasks remaining in the GPU worker queue. This extra rule prevents CPU workers from selecting all the few available tasks and leaving the GPU idle whereas it could have finished processing them all quickly.

3.5.3 Implementation and experimental results

We implemented the partitioning and scheduling methods described in the previous two sections within the `qr_mumps` solver. As explained above, only the `_gemqrt` (i.e., updates) and `do_subtree` tasks can be executed on the GPU. For the first, we used the GPU implementation of the LAPACK `_gemqrt` routine provided by the MAGMA library; note that this version does not take into account the staircase structure and, thus, performs more flops than necessary. For the second, we wrote our own GPU-enable implementation. This performs a sequential traversal of the associated subtree: frontal matrices are assembled on the CPU and then are factorized with a dense QR factorization on the GPU using the same method as in the MAGMA `_geqrt` routine which we modified to take advantage of the staircase structure. Note that this hybrid kernel is possible because in StarPU a GPU worker is associated with a GPU and a CPU core which is meant to drive it and thus both units can be used by a single task. At first, we only implemented the 1D partitioning scheme discussed in Section 3.3 because no `_tpmqrt` GPU routine is available in MAGMA; results obtained with the use of 2D communication-avoiding front factorization methods are presented in Section 3.5.4.

Tests were executed for a subset of the matrices of Table A.1 on the `sirocco` system which is equipped with two twelve-cores CPUs and 4 Nvidia K40 boards.

For the CPU-only experiments, we used a fine grained decomposition and the parameter values used for the experiments were `(nb,ib)={(128,64), (128,128), (192,64), (192,192), (256,64), (256,128), (256,256)}`.

For the heterogeneous experiments, we use the hierarchical-grain partitioning presented in section 3.5.1. The parameters defining a hierarchical block column partitioning are the size of the outer block column $\texttt{nb}_{\texttt{gpu}}$ and the size of the inner block column $\texttt{nb}_{\texttt{gpu}}$. The value `ib` is fixed such that $\texttt{ib} = \texttt{nb}_{\texttt{cpu}}$ because, as explained above the `_gemqrt` operation on the GPU cannot take advantage of the staircase structure and because, as seen in Section 3.3, this is commonly the choice which yields the best performance. As for the multicore case, we performed a large set of tests with several combinations for $\texttt{nb}_{\texttt{cpu}}$ and $\texttt{nb}_{\texttt{gpu}}$ and selected the best results in terms of factorization time. The values used for the experiments were $(\texttt{nb}_{\texttt{gpu}},\texttt{nb}_{\texttt{cpu}})$`={(256,128), (256,256), (384,128), (384,384), (512,128), (512,256), (512,512) (768,128), (768,256), (768,384), (896,128), (1024,128), (1024,256), (1024,512) }`. The scheduler used for the experiments in an heterogeneous context is HeteroPrio presented in Section 3.5.2.

The performance of the code in the multicore case using a fine-grain partitioning is reported in Table 3.8 with two configurations: the first using the twelve cores of a E5-2680 processor and the second using the twenty-four cores of two E5-2680 processors available on the machine. The table shows the shortest execution time along with the corresponding Gflop/s rates obtained for the factorization of the tested matrices and the optimal parameters `ib` and `nb` for which this performance was attained.

The performance of the code in the heterogeneous case using a hierarchical partitioning is reported in Table 3.9 with two configurations: first using the twelve cores of a E5-2680 processor plus one GPU K40M and second using the twenty-four cores of two E5-2680 processors available on the machine plus one GPU K40M. The table shows the shortest execution time along with the corresponding Gflop/s rates obtained for the factorization of the tested matrices and the optimal parameters $\texttt{nb}_{\texttt{gpu}}$ and $\texttt{nb}_{\texttt{cpu}}$ for which these performance were attained. Note that in the case were $\texttt{nb}_{\texttt{gpu}} = \texttt{nb}_{\texttt{cpu}}$, then the hierarchical-grain partitioning is equivalent to the fine-grain partitioning. Our implementation can also take advantage of multiple GPU streams. On a GPU, a stream is defined as a sequence of com-

	12 CPUs (1×E5-2680)				24 CPUs (2×E5-2680)			
Mat.	nb	ib	Time (s.)	Gflop/s	nb	ib	Time (s.)	Gflop/s
12	192	64	8.892E+00	138.6	128	128	4.922E+00	275.3
13	128	128	1.188E+01	226.7	128	128	8.525E+00	316.0
14	192	192	2.364E+01	221.0	128	128	1.444E+01	351.8
15	128	128	4.151E+01	252.1	128	128	2.468E+01	424.0
16	192	192	5.849E+01	272.3	128	128	3.734E+01	421.0
17	128	64	7.181E+01	271.7	128	64	4.270E+01	457.0
18	128	128	1.104E+02	248.8	128	128	6.209E+01	442.5
19	192	192	2.212E+02	284.0	128	128	1.269E+02	489.3
21	192	192	6.318E+02	294.6	192	192	3.352E+02	555.2

Table 3.8: Optimum performance for the STF fine-grain 1D factorization on `sirocco` in homogeneous case with both configurations 12 CPUs (1×E5-2680) and 24 CPUs (2×E5-2680).

mands that execute in order and is a feature available on relatively recent GPUs [5]. The use of multiple streams allows for the concurrent execution kernels on the device thus increasing the occupancy of the GPU when executing small grain tasks that are unable to feed all the available resources. This allows us to use smaller values for the parameters $\texttt{nb}_{\texttt{cpu}}$ and $\texttt{nb}_{\texttt{gpu}}$ leading to a greater concurrency in the DAG and thus potentially better performance. Table 3.9 reports the best execution time when using either one, two or four streams (the corresponding number of streams is reported in column "s").

The results in Tables 3.83.9 show that the proposed approach can make a good use of all the available computing resources because a significant speedup is achieved when more computational resources are available regardless of their type. This leads to a very good overall performance which reaches almost 800 Gflop/s.

	12 CPUs (1×E5-2680) + 1 GPU (1×K40M)					24 CPUs (2×E5-2680) + 1 GPU (1×K40M)				
Mat.	nb_{gpu}	nb_{cpu}	s	Time (s.)	Gflop/s	nb_{gpu}	nb_{cpu}	s	Time (s.)	Gflop/s
12	384	384	1	8.302E+00	214.1	384	384	1	6.960E+00	255.4
13	512	256	4	7.935E+00	368.3	256	256	4	8.117E+00	360.0
14	384	384	4	1.539E+01	365.3	256	256	1	1.404E+01	382.0
15	768	256	4	1.748E+01	616.1	768	128	2	1.839E+01	569.0
16	896	128	2	2.470E+01	636.4	896	128	4	2.558E+01	614.5
17	192	192	1	8.700E+01	246.7	256	256	1	6.797E+01	329.9
18	512	256	4	3.882E+01	710.2	512	256	4	3.838E+01	718.3
19	768	256	4	9.610E+01	661.3	768	384	2	8.142E+01	797.9
21	768	256	2	2.792E+02	671.4	1024	256	1	2.403E+02	780.1

Table 3.9: Optimum performance for the STF hierarchical-grain 1D factorization on `sirocco` in heterogeneous case with both configurations 12 CPUs (1×E5-2680) + 1 GPU (1×K40M) and 24 CPUs (2×E5-2680) + 1 GPU (1×K40M).

[5] Available on compute capability 2.x and higher devices but exploitable since compute capability 3.5 with the introduction of Hyper-Q technology.

3.5.4 Combining communication-avoiding methods and GPUs

The approach described above and that we evaluated on GPU-equipped machines relies on a 1D frontal matrices partitioning into block-columns. In more recent evolution of the `qr_mumps` code we have combined the use of a 2D decomposition with communication-avoiding front factorizations (as described in Section 3.4 for CPU-only machines) with the dynamic, hierarchical partitioning and the HeteroPrio scheduling (described in Sections 3.5.1 and 3.5.2, respectively) in order to achieve higher performance on systems equipped with multiple GPUs.

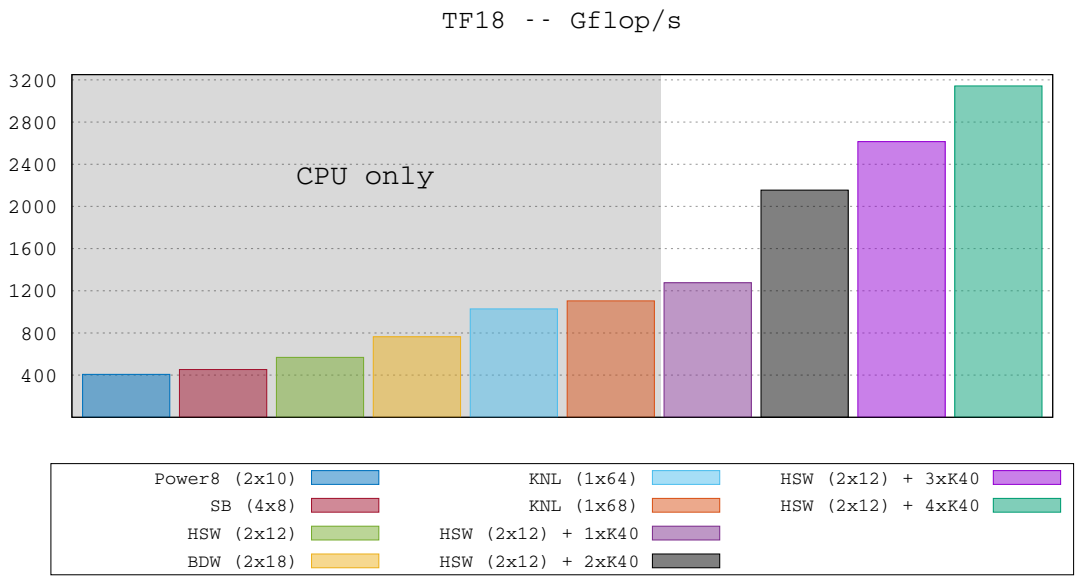


Figure 3.22: Performance achieved by the factorization of matrix #21 on a range of different machines, both CPU-only or CPU+GPU.

The results obtained with this new code are reported in Figure 3.22 for a single matrix (i.e., matrix #21) on both CPU-only and CPU+GPU systems. These results show that the proposed approach is able to achieve good absolute performance and great portability over a large set of systems. These include "traditional" multicore processors with up to 36 cores, Intel Knights Landing processors with 64 and 68 cores, and a system with 24 cores plus four Nvidia K40 GPUs. On this last system, when all the four GPUs are used, the achieved performance is close to 3.2 Tflop/s which is quite remarkable for a sparse linear algebra code on a single node machine.

3.6 A performance analysis approach for task-based parallelism

Performance profiling and analysis is one of the cornerstones of High Performance Computing. Nonetheless, it may be a challenging task to achieve, especially in the case of complex applications or algorithms and large or complex architectures. Therefore, it is no surprise that performance profiling has been the object of a very vast amount of literature.

In the case of sequential applications, a rather simple but effective approach consists in computing the efficiency of the code as a ratio of the attained speed and a reference

performance which depends on the peak capability of the underlying architecture. Because processing units and memories work at different speeds, this reference performance varies depending on whether the application is compute bound (i.e., limited by the speed of the processing unit) or memory bound and to what extent. The *Roofline model* [161] is a popular method for computing this performance upper bound as a function of the *arithmetic intensity*:

$$\begin{matrix}\text{Attainable}\\ \text{Gflop/s}\end{matrix} = min \left\{ \begin{matrix}\text{Peak Floating-point}\\ \text{Performance}\end{matrix} , \begin{matrix}\text{Peak Memory}\\ \text{Bandwidth}\end{matrix} \times \begin{matrix}\text{Arithmetic}\\ \text{intensity}\end{matrix} \right\}.$$

The peak floating-point performance and peak memory bandwidth can be set equal to the theoretical values of the architecture; these values, however, are commonly unattainable and therefore these parameter values are computed using benchmarks like the BLAS `_gemm` (matrix-matrix multiply) operation, which is commonly considered as the fastest compute-bound operation, or the STREAM [127] benchmark, respectively. The roofline model can also be used for shared-memory, parallel applications but cannot be extended to the case where accelerators are used or to distributed-memory, parallel codes. Moreover, its use is difficult in the case of complex applications, such as the multifrontal method, which include both memory and compute-bound operations whose relative weight varies depending on the input problem.

The performance of a parallel code, either shared or distributed-memory, on a homogeneous platform (i.e., where all the processing units are the same) is commonly assessed measuring its *speedup*, i.e., the ratio between the sequential and the parallel execution times $t(1)/t(p)$, or, equivalently, the *parallel efficiency*

$$e(p) = \frac{t(1)}{t(p) \times p}$$

where p is the number of processing units used. The *scalability* measures the ability of a parallel code to reduce the execution time as more resources are provided. *Amadahl's law* can be used to define a bound on the achievable speedup (or parallel efficiency or scaling) of a parallel code but, again, this is very hard to achieve for complex and irregular applications.

The emerging heterogeneous architectures represent a challenge for the performance evaluation of parallel algorithms compared to the uni-processor and parallel, homogeneous environments. Accelerators, not only process data at different speeds compared to the CPUs but also have different capabilities, i.e., are more or less suited to different types of operations, and are attached to their own memory which has different latency and bandwidth than that on the host. The performance analysis of codes that use accelerators is often limited to measuring the added performance brought by the accelerators, that is, a simple speed comparison with the CPU-only execution.

Although the above presented techniques can be used to achieve a rough evaluation of the performance of a parallel code, none of them provides any insight which can guide the HPC expert in reformulating or improving his algorithms or the programmer in optimizing his code in order to achieve better performance. Many factors play an important role in the performance and scalability of a code; among the others, we can mention the cost of data transfers and synchronizations, the granularity of operations, the properties of the algorithm and the amount of concurrency it can deliver. A quantitative evaluation of these factors can be extremely valuable.

In order to evaluate the effectiveness of the techniques proposed above as well as of the software that implements them, we developed a novel performance analysis approach [J2, C4]. First, we introduce a method for computing a relatively tight upper bound for the performance attainable by the parallel code, which is not merely a sum of the peak performance of the available processing units; this performance reference is computed by not only taking into account the features of the underlying architecture, but also the properties of the implemented algorithm and allows for evaluating the efficiency of a parallel code. Then we show how it is possible to factorize this efficiency measure into a product of terms that allow for assessing, singularly, the effect of several factors playing a role in the performance and scalability of a code. This analysis requires the ability to retrieve specific information from the execution that are easy to gather when using a runtime system.

Consider the problem of evaluating the execution of a parallel application on a target computing environment composed of p heterogeneous processors such as CPUs and GPUs workers. Critical factors playing a role on parallel executions must be considered to compute realistic performance bounds and understand to what extent each of these factors may limit the performance. Idle times and data transfers (communications, in general) for example represent a major bottleneck for performance of parallel executions. Also we seek to quantify the cost of the runtime system, if any, compared to the workload in order to evaluate the effectiveness of these tools and estimate the overhead they induce.

In the proposed performance evaluation approach we perform a detailed analysis of the execution times by considering the cumulative times spent by all threads in the main phases of the execution:

- $t_t(p)$: The time spent in tasks which represent the workload of the application;
- $t_r(p)$: The time spent in the runtime for handling the execution of the application (in our case, this includes building the DAG and scheduling the tasks);
- $t_c(p)$: The time spent performing communications that are not overlapped by computations. This corresponds to the time spent by workers waiting for data to be transferred on their associated memory node before being able to execute a task;
- $t_i(p)$: The idle time spent waiting for dependencies between tasks to be satisfied.

The execution time of the factorization $t(p)$, may be expressed, using these cumulative times, as follows:

$$t(p) = \frac{t_t(p) + t_r(p) + t_c(p) + t_i(p)}{p}$$

The efficiency of a parallel code can be defined as

$$e(p) = \frac{t^{min}(p)}{t(p)}$$

where $t^{min}(p)$ is a lower bound on the execution time with p processes. A possible way of computing $t^{min}(p)$ for a task graph is to measure the execution time associated with the optimal schedule. However, given the complexity of the task scheduling problem in the general case it is not reasonable to compute this $t^{min}(p)$ for any input problem. Instead we choose to use a looser bound that consists in computing the optimal value of $t^{min}(p)$ for a relaxed version of the initial scheduling problem built on the following assumptions:

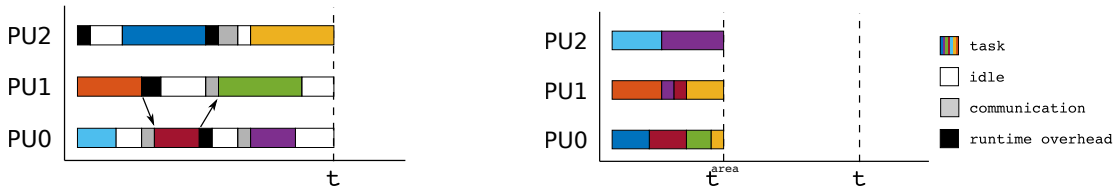
1. There are no dependencies between tasks which means that we consider an embarrassingly parallel problem, i.e., $t_i(p) = 0$;
2. The runtime does not induce any overhead on the execution time, i.e., $t_r(p) = 0$;
3. The cost of all data transfers is equal to zero, i.e., $t_c(p) = 0$;
4. Tasks are *moldable* meaning that they may be processed by multiple processors.

In order to compute $t^{min}(p)$ we introduce the following notation: for a set of tasks Ω running on a set of p resources denoted by R, we define t_r^ω as the time that resource r would take to process the entire task ω and α_r^ω as the share of work in task ω actually processed by resource r. Then $t^{min}(p)$ can be computed as the solution of the following linear program:

Linear Program 1— *Minimize T such that, for all $r \in R$ and for all $\omega \in \Omega$:*

$$\sum_{\omega \in \Omega} \alpha_r^\omega t_r^\omega = t_r \leq T \qquad \sum_{r=1}^{|R|} \alpha_r^\omega = 1$$

where the t_r^ω can be computed using a performance model. Note that the problem of finding $t^{min}(p)$ is equivalent to minimizing the area $\alpha_r^\omega t_r^\omega$ for all $\omega \in \Omega$ and $r \in R$. For this reason the optimal value $t^{min}(p)$ is replaced by $t^{area}(p)$ in the following. We illustrate how $t^{area}(p)$ is defined in Figure 3.23 on a simple execution with three resources. On the left of the figure is represented the trace of the real execution where $t(p)$ is measured. On the right is represented the optimal schedule for the relaxed version of the original scheduling problem where $t^{area}(p)$ is measured.



(a) Actual schedule. (b) Area schedule.

Figure 3.23: Illustration on a simple Gantt chart of a parallel execution with three workers.

Note that parallelism is normally achieved by partitioning operations and data; this implies a smaller granularity of tasks and thus, likely, a poorer performance of the operations performed by them. Moreover, parallel algorithms often trade floating-point operations for concurrency and therefore may perform more operations than the corresponding sequential ones (this is, for example, the case of 2D communication avoiding QR factorizations, as explained in Section 3.4.1). Based on these observations, we can further refine the efficiency definition above replacing $t^{area}(p)$ with $\tilde{t}^{area}(p)$ computed as the solution of Linear Program 1 assuming $\omega \in \tilde{\Omega}$, where $\tilde{\Omega}$ is the set of tasks of the sequential algorithm. In other words, in $\tilde{t}^{area}(p)$ we assume that there is no performance loss when working on partitioned data and that the parallel algorithm has the same cost as the sequential one. Please note that this also models the fact that in some cases data are inherently of small granularity and, therefore, tasks that work on them have a poor performance regardless of the partitioning; in the multifrontal method, for example, this is the case for the small frontal matrices at the bottom of the elimination tree.

By replacing the term $t(p)$ in the expression of the parallel efficiency using cumulative times and noting that $\tilde{t}_t^{area}(p) = p \times \tilde{t}^{area}(p)$ from the definition of our lower bound, we may express the parallel efficiency as

$$e(p) = \frac{\tilde{t}^{area}(p)}{t(p)} = \frac{\tilde{t}^{area}(p) \times p}{t_t(p) + t_r(p) + t_c(p) + t_i(p)} = \frac{\tilde{t}_t^{area}(p)}{t_t(p) + t_r(p) + t_c(p) + t_i(p)}$$
$$= \overbrace{\frac{\tilde{t}_t^{area}(p)}{t_t^{area}(p)}}^{e_g} \cdot \overbrace{\frac{t_t^{area}(p)}{t_t(p)}}^{e_t} \cdot \overbrace{\frac{t_t(p)}{t_t(p) + t_r(p)}}^{e_r} \cdot \overbrace{\frac{t_t(p) + t_r(p)}{t_t(p) + t_r(p) + t_c(p)}}^{e_c} \cdot \overbrace{\frac{t_t(p) + t_r(p) + t_c(p)}{t_t(p) + t_r(p) + t_c(p) + t_i(p)}}^{e_p}.$$

This expression allows us to decompose the parallel efficiency as the product of five well identified effects:

- e_g: the **granularity efficiency**, which measures how the overall efficiency is reduced by the data partitioning and the use of parallel algorithms. This loss of efficiency is mainly due to the fact that because of the partitioning of data into fine grained blocks, elementary operations do not run at the same speed as in the purely sequential code and also to the fact that the parallel algorithm may perform more flops than the sequential one;
- e_t: the **task efficiency**, measures how well the assignment of tasks to processing units matches the tasks properties to the units capabilities as well as the exploitation of data locality (more details are provided below);
- e_r: the **runtime efficiency**, which measures the cost of the runtime system with respect to the actual work done;
- e_c: the **communication efficiency**, which measures the cost of communications with respect to the actual work done due to data transfers between workers;
- e_p: the **pipeline efficiency**, which measures how well the tasks have been pipelined. This includes two effects. First, the quality of the scheduling because if the scheduling policy takes bad decisions (for example, it delays the execution of tasks along the critical path) many stalls can be introduced in the pipeline. Second, the shape of the DAG or, more generally, the amount of concurrency it delivers: for example, in the extreme case where the DAG is a chain of tasks, any scheduling policy will do as bad because all the workers except one will be idling at any time.

3.6.1 Analysis for homogeneous multicore systems

In the case of an homogeneous architecture such as a multicore system with p cores, the solution of Linear Program 1 greatly simplifies and can be easily found.

$\tilde{t}^{area}(p)$ and $t^{area}(p)$ do not have to be computed but can be measured by timing, respectively, the sequential execution of the sequential algorithm and the sequential execution of the parallel algorithm (with the corresponding data partitioning), that is

$$\tilde{t}^{area}(p) = \frac{\tilde{t}(1)}{p}, \quad t^{area}(p) = \frac{t(1)}{p}. \tag{3.2}$$

Note that in a sequential execution the cost of the communications and of the runtime as well as the idle times are all identically equal to zero and therefore $t(1) = t_t(1)$ and $\tilde{t}(1) = \tilde{t}_t(1)$.

Replacing $t(p)$ and $t^{area}(p)$ in the expression of the parallel efficiency using the cumulative times we obtain:

$$e(p) = \frac{\tilde{t}_t(1)}{t_t(p) + t_r(p) + t_i(p)}$$
$$= \overbrace{\frac{\tilde{t}_t(1)}{t_t(1)}}^{e_g} \cdot \overbrace{\frac{t_t(1)}{t_t(p)}}^{e_t} \cdot \overbrace{\frac{t_t(p)}{t_t(p) + t_r(p) + t_c(p)}}^{e_r} \cdot \overbrace{\frac{t_t(p) + t_r(p)}{t_t(p) + t_r(p) + t_i(p)}}^{e_p}.$$

Here we assumed $e_c = 1$ because there are no explicit or measurable data transfers. These, however, happen implicitly when tasks access data which are remotely located in the NUMA memory system; this makes the tasks execution time $t_t(p)$ increase as the number of cores p increases. This effect is measured by e_t.

Figure 3.24 shows the efficiency analysis related to the experimental results presented in Section 3.3 and 3.4 with the 1D and 2D algorithms, respectively. The 2D approach obviously have a lower task efficiency because of the smaller granularity of tasks and because of the extra flops. Note that the 1D code may suffer from a poor cache behavior in the case of extremely overdetermined frontal matrices because of the extremely tall-and-skinny shape of block-columns. This explains why the granularity efficiency for the 2D code on matrix #20 is better than for the 1D code unlike for the other matrices. 2D algorithms, however, achieve better locality efficiency than 1D most likely due to the 2D partitioning of frontal matrices into tiles which have a more cache-friendly shape and size than the extremely tall and skinny block-columns used in the 1D algorithm. The pipeline efficiency results confirm that the 1D approach produces less concurrency on matrices that are of relatively small size or on those whose fronts are extremely over-determined (e.g., matrix #20). Not surprisingly, the 2D code achieves much better pipeline efficiency (i.e., less idle time) than the 1D code on all matrices: this results from a much higher concurrency, which is the purpose of the 2D code. As for the runtime efficiency, it is in favor of the 1D implementation due to a much smaller number of tasks with bigger granularity and simpler dependencies. However the performance loss induced by the runtime is extremely small in both cases: less than 2% on average and never higher than 4% for the 2D implementation.

3.6.2 Analysis for heterogeneous systems

Compared to the homogeneous case, computing $\tilde{t}^{area}(p)$ and $t^{area}(p)$ in the context of heterogeneous systems is more complex because task execution times depend on the type of resources where they are executed. Therefore we need to solve Linear Program 1 to determine these values. The first step consists in gathering the execution times for every task on every possible computational unit. Then the linear program is built either statically if all tasks are known in advance or by registering it during an actual execution and finally solved using a linear program solver.

The tasks efficiency, in this case, still measures the effect of implicit communications but also, and more importantly, how well the heterogeneity of the processing units is exploited. In other words, e_t measures how well tasks have been mapped to computational units with respect to the optimal mapping computed by solving the linear program. Note that this efficiency is not necessarily lower than one and it is closely related to the pipeline efficiency:

- $e_t < 1$: tasks are globally being executed at a lower speed with respect to the optimal. This may happen because the tasks assigned to the different processing units are not

of the good type; this is, for example, the case where very small granularity tasks are executed by GPUs;

- $e_t > 1$: note that a particularly naive scheduling policy can map all the tasks to faster units. This would obviously result in a small cumulative tasks execution time $t_t(p)$ but would inevitably lead to the starvation of the slower units and, as a consequence, to a poor pipeline efficiency e_p.

As a result, the quality of the scheduling policy can be measured by the product of the tasks and pipeline efficiencies $e_t \cdot e_p$ which is always less than one.

This performance analysis can be used to evaluate the behavior of the code presented in Section 3.5 and explain the related results. We compute $\tilde{t}^{area}(p)$ as an upper bound on the performance of our application and the efficiency measures e_g, e_t, e_p, e_c and e_r to identify the main factors limiting the performance of the execution with respect to this upper bound. Technically, $\tilde{t}^{area}(p)$ and $t^{area}(p)$ are computed by running two instances of the code, respectively, one with coarse grain partitioning and the other with hierarchical partitioning and solving, in both cases, the Linear Program 1. Note that in order to generate the linear problem, it is necessary to run the code multiple times so that StarPU can build accurate performance profiles.

Figure 3.25 shows the efficiency analysis for our code on the test matrices. With a runtime efficiency e_r greater than 0.9 for the tested matrices we see that the cost of the runtime system is negligible compared to the workload. In addition, the runtime overhead becomes relatively smaller and smaller as the size of the problems increases. These results also show that our scheduling policy makes a good job in assigning tasks to the units where they can be executed more efficiently. The task efficiency e_t, lies between 0.8 and 1.2 for all tested matrices except for matrix #15, denotes a good balancing of the workload between the CPUs and the GPU. We observe in our experiments that the task efficiency may be greater than one as for matrix #15, #16 and #18. As explained above, this is simply due to the fact that too many tasks are affected to faster units, e.g., GPUs; this implies starvation of the slower units which translates into a weaker pipeline efficiency, as shown in Figure 3.25. The most penalizing effect on the global efficiency is the pipeline efficiency e_p. In addition, for all tested matrices except matrix #21 the pipeline efficiency decreases when the number of cores goes from twelve to twenty-four. This is mainly due to a lack of concurrency resulting from the choice of partitioning. This choice aims at achieving the best compromise between the efficiency of kernels and the amount of concurrency; it must be noted that e_p could certainly be improved by using a finer grain partitioning but this would imply a worse efficiency of the tasks and thus, as a consequence, higher values for both $t_t(p)$ and $t^{area}(p)$. In addition, smaller matrices do not deliver enough parallelism to feed all the resources which explains that the pipeline efficiency is greater on the biggest problems. Similarly to the runtime efficiency, the communication efficiency is rather good with values greater than 0.85. This shows that the scheduler is capable of efficiently overlapping task execution with communications thanks to the data prefetching capability enabled by the use of worker threads. All in all, we can observe that the parallelization efficiency is satisfactory especially on the biggest problems.

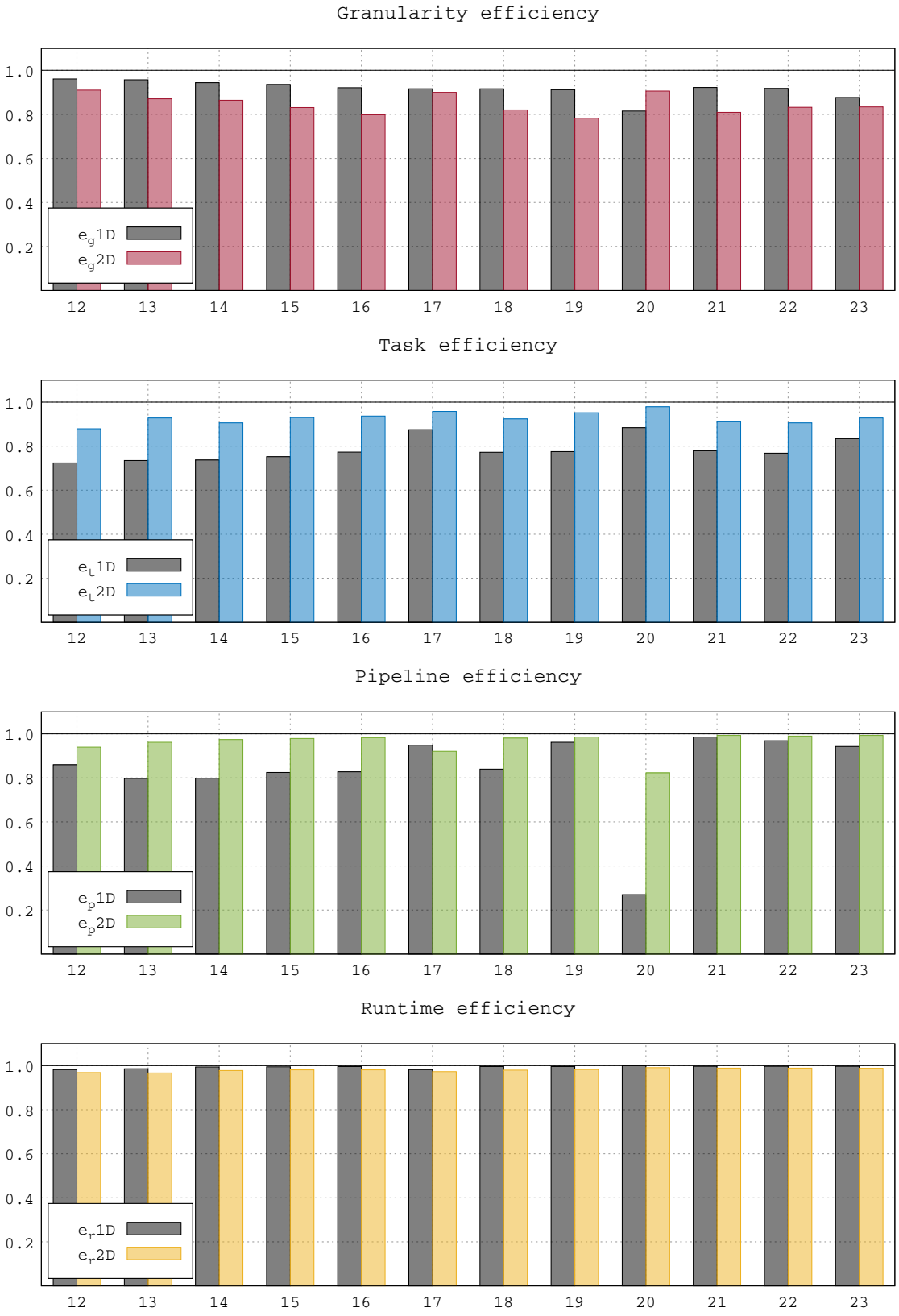


Figure 3.24: Efficiency measures for the STF 1D and 2D algorithms on `ada` (32 cores).

Figure 3.25: Efficiency measures for the STF heterogeneous algorithm on Sirocco with both configurations 12 CPUs (1×E5-2680) + 1 GPU (1×K40M) and 24 CPUs (2×E5-2680) + 1 GPU (1×K40M). Note that due to technical issues in StarPU, we are currently unable to obtain the efficiency measures for matrix # 17.

Chapter 4

Memory-aware

In this chapter we tackle the problem of memory scalability of a sparse, multifrontal solver. Frontal matrices are not statically allocated at once before the factorization begins; rather, each frontal matrix is allocated only when the corresponding node of the elimination tree is visited. When the processing of a front is finished, only part of this memory can be freed, i.e., the part containing the contribution block. Therefore, the memory used to store a frontal matrix can be logically split into two parts:

1. a persistent memory: once the frontal matrix is factorized, this part contains the factor coefficients and, therefore, once allocated it is never freed, unless in an out-of-core execution (where factors are written on disk)[1];

2. a temporary memory: this part holds the contribution block and is freed once the coefficients it contains have been assembled into the parent front.

At any time during a sequential multifrontal factorization, we refer to the memory containing the frontal matrix being processed plus all contribution blocks that are not yet assembled into their parents as *Active Memory*.

As a consequence of what said above, the memory footprint of the multifrontal factorization (QR as well as LU or other types) in a sequential execution varies greatly throughout the factorization. Starting at zero, it grows fast at the beginning as the first fronts are activated, it then goes up and down as fronts are activated and deactivated until it reaches a maximum value, which we refer to as the *sequential peak*, and eventually goes down towards the end of the factorization to the point where the only thing left in memory is the factors.

Remember that, as explained in Section 2.2, the elimination or assembly tree has to be traversed in topological order, i.e., from the bottom up. Yet, many topological orders are possible for a given tree and each of them will result in a different memory consumption profile and, ultimately in a different value for the sequential peak. Historically, postorders have been preferred: a postorder is a topological order where all the nodes in each subtree are numbered contiguously. This is because, if a topological order is followed in a sequential execution, the active memory behaves as a stack which is a very desirable property when dynamic allocation is not available and the storage of data has to be handled manually within a large, statically declared memory region. With the availability of fast and scalable dynamic memory allocators, this has become less of an issue; nonetheless, topological order are still preferred because of the better locality of access to contribution blocks which can improve the use of cache memories.

[1] In some other cases the factors can also be discarded as, for example, when the factorization is done for computing the determinant of the matrix.

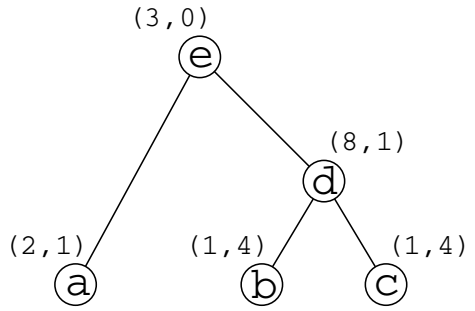
Reducing the memory requirements of the serial multifrontal method (in particular the active memory) has been extensively studied [88, 109, 116, 118]. Such problems were originally studied by Sethi et al. [147] for the evaluation of arithmetic expressions whose computation consists in the traversal of the tree representing them. The objective of the aforementioned work is to compute these arithmetic expressions using the minimal number of registers. This problem may be formulated as a *tree pebble game* which has a polynomial complexity for tree-shaped graphs [147] and is shown to be NP-hard by Sethi [146] for general DAGs if nodes cannot be pebbled more than once.

Consider the case where the memory for a frontal matrix is allocated after all its children subtrees have been processed (this is referred to as *terminal allocation scheme* [109]) and no overlap is possible between the memory of a front and of its children. In this case, the memory-minimizing postorder traversal of the multifrontal tree is given by Liu [118] which can be explained based on the relation between the tree pebble game and the memory usage of the multifrontal method. Consider a node i of the tree having nc_i children. We note m_i the memory needed to store frontal matrix i, cb_i the size of its contribution block. Liu proves that in order to minimize the total storage T or the total working storage S (i.e., the active memory), for every node i of the tree, its children have to be sorted in order to minimize the quantity $\max_{i=1,..,nc_i}(x_i + \sum_{j=1}^{i-1} y_j)$ where x_i represent the memory usage to process node i and $\sum_{j=1}^{i-1} y_j$ the memory remaining after processing the first $i-1$ nodes. Liu proves [118] that this quantity can be minimized by ordering the child nodes in descending order of $x_i - y_i$. The value associated with "x_i" and "y_i" depends on the memory management and assembly scheme as extensively presented and discussed by L'Excellent [109]. For example, if the objective is to minimize the total storage T, assuming a classical (i.e., non in-place) assembly is used, then $x_i = T_i$ and $y_j = cb_j + F_j$ where T_i is the total storage for processing the entire subtree rooted at node i and F_i is the size of all factors associated with the nodes in the subtree rooted at node i. If, instead, the objective is to minimize the active memory S, then $x_i = S_i$ and $y_j = cb_j$ were S_i is the active storage needed to process the entire subtree rooted at node i.

The optimal memory ordering is obtained using a depth-first traversal on the tree using the rearranged child sequences, assuming that for a leaf node i $T_i = S_i = m_i$. The minimal memory usage for the sequential traversal of the elimination tree is given by the memory peak computed at the root node. Figure 4.1 shows an example of how the memory usage varies during a multifrontal factorization. The table on the right side of the figure shows the memory consumption for a sequential execution where the tree is traversed in natural order which is a-b-c-d-e; the corresponding sequential peak is equal to 22 memory units. If we apply the above algorithm to the case of this figure then we find that the optimal memory traversal is b-c-d-a-e and the corresponding sequential memory peak is equals to 19 memory units.

Liu [116] observed that the postorder may not give the best memory usage among all topological orders and proposes an algorithm to find the optimal memory topological order for a given tree. Motivated by tree structures emerging from the multifrontal factorization, Jacquelin et al. [103] give an alternative algorithm for computing this memory optimal topological order. Their experimental findings are that, on trees associated with large sparse matrices coming from real applications, the optimal traversal is a postorder 95% of the time. In the worst case, the memory overhead induced by using the best postorder instead of the best order is 18%. This confirms that using a postorder traversal (in the serial case) is a reasonable choice, especially since it allows for an efficient stack mechanism.

Other allocation schemes exist and they have different properties with respect to the memory usage and, in the out-of-core case, disk traffic [8, 88, 109].



Task	Memory
activate(a)	3
activate(b)	8
activate(c)	13
activate(d)	**22**
deactivate(b)	18
deactivate(c)	14
activate(e)	17
deactivate(a)	16
deactivate(d)	15

Figure 4.1: The memory consumption (*right*) for a 5-nodes elimination tree (*left*) assuming a sequential traversal in natural order. Next to each node of the tree the two values corresponding to the factors (permanent) and the contribution block (temporary) sizes in memory units.

The use of parallelism increases the peak of active memory. Specifically, this is due to tree parallelism because it implies traversing multiple branches of the elimination tree concurrently which results in more contribution blocks and fronts being stored in the global active memory.

The memory-minimization problem, extensively studied in sequential, has received little attention in the parallel case. In recent work, Eyraud-Dubois et al. [72] show that the parallel variant of the *pebble game* is NP-complete and prove that there is no approximation algorithm that can be designed to tackle the problem. They propose, instead, several heuristics for scheduling task trees which aim at reducing the memory consumption of a multifrontal factorization[2] in a shared memory, parallel setting. One of these heuristics, MemBookingInnerFirst is such that the parallel processing of the elimination tree can be achieved while respecting a prescribed memory bound. This is achieved through a memory reservation system; roughly speaking, when a tree node is activated not only is the memory needed for its processing allocated but the memory needed to process its parent is reserved. This reservation ensures that, when the parent node becomes ready, enough memory is available to process it; this prevents memory deadlocks (see the next section for an explanation of this issue). The authors validated experimentally this heuristic, as well as the others, by simulating a rather simplistic shared memory parallelization scheme.

It must be noted that this problem is quite common in parallel computing whenever temporary data is generated as a result or the parallel processing. Sid-Lakhdar [148], for example, deals with memory deadlocks that may occur because of the temporary buffers used for exchanging messages in distributed memory parallel codes.

We will describe methods for controlling the memory consumption in the multifrontal factorization first in a shared-memory, parallel setting and then in a distributed-memory one in Sections 4.1 and 4.2, respectively.

4.1 Memory-aware scheduling in shared-memory systems

Figure 4.2 shows, in the blue curve, the typical memory consumption profile of a sequential multifrontal factorization; the data in this figure results from the multifrontal QR

[2]Note that the problem is presented in a much more general form in their paper and not necessarily related to the multifrontal factorization.

factorization of matrix #12 from Table A.1. The memory footprint varies considerably throughout the factorization and presents spikes immediately followed by sharp decreases; each spike corresponds to the activation of a front whereas the following decrease corresponds to the deactivation of its children once its assembly is completed.

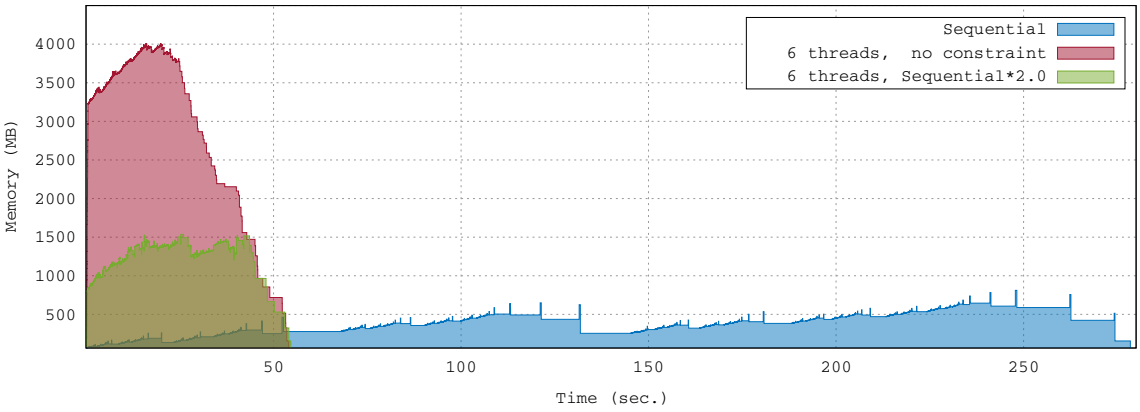


Figure 4.2: The memory profiles for matrix #12 from Table A.1 for a sequential (in blue) and 6-threaded parallel QR factorization without memory constraint (in red) and with a constraint equal to 2.0 times the sequential peak (in green).

As explained above, the memory consumption of the STF parallel code discussed so far can be considerably higher than the sequential peak (up to 3 times or more). This is due to the fact that the runtime system tries to execute tasks as soon as they are available and to the fact that activation tasks are extremely fast and only depend upon each other; as a result, all the fronts in the elimination tree are almost instantly allocated at the beginning of the factorization. This behavior, moreover, is totally unpredictable because of the very dynamic execution model of the runtime system. This is depicted in Figure 4.2 with the red curve which shows the memory consumption for the case where our runtime based STF factorization is run with six threads; in this case the memory consumption is roughly five times higher than the sequential peak. Figure 4.3 shows the memory consumption of the STF parallel code relative to the sequential code on a number of matrices when executed on the `ada` computer using 32 threads. As shown, in several cases the memory increase can be considerable, especially in the case where the factors are discarded due to the fact that, in this case, the relative weight of the temporary memory is smaller.

This section proposes a method for limiting the memory consumption of parallel executions of our STF code by forcing it to respect a prescribed memory constraint which has to be equal to or bigger than the sequential peak. This technique shares commonalities with the MemBookingInnerFirst heuristic proposed by Eyraud-Dubois et al. [72]. On the other hand, whereas Eyraud-Dubois et al. [72] only consider the theoretical problem, the present study proposes a new and robust algorithm to ensure that the imposed memory constraint is guaranteed while allowing a maximum amount of concurrency on shared-memory multicore architectures. In the remainder of this section we assume that the tree traversal order is fixed and we do not tackle the problem of finding a different traversal that minimizes the memory footprint.

We rely on the STF model to achieve this objective with a relatively simple algorithm. In essence, the proposed technique amounts to subordinating the submission of tasks to the availability of memory. This is done by suspending the execution of the outer loop in Figure 3.2 if not enough memory is available to activate a new front until the required

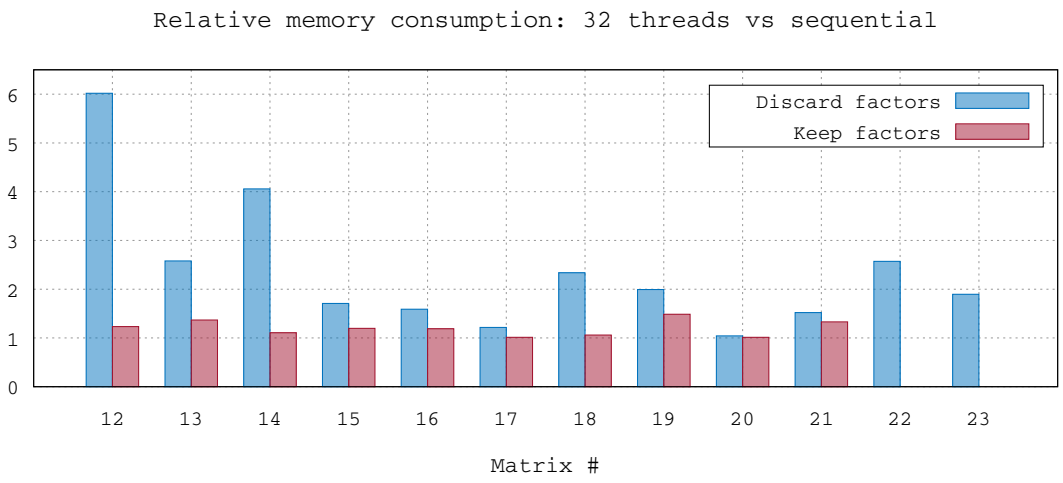


Figure 4.3: Memory consumption of the parallel STF multifrontal QR factorizations with 32 threads relative to the sequential memory peak on the `ada` computer for matrices in Table A.1 either discarding or keeping the factors in memory. Matrices #22 and #23 could only be factorized discarding the factors due to limited memory availability on the machine.

memory amount is freed by already submitted `deactivate` tasks. Special attention has to be devoted to avoiding memory deadlocks, though. A memory deadlock may happen because the execution of a front deactivation task depends (indirectly, through the assembly tasks) on the activation of its parent front; therefore the execution may end up in a situation where no more fronts can be activated due to the unavailability of memory and no more deactivation tasks can be executed because they depend on activation tasks that cannot be submitted. An example of memory deadlock may be shown using Figure 4.1. Remember that the memory-minimizing traversal for the tree in this figure is b-c-d-a-e, which leads to a memory consumption of 19 units. Assume a parallel execution with a memory constraint equal to the sequential peak. If no particular care is taken, nothing prevents the runtime system from activating nodes a, b and c at once thus consuming 13 memory units; this would result in a deadlock because no other front can be activated nor deactivated without violating the constraint.

This problem can be addressed by ensuring that the fronts are allocated in exactly the same order as in a sequential execution: this condition guarantees that, if the tasks submission is suspended due to low memory, it will be possible to execute the deactivation tasks to free the memory required to resume the execution. Note that this only imposes an order in the allocation operations and that all the submitted tasks related to activated fronts can still be executed in any order provided that their mutual dependencies are satisfied. This strategy is related to the Banker's Algorithm proposed by Dijkstra in the early 60's [59, 60].

In our implementation this was achieved as shown in Figure 4.4. Before performing a front activation (line 4), the master thread, in charge of the submission of tasks, checks if enough memory is available to perform the corresponding allocations (line 2); if this is the case, the allocation of the frontal matrix (and the other associated data) is performed within the `activate` routine. This activation is a very lightweight operation which consists in simple memory bookkeeping (due to the first-touch rule) and therefore does not

```
forall fronts f in topological order
   do while (size(f) > avail_mem) wait
   ! allocate and initialize front: avail_mem -= size(f)
   call activate(f)

   ! initialize the front structure
   call submit(init, f:RW, children(f):R)

   ! front assembly
   forall children c of f
      ...
      ! Deactivate child: avail_mem += size(cb(f))
      call submit(deactivate, c:RW)
   end do

   ! front factorization
   ...

end do
call wait_tasks_completion()
```

Figure 4.4: Pseudo-code showing the implementation of the memory-aware task submission.

substantially slow down the task submission. The front initialization is done in the `init` task (line 7) submitted to the runtime system which can potentially execute it on any worker thread, as described in Section 3.3. If the memory is not available, the master thread suspends the submission of tasks until enough memory is freed to continue. In order not to waste resources, the master thread is actually put to sleep rather than left sitting on active wait. This was manually implemented through the use of POSIX thread locks and condition variables: the master thread goes to sleep waiting for a condition which is signaled by any worker thread that frees memory by executing a `deactivate` task. When woken up, the master checks again for the availability of memory. This work has prompted the StarPU developers to extend the runtime API with routines (the `starpu_memory_allocate/deallocate()` and `starpu_memory_wait_available()`) that implement this mechanism and easily allow for implementing memory-aware algorithms.

The use of this technique on the case of Figure 4.2, leads to memory profile depicted by the green curve reported in the figure: the imposed memory constraint equal to twice the sequential peak is never exceeded whereas the execution time is barely affected. A more detailed analysis is provided in the experimental section below.

4.1.1 Experimental results

This section describes and analyses experiments that aim at assessing the effectiveness of the memory aware scheduling presented above. Here we present only results related to an Out-Of-Core (OOC) QR factorization of matrices from Table A.1 ; in this scenario the factors are written to disk as they are computed in order to save memory. In this case the memory consumption is more irregular and more considerably increased by parallelism. We simulate this scenario by discarding the factors as we did in Sections 3.3 and 3.4; note that by doing so we are assuming that the overhead of writing data to disk has a negligible effect on the experimental analysis reported here. We refer the reader to our

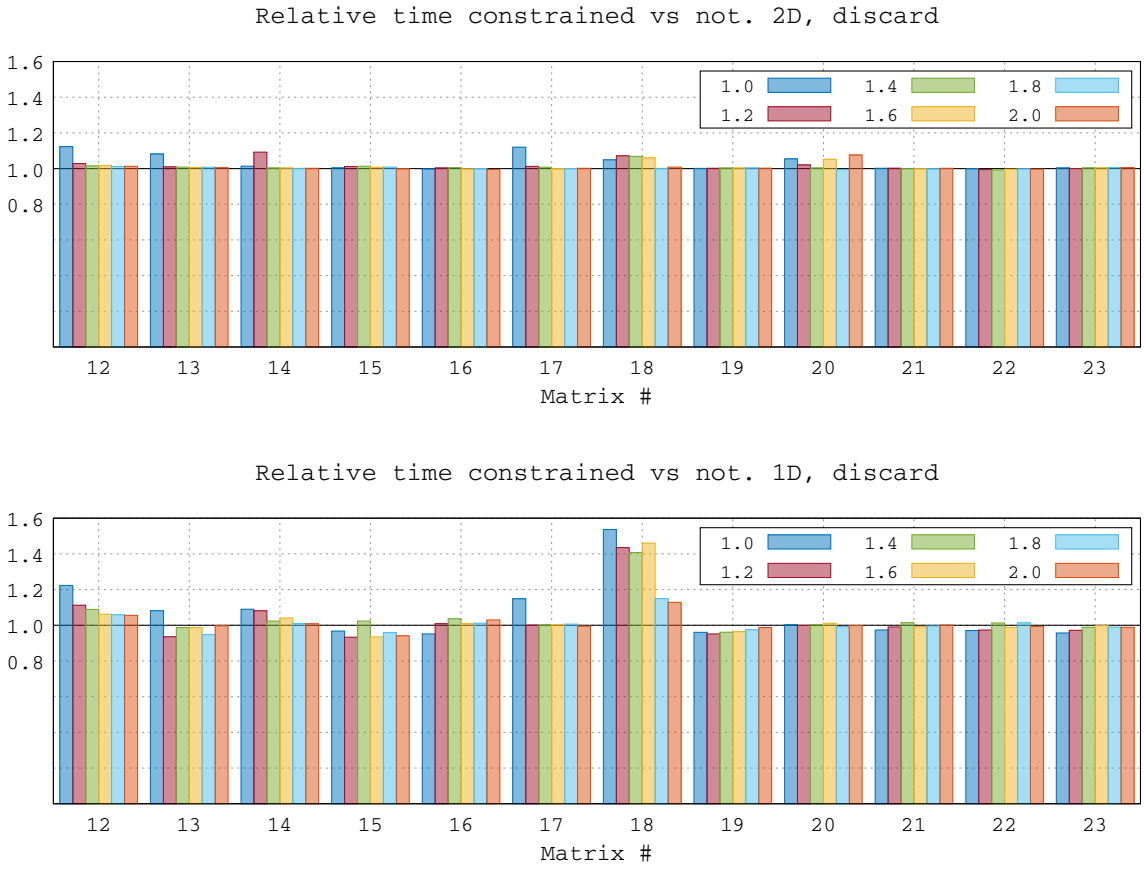


Figure 4.5: Memory-constrained factorization times relative to the unconstrained execution for the 2D and 1D methods in an OOC execution.

original paper on this subject [J2] or to Florent Lopez's PhD thesis [120] for results related to the In-Core case.

These experiments measure the performance of both the 1D and 2D factorization (with the parameter values in Table 3.5 and Table 3.6) within an imposed memory footprint. Experiments were performed using 32 cores with memory constraints equal to $\{1.0, 1.2, 1.4, 1.6, 1.8, 2.0\} \times S_{seq}$ where with S_{seq} we denote the peak of active memory from a sequential execution. The parameter settings used for these experiments are those reported in Tables 3.5 and 3.6.

For almost all 1D and 2D tests, a performance as high as the non constrained case (presented in Sections 3.3 and 3.4) could be achieved with a memory exactly equal to the sequential peak, which is the lower bound that a parallel execution can achieve. This shows the extreme efficiency of the memory-aware mechanism for achieving high-performance within a limited memory footprint. Combined with the 2D numerical scheme, which delivers abundant node parallelism, the memory-aware algorithm is thus extremely robust since it could process all considered matrices at maximum speed with the minimum possible memory consumption. In a few cases a slight increase (always lower than 20%) in the factorization time can be observed (especially when the constraint is set equal to the sequential peak). In only three cases it is possible to observe a smooth decrease of the factorization time as the constraint on the memory consumption is relaxed: these are the 1D factorization of matrices #12, #14 and #18.

To explain this extreme efficiency, we performed the following analysis. As explained in the previous section, prior to activating a front, the master thread checks whether enough memory is available to achieve this operation. If it is not the case, the master thread is put to sleep and later woken up as soon as one `deactivate` task is executed; at this time the master thread checks again for the availability of memory. The master thread stays in this loop until enough deactivation tasks have been executed to free up the memory needed to proceed with the next front activation. Every time the master thread was suspended or resumed we recorded the time stamp and the number of ready tasks (i.e., those whose dependencies were all satisfied).

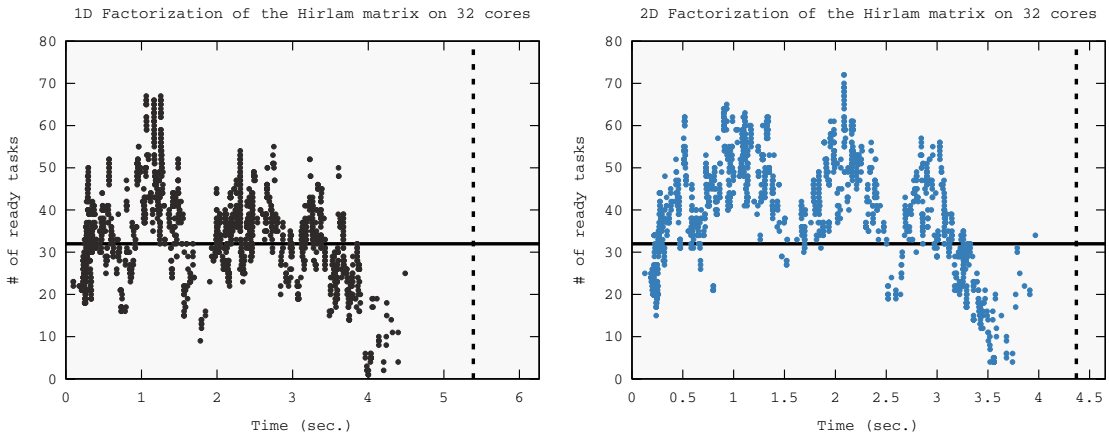


Figure 4.6: Concurrency under a memory constraint for the Hirlam matrix on `ada` (32 cores).

Figure 4.6 shows the collected data for matrix #12 with an imposed memory consumption equal to the sequential peak, in the OOC case using both the 1D (*left*) and 2D (*right*) methods. In this figure, each (x, y) point means that at time x the master thread was suspended or resumed and that, at that time, y tasks where ready for execution or being executed. The width of each graph shows the execution time of the memory constrained factorization whereas the vertical dashed line shows the execution time when no limit on the memory consumption is imposed. The figure leads to the following observations:

- in both the 1D and 2D factorizations, the number of ready tasks falls, at some point, below the number of available cores (the horizontal, solid line); this lack of tasks is responsible for a longer execution time with respect to the unconstrained case.
- in the 1D factorization this lack of tasks is more evident; this can be explained by the fact that the 1D method delivers much lower concurrency than the 2D one and therefore, suspending the submission of tasks may lead more quickly to thread starvation. As a result, the difference in the execution times of the constrained and unconstrained executions is more evident in the 1D factorization.

For all other tests, either the number of tasks is always (much) higher than the number of workers or the tasks submission is never (or almost never) interrupted due to the lack of memory; as a result, no relevant performance degradation was observed with respect to the case where no memory constraint is imposed. This behavior mainly results from two properties of the multifrontal QR factorization:

1. the size of the contribution blocks is normally very small compared to the size of factors, especially in the case where frontal matrices are overdetermined;

2. the size of a front is always greater than or equal to the sum of the sizes of all the contribution blocks associated with its children (because in the assembly operation, contribution blocks are not summed to each other but stacked).

As a result, in the sequential multifrontal QR factorization, the memory consumption grows almost monotonically and in most cases the sequential peak is achieved on the root node or very close to it. For this reason, when the tasks submission is interrupted in a memory-constrained execution, a large portion of the elimination tree has already been submitted and the number of available tasks is considerably larger than the number of working threads. Other types of multifrontal factorizations (LU, for instance) are likely to be more sensitive to the memory constraint because they do not possess the two properties described above. By the same token, it is reasonable to expect that imposing a memory constraint could more adversely affect performance when larger numbers of threads are used.

4.2 Memory-aware scheduling and mapping in distributed-memory systems

In this section we turn our attention to the problem of memory consumption in a distributed-memory parallel context which is much harder to address with respect to the case of shared-memory parallelism discussed in the previous section. Obviously, in a distributed-memory context, minimizing the total memory consumption is desirable but it is also crucial to maintain a balanced memory usage between the processes, to prevent a process from running out of memory (assuming that the same amount of main memory is available on all the processes, which is the most frequent setting).

Minimizing the overall memory consumption in a parallel setting is complicated unless a schedule with very specific properties is used. In our distributed memory context, the maximum peak of active memory (over the set of processes) is the target for our optimization problem. Note that the sum of the peaks of the different processes provides an upper bound on the total consumption (which might be loose).

For a parallel execution on p processes, we denote by $S_{max}(p)$ and $S_{avg}(p)$ the maximum and average peaks of active memory among the p processes, respectively. $S_{avg}(p)$ is computed as the sum of the p peaks divided by p; with the above observation $p \cdot S_{avg}(p)$ is an upper bound on the total consumption.

The performance of a parallel algorithm executed on p processes is often assessed using a notion of *efficiency*; given the above observations, we consider two kinds of *memory efficiency* metrics that depend on p:

- $e_{avg}(p) = \dfrac{S_{seq}}{p \cdot S_{avg}(p)}$; this metric compares the total memory usage (using an upper bound, as described above) to that of a sequential execution and is relevant in a shared-memory context as well as in a distributed-memory one.

- $e_{max}(p) = \dfrac{S_{seq}}{p \cdot S_{max}(p)}$; this metric detects that one (or more) process(es) use(s) too much memory, which is only relevant in a distributed-memory context.

Assuming p_i processes are mapped on the subtree rooted at node i of the tree, a lower bound on the average storage needed by these processes to process the subtree is defined as $\bar{S}_i = S_i/p_i$. This quantity corresponds to the memory consumption per process in the ideal case where the peak S_i is uniformly distributed; as we explain below, this case is very unlikely in practice, unless very specific mapping schemes are used within the subtree.

Consider the following parallel scheme. The tree is processed following the postorder used in the sequential case (*tree serialization*) and each node of the tree is mapped on all processes; this is what we will refer to as *tree serialization mapping*. If we assume a perfect memory balance within each node and within each contribution block, this tree serialization mapping technique clearly provides a perfect memory scalability ($S_{avg} = S_{max} = \bar{S}_i$ and $e_{avg}(p) = e_{max}(p) = 1$). However, it does not take advantage of tree parallelism because all the branches are serialized. It also leads, by forcing a large number of processes to be used at each node, to an unnecessary increase in communication (both within nodes when performing dense partial factorizations, and between nodes when communicating contribution blocks). Finally, it does not deliver an adequate granularity of operations within nodes. These drawbacks are likely to induce a significant performance penalty, as we assess in Section 4.2.3. This is why no solver that we are aware of relies on this strategy. In practice, minimizing the active memory is not the best formulation of the problem we want to solve. Instead of minimizing it, we would rather *control* the active memory by enforcing a given memory constraint that would be provided by the user or defined by hardware specifications, as in the approach described in the previous section. Then, for this memory constraint, we would like to maximize tree parallelism and parallelism granularity, in order to avoid as much as possible the aforementioned drawbacks related to communication volume and small task granularity; this is exactly the aim of the mapping technique we describe in Section 4.2.2.

4.2.1 Mapping techniques

Different mapping strategies are commonly used in distributed memory, parallel multifrontal codes. The subtree to subcube mapping by Liu, George and Ng [115] and the proportional mapping by Pothen and Sun [136] are popular strategies and are the basis for more sophisticated schemes. These are special cases of the wider family of *tree partitioning* methods where the set of processes mapped on a node of the tree is partitioned into disjoint subsets and each subset is affected to a child subtree. Tree partitioning mappings are well appreciated because they help reducing the volume of data transfers thanks to a good data locality and because they allow for a good use of both tree and node parallelism. In the proportional mapping method, this recursive splitting of processes sets is guided by a balancing criterion. This mapping technique consists in a top-down traversal of the tree where every node is assigned a set of processes. All the processes are assigned to work on the root node. This is a natural choice since the root node is the last task to be performed in the factorization. Then, for every node in the tree, the set of processes working at that node is split among its children, proportionally to the weights (determined according to a given metric) of the subtrees rooted at these children. Denoting by w_i the weight of the subtree rooted at node i, and by $par(i)$ the parent of node i, the number of processes p_i given to node i is then

$$p_i = \frac{w_i}{\sum_{j;\, par(j)=par(i)} w_j} \cdot p_{par(i)}. \tag{4.1}$$

This procedure is applied in a recursive fashion to the whole tree, starting from the root r; the recursion stops when leaf nodes are reached or entire subtrees are mapped onto single

processes, which happens because the number of nodes in the tree is commonly much larger than the number of processes. Not considering the case where fractions of processes are allowed to be mapped on different subtrees (meaning that such a process would work less than the others on a given subtree, and be assigned less memory), rounding is performed in (4.1) in order to ensure that:

- p_i is an integer for all nodes i; and
- the tree partitioning property holds, i.e., $p_{par(i)} = \sum_{j;\, par(j)=par(i)} p_j$.

The metric used at each step of the mapping in the original method proposed by Pothen and Sun is the workload of each subtree; we refer to this case as the *workload-based proportional mapping*. Clearly, this criterion can be replaced by another one depending on the objectives. If one aims to achieve a memory balance rather than a load balance, a possible heuristic consists in using a *memory-based* variant; we report on experimental results using this strategy in the experimental section. Prasanna and Musicus proposed a scheduling strategy for tree-shaped task graphs when the time for computing a parallel task (a *malleable task*) using p processes is exactly $\frac{L}{p^\alpha}$ (with $0 < \alpha \leqslant 1$) where L is the length of the task [137]. Beaumont and Guermouche evaluated this approach in the multifrontal method [30].

An interesting property of the tree partitioning mapping is that the traversal of every process, i.e., the set of tasks that a process executes and the order in which they are processed, is deterministic. Indeed, every process is in charge of a sequential subtree and takes part in the computation of the parallel nodes in the path between that subtree and the root of the elimination tree; this defines a single possible traversal.

In the remainder, we will thus use the proportional mapping as a representative of the whole family of tree partitioning mappings; it must be noted, though, that the novel techniques proposed in Section 4.2.2 are perfectly compatible with any tree partitioning technique.

Proportional mapping (tree partitioning mappings in general) is not a memory-friendly strategy. Actually, it is possible to demonstrate that the memory efficiency resulting from this mapping technique tends to zero as the number of processors goes to infinity as established by Theorem 4.1.

Theorem 4.1 Sub-optimality of the proportional mapping ([140, Theorem 8.1]).— *Let T be an elimination tree corresponding to a nested dissection of a regular 2D square grid with a nine-point stencil. For a given number of processes p, the memory efficiency of a strict memory-based proportional mapping of T verifies:*

$$e_{max}(p) = \frac{S_{seq}}{p \cdot S_{max}(p)} \leqslant \frac{6.25}{29} p^{-0.2} \simeq 0.22\, p^{-0.2}$$

(for p sufficiently large).

The proof of Theorem 4.1 is very long and tedious and thus we do not report it for the sake of conciseness; we refer the interested reader to the PhD thesis of Rouet [140]. Nonetheless, in the next section we show through an example how the proportional mapping may result in very poor memory scalability.

4.2.1.1 Memory scalability of proportional mapping: an example

We illustrate the behavior of the proportional mapping on a simple yet realistic elimination tree. We consider a memory-based proportional mapping strategy, but we could make the same observations about a workload-based strategy. We consider the tree in Figure 4.7(a), which is to be mapped on 64 processes. First, these 64 processes are assigned to the root node l. Then, a first step of memory-based proportional mapping is used to distribute these 64 processes among the four children of l: a, e, f, and k, as illustrated in Figure 4.7(b). The sequential peaks of active memory of the subtrees rooted at a, e, f, and k are 8 GB, 5 GB, 3 GB and 3 GB, respectively. Therefore, a gets $\frac{8}{8+5+3+3} \cdot 64 \approx 27$ processes; e gets $\frac{5}{8+5+3+3} \cdot 64 \approx 17$ processes and f and k get $\frac{3}{8+5+3+3} \cdot 64 \approx 10$ processes. At this stage, we can compute a lower bound of the peak of active memory of every process. Consider the 27 processes working on the subtree rooted at a. The sequential peak of active memory of this subtree is 8 GB. Therefore, at best, i.e., assuming a perfect memory scalability can be attained for this subtree, the maximum peak of active memory among these 27 processes will be $\bar{S}_a = \frac{8 \text{ GB}}{27} = 0.296$ MB. Similarly, the maximum peaks of active memory for the processes working on the subtree rooted at e (f and k respectively) are bounded from below by $\bar{S}_e = \frac{5\text{GB}}{17}$ ($\bar{S}_f = \frac{3\text{GB}}{10}$ and $\bar{S}_k = \frac{3\text{GB}}{10}$, respectively); ignoring the rounding applied to obtain integer numbers of processes, all these peaks are the same ($\bar{S}_a \approx \bar{S}_e \approx \bar{S}_f \approx \bar{S}_k \approx 0.3$ GB) since we have applied a memory-based proportional mapping. We can, thus, derive a first lower bound on the memory efficiency for this problem; since the sequential peak of active memory for the whole tree is 8 GB, the memory efficiency is bounded as follows:

$$e_{max}(p) = \frac{S_{seq}}{p \cdot S_{max}(p)} \leqslant \frac{8 \text{ GB}}{64 \cdot 0.3} \leqslant 0.42$$

It is fairly easy to see why the efficiency is low. The sequential peaks of the whole tree and the subtree rooted at a are the same; however, only a small subset of the processes work on the latter subtree. Therefore, even if a perfect memory scalability is attained on the subtree rooted at a, the memory usage for the 27 processes working on that subtree is more than twice what we are targeting ($\frac{8 \text{ GB}}{27}$ instead of $\frac{8 \text{ GB}}{64}$).

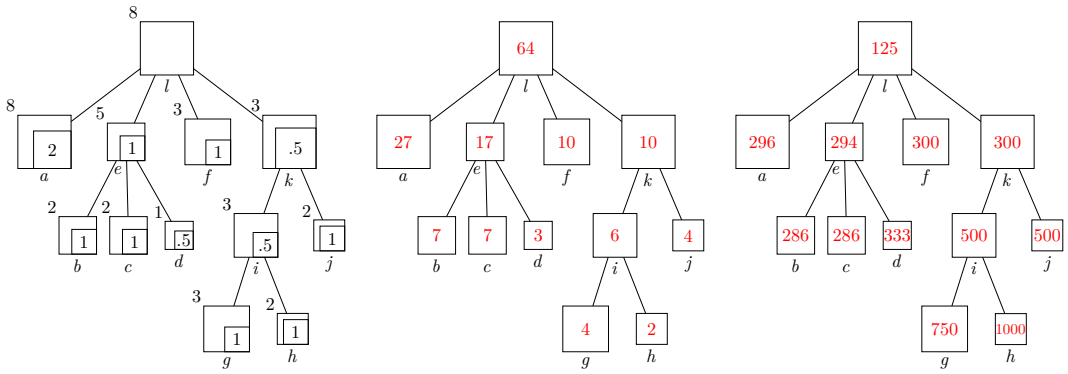
We can refine this upper bound on memory efficiency by looking at the lower levels of the tree. By considering the grandchildren of the root node, we see in Figure 4.7(c) that the memory usage $S_{max}(p)$ is at least $\max\left(\bar{S}_b, \bar{S}_c, \bar{S}_d, \bar{S}_i, \bar{S}_j\right) = 0.5$ GB, yielding

$$e_{max}(p) \leqslant \frac{S_{seq}}{p \cdot 0.5} = 0.25$$

We assumed that a perfect memory scalability was reached in the different subtrees rooted at the grandchildren of the root node, which means that this bound on memory efficiency is likely to be optimistic. Finally, by considering the lowermost level (see node h in Figure 4.7(c)), we have $e(p) \leqslant 0.125$.

4.2.2 Memory-aware mapping algorithms

We demonstrated that a memory-based proportional mapping leads to a low scalability of the active memory. However this mapping is interesting in terms of performance since it maximizes tree parallelism and reduces the volume of communication within parallel nodes and between nodes of the tree. We also introduced a “tree serialization mapping”



(a) Elimination tree. (b) Proportional mapping. (c) Best-case average peak.

Figure 4.7: An example of memory-based proportional mapping on $p = 64$ processes. In (a), we indicate next to each node the sequential peak of active memory S_i (in GB) of the corresponding subtree, and within each node, the size of its contribution block cb_i (in GB). In (b), we indicate within each node the number of processes p_i assigned by the proportional mapping. In (c), we indicate within each node the lower bound $\bar{S}_i = \frac{S_i}{p_i}$ on the average storage required per process (in MB).

which consists in a constrained traversal of the tree where all the processes work at every node, following a postorder. However, this solution generates prohibitive amounts of communications, it does not exploit tree parallelism and yields small granularity of computations on nodes at the bottom of the tree. Therefore it is not time efficient; we assess this in the experimental section. Here, we introduce a "memory-aware mapping" that hybridizes these two techniques and tries to enforce a given memory constraint (the maximum amount of active memory that a process can use). The idea was first presented in Agullo's PhD thesis [2] and is described in Section 4.2.2. It consists in a tree serialization mapping in memory demanding parts of the tree, and a proportional mapping whenever we can be sure that it will not violate the memory constraint.

We assume that we are given a memory constraint M_0 that represents the maximum amount of active memory that a process is allowed to use. We will also use the notation M_i, $i > 0$, to denote the memory constraint for a subtree $\mathcal{T}_i$ rooted at i ($M_0 = M_r$ in case r is the root of the entire tree). The memory-aware mapping works as follows. We assume that the tree has been reordered to reduce the sequential peak of active memory and that the sequential peaks S_i have been computed for every subtree $\mathcal{T}_i$. Then, a top-down traversal of the tree is performed to compute the mapping. All the processes are first assigned to the root node r. Then, recursively, once a subtree $\mathcal{T}_r$ rooted at r is mapped on p_r processes, its children are mapped as follows. We first check whether a proportional mapping of the children is feasible by simulating a proportional mapping and verifying that the memory constraint is respected at every child i. Denote p_i the number of processes that a proportional mapping would assign to child i (Equation 4.1). For every child i, we check the condition $\bar{S}_i = \frac{S_i}{p_i} \leqslant M_r$:

- If all child subtrees $\mathcal{T}_i$ respect this condition, then the step of proportional mapping is accepted; the child subtrees will be processed in parallel on the number of processes provided by the step of proportional mapping. For the subsequent steps of the mapping procedure, the memory constraint is unchanged: $M_i = M_r$.

- If at least one of the subtrees does not respect the condition, then the step of pro-

portional mapping is rejected. All the child subtrees $\mathcal{T}_i$ inherit the processes of their parent ($p_i = p_r$) and will be processed one after another during the factorization, following the same order as the one of the sequential execution. In this case, when a child subtree $\mathcal{T}_i$ is processed, the contribution blocks of the previous siblings j ($par(j) = par(i) = r, j < i$, assuming the order of the siblings is in agreement with the postorder) are stacked and equally distributed in the memory of the $p_j = p_i = p_r$ processes. Therefore, for the next steps of the mapping procedure, the memory constraint is modified in order to take into account these contributions blocks: $M_i = M_r - \sum_{par(j)=r, j<i} \frac{cb_j}{p_j}$ (where $p_j = p_i = p_r$).

At each step of the traversal, the condition $\bar{S}_i \leqslant M_r$ means "is it possible to process the subtree $\mathcal{T}_i$ on p_i processes, using a memory at most equal to $M_i = M_r$ on each process?". Thus, when a step of proportional mapping is accepted, we ensure that every subtree will respect the memory constraint. In the end, this algorithm yields a hybrid mapping in-between a proportional mapping and a tree serialization mapping.

We illustrate a few steps of memory-aware mapping in Figure 4.8, using the tree of Figure 4.7(a) again. The tree is to be mapped on $p = 64$ processes and we choose a rather tight memory constraint $M_0 = 160$ MB (i.e., we target $e_{max} = \frac{8 \text{ GB}}{64 \cdot 160 \text{ MB}} = 0.8$). First, the 64 processes are assigned to the root node l; then the four children a, e, f and k of l are mapped. The first step consists in computing a proportional mapping of the four child nodes. a is given 27 processes, e is given 17 processes and f and k are given 10 processes each. Then, the memory constraint is checked for the subtrees. At node a, the sequential peak of active memory is 8 GB; thus $\bar{S}_a = \frac{8 \text{ GB}}{27} = 296$ MB is greater than M_0. Therefore, the subtree rooted at a cannot be processed using 27 processes without violating the memory constraint. Thus the step of proportional mapping is rejected; the four child subtrees are mapped on the 64 processes and are serialized ($\mathcal{T}_a$ will be processed first, then $\mathcal{T}_e$, and so on). Then the four subtrees are mapped using the same procedure taking into account the fact that the memory constraint has to updated because of the stacked contriution blocks. For example consider the mapping of the subtree rooted at e. Since we have serialized the four child subtrees of l, we have to take into account that, when processing $\mathcal{T}_e$, the contribution block of a is stacked in memory and equally distributed among the processes. For a given process, the available memory is no longer $M_l = M_0$ but $M_e = M_l - \frac{2 \text{ GB}}{64} = 129$ MB.

It must be noted that, whenever a proportional mapping step is rejected, even if the memory constraint is violated on a single subtree, a whole set of siblings is serialized. This may lead to an excessive loss of tree parallelism. Rather, the set of siblings can be split into groups; groups are treated sequentially one after the other but within each group proportional mapping is still possible. This clearly allows for a better compromise between memory consumption and concurrency. Forming groups, though, is not trivial and can actually be modeled as a bin-packing problem; heuristics can be used instead. We call this technique *Aggregated Memory Aware Mapping* and, for further details, we refer the reader to our original work [J1] describing this method as well heuristics we implemented for forming groups; experimental results extracted from this article will be presented below.

4.2.3 Experiments

We implemented the proposed algorithms within the MUMPS (MUltifrontal Massively Parallel Solver) software package [13, 16].

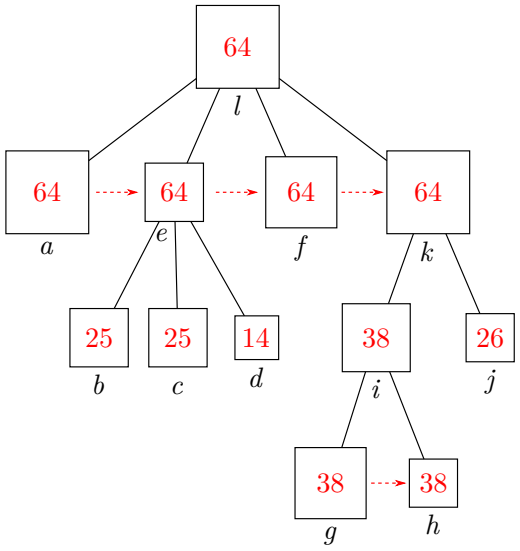


Figure 4.8: Simple example of memory-aware mapping. The tree shown in Figure 4.7(a) is mapped using the flat memory-aware mapping with $M_0 = 160$ MB. The scheduling constraints resulting from tree serialization are shown with arrows. The subtree rooted at a is processed first and is followed by the subtrees rooted at e and f, and the subtree rooted at k is processed last. Similarly, the subtree rooted at i is mapped using a local tree serialization mapping.

In this section, we assess different mapping strategies on the set of matrices described in Table 4.1. The largest problem (Geoazur_192) is processed using 256 MPI processes while the other matrices are processed using 64 MPI processes. The Geoazur_192 matrix [131] is unsymmetric and complex and corresponds to a 27-point stencil discretization of a 3D visco-acoustic wave propagation model on a grid of size $192 \times 192 \times 192$. All the matrices are ordered using MeTiS. The experiments were carried out on the `eos` system (see Appendix A.2). For the Geoazur_192 problem, we used 16 nodes of the system, and we used 4 nodes for the other matrices (i.e., we used 16 MPI processes per node). In Table 4.1, we present the characteristics of the matrices and also statistics for the amount of active memory used by the MUMPS solver, version 5.0.0, for the factorization. The mapping strategy used in MUMPS is described by Amestoy et al. [16]: at the top of the tree, a relaxed proportional mapping is used whereas on lower layers of the tree, a strategy that aims at balancing memory requirements and workloads is used. The results reported in the table show that the memory efficiency of MUMPS, although slightly better than with a strict proportional mapping, is quite low: e_{max} lies between 0.07 and 0.32 (the average is 0.19).

First, we assess the behavior of the following strategies for matrix Geoazur_192 in Table 4.2:

- A plain, memory-based, proportional mapping;
- A flat memory-aware mapping, with different constraints M_0;
- An aggregated memory-aware mapping, with different constraints M_0;
- A tree serialization mapping on the whole tree, where all processes work at every node (except in case of frontal matrices with less rows than processes).

Matrix name	Order N	Entries ($\times 10^6$)	Factors (GB)	S_{seq} (GB)	S_{max} (MB)	e_{max}	S_{avg} (MB)	e_{avg}	Time (s)	Description; origin
cage13	445,315	7.5	30.7	21.4	1050	0.32	709	0.47	385.1	Directed weighted graph; Utrecht Univ.
pancake2_3	1,004,060	49.1	39.8	10.5	832	0.21	481	0.36	278.0	3D electromagnetism; Padova Univ.
as-Skitter	1,696,415	23.9	17.7	3.8	566	0.11	197	0.30	171.1	Internet topology graph; SNAP
HV15R	2,017,169	283.1	366.4	88.6	10638	0.13	4883	0.28	N/A	CFD, 3D engine fan; FLUOREM
MORANSYS1	2,734,008	81.3	63.5	17.9	998	0.28	715	0.39	390.3	Model Order Reduction; CADFEM
meca_raff6	3,269,763	130.2	63.5	6.6	1393	0.07	859	0.12	335.3	Thermo-mechanical coupling; EDF
Geoazur_192	7,077,888	189.1	251.9	65.9	1151	0.22	827	0.31	1308.2	3D Geophysics; Seiscope consortium

Table 4.1: Set of matrices used for the experiments; active memory (S_{max} and S_{avg}) and run time for the factorization using MUMPS (with $p = 64$ except for Geoazur_192 for which $p = 256$). For matrices as-Skitter and meca_raff6, symmetric, an LDL^t decomposition is computed; for the other matrices (unsymmetric) an LU decomposition is computed. Matrices pancake_2 and Geoazur_192 use single precision, complex, arithmetic; the other matrices use double precision, real, arithmetic.

Mapping	M_0	S_{max} (MB)	e_{max}	S_{avg} (MB)	e_{avg}	Time (s)
Proportional		4417	0.06	1852	0.14	1465
Memory-aware	1288	940	0.27	672	0.38	1369
Aggregated memory-aware	1288	875	0.29	676	0.38	1381
Memory-aware	644	478	0.54	364	0.71	2061
Aggregated memory-aware	644	399	0.65	390	0.66	1961
Memory-aware	429	453	0.57	294	0.87	2695
Aggregated Memory-aware	429	392	0.66	289	0.89	2301
Memory-aware	322	292	0.88	258	0.99	3141
Aggregated memory-aware	322	279	0.92	258	0.99	2799
Memory-aware	258	259	0.99	258	1.00	3765
Aggregated memory-aware	258	259	0.99	258	1.00	3370
Tree serialization		260	0.99	258	1.00	9070

Table 4.2: Experiments with the Geoazur_192 matrix. For the memory-aware mapping, the imposed memory bounds of 1288, 644, 429, 322, and 258 MB correspond, respectively, to a memory efficiency of 0.2, 0.4, 0.6, 0.8 and 1.0.

As expected, the tree serialization approach delivers a near-perfect memory scalability ($e_{max} = 0.99$ and $e_{avg} = 1.00$). However, the run time is over six times higher than the one obtained with the plain proportional mapping strategy. This is due to the prohibitive amount of communications generated by this mapping. For the Geoazur_192 problem, and using 256 MPI processes, the volume of communication with the tree serialization mapping is roughly nine times the volume generated by the proportional mapping strategy, and the number of messages is over 30 times larger than the number of messages with the default mapping. We also experimented with different numbers of processes and observed that the performance of the tree serialization strategy degrades significantly when the number of processes increases, which is due to the large number of messages and the small granularity of tasks. These observations show that the tree serialization mapping is in general prohibitive in practice, especially for large number of processes.

For matrix Geoazur_192, we also report results with four values of M_0 to illustrate how the memory-aware algorithm can be used. The four values that we use correspond to constraining the mapping such that $e_{max} \geqslant 0.2$, $e_{max} \geqslant 0.4$, $e_{max} \geqslant 0.6$, $e_{max} \geqslant 0.8$, and $e_{max} \approx 1.00$ respectively. Note that value 0.2 is similar to the value of e_{max} obtained using the default mapping in MUMPS (see Table 4.1), while the others are significantly higher. Thanks to the memory-aware mapping, the memory constraint is respected and performance is interesting since the run time remains comparable to our references (proportional mapping and default mapping in MUMPS 5.0.0) which need significantly more memory. On this matrix, we also observe that the larger the size of the memory is, the lower the run time is. This makes sense although it is not guaranteed by our memory-based mapping heuristics. It is also interesting to see that, with $M_0 = 258$ MB, we indeed reach a near-perfect memory scalability, but with a much better performance than the one obtained with the tree serialization strategy.

When groups are added to our memory-aware algorithm, we generally observe an improvement in the run time. This strategy exploits more tree parallelism but enforces the same memory constraints as the baseline approach. Aggregated memory-aware mapping is thus the most robust approach and will be used in the rest of this experimental section.

In Table 4.3 we report results on a large class of matrices; we compare a proportional mapping strategy and the aggregated memory-aware algorithm. We observe that using a memory-aware strategy that targets $e_{max} = 0.8$, we are able to decrease the memory peak by factors between 2.5 (matrix cage13) and 23.8 (matrix meca_raff6). This comes at the price of a moderate increase in run time for most problems. The worst case is meca_raff6 for which the increase in run time is about 70%. For matrix cage13, the memory-aware mapping actually delivers slightly better performance with $e_{max} = 0.8$ than with both $e_{max} = 0.4$ and proportional mapping. Although counter-intuitive, it may happen that smaller task granularities yield better time performance, especially since the proportional mapping heuristic is designed to balance memory rather than optimize time. Matrix HV15R is particularly interesting because the factorization cannot complete when proportional mapping (or the default mapping in MUMPS 5.0.0) is used. Indeed, we use 16 MPI processes per node and the average memory peak (estimated during the analysis phase) is 23.6 GB per MPI process with the default strategy, while each node of our system only has 64 GB of memory.

Overall, the memory-aware mapping exhibits very interesting results compared to the default strategy in MUMPS, since we are able to significantly decrease the memory footprint without dramatically decreasing performance. For all matrices, we have decreased the maximum memory peak by an important factor that is increasing with our target for memory efficiency e_{max}. The penalty in run time also depends on the target for memory efficiency and is typically between 40% and 60% on 64 processes with respect to the

Matrix	Mapping	S_{max}	e_{max}	S_{avg}	e_{avg}	Time (s)
cage13	PM	912	0.37	737	0.45	406
	MA $e = 0.4$	639	0.52	516	0.66	381
	MA $e = 0.8$	360	0.93	338	0.99	372
pancake2_3	PM	1723	0.10	619	0.27	425
	MA $e = 0.4$	324	0.51	257	0.63	538
	MA $e = 0.8$	177	0.93	176	0.93	560
as-Skitter	PM	556	0.11	190	0.31	144
	MA $e = 0.4$	142	0.42	76	0.78	168
	MA $e = 0.8$	72	0.83	61	0.98	232
HV15R	PM	23624	0.07	10126	0.15	N/A
	MA $e = 0.4$	2778	0.50	1855	0.75	4718
	MA $e = 0.8$	1407	0.98	1390	0.99	4511
MORANSYS1	PM	1733	0.16	939	0.30	320
	MA $e = 0.4$	695	0.40	477	0.59	392
	MA $e = 0.8$	322	0.87	285	0.98	475
meca_raff6	PM	2951	0.04	1741	0.06	305
	MA $e = 0.4$	226	0.46	128	0.81	433
	MA $e = 0.8$	124	0.84	103	1.00	514

Table 4.3: Comparison of the memory-based proportional mapping with the aggregated memory-aware algorithm for two target efficiencies.

performance of the proportional mapping strategy. When comparing with the default mapping used in MUMPS 5.0.0 (Table 4.1), we observe similar results. Although the mapping strategy of MUMPS typically yields lower memory consumption than a proportional mapping, using a memory-aware algorithm can significantly reduce memory usage, at the price of a moderate penalty in factorization time.

Chapter 5

Low-rank approximation techniques for sparse, direct solvers

Sparse matrices that we commonly have to deal with only have a few nonzeros per row. For example, if the matrix results from the discretization of a Partial Differential Equation, the number and position of the nonzero coefficients is determined by the discretization mesh. We commonly refer to this property of the matrix as its *structural sparsity*. In Section 2.2 we have discussed how, once the elimination order is fixed, also the position of nonzeros in the factors resulting from a sparse factorization is fully determined; that is to say that also the factors have a well determined structural sparsity, despite they are much denser than the factorized matrix because of fill-in. In some cases, however, some of the information carried by these factors can be considered irrelevant. This may be, for example, the case in applications where this information has lower magnitude that the measurement errors of the instruments that were used to generate the input data, or because only a very approximate factorization is needed to be used as a preconditioner for an iterative method. We refer to this property as *data sparsity*. In this chapter we will discuss how to take advantage of this property to reduce the cost (both in terms of floating point operations and memory) of sparse factorizations through the use of *low-rank approximation* techniques.

5.1 Low-rank approximations

5.1.1 Low-rank matrices

Low-rank approximations techniques rely on the definition of *low-rank matrix* which is introduced below.

Definition 5.1 Range and null space of a matrix.— *Let $A \in \mathbb{R}^{m \times n}$. The range of A is*

$$range(A) \coloneqq \{Ax \in \mathbb{R}^m \ : \ x \in \mathbb{R}^n\}.$$

The null space of A is

$$null(A) \coloneqq \{x \in \mathbb{R}^n \ : \ Ax = 0\}.$$

Definition 5.2 Rank of a matrix.— *Let $A \in \mathbb{R}^{m\times n}$. The rank of A is*

$$rank(A) \coloneqq dim(range(A)).$$

Note that $\text{rank}(A) + \dim(\text{null}(A)) = n$. Any matrix A of rank $r = \text{rank}(A)$ admits a factorization of the type

$$A = XY^T, \quad X \in \mathbb{R}^{m\times r}, \; Y \in \mathbb{Y}^{n\times r} \tag{5.1}$$

that is to say, matrix A can be represented in the form of a *rank-r* product; note that this representation takes $O((m+n)r)$ memory as opposed to $O(mn)$ for the standard form. This observation leads to the definition of low-rank matrix

Definition 5.3 Low-rank matrix.— *Let $A \in \mathbb{R}^{m\times n}$ be a matrix of rank r. A is low-rank if*

$$(m+n)r < mn. \tag{5.2}$$

With a slight abuse of notation, we will refer to matrices that do not respect Equation (5.2) as *full-rank* matrices even though their rank is not actually full (that is $r = n$).

5.1.2 Basic linear algebra operations on low-rank matrices

Note that Definition 5.3 simply states that a low-rank matrix is one that can be more conveniently stored in low-rank form (as in Equation (5.1)) rather than in standard (or full-rank) form; however, the low-rank representation not only allows for reducing the storage but also the complexity of operations; in the reminder of this section we present some basic linear algebra operations involving low-rank matrices and the associated cost.

Triangular solve. This operation computes $A \leftarrow L^{-1}A$ where L is a full-rank lower-triangular matrix and $A \in \mathbb{R}^{m\times n}$ is of rank r_A. It can be computed as

$$A \leftarrow L^{-1}A = (L^{-1}X_A)Y_A^T = WY_A^T, \quad \text{with } W = (L^{-1}X_A) \in \mathbb{R}^{m\times r}.$$

This costs $m^2 r_A$ floating point operations as opposed to $m^2 n$ for the case where A is in full-rank form. The cases where L^{-1} is multiplied on the right or where an upper-triangular matrix U is used instead of L can be treated in a similar way.

Matrix sum. This operation computes $C = A + B$ where A, B and C are all in $\mathbb{R}^{m\times n}$ and $A = X_A Y_A^T$ and $B = X_B Y_B^T$ are of rank r_A and r_B, respectively. This can be computed as

$$C = A + B = X_A Y_A^T + X_B Y_B^T = [X_A X_B]\,[Y_A Y_B]^T\,.$$

This does not involve any floating point operation but leads to a rank-$(r_A + r_B)$ representation for C where $r_A + r_B$ may be greater that the actual rank of C.

Matrix product This operation computes $C = C + AB$ with $A \in \mathbb{R}^{m\times k}$, $B \in \mathbb{R}^{k\times n}$ and $C \in \mathbb{R}^{m\times n}$. When all the three matrices are full-rank, the cost of this operation is $2mnk$. We will examine the following three cases

- A, B and C are all low-rank of ranks r_A, r_B and r_C, respectively. This is computed as

  $$X_C Y_C^T \leftarrow X_C Y_C^T + X_A (Y_A^T X_B) Y_B^T.$$

  The inner product $Y_A^T X_B$ costs $2kr_A r_B$ flops. The result of this product is a matrix W of size $r_A \times r_B$ and can then be multiplied either to the right or to the left:

  - right: WY_B^T costs $2nr_A r_B$ flops and yields a matrix Z of size $r_A \times n$. The resulting rank-r_A matrix $X_A Z$ can then be summed with $X_C Y_C^T$ at no cost. The final cost is thus $2r_A r_B(k+n)$ and the result is expressed in rank-(r_C+r_A) form.
  - left: $X_A W$ costs $2mr_A r_B$ flops and yields a matrix Z of size $m \times r_B$. The resulting rank-r_B matrix ZY_B^T can then be summed with $X_C Y_C^T$ at no cost. The final cost is thus $2r_A r_B(k+m)$ and the result is expressed in rank-(r_C+r_B) form.

  As a result the right or the left case can be more convenient depending on the sign of $(m-n)$.

- A, B are low-rank of ranks r_A and r_B , respectively, and C is full-rank. This is similar to the above case except that the final sum is not free. Before being added to C, the low-rank matrix resulting from the $X_A(Y_A^T X_B)Y_B^T$ product has to be brought back into full-rank form by means of an outer product operation. This costs $2mnr_A$ or $2mnr_B$ in the right or left cases described above, respectively. Therefore, the choice between the right or left cases depends on the sign of

  $$(m-n)r_A r_B + mn(r_A - r_B).$$

- A, is low-rank of rank r_A and B and C are full-rank. This is computed as

  $$C \leftarrow C + X_A (Y_A^T B)$$

  The product $W = Y_A^T B$ costs $2r_A kn$ flops. It is then followed by the outer product $C = C + X_A W$ which costs $2mr_A n$. The total cost is thus $2r_A n(m+k)$. The case where B is low-rank and A and C full-rank can be treated the same way. Also, the case where C is low-rank is treated by performing a low-rank sum (at no cost) without the outer-product.

5.1.3 Numerically low-rank matrices

A matrix A which is not low-rank can be approximated with a low-rank matrix $\tilde{A}$ with controlled accuracy. To see how this is possible, we first have to introduce the *singular value decomposition* of a matrix.

Theorem 5.1 Singular value decomposition [86, Theorem 2.4.1].— *If A is a real $m \times n$ matrix then there exist orthogonal matrices*

$$U = [u_1 \cdots u_m] \in \mathbb{R}^{m \times m} \quad \textit{and} \quad V = [v_1 \cdots v_n] \in \mathbb{R}^{n \times n}$$

such that

$$U^T A V = \Sigma = diag(\sigma_1, \cdots, \sigma_p) \in \mathbb{R}^{m \times n}, \quad p = min(m, n),$$

where $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_p \geq 0$.

The values $\sigma_1, \cdots, \sigma_p$ are called the *singular values* of A and $U\Sigma V^T$ its singular value decomposition. The following theorem defines how to compute the best rank-r approximation of a matrix A.

Theorem 5.2 Best rank-r approximation [67, 95].— *Let A be a matrix in $\mathbb{R}^{m\times n}$ and $A = U\Sigma V^T$ its singular value decomposition. The minimization problem*

$$\min_{rank(\tilde{A})\leq r} \|A - \tilde{A}\|_2$$

is solved by

$$\tilde{A} \coloneqq U_r\Sigma_r V_r^T, \quad U_r = [u_1 \cdots u_r]\,,\ V_r = [v_1 \cdots v_r]\,,\ \Sigma_r = diag(\sigma_1, \cdots, \sigma_r).$$

The arising error is $\|A - \tilde{A}\|_2 = \sigma_{r+1}$.

Proof. Let $U_{r+1} = [u_{r+1} \cdots u_m]$, $V_{r+1} = [v_{r+1} \cdots v_n]$ and $\Sigma_{r+1} = diag(\sigma_{r+1}, \cdots, \sigma_p)$. Then

$$\|A - \tilde{A}\|_2 = \|A - U_r\Sigma_r V_r^T\|_2 = \|U_{r+1}\Sigma_{r+1}V_{r+1}^T\|_2 = \sigma_{r+1}.$$

In order to prove that $\tilde{A}$ is the best rank-r approximation of A we need to prove that for any other rank-r approximation of $B = XY^T$ of A, the error is higher than σ_{r+1}. Because Y is of rank r, $\dim(\text{null}(Y^T)) = n - r$; therefore

$$\text{span}(v_1, \cdots, v_{r+1}) \cap \text{null}(Y^T) \neq \emptyset.$$

That is to say, there exist a linear combination $w = \gamma_1 v_1 + \cdots + \gamma_{r+1}v_{r+1}$ such that $Y^T w = 0$. Without loss of generality, we can assume that $\|w\|_2 = 1$, which implies $\gamma_1^2 + \cdots + \gamma_{r+1}^2 = 1$. Therefore

$$\|A - B\|_2^2 \geq \|(A - B)w\|_2^2 = \|Aw\|_2^2 = \gamma_1^2\sigma_1^2 + \cdots + \gamma_{r+1}^2\sigma_{r+1}^2 \geq \sigma_{r+1}^2$$

which concludes the proof. □

Theorem 5.2 states that the best rank-r approximation of A is a matrix $\tilde{A}$ obtained by computing the singular value decomposition of A and by dropping all the last $p - r$ singular values. Note that $U_r\Sigma_r V_r^T$ is a rank-r representation of A; alternatively one can use $U_r W^T$ or ZV_r^T with $W = V_r\Sigma_r$ and $Z = U_r\Sigma_r$, respectively. In practice, however, it is more desirable to control the accuracy of the approximation rather than its rank: given a threshold ε we want to compute the representation of smallest rank with an accuracy that is better than or equal to ε.

Definition 5.4— *Let A be a matrix in $\mathbb{R}^{m\times n}$ and ε a prescribed threshold. The ε-rank of A (also called numerical rank with threshold ε) is*

$$r_\varepsilon = \min\{rank(B)\ :\ \|A - B\|_2 \leq \varepsilon\}.$$

Theorem 5.2 implies that r_ε is equal to the number of singular values of A that are greater than ε. The B matrix for which the minimum in Definition 5.4 is achieved is thus

$$B = U_{r_\varepsilon}\Sigma_{r_\varepsilon}V_{r_\varepsilon}^T, \quad U_{r_\varepsilon} = [u_1 \cdots u_{r_\varepsilon}]\,,\ V_{r_\varepsilon} = [v_1 \cdots v_{r_\varepsilon}]\,,\ \Sigma_{r_\varepsilon} = diag(\sigma_1, \cdots, \sigma_{r_\varepsilon})$$

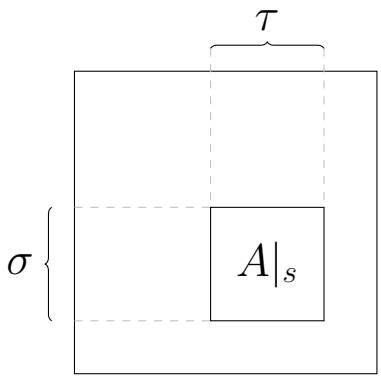


Figure 5.1: Clusters, block-cluster and matrix block.

with

$$\sigma_1 \geq \cdots \geq \sigma_{r_\varepsilon} > \varepsilon \geq \sigma_{r_\varepsilon+1} \geq \cdots \geq \sigma_p.$$

Clearly, if r_ε fulfills the condition in Equation (5.2), B is a low-rank approximation of A with accuracy ε. For the sake of readability, we will drop the ε in r_ε.

The operation of computing a low-rank approximation of a matrix A is referred to as *compression* in the remainder of this document. As we have seen above, this can be achieved by computing a SVD decomposition of A but this implies a cost of $O(mn^2)$ flops, assuming $m \geq n$. Other cheaper but less accurate methods exist. One commonly used technique uses a QR factorization with column pivoting $AP = QR$, such as the one described in Section 2.1.4.3. This factorization is stopped at step k if the diagonal coefficient $r_{k,k}$ of R is smaller than the prescribed threshold ε. The trailing submatrix is discarded and the partially computed factors $Q_{:,1:k}$ and $(RP^T)_{1:k,:}$ are used as a rank-k approximation of A. Because of the early stoppage, this operation only costs $O(mnk)$ flops.

Other well known and commonly used methods include randomized sampling [97, 114] or Adaptive Cross Approximation [32].

5.2 Low-Rank formats

In most practical cases, the matrix A that we have to deal with, not only is of full rank but its singular values decay so slowly that it is not possible to compute a compact and accurate low-rank approximation. In some cases, however, it has been shown that conveniently defined blocks of A or of its Schur complement can be effectively approximated by a low-rank product [31]. Assume A has row index set $\mathcal{I}$ and column index set $\mathcal{J}$. Assume $\sigma \subset \mathcal{I}$ and $\tau \subset \mathcal{J}$ are subsets of $\mathcal{I}$ and $\mathcal{J}$; we refer to them as *clusters* and to $s = \sigma \times \tau$ as a *block-cluster* or, simply, block. A matrix block of A is thus defined as the restriction of A on the index subsets σ and τ: $A|_s$. This notation is illustrated in Figure 5.1. When it is clear from the context we will use the word block to refer to a block-cluster or a matrix block.

Different low-rank matrix formats can be used to take advantage of this property which can be classified based on these three properties:

- *block-admissibility condition*: a criterion that tells whether a matrix block can be effectively approximated with a low-rank product;

- *block-partitioning*: a method for defining matrix blocks;
- *low-rank bases*: how matrix blocks should be compressed.

In the following section we will provide some details of these properties and we refer the reader to the book by Hackbusch [95] for a thorough analysis of the subject.

5.2.1 Block-admissibility condition

A block-admissibility condition states whether a block can be approximated with a low-rank product or not. In the following, we will assume that each index of $\mathcal{I}$ is related to a point in the domain where the problem is defined. We will denote $\text{diam}(\sigma)$ the diameter of the subdomain formed by the points associated with σ and $\text{dist}(\sigma,\tau)$ the distance between the two subdomains associated with σ and τ.

One commonly employed condition is the so called *strong* block-admissibility condition:

$$s = \sigma \times \tau \text{ is admissible iff } \max(\text{diam}(\sigma), \text{diam}(\tau)) \leq \eta \ \text{dist}(\sigma,\tau). \qquad (Adm_b^s)$$

This formalizes a very simple intuition. A matrix block defines the interaction between the two subdomains related to σ and τ; if these are relatively small and far away, their interaction is weak and the corresponding matrix block has low numerical rank. More or less admissible blocks can be obtained depending on the value of the scalar η. The choice

$$\eta = \eta_{max} = \max_{\text{dist}(\sigma,\tau)>0} \frac{\max(\text{diam}(\sigma), \text{diam}(\tau))}{\text{dist}(\sigma,\tau)}$$

amounts to saying that all non contiguous clusters define an admissible block. This is referred to as *least-restrictive* strong block admissibility condition

$$s = \sigma \times \tau \text{ is admissible iff } \text{dist}(\sigma,\tau) > 0. \qquad (Adm_b^{lrs})$$

Finally, we can be even more tolerant and assume that all disjoint clusters define an admissible block:

$$s = \sigma \times \tau \text{ is admissible iff } \sigma \cap \tau = \emptyset. \qquad (Adm_b^w)$$

This is called *weak* admissibility condition.

5.2.2 Block-partitioning

Without loss of generality, we assume that $\mathcal{I} = \mathcal{J}$. Here we are interested in finding *block-partitions*, of $\mathcal{I} \times \mathcal{I}$ to identify matrix blocks that can be compressed. First, we introduce the definition of *partition* of an index set.

Definition 5.5 Partition.— *A partition* $\mathfrak{S}(\mathcal{I})$ *of an index set* $\mathcal{I}$ *is*

1) $\mathfrak{S}(\mathcal{I}) = \{\sigma_1, \cdots, \sigma_p\}, \quad \sigma_i \subset \mathcal{I}$
2) $\sigma_i \cap \sigma_j = \emptyset \ \ for \ \ i \neq j$
3) $\bigcup_{i=1}^{p} \sigma_i = \mathcal{I}.$

This leads to the definition of a block-partition.

Definition 5.6 Block-partition.— *A block-partition* $\mathfrak{S}(\mathcal{I}\times\mathcal{I})$ *is*

1) $\mathfrak{S}(\mathcal{I}\times\mathcal{I}) = \{s_1, \cdots, s_p\}, \quad s_k = \sigma_i \times \tau_j, \quad \sigma_i \subset \mathcal{I},\ \tau_j \subset \mathcal{I}$
2) $s_i \cap s_j = \emptyset \ \ for \ \ i \neq j$
3) $\bigcup_{k=1}^{p} s_k = \mathcal{I}\times\mathcal{I}.$

The most commonly used methods to compute a suitable block-clustering rely on the use of *cluster trees* and *block-cluster trees*.

Definition 5.7 Cluster tree.— *Given an index set* $\mathcal{I}$*, a cluster tree* $T(\mathcal{I})$ *is a tree structure whose nodes are clusters of* $\mathcal{I}$ *that respect the following conditions:*

1) $\mathcal{I} \in T(\mathcal{I})$ *is the root of the tree.*
2) $\sigma_1 \cup \sigma_2 = \emptyset \ \ for \ \ \sigma_1, \sigma_2 \in S_{T(\mathcal{I})}(\tau)$ *and* $\sigma_1 \neq \sigma_2$
3) $\bigcup_{\sigma\in S_{T(\mathcal{I})}(\tau)} \sigma = \tau \ \ for\ all \ \ \tau \in T(\mathcal{I}) \setminus L_{T(\mathcal{I})}$

where $S_{T(\mathcal{I})}(\tau)$ *is the set of child nodes of* τ *and* $L_{T(\mathcal{I})}$ *is the set of leaves of the tree.*

Note that, by definition, the sons of node τ define a partition of τ and thus, by recursion, the leaves of the cluster tree define a partition of $\mathcal{I}$. The definition of block-cluster tree can now be derived.

Definition 5.8 Block-cluster tree.— *Le* $T(\mathcal{I})$ *be a cluster tree for the index set* $\mathcal{I}$*. A block-cluster tree* $T(\mathcal{I}\times\mathcal{I})$ *is a tree structure whose nodes are block-clusters of* $\mathcal{I}\times\mathcal{I}$ *that respect the following conditions:*

1) $\mathcal{I}\times\mathcal{I} \in T(\mathcal{I}\times\mathcal{I})$ *is the root of the tree.*
2) *for every non-leaf node* $s = \sigma\times\tau$ *:*
$$S_{T(\mathcal{I}\times\mathcal{I})}(s) \coloneqq \{\sigma'\times\tau' \,:\, \sigma' \in S_{T(\mathcal{I})}(\sigma), \quad \tau' \in S_{T(\mathcal{I})}(\tau)\}$$

Note that $S_{T(\mathcal{I})}(\sigma) = \emptyset$ *or* $S_{T(\mathcal{I})}(\tau) = \emptyset$ *imply that* s *is a leaf.*

Because of the recursive nature of Definition 5.8, we will refer to matrix block-partitions obtained through the use of a block-cluster tree as *hierarchical*.

A much simpler block-partitioning technique, which we refer to as *flat*, consists in defining a partition $\mathfrak{S}(\mathcal{I})$ of the index space $\mathcal{I}$ into subsets of (roughly) equal size b; the block-partition is then obtained as the cross-product of $\mathfrak{S}(\mathcal{I})$ by itself, i.e., $\mathfrak{S}(\mathcal{I}\times\mathcal{I}) = \mathfrak{S}(\mathcal{I})\times\mathfrak{S}(\mathcal{I})$. Note that a flat block-partition can be thought of resulting from a two-level block-cluster tree made of a root node $\mathcal{I}\times\mathcal{I}$ with as many sons as $\lceil \#\mathcal{I}/b \rceil^2$ but this is unnecessarily complex; its simplicity, instead, is one of the properties that make a flat block-partitioning preferable to a hierarchical one in some situations.

5.2.3 Nested bases

Nested bases are commonly used in combination with hierarchical block-partitionings so let $T(\mathcal{I}\times\mathcal{I})$ be the block-cluster tree resulting from a cluster tree $T(\mathcal{I})$. Nested bases rely on the following two properties:

Format	Admissibility	Partitioning	Nested Bases	Authors
BS	Weak	Flat	no	Cheng et al. [49]
BLR	Strong/Weak	Flat	no	Amestoy et al. [J4]
$\mathcal{H}$	Strong	Hierarchical	no	Hackbusch [94]
HODLR	Weak	Hierarchical	no	Aminfar et al. [19]
$\mathcal{H}^2$	Strong	Hierarchical	yes	Börm et al. [37]
HSS	Weak	Hierarchical	yes	Chandrasekaran et al. [47]
HBS	Weak	Hierarchical	yes	Gillman et al. [82]

Table 5.1: A taxonomy of the most well known low-rank formats.

- The same row-basis X_σ can be used for all blocks $A|_{\sigma\times\tau}$ with the same σ, i.e., all the $A|_{\sigma\times\tau}$ with the same σ will have a rank-r approximation of the form $X_\sigma Y_{\tau_i}^T$, $i = 1, 2, \ldots$. The same holds for the column-basis Y_τ.

- Let σ be a node of the cluster tree $T(\mathcal{I})$ and σ' be its son; assume, also, that $\mathcal{X}_\sigma = range(X_\sigma)$ and $\mathcal{X}_{\sigma'} = range(X_{\sigma'})$, Then $\mathcal{X}_\sigma|_{\sigma'} = \mathcal{X}_{\sigma'}$. This means that the row-basis of a large block can be defined as a combination of the row-bases of the smaller blocks associated with the clusters in $S_{T(\mathcal{I})}(\sigma)$.

These two properties can be used to achieve further reductions in the complexity of operations and in the storage.

5.2.4 A taxonomy of low-rank formats

Table 5.1 contains a taxonomy of the most commonly used low-rank formats along with the admissibility condition, partitioning technique and nested bases use they rely on; references to the authors of these formats are also provided. We discuss the details of two of them, namely the BLR and $\mathcal{H}$ formats, below.

5.2.4.1 The Block Low-Rank format

The Block Low-Rank format, introduced by Amestoy et al. [J4], uses a flat block-partitioning scheme and, thus, it does not use nested bases. BLR can use either a weak or a strong admissibility condition.

The construction of a BLR matrix is achieved by simply partitioning the index set $\mathcal{I}$ into parts of a prescribed size b, which induces matrix block-partition illustrated in Figure 5.2.

Definition 5.9 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 $\tilde{A}$ of a dense matrix A is of the form:*

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

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* $\tilde{A}_{i,j} = X_{i,j}Y^T_{i,j}$ *at accuracy* ε*, where* $X_{i,j}$ *is a* $m_{i,j} \times k^{\varepsilon}_{i,j}$ *matrix and* $Y_{i,j}$ *is a* $n_{i,j} \times k^{\varepsilon}_{i,j}$ *matrix.*

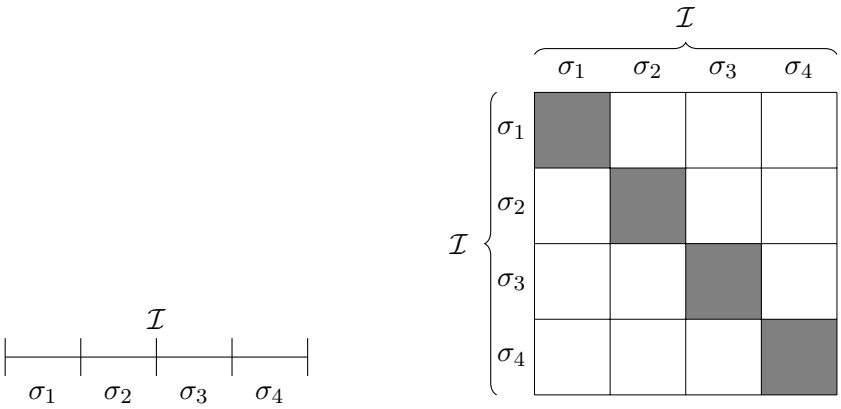


Figure 5.2: An example of clustering and its associated flat block-clustering and BLR representation where we have assumed a weak admissibility condition.

Although either weak or strong admissibility conditions can be used, in order to achieve a favorable theoretical properties, the block-partition has to respect some constraints; specifically, we will consider a block-partition *BLR-Admissible* if the maximum number of non-admissible blocks in every block-rows or block columns is bounded by a constant (more on this will be discussed in Section 5.4).

5.2.4.2 The $\mathcal{H}$-matrix format

The $\mathcal{H}$-matrix, introduced by Hackbusch [94], is probably the most well known and widely used low-rank approximation format. It relies on a hierarchical blocking with a strong admissibility condition and does not use nested bases. The block-partition is built through a block-cluster tree starting from the root $\mathcal{I} \times \mathcal{I}$ and applying the recursive process in Definition 5.7: if a block-cluster is admissible then it is kept in the block-partition otherwise it is replaced by its sons computed as in step 2) of Definition 5.7. The recursion is stopped when a minimum prescribed block size is reached. The resulting block-partition is called $\mathcal{H}$-admissible.

Definition 5.10 $\mathcal{H}$-admissible block partition.— *A block-partition is $\mathcal{H}$-admissible if all of its locks are either admissible, according to a strong admissibility condition, or of size smaller than* c_{min}*:*

$$\mathfrak{S}(\mathcal{I} \times \mathcal{I}) \text{ is admissible } \Leftrightarrow \forall \sigma \times \tau \in \mathfrak{S}(\mathcal{I} \times \mathcal{I}), \quad \sigma \times \tau \text{ is admissible or } \min(\#\sigma, \#\tau) \leq c_{min}$$

Note that replacing the strong admissibility condition with the weak one the HODLR format is obtained rather than the $\mathcal{H}$-matrix one.

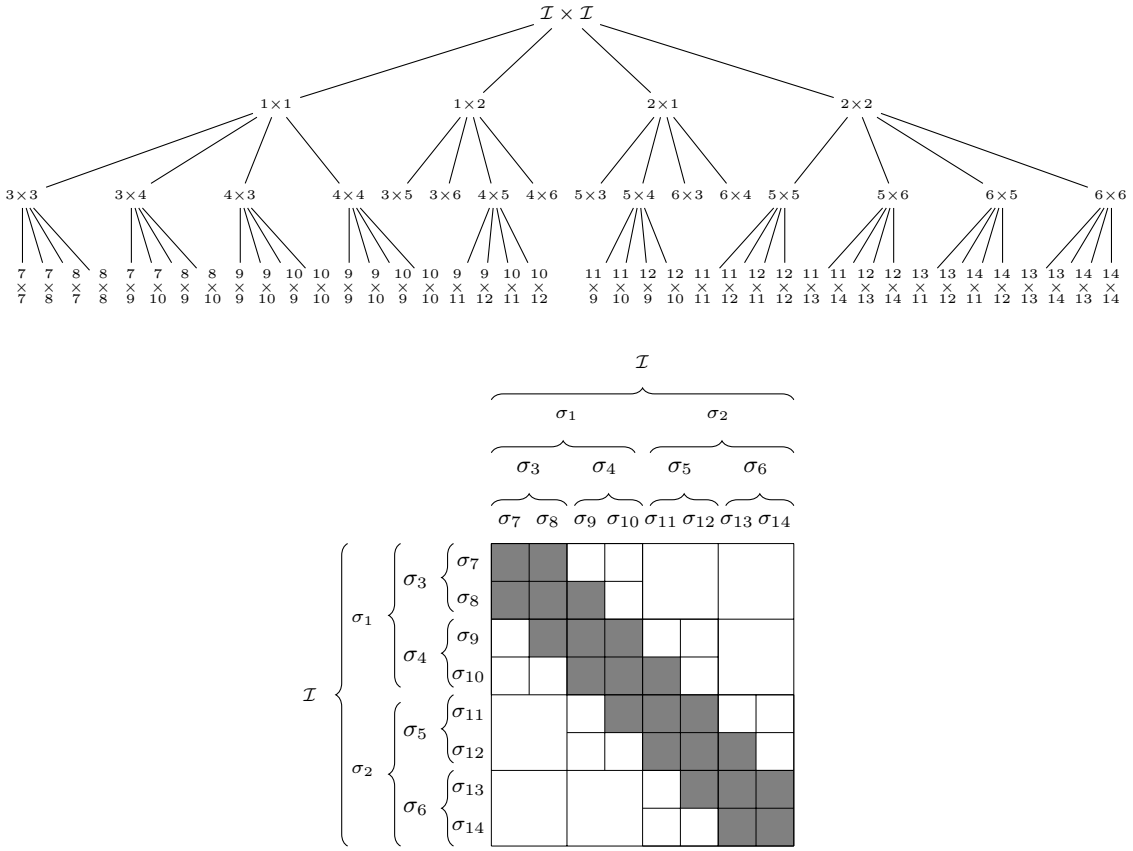


Figure 5.3: An example of block-cluster tree on top ($\sigma_i \times \sigma_j$ has been abbreviated as $i \times i$) and its associated $\mathcal{H}$-admissible block-partition at the bottom; the white blocks are admissible whereas the gray ones are non admissible but smaller than c_{min}.

5.3 BLR multifrontal

It has been shown that, for matrices issued from the discretization of elliptic PDEs, conveniently defined off-diagonal blocks of Schur complements can be accurately approximated with low-rank products [46]. Because fronts are closely related to Schur complements and because they are dense matrices, any of the low-rank formats discussed above can be used to compress them in order to reduce the storage and the complexity of the multifrontal factorization and solve operations. The use of low-rank approximations in direct solvers for sparse linear systems has received considerable attention in recent years. For example Wang et al. [158], Xia [163], and Xia et al. [164] or Ghysels et al. [78] have investigated the use of the HSS format in the multifrontal method, Gillman et al. [82] has studied the use of the HBS format, Aminfar et al. [19] and Coulier et al. [52] have focused on the use of HODLR and $\mathcal{H}^2$, respectively. Other recent efforts by Pichon et al. [134] have investigated the use of the HODLR format in a right-looking supernodal solver.

We are interested in the use of the BLR format within a general purpose, algebraic, parallel multifrontal solver. The main advantage of this format over hierarchical ones is its simplicity which makes it a better candidate for being used in a context where the geometry of the problem is not known (and, therefore, a weak admissibility is easier to check) and where special care must be taken to the efficiency of operations and the

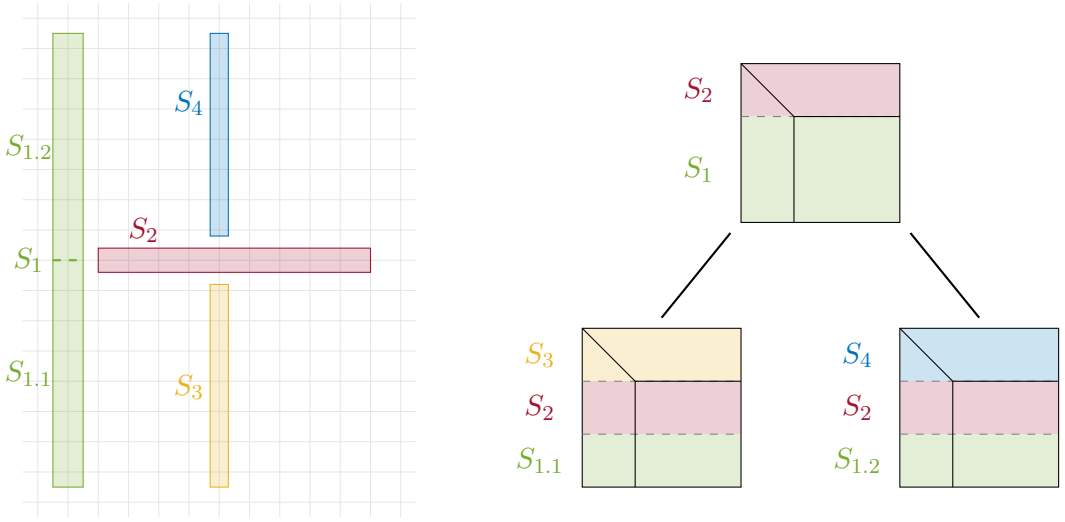


Figure 5.4: Some separators of a 2D regular grid (S_1 being the topmost) and the associated fronts.

possibility to develop and implement scalable parallel algorithms.

In order to integrate the BLR format within a multifrontal solver, three main issues have to be tackled. First it is necessary to develop a method for computing the block-partition of all the frontal matrices, second, we must conceive methods for the factorizations of frontal matrices that can take advantage of the low-rank property and, finally, we have to take care of how to perform assembly operations.

In the following section we will provide details of how we dealt with these issues. In Section 5.4 we will present an analysis of the complexity of the BLR-based multifrontal solver and then, in Section 5.5, we will address aspects related to the efficiency and scalability of the code in a shared-memory, parallel setting.

5.3.1 Block-partitioning of fronts

Remember, from Section 2.2, that the unknowns associated with a frontal matrix can be split in two parts. The first contains the so-called fully-summed variables which are eliminated in that front; the second contains the non fully-summed which are only updated and which appear as fully-summed variables in an ancestor node of the elimination tree and as non fully-summed on all the nodes lying along the path that connects the node to that ancestor. If, for example, the input sparse matrix has been permuted using a nested dissection method (see Section 2.2.3.1), the fully-summed variables of a front correspond to a separator whereas the non fully-summed correspond to pieces of other separators associated with ancestor nodes; this is illustrated in Figure 5.4. For the sake of simplicity, in the rest of this chapter we will always assume that the matrix was permuted according to a nested dissection ordering and, thus, that the fully-summed variables of a front are associated with a separator. Although all the methods and results described below also apply to the case of other matrix permutations, we observed experimentally that using nested dissection resulted in the best compromise between reduction of structural fill-in and improvements of the low-rank approximations.

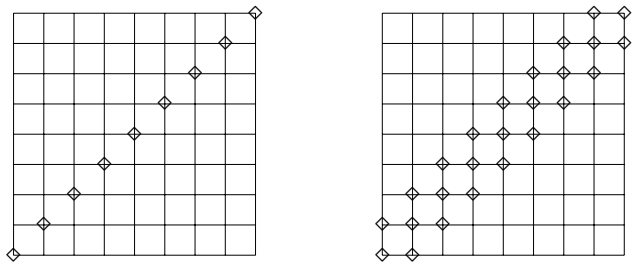
Therefore, defining a blocking of a frontal matrix implies computing a clustering of two different index sets, which we will explain in the following two sections.

5.3.1.1 Clustering of the fully-summed variables

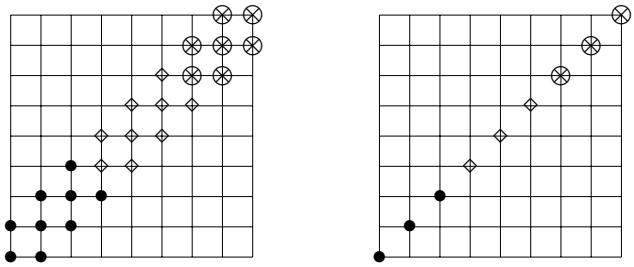
Defining a blocking of the fully-summed part of a frontal matrix (i.e., the left-topmost submatrix) amounts to clustering the variables of the associated separator. It must be noted, though, that in a completely algebraic setting, which we assume, the geometry of the problem is not available and, thus, the admissibility condition cannot be checked. Grasedyck et al. [87] propose to compute the clustering of variables using the adjacency graph $\mathcal{G}(A)$ of the matrix rather than the geometry of the problem; in this approach, the admissibility condition can thus be checked using the distance $\text{dist}_{\mathcal{G}}$ and diameter $\text{diam}_{\mathcal{G}}$ as defined in standard graph theory. Therefore, a blocking of the fully-summed part of a frontal matrix is achieved by extracting the subgraph associated with the corresponding separator and clustering it. This is guided by a several criteria. First of all, as explained above, we will use a weak admissibility condition because this is easier to check when the geometry is not available. Moreover, as mentioned above and explained in Section 5.4, it is important to minimize the number of neighbors of each cluster, which can be achieved by defining clusters with a low aspect ratio. Finally, the size of clusters has to be carefully chosen. In Section 5.4 we will explain why this size has to grow with the size of the separator; moreover, in practice, we want this size not to be excessively small because it defines the granularity of computations (and thus the efficiency of Level-3 BLAS operations) nor too big in order to achieve a satisfactory amount of concurrency in a multithreaded parallelization (see Section 5.5).

Once, for a given separator S, the desirable cluster size is defined, the clustering can be achieved using on the related subgraph $\mathcal{G}_S$ a K-Way partitioning method such as those implemented in the METIS or SCOTCH tools. It must be noted, however, that if nested dissection was used to permute the matrix in an algebraic setting (e.g., using, again, METIS or SCOTCH) the separators may be disconnected. Take, as an extreme example, the square domain depicted in Figure 5.5(a) for which a separator is defined by the nodes lying on the diagonal; this separator is totally disconnected and, thus, running a graph partitioning tool on its subgraph $\mathcal{G}_S$ yields, trivially, clusters of size one, which is clearly undesirable. Although in reality such an extreme situation would rarely occur, it is quite commonly the case where a separator is formed by multiple connected components that are close to each other in the global graph $\mathcal{G}$. This may lead to a sub-optimal clustering because variables that strongly interact due to their adjacency will end up in different clusters. In order to overcome this issue, we have developed a method which allows for reconnecting the disconnected components of the separator subgraph [J4, 159] in a way that takes into account the geometry and shape of the separator: variables close to each other in the original graph $\mathcal{G}$ have to be close to each other in the reconnected $\mathcal{G}_S$ in order to satisfy the admissibility condition. We describe an approach that achieves this objective by extending the subgraph induced by each separator with a relatively small number of level sets (a vertex v belongs to the level set $\mathcal{L}_i$ of S if and only if there exist $s \in S$ such that the distance between v and s is exactly i). The union of these level sets for $i = 1$ to p (together with the original nodes of the separator) is called a halo and p its depth. The graph of the halo will be referred to as $\mathcal{G}_H$. Note that $\mathcal{G}_S$ is included in $\mathcal{G}_H$. The graph partitioning tool is then run on $\mathcal{G}_H$ and the resulting partitioning projected back on the original subgraph $\mathcal{G}_S$. Figure 5.5 shows how this is done on the example above using just one level set. For a limited number of level sets, the extended graph preserves the shape of the separator, keeps the cost of computing the clustering limited and allows us to compute clusterings that better comply with the strategy presented in the previous section. In practice, we observed [125, 159] that one or two layers are enough to reconnect the separator and to obtain good performance (in 2D, one layer is sufficient). However,

on very complex domains, the optimal value may have to be found experimentally.



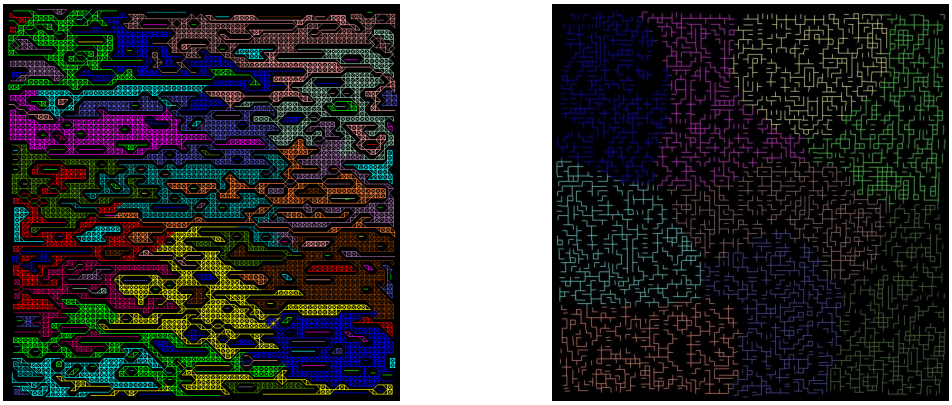
(a) separator subgraph $\mathcal{G}_S$ (b) halo subgraph $\mathcal{G}_H$



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

Figure 5.5: Halo-based partitioning of $\mathcal{G}_S$ with depth 1. $\mathcal{G}_H$ is partitioned using METIS.

Figures 5.6(a) and 5.6(b) show the result of the graph partitioning tool on the topmost separator of a cubic 3D domain discretized with a 7-point stencil without and with the halo, respectively; it is easy to see why the clustering on the right figure better complies with the weak admissibility condition. Figure 5.7, instead, shows a 3D view of the topmost separator with the computed clustering. In this figure we can see that the separator itself is skewed and not just a flat surface as one would obtain with a geometric nested dissection; nonetheless, the halo method can achieve a clustering which is suited to the BLR format.



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

Figure 5.6: The clustering of the top level separator computed with `SCOTCH` on a 3D 7-points stencil Laplacian problems, with and without the halo method.

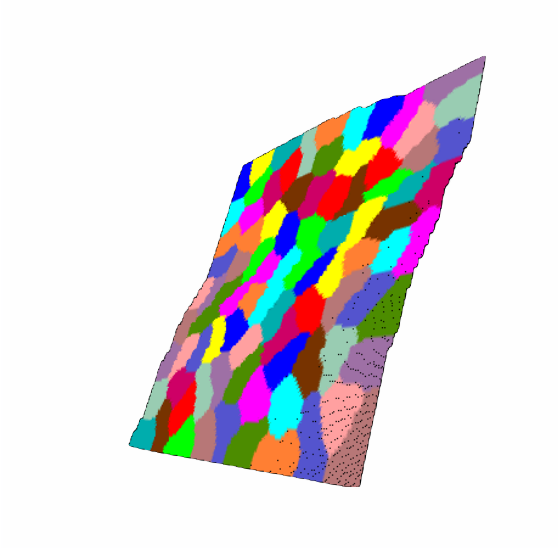


Figure 5.7: BLR clustering of the root separator of a 128^3 3D cubic problem.

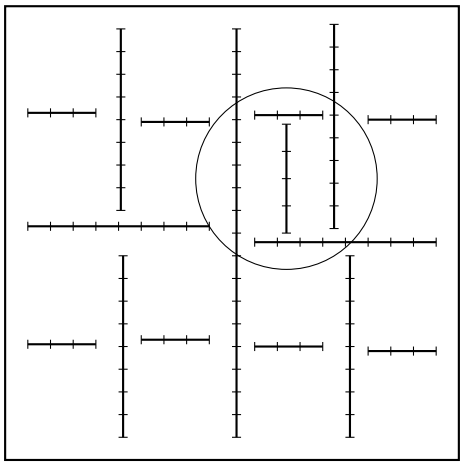
5.3.1.2 Clustering of the non fully-summed variables

One straightforward approach to clustering the non fully-summed variables amounts to extracting the associated subgraph, extending it with a halo and partitioning it exactly the as it is done for the fully-summed ones; we refer to this approach as the *explicit* clustering. Although this approach yields good clustering, it can incur excessive cost because each variable is clustered multiple times, i.e., one for each contribution block it appears in. This issue can be overcome by observing that, as explained above, the subgraph associated with the non fully-summed variables of a front is made of pieces of separators that are higher in the tree. Therefore, clustering all the separators in the tree also induces a clustering of the contribution blocks; we refer to this approach as *inherited* clustering since the clustering of a contribution block is defined by the clustering of separators in ancestor nodes. This is illustrated in Figure 5.8.

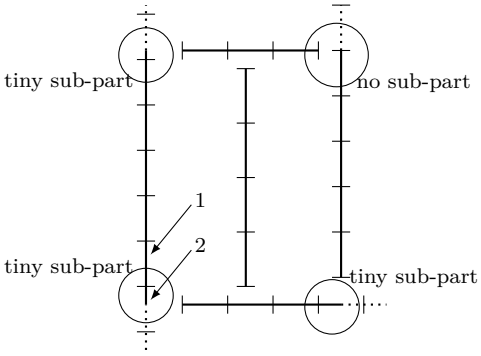
Depending on where the separators intersect, small clusters, on which the compression cannot provide much memory and operation reduction, may be formed; as a result, the CB may include blocks which are too small to be effectively compressed and to achieve a good BLAS efficiency for the related operations. Note that this problem also affects the blocking of the bottom-left submatrix (i.e., interaction between separator and CB variables), which we refer to as the $L_{2,1}$ block, but to a lower extent because the effect is damped by the good clustering computed for the separator (see Figure 5.9).

To recover BLAS efficiency, a reclustering step can be performed by merging neighbor clusters together in order to increase the block size, as Figure 5.9(c) shows. Recovering the low-rank compression is less straightforward. Indeed, it is not guaranteed that two neighbor blocks in the front correspond to two neighbor clusters in the graph. Constraining the clustering strategy to obtain this property considerably increases its complexity (by the addition of notions such as global cluster ordering, recursive ordering as in $\mathcal{H}$-matrices for instance) with a small payoff since the proportion of small clusters in a front is usually very small.

It has to be noted that the inherited clustering also provides another convenient property: the blocking of a frontal matrix is compatible with the blocking of its parent front. This translates into the fact that one block of its Schur complement will be assembled

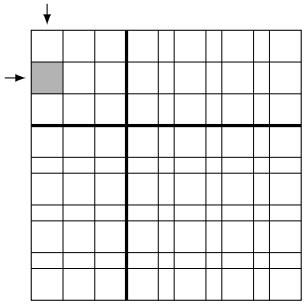


(a) A nested dissection of the domain where the variables of each separator have been clustered with the halo method, i.e. each segment is a group of variables.

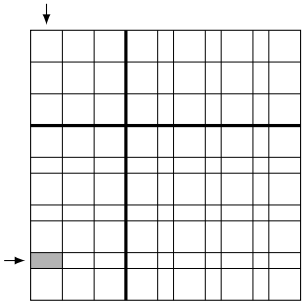


(b) Zoom on current separator and border. The separator's clustering has been computed with the halo method. The border's clustering is inherited from several separator clusterings. At some corners, a cluster maybe not be entirely involved in current front so that tiny clusters may appear.

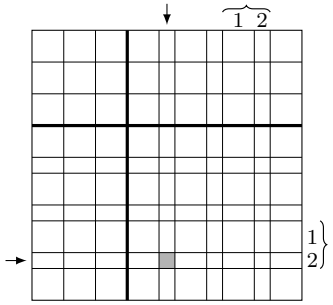
Figure 5.8: Inherited clustering where the separators are partitioned with the halo method and the borders inherit their clustering.



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



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



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

Figure 5.9: Relation between inherited clustering and front blocking. The small clusters correspond to clusters located in corners in Figure 5.8(b). The large clusters correspond to optimal clusters which are integrally kept in the current front. Note that not all the large clusters of Figure 5.8(b) are represented here.

into exactly one block of the parent front because, by construction, a block of a Schur complement is always included in one single block in the fully-summed part of another frontal matrix. This considerably eases the assembly of frontal matrices in the case of fully-structured solvers (more details in the next section).

5.3.2 Assembly operations

In our approach frontal matrices are always assembled in full-rank (i.e., uncompressed form) and gradually converted to BLR form in the course of the frontal matrix factorization; this is achieved by interleaving factorization operations with compress ones according to several variants described in Section 5.3.3. Note that full-rank assemblies imply that if a contribution block has been compressed (it does not have to be the case as explained in Section 5.3.3) then it has to be decompressed, possibly block by block, prior to being assembled into the father front. This approach allows for easier and more efficient assembly operations and an easier handling of the pivoting, as explained in the next section, but may incur in excessive storage requirements as the peak memory consumption may not be smaller than the size of the largest full-rank front in the assembly tree.

As an alternative, assembly operations can be performed using low-rank blocks which allows for always storing frontal matrices in compressed form which ultimately results in a better memory consumption. This yields what is commonly referred to as a *fully-structured* solver (see, for example, the work by Xia [163]). As explained in the previous section, by using an inherited clustering we can ensure that a block of a contribution block is assembled in only one block of the father front; nonetheless, this block can be of smaller size of that of the father (see Figure 5.8(b)) and, therefore, it has to be padded with explicit zeros in order to make its size conform to that of the father block and make the low-rank sum possible. As a result, the low-rank extend-add operation involves extra computations which can considerably limit the gains provided by the use of low-rank approximations as shown by Pichon et al. [134]. This issue can be overcome using randomization techniques [123] which make low-rank assembly operations easier, as shown by Martinsson [124], Xia [163] or Ghysels et al. [78].

A further option consists in assembling a frontal matrix in full rank form one block at a time; once a block is assembled, it is immediately compressed which means that the whole frontal matrix is compressed prior to its factorization. This allows for achieving the same memory consumption as a fully-structured solver while taking advantage of the simpler full-rank assembly operations; the handling of pivoting in the subsequent factorization may be harder though.

5.3.3 BLR fronts factorization

As explained in the previous section, we assume that a frontal matrix F has been assembled in full-rank form. We will also assume that it has been partitioned (either logically or actually) into blocks according to the clustering of the associated index set; $F_{i,j}$ denotes the block along block-row i and block-column j whereas $\tilde{F}_{i,j}$ denotes its low-rank form. We also assume that a frontal matrix is of size $p = p_{fs}+p_{nfs}$ blocks where p_{fs} and p_{nfs} are the number of blocks in the fully-summed and non fully-summed parts, respectively. Algorithm 5.1 describes the LDL^T full-rank factorization of a frontal matrix. The Factor+Solve step is essentially equivalent to the operation in Equation (2.16a) except that it takes advantage of the symmetry and uses a Bunch-Kaufman type of pivoting in order to preserve it; this operation is based on Level-2 BLAS operations and thus is not very efficient. Note that, if pivoting is not done or restricted to the diagonal block, this step can be split into distinct operations, one that factorizes the diagonal block and others that update the sub-diagonal blocks by means of triangular solve operations; these last can be done using Level-3 BLAS routines are, thus, are very efficient.

In order to take advantage of low-rank approximations, the blocks of the frontal matrix have to be compressed. This does not necessarily have to be done before the factorization

Algorithm 5.1 Frontal full-rank LDL^T (Right-looking) factorization.

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: **for** $j = k + 1$ **to** i **do**
6: Update: $F_{i,j} \leftarrow F_{i,j} - F_{i,k} F_{j,k}$
7: **end for**
8: **end for**
9: **end for**

starts but Compress operations can be interleaved with Factor, Solve and Update ones; depending on when the compress operations are done, different variants can be defined. In the following sections we describe some of them but for a complete taxonomy of all the BLR factorization variants we refer the reader to the PhD thesis of Mary [125].

5.3.3.1 The FSCU and UFSC variants

The objective of this variant, introduced by Amestoy et al. [J4] is to preserve the ability to perform pivoting as is done in Algorithm 5.1, which implies that the Factor and Solve operations have to be fused together; therefore, a block cannot be compressed until the panel it belongs to is reduced. The name FSCU results from the order in which operations are done, i.e., Factor+Solve, Compress and Update. This variant is illustrated in Algorithm 5.2. In this variant, the only operation to benefit from the low-rank approximation is the Update one, while the Factor+Solve is still done in full-rank. It must be noted that, the $\tilde{C}_{i,j}^{(k)}$ block is actually full-rank if both $F_{i,k}$ and $F_{j,k}$ are and low-rank otherwise. In the latter case it must be decompressed by means of an outer product operation in order to be summed to the $F_{i,j}$ block. This also holds for the other variants presented below but, as discussed by Mary [125], it is possible to conceive alternatives where the block being updated is already compressed and thus the outer product (and the associated cost) unnecessary.

The UFSC variant is the left-looking equivalent of the FSCU one and is presented in Algorithm 5.3. These two variants are mathematically equivalent but the left-looking one has a different data access pattern which makes it possible to use the LUAR technique described in Section 5.3.3.3 and allows for achieving better performance as shown in Section 5.5.

As explained above, the FSCU variant, as well as the UFSC one, are designed to be compatible with the Delayed pivoting technique described in Section 2.2.3.2. In these variants, pivoting is performed in the panel reduction, i.e., the Factor+Solve operation. A pivot can only be chosen within the diagonal BLR block using a threshold partial pivoting technique but its quality is checked against all the coefficients in the column. If no more satisfactory pivots can be found before the panel reduction is finished, the uneliminated pivots are merged to the following BLR panel, if any, or delayed to the parent front, as explained in Section 2.2.3.2. This is illustrated in Figure 5.10. A consequence of postponing rows and columns from one BLR panel to the next is that the defined frontal matrix blocking is not respected anymore, which may have an impact on the effectiveness of the low-rank approximations.

A second consequence of postponing pivots is that, within a front, BLR blocks become unaligned with respect to previously eliminated ones. This raises an issue when a

Algorithm 5.2 Frontal BLR LDL^T (Right-looking) factorization: standard FSCU 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**
  Factor+Solve: $F_{k:p,k} \leftarrow L_{k:p,k} D_{k,k} L_{k,k}^T$
  for $i = k+1$ **to** p **do**
    Compress: $F_{i,k} \approx \tilde{F}_{i,k} = X_{i,k} Y_{i,k}^T$
  end for
  for $i = k+1$ **to** p **do**
    for $j = k+1$ **to** i **do**
      Update $F_{i,j}$:
        Inner Product: $\tilde{C}_{i,j}^{(k)} \leftarrow X_{i,k}(Y_{i,k}^T D_{k,k} Y_{j,k}) X_{j,k}^T$
        Outer Product: $C_{i,j}^{(k)} \leftarrow \tilde{C}_{i,j}^{(k)}$
        $F_{i,j} \leftarrow F_{i,j} - C_{i,j}^{(k)}$
    end for
  end for
end for

Algorithm 5.3 Frontal BLR LDL^T (Left-looking) factorization: UFSC 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**
    for $j = 1$ **to** $\min(k-1, p_{fs})$ **do**
      Update: $F_{i,k}$:
        Inner Product: $\tilde{C}_{i,k}^{(j)} \leftarrow X_{i,j}(Y_{i,j}^T D_{j,j} Y_{k,j}) X_{k,j}^T$
        Outer Product: $C_{i,k}^{(j)} \leftarrow \tilde{C}_{i,k}^{(j)}$
        $F_{i,k} \leftarrow F_{i,k} - C_{i,k}^{(j)}$
    end for
  end for
  Factor+Solve: $F_{k:p,k} \leftarrow L_{k:p,k} D_{k,k} L_{k,k}^T$
  for $i = k+1$ **to** p **do**
    Compress: $F_{i,k} \approx \tilde{F}_{i,k} = X_{i,k} Y_{i,k}^T$
  end for
end for

LAPACK-style pivoting is used as in Equation (2.2) or Algorithm 2.2 where the swap operations also concern the submatrix left to the panel being factorized. Indeed, this implies that rows belonging to different compressed blocks may have to be swapped. Mary [125] proposes two different techniques for dealing with this issue.

5.3.3.2 The UFCS variant with restricted pivoting

For many real-life problems, a practically stable solution can be obtained even if pivoting is restricted to a smaller area of the current panel, such as the BLR diagonal block, or completely avoided. Without the need for pivoting the Factor and Solve operations need not be fused: the Factor operation only reduces the diagonal block (possibly with pivoting) and is followed by a sequence of Solve operations that concern the subdiagonal

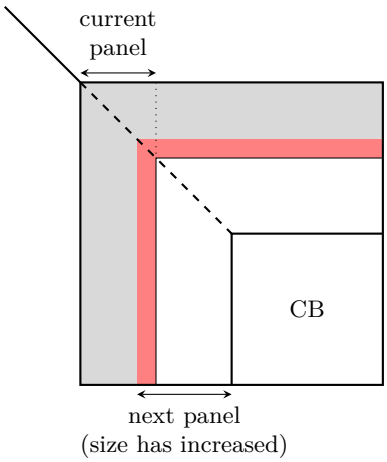


Figure 5.10: BLR factorization with numerical pivoting scheme. Postponed pivots after the elimination of the current panel are in red; they are merged with the next panel, whose size increases.

blocks. This has a twofold advantage. First the subdiagonal blocks can be compressed prior to the Solve operations, thereby reducing their cost; second, these can be done using Level-3 BLAS operations which ultimately results in better performance. Note that this last advantage also applies to the standard full-rank factorization.

The resulting variant, called UFCS, is illustrated in Algorithm 5.4. In Section 5.4 we will show that this variant achieves better asymptotic complexity than the ones in the previous section thanks to the reduced complexity of the Solve operations. In Section 5.5 we will show that this variant can also improve performance thanks to the higher efficiency of the Level-3 BLAS operations.

It must be noted that, in this variant, more operations (specifically, the Solve ones) are done in low-rank with respect to the FSCU and UFSC ones. This may lead to an additional loss of accuracy of the solver, as speculated by Amestoy et al. [J4] and Weisbecker [159]. Our experimental results in Section 5.5 show only a moderate effect on the scaled residual for a number of real-life problems, though.

Mary [125] proposes a variant, called UCFS, that can perform pivoting even if subdiagonal blocks are compressed before the solve operation. He presents experimental results on a number of real-life problems showing that this variant is comparable to the FSCU and UFSC ones in terms of stability while achieving the same complexity ah the UFCS one.

5.3.3.3 LUAR

One of the limiting factors of the previously presented variants is the outer product operation which is needed to decompress a low-rank update in order to sum it to a full-rank block. The LUAR (Low-rank Update Accumulation and Recompression) technique allows for reducing the cost of the outer product and improve its efficiency. By this technique, which can only be applied to left-looking variants, lines (4-9) of Algorithm 5.3 and lines (4-9) of Algorithm 5.4 are replaced by Algorithm 5.5

LUAR consists in accumulating the update matrices $\tilde{C}_{ik}^{(j)}$ together, as shown on line 5 of Algorithm 5.5:

$$\tilde{C}_{ik}^{(acc)} := \tilde{C}_{ik}^{(acc)} + \tilde{C}_{ik}^{(j)}$$

Algorithm 5.4 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**
    for $j = 1$ **to** $\min(k-1, p_{fs})$ **do**
      Update: $F_{i,k}$:
        Inner Product: $\tilde{C}_{i,k}^{(j)} \leftarrow X_{i,j}(Y_{i,j}^T D_{j,j} Y_{k,j}) X_{k,j}^T$
        Outer Product: $C_{i,k}^{(j)} \leftarrow \tilde{C}_{i,k}^{(j)}$
        $F_{i,k} \leftarrow F_{i,k} - C_{i,k}^{(j)}$
    end for
  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: $F_{i,k} \approx \tilde{F}_{i,k} = X_{i,k} Y_{i,k}^T$
  end for
  for $i = k+1$ **to** p **do**
    Solve: $\tilde{F}_{i,k} \leftarrow \tilde{F}_{i,k} L_{k,k}^{-T} D_{k,k}^{-1} = X_{i,k}(Y_{i,k}^T L_{k,k}^{-T} D_{k,k}^{-1})$
  end for
end for

Algorithm 5.5 LUAR-Update step.

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

Note that in the previous equation, the + sign denotes a low-rank sum operation. Specifically, if we note $A = C_{ik}^{(acc)}$ and $B = C_{ik}^{(j)}$, then

$$\tilde{B} = \tilde{C}_{ik}^{(j)} = X_{ij}(Y_{ij}^T D_{jj} Y_{jk}) X_{jk}^T = X_B C_B Y_B^T$$

with $X_B = X_{ij}$, $C_B = Y_{ij}^T D_{jj} Y_{kj}$, and $Y_B = X_{kj}$. Similarly, $\tilde{A} = \tilde{C}_{ik}^{(acc)} = X_A C_A Y_A^T$. Then the low-rank sum operation is defined by:

$$\tilde{A} + \tilde{B} = X_A C_A Y_A^T + X_B C_B Y_B^T = \begin{pmatrix} X_A & X_B \end{pmatrix} \begin{pmatrix} C_A & \\ & C_B \end{pmatrix} \begin{pmatrix} Y_A & Y_B \end{pmatrix}^T = X_S C_S Y_S^T = \tilde{S}$$

where $\tilde{S}$ is a low-rank approximant of $S = A + B$.

Accumulated updates can, optionally, be recompressed in order to reduce the complexity of the outer product. Many different strategies are possible to recompress the accumulated updates as discussed by Mary [125] and Anton et al. [21]; in the rest of this

document and, especially, in the experimental results of Sections 5.4 and 5.5 we will assume that this is achieved by recompressing only the middle block C_S (see Figure 5.11) as this is the strategy that captures most of the recompression potential with an acceptable overhead:

$$\begin{aligned}\tilde{A} \; + \; \tilde{B} =& X_S C_S Y_S^T \approx X_S X_{C_S} Y_{C_S}^T Y_S^T = \hat{X}_S \hat{Y}_S^T, \\ \tilde{C}_S =& X_{C_S} Y_{C_S}^T, \quad \hat{X}_S = X_S X_{C_S}, \quad \hat{Y}_S = Y_S Y_{C_S}.\end{aligned}$$

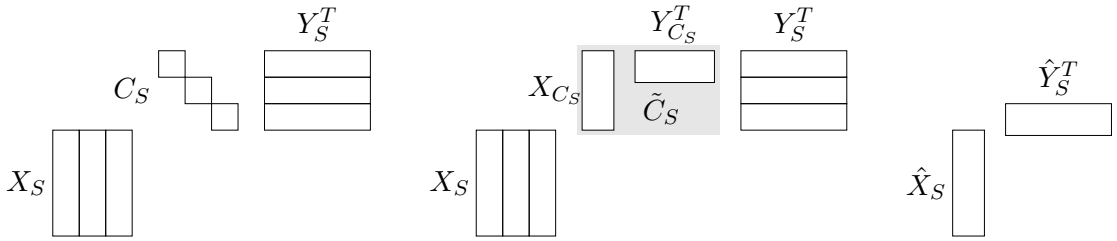


Figure 5.11: Low-rank Updates Accumulation.

LUAR has a twofold advantage. First, it improves the granularity of the outer product operation, especially in the case where the recompression is not done because, in essence, multiple outer products are done with a single kernel; this will be discussed and illustrated by means of experimental results in Section 5.5. Second, when recompression is done, it reduces the complexity of the outer product operation which yield a reduction of the overall complexity of the multifrontal factorization; this will be analyzed theoretically in Section 5.4 and assessed experimentally.

5.3.3.4 Compression of the contribution block

It must be noted that upon completion of Algorithms 5.2, 5.3 and 5.4, the contribution block is left uncompressed, i.e., in full-rank form. This, however, does not have to be the case because the contribution block can be compressed exactly the same as the fully-summed part of a frontal matrix. This can easily be achieved by appending to Algorithms 5.2, 5.3 and 5.4 a doubly-nested loop that sweeps over all the blocks of the BLR contribution block and compresses them one by one. As explained in Section 5.3.2, compressing the contribution block does not necessarily imply that the assembly operations are done in low-rank because it can be decompressed (either block-by-block or all at once) prior to being assembled into the parent front.

Compressing the contribution block has a number of consequences:

1. Unless in a fully-structured factorization variant, the overall number of floating-point operations is not decreased but, rather, increased. This is because compression clearly has a cost but, once compressed, the contribution block is not used in other operations apart from assemblies. If these are done in full-rank, decompression implies an additional cost; if, instead, they are done in low-rank (as in a fully-structured solver) the overhead may be even higher due to the padding of BLR blocks (see Section 5.3.2).

2. Memory consumption is reduced. Although a contribution block is a temporary data that is discarded after being assembled into the parent front, its storage can consume

a large amount of memory. Actually, as explained in Chapter 4, the memory used to store all the contribution blocks – the so called *active memory* (see Chapter 4 for a more accurate definition) – can represent a large fraction of the total memory consumption in a sequential execution, can become dominant in a parallel execution or can even be the totality of the memory consumption in an Out-Of-Core [2] execution. Therefore, compressing the contribution block is the only effective way of reducing the memory footprint of a multifrontal solver.

3. Communications are reduced in a distributed memory, parallel setting. Compressing the fully-summed part reduces the total volume of communications associated with the factorization of frontal matrices. Assembly operations also involve communications if a front and its parent are not mapped on the same process. Mary [125] showed that these communications can become dominant and harm the scalability. Compressing the contribution block allows for reducing these communications.

4. A for the execution time, this is clearly increased in a serial or shared-memory parallel execution because of point 1) above but in a distributed-memory parallel setting it can be decreased because of a lower volume of communications as explained in the previous point.

In the remainder of this work we will assume that the contribution block is not compressed.

5.3.4 Experimental results on applications

In this section, we evaluate the effectiveness of the BLR approximations on two real-life applications.

The use of the BLR format through the techniques presented in the previous section was integrated in the MUMPS [13] parallel, multifrontal solver. Note, however, that low-rank approximations are not applied to the smallest frontal matrices where the compression overhead exceeds the actual benefits brought by the LR format. In the BLR-based version, distributed memory parallelism is achieved as is the standard, full-rank (FR), MUMPS solver with the exception that messages exchanged during the factorization of each front are reduced thanks to the low-rank data compression. Note that in the code used for the experiments reported in this section, the contribution block is not compressed and, therefore, the volume of messages related to the assembly of fronts is not reduced. For a thorough discussion of this issue, please refer to Mary's PhD thesis [125]. As for shared memory parallelism, the FR MUMPS code heavily relies on multithreaded BLAS routines; this approach is not suited to the BLR factorization due to the small granularity of the compressed blocks. Therefore, multithreading is achieved by parallelizing the loops on lines 4 and 7 of the FSCU factorization in Algorithm 5.2; more details on this will be provided in Section 5.5.

The BLR factorization variant used for these experiments is the FSCU; the details of a full-featured, distributed memory parallel implementation of the other variants can be found in the PhD thesis of Mary [125].

5.3.4.1 3D frequency-domain Full Waveform Inversion

This section discusses the use of a BLR multifrontal solver in 3D seismic modeling by means of frequency-domain Full Waveform Inversion (FWI) [151]; this evaluation was

conducted in collaboration with the SEISCOPE consortium[1]. Although this method is now routinely used in the oil industry as part of the seismic imaging work-flow, 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. In our case, both seismic modeling and FWI are performed in the frequency domain [156]. 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 [122]. 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. Processing a large number of right-hand sides leads us naturally toward direct solvers because the computation of the solutions by forward/backward substitutions is quite efficient once a LU factorization of the impedance matrix has been performed.

In the following analysis, the subsurface target (the Valhall oil field located in the North Sea in a shallow water environment) and the dataset are the same as in the work by Operto et al. [130]. For the details of the 3D visco-acoustic equation, instead, we refer the reader to the paper by Amestoy et al. [J3] from where this section was extracted. This leads to the three matrices $5Hz$, $7Hz$ and $10Hz$ reported in Appendix A.1, Table A.2. For each of these matrices 4604 right-hand sides were used. The following results were measured on the `licallo` machine described in Appendix A.2. For each node we used two MPI processes, each using ten cores by means of multithreading. For the $5Hz$, $7Hz$ and $10Hz$ matrices we used 12, 16 and 32 nodes for a total of 240, 320 and 680 cores, respectively. Computations are done is single precision, complex arithmetic. A resume of the experimental setting is provided in Table 5.2.

Freq. (Hz)	h(m)	Grid dimensions	n_{pml}	$\#u$	$\#n$	$\#MPI$	$\#th$	$\#c$	$\#rhs$
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 5.2: North Sea case study. Problem size and computational resources. 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.

First, we show the nature of the errors introduced in the wavefield solutions by the BLR approximation. Figure 5.12(a) shows a 5Hz monochromatic common-receiver gather computed with the FR solver in the FWI models obtained after the inversions. Figure 5.12(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 in Figure 5.12(e-g). Similar figures can be drawn for the 7Hz and 10Hz problems.

[1] https://seiscope2.osug.fr

Table 5.3 reports 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.

Three conclusions can be drawn for this case study: for the used 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 δ_{BLR}/δ_{FR} decreases with frequency as shown by the results in Table 5.3. From these results and from an analysis of the overall FWI ones [J3], it can concluded that a BLR threshold of 10^{-3} produces satisfactory results.

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 5.3: 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.

Table 5.4 shows the details of the computational savings provided by te BLR approximation techniques. 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 the table) 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 5.4). The time reduction achieved by the BLR solver tends to increase with the frequency. The elapsed time to perform the FWI is provided for each grid in Table 5.4. 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 conclude from this analysis that, for this case study, the BLR solver with $\varepsilon = 10^{-4}$ provides the best FWI time. 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.

It must be noted that the implementation used for these tests does not take advantage of the BLR format in the solve phase whose computational cost is relatively high due to the high number of right-hand sides. Mary [125] provides indications of the potential time reductions that can be achieved with a BLR-based solve operation.

Finally, in Figure 5.13, we provide a strong scalability analysis of both the FR and BLR factorizations with the FSCU variant. These experiments were run on the `eos` supercomputer (see Appendix A.2) which has similar characteristics than the `licallo` machine. 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 and the ratio between FR and BLR factorization times is relatively stable, the FR strong scalability is clearly better than the BLR one. This can be ascribed to a number of issues. For example, this can be due to the fact that the relative weight of communications is higher in BLR than in FR; this issue is particularly important in these tests because the

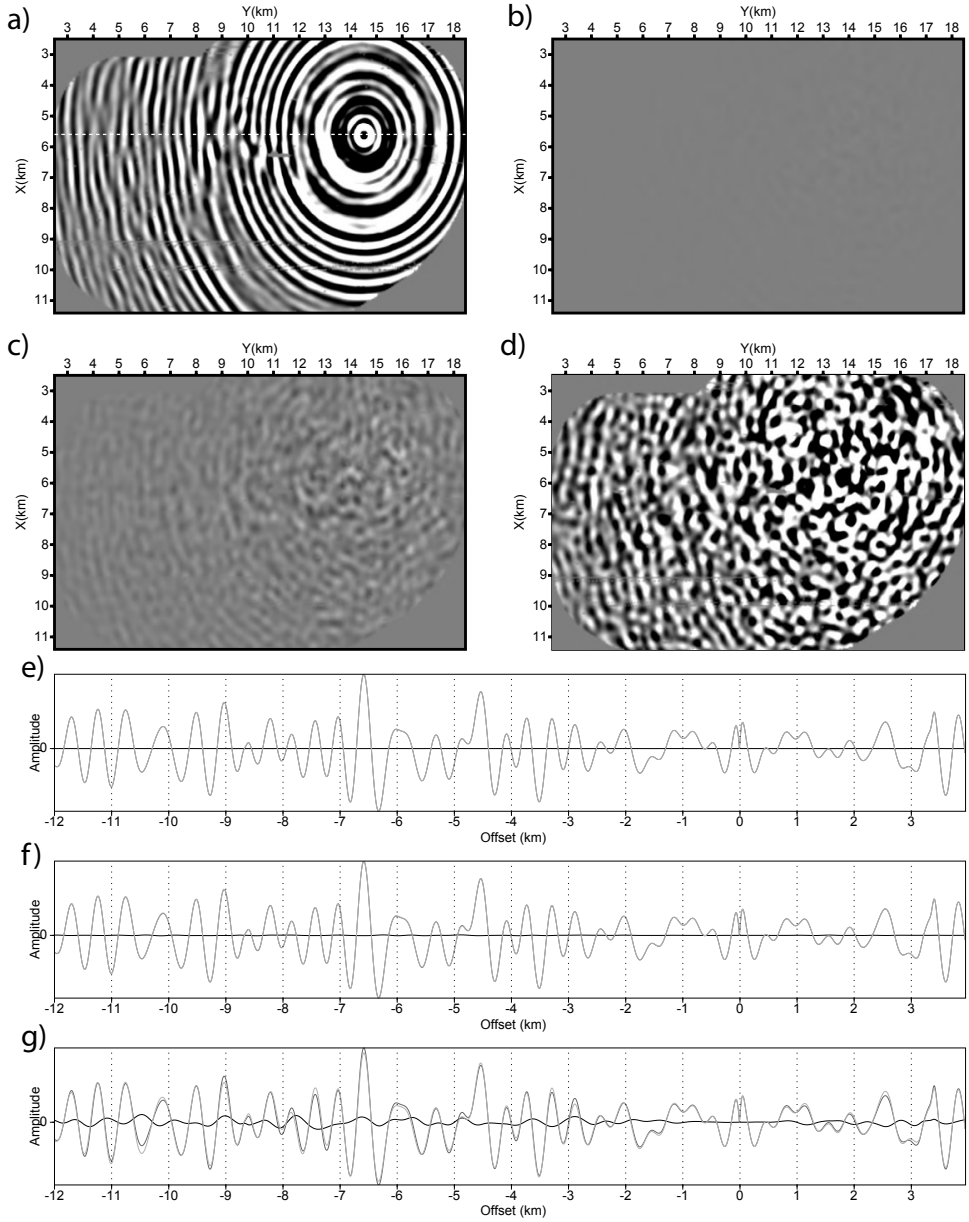
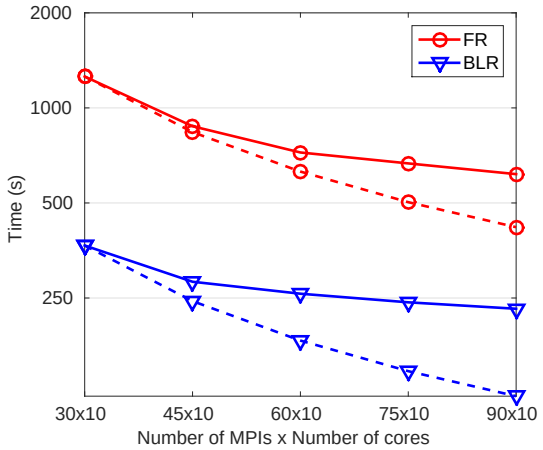


Figure 5.12: North Sea case study. BLR modeling errors. (a) 5Hz receiver gather (real part) computed with the FR solver. (b) Difference between the receiver gathers computed with the BLR ($\varepsilon = 10^{-5}$) and the FR (a) solvers. (c) Same as (b) for $\varepsilon = 10^{-4}$. (d) Same as (b) for $\varepsilon = 10^{-3}$. 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.

F(Hz)/h(m)	ε	$F_{LU}(\times 10^{12})$	$T_{LU}(s)$	T_{FWI} (hr)
5Hz/70m (240 cores)	FR	66 (1.0)	78 (1.0)	14.0
	10^{-5}	17 (3.8)	48 (1.6)	12.9
	10^{-4}	12 (5.3)	46 (1.7)	12.7
	10^{-3}	8 (8.0)	41 (1.9)	14.0
7Hz/50m (320 cores)	FR	410 (1.0)	322 (1.0)	14.5
	10^{-5}	90 (4.5)	157 (2.1)	12.0
	10^{-4}	63 (6.5)	136 (2.4)	9.1
	10^{-3}	38 (10.7)	121 (2.7)	9.0
10Hz/35m (680 cores)	FR	2600 (1.0)	1153 (1.0)	20.7
	10^{-5}	520 (4.9)	503 (2.3)	14.5
	10^{-4}	340 (7.5)	442 (2.6)	14.2
	10^{-3}	190 (13.3)	424 (2.7)	14.8

Table 5.4: 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_{FWI}: the overall FWI time.

CBs are not compressed and, thus, the volume of messages related to the assembly of front is not reduced. Moreover, because the BLR compression rate cannot be estimated prior to the actual factorization, the workload mapping is based of full-rank estimates which can be very inaccurate and lead to a load imbalance. Mary [125] provides a detailed analysis of these issues (and more) and proposes methods to overcome them.



(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 5.13: Strong scalability of the FR and BLR factorizations (10Hz matrix).

5.3.4.2 3D Controlled-source Electromagnetic inversion

Our second application is marine Controlled-Source ElectroMagnetic (CSEM) surveying which is a widely used method for detecting hydrocarbon reservoirs and other resistive structures embedded in conductive formations [51, 70, 106]; the analysis reported in this section was achieved in collaboration with the EMGS company[2]. 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. 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. Note that, because may sources are used, the M matrix has to be solved against multiple right-hand sides which, as in the case of the application discussed in the previous section, suggests that iterative methods may behave well.

In all simulations the system matrix was generated using the finite-difference modeling code presented by Jaysaval et al. [104]. The simulations were carried out on either the `eos` supercomputer or the `farad` machine (see Appendix A.2).

Two different models were used for this analysis. The first, referred to as H-model is a simple, synthetic model, whereas the second, referred to as the S-model, is the SEAM (SEG Advanced Modeling Program) Phase I salt resistivity model representative of the geology in the Gulf of Mexico. It is a complex 3D earth model designed by the hydrocarbon exploration community and widely used to test 3D modeling and inversion algorithms. This lead to the generation of four matrices, namely the H3, H17 (for the H-model) and S3, S21 (for the S-model) reported in Table A.2. Computations are done in double precision, complex arithmetic.

For the details of the equations, the discretization approach and the models used in this study, we refer the reader to the paper by Shantsev et al. [J20], from where this section was extracted.

First, we 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 :

$$\xi_{m,i,j,k} = \sqrt{\frac{|x^\varepsilon_{m,i,j,k} - x_{m,i,j,k}|^2}{\left(|x^\varepsilon_{m,i,j,k}|^2 + |x_{m,i,j,k}|^2\right)/2 + \eta^2}}, \tag{5.3}$$

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 5.14 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

[2] http://www.emgs.com/

maps, the relative error in the air is orders of magnitude larger than in the water or formation. 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 5.14 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.

Next, we turn our attention to the gains provided by the BLR multifrontal factorization to the CSEM inversion. We consider the inversion of synthetic CSEM data over the S-model. We assume that $n_s = 3300$ horizontal electric dipole (HED) sources are used and $n_r = 121$ receivers are used to record simulated responses and the model is discretized with grid $181 \times 160 \times 237$ 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 and (2) a Gauss-Newton inversion scheme. 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 et al. [92] 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 5.5, 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 $\times$ 10 threads setting and ParMETIS [105] for ordering. For comparison, time estimates for an iterative solver are also presented. This iterative solver was developed following the ideas of Mulder [129]: a complex biconjugate-gradient-type solver, BICGStab(2) [91, 155] is used in combination with a multigrid preconditioner and a block Gauss-Seidel type smoother.

The first two rows of Table 5.5 show the result reported in Shantsev et al. [J20], 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 5.5 show the new results, obtained with the development

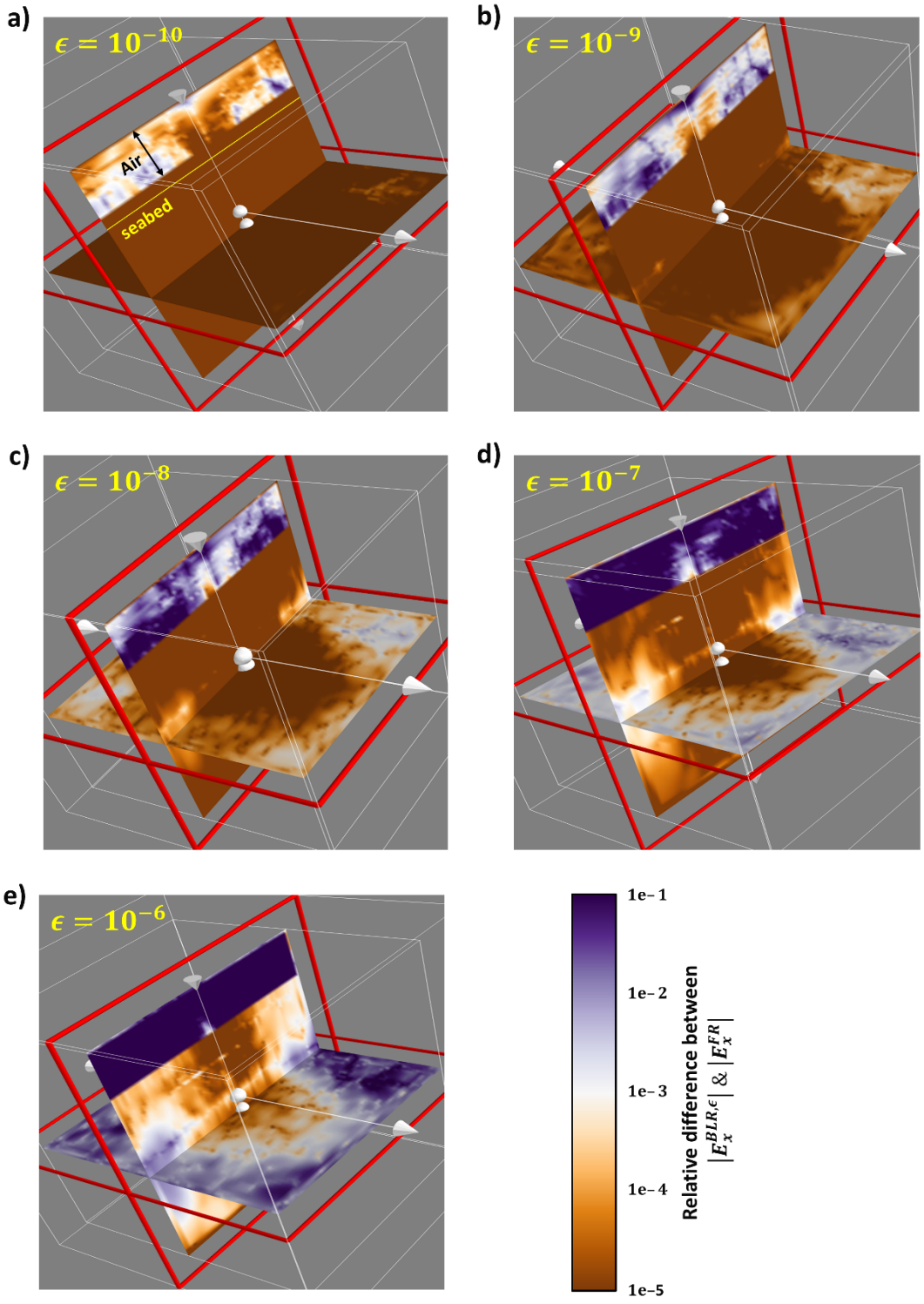


Figure 5.14: Relative difference between the BLR solution x^{ε} for different low-rank thresholds ε, and the FR solution x for a linear system corresponding to matrix H3. For $\varepsilon = 10^{-7}$, the solution accuracy is acceptable everywhere except in the air layer at the top. The results are for the x-component of the electric field. The air and PML layers are not to scale.

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 5.5: Run times for the FR and BLR direct solvers on the `eos` supercomputer and for a multigrid preconditioned iterative solver to perform CSEM simulations. On top, the old results presented in Shantsev et al. [J20] (based on MUMPS 5.0 and using the FSCU BLR variant); at the bottom the new ones (MUMPS development 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.

version of MUMPS. Both the factorization and solution phases of the FR solver have been accelerated although the solve solve phase is still carried in full-rank; 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.

5.4 Complexity of the BLR multifrontal factorization

In this section we develop a theoretical analysis of the complexity (both in terms of floating-point operations and size of factors) of dense and sparse matrices based on the BLR format. This analysis is based on the work by Bebendorf et al. [35] which assumes the use of the strong admissibility condition. Hackbusch et al. [93], however, show 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 our work.

In the following, when referring to the BLR case, we simplify the notation $\mathfrak{S}(\mathcal{I} \times \mathcal{I})$ to $\mathfrak{S}(\mathcal{I})$, as in most cases we do not need a different partitioning of the row and column variables.

5.4.1 FE discretization of elliptic PDEs

We consider a Partial Differential Equation of the form:

$$\begin{aligned} Lu &= f \quad \text{in } \Omega \subset \mathbb{R}^d, \Omega \text{ convex }, d \geq 2 \\ u &= g \quad \text{on } \partial\Omega \end{aligned} \tag{5.4}$$

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 $c_{ij} \in L^\infty(\Omega)$. Furthermore, $\boldsymbol{c_1}(x), \boldsymbol{c_2}(x) \in \mathbb{R}^d$ and $c_3(x) \in \mathbb{R}$.

We consider the resolution of problem 5.4 by the Finite Element (FE) method. Let $\mathcal{D} = H_0^1(\Omega)$ be the domain of definition of operator L. We consider a FE discretization, with step size h, that defines the associated approximation of $\mathcal{D}$, the space $\mathcal{D}_h$. Let $n = N^d = \dim \mathcal{D}_h$ be its dimension and $\{\varphi_i\}_{i\in\mathcal{I}}$ the basis functions, with $\mathcal{I} = [1, n]$ the index set. Similarly as in the work of Bebendorf et al. [35], 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:

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

We note J the bijection defined by:

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

To compute an approximated solution of Equation 5.4, we solve the discretized problem $Ax = b$ where A is the stiffness matrix defined by $A = J^*LJ$. We assume that this linear system of equations is solved 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

$$S = A_{\Psi\Psi} - A_{\Psi\Phi}A_{\Phi\Phi}^{-1}A_{\Phi\Psi} \tag{5.5}$$

for some $\Phi, \Psi \subset \mathcal{I}$ such that $\Phi \cup \Psi = \mathcal{I}$ is called a Schur complement of A. One of the main results of Bebendorf [34], 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 $\tilde{A}^{-1}$, approximant of the inverse stiffness matrix A^{-1}. The following result from FE theory will be used ([35], Subsection 5.2): the discretization of the inverse of the operator is approximated by the inverse of the discretized operator, i.e.,

$$\|A^{-1} - M^{-1}BM^{-1}\|_2 \leq O(\varepsilon_h) \tag{5.6}$$

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 $\tilde{M}^{-1}$ and $\tilde{B}$, approximants of the inverse mass matrix M^{-1} and of the B matrix, we have ([35], Subsection 5.3):

$$\begin{aligned} M^{-1}BM^{-1} - \tilde{M}^{-1}\tilde{B}\tilde{M}^{-1} &= (M^{-1} - \tilde{M}^{-1})BM^{-1} \\ &\quad + \tilde{M}^{-1}(B - \tilde{B})M^{-1} + \tilde{M}^{-1}\tilde{B}(M^{-1} - \tilde{M}^{-1}) \end{aligned} \tag{5.7}$$

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

5.4.2 From Hierarchical to BLR bounds

The existence of $\mathcal{H}$-matrix approximants of the Schur complements of A has been shown by Bebendorf et al. [35] and Bebendorf [34]. 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 the work by Hackbusch & Bebendorf [33, 34, 35] for the details of the proof for hierarchical matrices.

5.4.2.1 $\mathcal{H}$-admissibility and properties

We assume that a $\mathcal{H}$-matrix is built as described in Section 5.2.4.2 and, thus, its block-partition is $\mathcal{H}$-admissible as in Definition 5.10. We note $\mathcal{H}(\mathfrak{S}(\mathcal{I}\times\mathcal{I}), r)$ the set of hierarchical matrices such that r is the maximal rank of the blocks defined by the admissible partition $\mathfrak{S}(\mathcal{I}\times\mathcal{I})$. We define the so-called *sparsity constant*:

$$c_{sp} = \max\left(\max_i \#\{I_j; I_i\times I_j \in \mathfrak{S}(\mathcal{I}\times\mathcal{I})\}, \max_j \#\{I_i; I_i\times I_j \in \mathfrak{S}(\mathcal{I}\times\mathcal{I})\}\right) \tag{5.8}$$

where $\#E$ denotes the cardinality of a set E (we will use this notation from now on). 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.

Hackbusch & Bebendorf [33, 34, 35] prove that the Schur complements of A possess $\mathcal{H}$-approximants using 5.6.

They first establish that B and M^{-1} possess $\mathcal{H}$-approximants ([35], Theorems 3.4 and 4.3). More precisely, they can be approximated with accuracy ε by $\mathcal{H}$-matrices $\tilde{B}$ and $\tilde{M}^{-1}$ such that

$$\tilde{B} \in \mathcal{H}(\mathfrak{S}(\mathcal{I}\times\mathcal{I}), r_G) \tag{5.9}$$

$$\tilde{M}^{-1} \in \mathcal{H}(\mathfrak{S}(\mathcal{I}\times\mathcal{I}), |\log\varepsilon|^d) \tag{5.10}$$

where $\mathfrak{S}(\mathcal{I}\times\mathcal{I})$ is an $\mathcal{H}$-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 [33, 35].

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

Theorem 5.3 $\mathcal{H}$-matrix product, Theorem 2.20 in Börm et al. [37].— *Let H_1 and H_2 be two hierarchical matrices of order n, such that $H_1 \in \mathcal{H}(\mathfrak{S}(\mathcal{I}\times\mathcal{I}), r_1)$ and $H_2 \in \mathcal{H}(\mathfrak{S}(\mathcal{I}\times\mathcal{I}), r_2)$. Then, their product is also a hierarchical matrix and it holds:*

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

In Theorem 5.3, c_{sp} is the sparsity constant, defined by 5.8.

Then, using the fact that $r_G > |\log\varepsilon|^d$ [35], and applying 5.6, 5.7, and Theorem 5.3, it is established ([35], Theorem 5.4) that

$$\tilde{A}^{-1} \in \mathcal{H}(\mathfrak{S}(\mathcal{I}\times\mathcal{I}), r_{\mathcal{H}}), \quad \text{with } r_{\mathcal{H}} = c_{sp}^2 r_G \log^2 n \tag{5.11}$$

Furthermore, if an approximant $\tilde{A}^{-1}$ exists, then for any $\Phi \subset \mathcal{I}$, an approximant of $A_{\Phi\Phi}^{-1}$ must also exist, since $A_{\Phi\Phi}$ is simply the restriction of A to the subdomain X_Φ [34].

Thus, using 5.5, in combination to the fact that the stiffness matrix A can also be approximated by $\tilde{A} \in \mathcal{H}(\mathfrak{S}(\mathcal{I}\times\mathcal{I}), O(1))$, the existence of $\tilde{S}$, $\mathcal{H}$-approximant of any Schur

complement S of A, is guaranteed by 5.11 and it is shown [34] that the maximal rank of the blocks of $\tilde{S}$ is $r_{\mathcal{H}}$, i.e.

$$\tilde{S} \in \mathcal{H}(\mathfrak{S}(\mathcal{I} \times \mathcal{I}), r_{\mathcal{H}}) \tag{5.12}$$

Finally, it can be shown that the complexity of factorizing an $\mathcal{H}$-matrix of order m and of maximal rank r is [35, 94]:

$$\mathcal{C}(m) = O(c_{sp}^2 r^2 m \log^2 m) \tag{5.13}$$

Equation 5.13 relies on the assumption that the factorization is fully-structured, i.e. the compressed form $\tilde{A}$ of A is available at no cost.

To conclude, in the $\mathcal{H}$ case, applying 5.13 to the (dense) factorization of $\tilde{S}$ leads to a cost which is almost linear when $r = O(1)$ and almost in $O(mN^2)$ when $r = O(N)$. As will be explained in Section 5.4.4, both cases lead to near-linear complexity of the multifrontal (sparse) factorization [163].

5.4.2.2 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 because it is equivalent to bounding *all the ranks* k_{ij}^{ε} 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 $\text{dist}(X_\sigma, X_\tau) = 0$ (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 5.8) is not bounded, as it is equal to $p = m/b$. Thus, 5.13 leads to a factorization complexity bound in $O((m/b)^2 b^2 m \log^2 m) = O(m^3 \log^2 m)$, even worse than the full-rank factorization.

5.4.2.3 BLR-admissibility and properties

To compute a meaningful complexity bound for BLR, we divide the BLR blocks into two groups: the blocks who satisfy the block-admissibility condition (whose rank r can be bounded by a meaningful bound), and those who 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 $\mathfrak{S}(\mathcal{I})$. This leads us to introduce the notion of BLR-admissibility of a partition $\mathfrak{S}(\mathcal{I})$, and we establish for such a partition a bound on the maximal rank of the admissible blocks.

In the following, we note $\mathcal{B}_A$ the set of admissible blocks. We also define

$$N_{na} = \max_{\sigma \in \mathfrak{S}(\mathcal{I})} \#\{\tau \in \mathfrak{S}(\mathcal{I}), \sigma \times \tau \notin \mathcal{B}_A\} \tag{5.14}$$

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 $\mathcal{H}$-admissibility of a partition to the BLR uniform blocking. We propose the following BLR-admissibility condition:

$$\mathfrak{S}(\mathcal{I}) \text{ is admissible } \Leftrightarrow N_{na} \leq q \tag{Adm_{BLR}}$$

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 Adm_b^{lrs} 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 5.15, where we have assumed that Adm_b^{lrs} is used for simplicity. In Figure 5.15 (*left*), the vertical subdomain (in gray) is at distance zero of $O(m/b)$ blocks and thus N_{na} is not constant. In Figure 5.15 (*right*), 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 Adm_b^s is used, the same reasoning applies, as in Figure 5.15 (*right*), N_{na} only depends on η and d, which are both constant.

We note $\mathcal{BLR}(\mathfrak{S}(\mathcal{I}), r, q)$ the set of BLR matrices such that r is the maximal rank of the admissible blocks defined by the BLR-admissible partition $\mathfrak{S}(\mathcal{I})$.

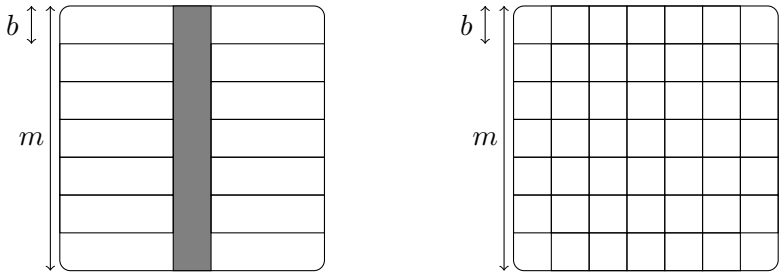


Figure 5.15: Illustration of the BLR-admissibility condition. On the left an example of a non-BLR-admissible partition. Here, the gray subdomain has $N_{na} = O(m/b) \neq O(1)$ neighbors. On the right example of a BLR-admissible partition when $q \geq N_{na} = 9 = O(1)$, the maximal number of neighbors of any block.

The following lemma, whose proof is provided by Amestoy et al. [J6], proves that a BLR-admissible partitioning exist and can be obtained by refining a $\mathcal{H}$-admissible one.

Lemma 5.1— *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 $N_{na}^{(\mathcal{H})} = O(1)$.*

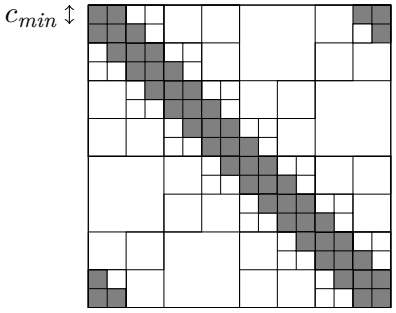
In view of this lemma, we assume in the following that the partition $\mathfrak{S}(\mathcal{I})$ is defined by a geometrically balanced cluster tree and is thus admissible for $q = N_{na} = O(1)$.

The next step is to find BLR approximants $\tilde{B}$ and $\tilde{M}^{-1}$ of B and M^{-1}, respectively, that verify:

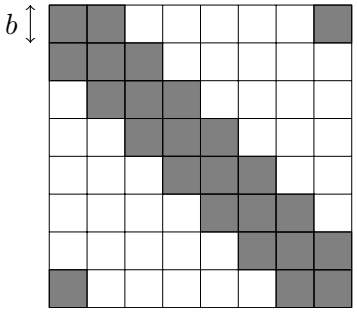
$$\tilde{B} \in \mathcal{BLR}(\mathfrak{S}(\mathcal{I}), r_G, N_{na}) \tag{5.15}$$

$$\tilde{M}^{-1} \in \mathcal{BLR}(\mathfrak{S}(\mathcal{I}), 0, N_{na}) \tag{5.16}$$

The construction of $\tilde{B}$ is the same for a BLR or an $\mathcal{H}$-partitioning, and we can thus rely on the work of Hackbusch and Bebendorf [35]. The main idea behind this construction



(a) Example of $\mathcal{H}$-partitioning, with $N_{na}^{(\mathcal{H})} = 4 \leq c_{sp} = 6 = O(1)$. 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 5.16(a). $c_{sp} = O(m/b)$ is not bounded anymore but $N_{na}^{(BLR)} = 3 = O(1)$ remains constant.

Figure 5.16: Illustration of Lemma 5.1 (proof of the boundedness of N_{na}).

is to exploit the decay property of Green functions. As shown by Hackbusch and Bebendorf ([35], 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 $\widetilde{B} \in \mathcal{BLR}(\mathfrak{S}(\mathcal{I}), r_G, N_{na})$ as follows

$$\forall \sigma \times \tau \in \mathfrak{S}(\mathcal{I}),\ \widetilde{B}_{\sigma\times\tau} = \begin{cases} B^{\varepsilon}_{\sigma\times\tau} & \text{if } \sigma \times \tau \in \mathcal{B}_A \\ B_{\sigma\times\tau} & \text{otherwise} \end{cases} \tag{5.17}$$

The construction of $\widetilde{M}^{-1}$ is also very similar to the one in Hackbusch & Bebendorf [35]. 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 $\sigma \times \tau \in \mathfrak{S}(\mathcal{I})^2$,

$$\|M^{-1}_{\sigma\times\tau}\| \leq O(\sqrt{\#\sigma\#\tau}\ c^{\sqrt[2d]{\#\sigma\#\tau}\ \mathrm{dist}(X_\sigma, X_\tau)})\|M^{-1}\|$$

where $c < 1$ ([35], 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 $\sigma \times \tau \in \mathcal{B}_A$.

Therefore, we construct $\widetilde{M}^{-1} \in \mathcal{BLR}(\mathfrak{S}(\mathcal{I}), 0, N_{na})$ as follows:

$$\forall \sigma \times \tau \in \mathfrak{S}(\mathcal{I}),\ \widetilde{M}^{-1}_{\sigma\times\tau} = \begin{cases} 0 & \text{if } \sigma \times \tau \in \mathcal{B}_A \\ M^{-1}_{\sigma\times\tau} & \text{otherwise} \end{cases} \tag{5.18}$$

Note that Equations 5.15 and 5.16 are the BLR equivalents of Equations 5.9 and 5.10, respectively. It now remains to derive a BLR arithmetic property similar to Theorem 5.3 which is given by Theorem 5.4 (for its proof we refer the reader to the appendix of the work by Amestoy et al. [J6]).

Theorem 5.4 BLR matrix product.— *If $A \in \mathcal{BLR}(\mathfrak{S}(\mathcal{I}), r_A, q_A)$ and $B \in \mathcal{BLR}(\mathfrak{S}(\mathcal{I}), r_B, q_B)$ are BLR matrices then their product $P = AB$ is a BLR matrix such that*

$$P \in \mathcal{BLR}(\mathfrak{S}(\mathcal{I}), r_P, q_P)$$

with $r_P = c_{sp} \min(r_A, r_B) + q_A r_B + q_B r_A$ and $q_P = q_A q_B$.

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 5.6), applying Theorem 5.4 on Equations 5.15 and 5.16 leads to

$$\tilde{A}^{-1} \in \mathcal{BLR}(\mathfrak{S}(\mathcal{I}), N_{na}^2 r_G, N_{na}^3) \tag{5.19}$$

and from this approximant of A^{-1} we can derive an approximant of $A_{\Phi\Phi}^{-1}$ for any $\Phi \subset \mathcal{I}$. For any $\Phi, \Psi \subset \mathcal{I}$, we also have $A_{\Phi\Psi} \in \mathcal{BLR}(\mathfrak{S}(\mathcal{I}), 0, N_{na})$, and therefore applying Theorem 5.4 on 5.5 and 5.19 implies in turn:

$$\tilde{S} \in \mathcal{BLR}(\mathfrak{S}(\mathcal{I}), N_{na}^4 r_G, N_{na}^5) \tag{5.20}$$

Therefore, there are at most $N_{na}^5 = O(1)$ 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 [33], the constant N_{na}^4 can be very large. However, our bound is extremely pessimistic. In Section 5.4.5, 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 $O(r_G)$, 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.

5.4.3 Complexity of the dense BLR factorization

In this section we compute the complexity of the BLR-based factorization of a dense matrix. As long as a bound on the ranks holds, similar to the one we have established in Section 5.4.2, the complexity computations reported in this section hold, and thus, the following results may be applicable to a broader context than the resolution of discretized PDEs.

Here we derive bounds for the complexity of the factorization of a dense matrix by means of the FSCU or UFSC methods (which are equivalent) described in Algorithms 5.2 and 5.3. Although we compute the complexity for the LDL^T factorization, note that the complexity of the BLR factorization is the same in LU or LDL^T, up to a constant. Note also that, in Algorithms 5.2 and 5.3, operations on non-admissible blocks are omitted for the sake of simplicity (but are taken into account in the complexity computations).

We will extend the computation of the complexity to the sparse multifrontal case in Section 5.4.4.

First, we compute the complexity of factorizing a dense 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 5.6 (third column) and are derived from Section 5.1.2. 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 low-rank blocks (LR-LR), two full-rank blocks (FR-FR) or a low-rank block and a full-rank one (LR-FR). We note b the block size and $p = m/b$ the number of blocks per row and/or column. We assume here that the cost of compressing an admissible block is $O(b^2 r)$ which is the case if a truncated rank-revealing QR factorization is used.

We can then use 5.20 to compute the cost of the factorization: the boundedness of $N_{na}^5 = O(1)$ ensures that only a constant number of blocks on each line are full-rank.

step	type	cost	number	$\mathcal{C}_{step}(b,p)$	$\mathcal{C}_{step}(m,x)$
Factor	FR	$O(b^3)$	$O(p)$	$O(pb^3)$	$O(m^{1+2x})$
Solve	FR-FR	$O(b^3)$	$O(p^2)$	$O(p^2b^3)$	$O(m^{2+x})$
Compress	LR	$O(b^2r)$	$O(p^2)$	$O(p^2b^2r)$	$O(m^2r)$
Inner Prod.	LR-LR	$O(br^2)$	$O(p^3)$	$O(p^3br^2)$	$O(m^{3-2x}r^2)$
	LR-FR	$O(b^2r)$	$O(p^2)$	$O(p^2b^2r)$	$O(m^2r)$
	FR-FR	$O(b^3)$	$O(p)$	$O(pb^3)$	$O(m^{1+2x})$
Outer Prod.	LR	$O(b^2r)$	$O(p^3)$	$O(p^3b^2r)$	$O(m^{3-x}r)$

Table 5.6: Main operations for the BLR (FSCU and UFSC variants) 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. $\mathcal{C}_{step}(b,p)$: obtained by multiplying the *cost* and *number* columns (Equation 5.21). $\mathcal{C}_{step}(m,x)$: obtained with the assumption that $b = O(m^x)$ (and thus $p = O(m^{1-x})$), for some $x \in [0,1]$.

From that we derive the fourth column of Table 5.6, which counts the number of blocks on which the step is performed.

The BLR factorization cost of each step is then equal to

$$\mathcal{C}_{step}(b,p) = cost_{step} * number_{step} \tag{5.21}$$

and is reported in the fifth column of Table 5.6. Then, we assume the block size b is of order $O(m^x)$, where x is a real value in $[0,1]$, and thus the number of blocks p per row and/or column is of order $O(m^{1-x})$. Then by substituting b and p by their value, we compute $\mathcal{C}_{step}(m,x)$ 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:

$$\mathcal{C}(m,x) = O(rm^{3-x} + m^{2+x}) \tag{5.22}$$

Similarly, the factor size complexity of a dense BLR matrix can be computed as

$$O(N_{LR} * br + N_{FR} * b^2) = O(p^2br + N_{na}^5 pb^2) = O(p^2br + pb^2) \tag{5.23}$$

where $N_{LR} = O(p^2)$ and $N_{FR} = O(p)$ are the number of low-rank and full-rank blocks in the matrix, respectively. Thus, the factor size complexity is:

$$\mathcal{M}(m,x) = O(rm^{2-x} + m^{1+x}) \tag{5.24}$$

It then remains to compute the optimal x^* which minimizes the complexity. We consider a general rank bound $r = O(m^\alpha)$, with $\alpha \in [0,1]$. Equations (5.22) and (5.24) become

$$\mathcal{C}(m,x) = O(m^{3+\alpha-x} + m^{2+x}) \tag{5.25}$$
$$\mathcal{M}(m,x) = O(m^{2+\alpha-x} + m^{1+x}) \tag{5.26}$$

respectively. Then, the optimal x^* is given by

$$x^* = \frac{1+\alpha}{2} \tag{5.27}$$

which leads to optimal complexities

$$\mathcal{C}(m) = \mathcal{C}(m, x^*) = O(m^{2.5+\alpha/2}) = O(m^{2.5}\sqrt{r}), \tag{5.28}$$

$$\mathcal{M}(m) = \mathcal{M}(m, x^*) = O(m^{1.5+\alpha/2}) = O(m^{1.5}\sqrt{r}). \tag{5.29}$$

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 $r = O(1)$ leads to complexities in $O(m^{2.5})$ for flops and $O(m^{1.5})$ for factor size, while the case $r = O(\sqrt{m})$ leads to $O(m^{2.75})$ for flops and $O(m^{1.75})$ for factor size. The link between dense and sparse rank bounds will be made in Section 5.4.4.

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 for hierarchical formats, where the construction of the compressed matrix, whose cost is in $O(m^2 r)$ [37], becomes the bottleneck when it has to be performed.

5.4.4 From dense to sparse BLR complexity

The results derived in the previous section can be readily used to compute the complexity of a multifrontal factorization based on the use of the BLR format at each front. Here we rely on the result by George [74] discussed in Section 2.2.3.1. For the sake of readability, we report it below.

We deal with a sparse matrix resulting from a square ($d = 2$) or cubic ($d = 3$) domain of dimension$N$. At each level ℓ of the separators tree, we need to factorize $(2^d)^\ell$ fronts of order $O((\frac{N}{2^\ell})^{d-1})$, for ℓ ranging from 0 to $L = \log_2(N)$. Therefore, the flop complexity $\mathcal{C}_{MF}(N)$ to factorize a sparse matrix of order N^d is

$$\mathcal{C}_{MF}(N) = \sum_{\ell=0}^{L} \mathcal{C}_\ell(N) = \sum_{\ell=0}^{L} (2^d)^\ell \mathcal{C}((\frac{N}{2^\ell})^{d-1}), \tag{5.30}$$

where $\mathcal{C}_\ell(N)$ is the cost of factorizing all the fronts on the ℓ-th level, i.e. $\mathcal{C}_\ell(N) = (2^d)^\ell \mathcal{C}(m_\ell)$ with $m_\ell = (\frac{N}{2^\ell})^{d-1}$. Using the dense complexity Equation 5.28, we compute and report the value of $\mathcal{C}_\ell(N)$ in Table 5.7 (second column). The overall complexity of the multifrontal factorization is obtained by solving the geometric series in Equation (5.30) and is reported in Table 5.7 (third column).

Using Equation 5.29, we similarly compute the factor size complexity:

$$\mathcal{M}_{MF}(N) = \sum_{\ell=0}^{L} \mathcal{M}_\ell(N) = \sum_{\ell=0}^{L} (2^d)^\ell \mathcal{M}((\frac{N}{2^\ell})^{d-1}), \tag{5.31}$$

and report the results in Table 5.7.

5.4.4.1 Complexity of the BLR variants

In this section we focus on the complexity of the variants described in Section 5.3.3. This is computed in a very similar way as for the standard version (see Section 5.4.3). We provide the equivalent of Tables 5.6 and 5.7 for the BLR variants in Tables 5.8 and 5.9, respectively.

d	$\mathcal{C}_\ell(N)$	$\mathcal{C}_{MF}(N)$	
2D	$O(2^{-(\ell+\alpha)/2}N^{2.5+\alpha/2})$	$O(N^{2.5+\alpha/2}) = O(N^{2.5}\sqrt{r})$	
3D	$O(2^{-(2+\alpha)\ell}N^{5+\alpha})$	$O(N^{5+\alpha}) = O(N^{5}\sqrt{r})$	
d	$\mathcal{M}_\ell(N)$	$\mathcal{M}_{MF}(N)$	
		$\alpha = 0$	$\alpha > 0$
2D	$O(2^{\ell(1-\alpha)/2}N^{1.5+\alpha/2})$	$O(N^2)$	$O(N^2)$
3D	$O(2^{-\alpha\ell}N^{3+\alpha})$	$O(N^3 \log N)$	$O(N^{3+\alpha}) = O(N^3\sqrt{r})$

Table 5.7: Flop and factor size complexity of the BLR (standard UFSC variant) multifrontal factorization of a sparse matrix of order N^d. d: dimension. $\mathcal{C}_\ell(N)/\mathcal{M}_\ell(N)$: flop/factor size complexity at level ℓ in the separator tree, computed using the dense complexity equations 5.28 and 5.29. $\mathcal{C}_{MF}(N)/\mathcal{M}_{MF}(N)$: total multifrontal flop/factor size complexity, computed using equations 5.30 and 5.31.

In Table 5.8, we report the cost of each step of the factorization. Using the LUAR technique implies that the cost of the Outer Product operation is reduced thanks to the recompression of the accumulated updates; the recompression, however has a cost which has to be accounted for in the complexity. This is illustrated in Table 5.8 with grayed-out rows. The UFCS (possibly with LUAR) variant, instead, reduces the complexity further thanks to the fact that Solve operations are mostly done on low-rank blocks rather than full-rank ones.

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:

$$\mathcal{C}(m, x) = O(r^2 m^{3-2x} + m^{2+x}) \tag{5.32}$$

Compared to 5.22, the low-rank term of the complexity has thus been reduced from $O(rm^{3-x})$ to $O(r^2m^{3-2x})$ thanks to the recompression of the accumulated updates. The full-rank term $O(m^{2+x})$ remains the same. By recomputing the value of x^*, we achieve flop complexity gains: For $r = O(m^\alpha)$, $\mathcal{C}(m)$ becomes

$$\mathcal{C}(m) = O(m^{2+(2\alpha+1)/3}) = O(m^{7/3}r^{2/3}), \tag{5.33}$$

which yields in particular $O(m^{7/3})$ for $r = O(1)$ and $O(m^{8/3})$ for $r = O(\sqrt{m})$.

In the same way, the flop complexity for the dense factorization with the UFCS+LUAR variant is given by

$$\mathcal{C}(m, x) = O(r^2 m^{3-2x} + m^{1+2x}). \tag{5.34}$$

This time, the full-rank term has been reduced from $O(m^{2+x})$ to $O(m^{1+2x})$. By recomputing x^*, we achieve further flop complexity gains:

$$\mathcal{C}(m) = O(m^{2+\alpha}) = O(m^2 r), \tag{5.35}$$

which yields in particular $O(m^2)$ for $r = O(1)$ and $O(m^{2.5})$ for $r = O(\sqrt{m})$.

Note that for the UFCS+LUAR variant, the Compress step has become asymptotically dominant and thus the assumption that its cost is $O(b^2 r)$ is now necessary to obtain the complexity reported in equation 5.34.

Note that the factor size complexity is not affected by the BLR variant used.

UFSC+LUAR variant					
step	type	cost	number	$\mathcal{C}_{step}(b,p)$	$\mathcal{C}_{step}(m,x)$
Factor	FR	$O(b^3)$	$O(p)$	$O(pb^3)$	$O(m^{1+2x})$
Solve	FR-FR	$O(b^3)$	$O(p^2)$	$O(p^2b^3)$	$O(m^{2+x})$
Compress	LR	$O(b^2r)$	$O(p^2)$	$O(p^2b^2r)$	$O(m^2r)$
Inner Prod.	LR-LR	$O(br^2)$	$O(p^3)$	$O(p^3br^2)$	$O(m^{3-2x}r^2)$
	LR-FR	$O(b^2r)$	$O(p^2)$	$O(p^2b^2r)$	$O(m^2r)$
	FR-FR	$O(b^3)$	$O(p)$	$O(pb^3)$	$O(m^{1+2x})$
Recompress	LR	$O(bpr^2)$	$O(p^2)$	$O(p^3br^2)$	$O(m^{3-2x}r^2)$
Outer Prod.	LR	$O(b^2r)$	$O(p^2)$	$O(p^2b^2r)$	$O(m^2r)$
UFCS+LUAR variant					
step	type	cost	number	$\mathcal{C}_{step}(b,p)$	$\mathcal{C}_{step}(m,x)$
Factor	FR	$O(b^3)$	$O(p)$	$O(pb^3)$	$O(m^{1+2x})$
Solve	FR-FR	$O(b^3)$	$O(p)$	$O(pb^3)$	$O(m^{1+2x})$
	LR-FR	$O(b^2r)$	$O(p^2)$	$O(p^2b^2r)$	$O(m^2r)$
Compress	LR	$O(b^2r)$	$O(p^2)$	$O(p^2b^2r)$	$O(m^2r)$
Inner Product	LR-LR	$O(br^2)$	$O(p^3)$	$O(p^3br^2)$	$O(m^{3-2x}r^2)$
	LR-FR	$O(b^2r)$	$O(p^2)$	$O(p^2b^2r)$	$O(m^2r)$
	FR-FR	$O(b^3)$	$O(p)$	$O(pb^3)$	$O(m^{1+2x})$
Recompress	LR	$O(bpr^2)$	$O(p^2)$	$O(p^3br^2)$	$O(m^{3-2x}r^2)$
Outer Product	LR	$O(b^2r)$	$O(p^2)$	$O(p^2b^2r)$	$O(m^2r)$

Table 5.8: 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. $\mathcal{C}_{step}(b,p)$: obtained by multiplying the *cost* and *number* columns (equation 5.21). $\mathcal{C}_{step}(m,x)$: obtained with the assumption that $b = O(m^x)$ (and thus $p = O(m^{1-x})$), for some $x \in [0,1]$.

The sparse flop complexities are derived from the dense ones in the same way as they are for the standard FSCU/UFSC variant. The results are reported in Table 5.9.

A summary of the sparse complexities for all BLR variants, as well as the full-rank and $\mathcal{H}$ complexities, is given in Table 5.10 for the cases where $r = O(1)$ and $r = O(\sqrt{m})$.

5.4.5 Experimental results

In this section we compare the experimental complexity of the full-rank solver with each of the previously presented BLR variants (FSCU/UFSC, UFSC+LUAR, UFCS+LUAR).We refer the reader to our original paper on the complexity of the BLR factorization [J6] or to Mary's PhD thesis[125] for a richer set of experiments that include an analysis of the influence of the threshold ε and the lock size.

All the experiments in this chapter were performed on the `brunch` system (see description in Section A.2).

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:

$$X_{fit} = e^{\beta_1^*} N^{\beta_2^*} \text{ with } \beta_1^*, \beta_2^* = \arg\min_{\beta_1,\beta_2} \| \log X_{obs} - \beta_1 - \beta_2 \log N \|^2. \tag{5.36}$$

	UFSC+LUAR	
d	$\mathcal{C}_\ell(N)$	$\mathcal{C}_{MF}(N)$
2D	$O(2^{-(1+2\alpha)\ell/3}N^{2+(2\alpha+1)/3})$	$O(N^{2+(2\alpha+1)/3}) = O(N^{7/3}r^{2/3})$
3D	$O(2^{-(5+4\alpha)\ell/3}N^{4+(4\alpha+2)/3})$	$O(N^{4+(4\alpha+2)/3}) = O(N^{14/3}r^{2/3})$

	UFCS+LUAR/UCFS+LUAR/CUFS		
d	$\mathcal{C}_\ell(N)$	$\mathcal{C}_{MF}(N)$	
		$\alpha = 0$	$\alpha > 0$
2D	$O(2^{-\alpha\ell}N^{2+\alpha})$	$O(N^2 \log N)$	$O(N^{2+\alpha}) = O(N^2 r)$
3D	$O(2^{-(1+2\alpha)\ell}N^{4+2\alpha})$	$O(N^{4+2\alpha}) = O(N^4 r)$	

Table 5.9: Flop and factor size complexity of the BLR multifrontal factorization of a sparse matrix of order N^d. d: dimension. $\mathcal{C}_\ell(N)$: flop complexity at level ℓ in the separator tree, computed using the dense complexity equations 5.33 and 5.35 for the UFSC+LUAR and UFCS+LUAR variants, respectively. $\mathcal{C}_{MF}(N)$: total multifrontal flop complexity, computed using equation 5.30.

	operations		factor size	
	2D	3D	2D	3D
$r = O(1)$				
BLR UFSC	$O(n^{1.25})$	$O(n^{1.67})$	$O(n)$	$O(n \log n)$
BLR UFSC+LUAR	$O(n^{1.17})$	$O(n^{1.56})$	$O(n)$	$O(n \log n)$
BLR UFCS+LUAR	$O(n \log n)$	$O(n^{1.33})$	$O(n)$	$O(n \log n)$
$\mathcal{H}$	$O(n \log n)$	$O(n^{1.33})$	$O(n)$	$O(n)$
$\mathcal{H}$ (fully-structured)	$O(n)$	$O(n)$	$O(n)$	$O(n)$
$r = O(\sqrt{m})$				
BLR UFSC	$O(n^{1.5})$	$O(n^{1.83})$	$O(n \log n)$	$O(n^{1.17})$
BLR UFSC+LUAR	$O(n^{1.5})$	$O(n^{1.78})$	$O(n \log n)$	$O(n^{1.17})$
BLR UFCS+LUAR	$O(n^{1.5})$	$O(n^{1.67})$	$O(n \log n)$	$O(n^{1.17})$
$\mathcal{H}$	$O(n^{1.5})$	$O(n^{1.67})$	$O(n)$	$O(n)$
$\mathcal{H}$ (fully-structured)	$O(n)$	$O(n^{1.33})$	$O(n)$	$O(n)$

Table 5.10: Flop and factor size complexities of the BLR and $\mathcal{H}$ multifrontal factorization of a system of n unknowns, considering the case $r = O(1)$ and $r = O(\sqrt{m}) = O(N^{d-1})$.

We provide the experimental complexities for two different problems: the Poisson problem and the Helmholtz problem.

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

$$\Delta u = f.$$

We perform the computations in real double-precision arithmetic. For the Poisson problem, we will use a low-rank threshold $\varepsilon = 10^{-10}$ with no particular application in mind.

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

for pressure p

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

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 aim is the modeling of visco-acoustic wave propagation in a 3D visco-acoustic homogeneous medium (see Section 5.3.4.1) parameterized by wavespeed (4000 m/s), density (1 kg/m^3), and quality factor (10000, no attenuation). 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 complex single-precision arithmetic and the chosen low-rank threshold is $\varepsilon = 10^{-4}$.

For the Poisson problem, the rank bound is in $O(1)$ [35]. For the Helmholtz problem, although there is no rigorous proof of it, the rank bound is assumed to be $O(N) = O(\sqrt{m})$ in the related literature [71, 157, 163]. Thus, we will use the Poisson and Helmholtz problems to experimentally validate the complexities computed in Table 5.10.

For both Poisson and Helmholtz, in all the following experiments, the backward error is in good agreement with the low-rank threshold used.

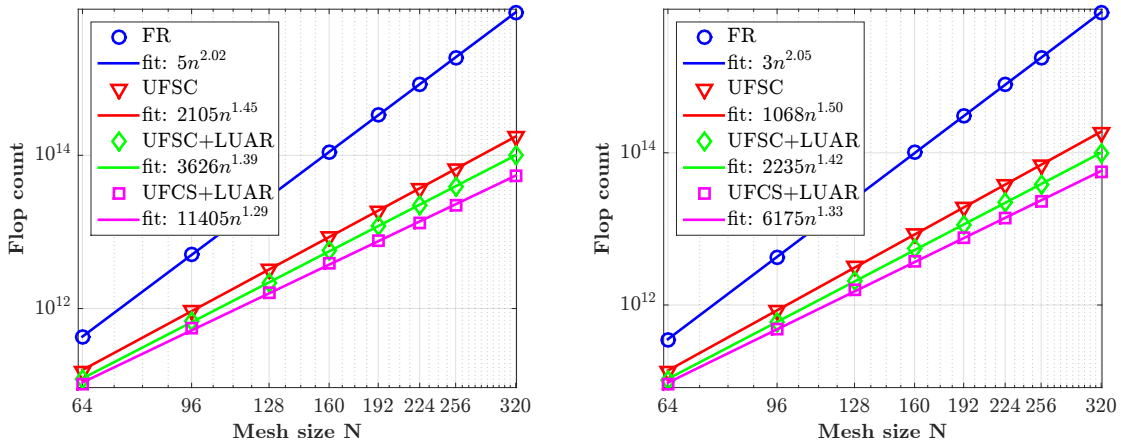
Both the Poisson and Helmholtz problem were discretized using the finite-difference method rather the finite-elements one, but this is acceptable as both methods are equivalent on equispaced meshes [132].

In Figures 5.17 and 5.18, 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. The FSCU variant is equivalent to the UFSC one and, thus, it will not be presented here.

The results show that each new variant improves the complexity. Note that we obtain the expected quadratic complexity of the full-rank version. Results with both geometric nested dissection (Figures 5.17(a) and 5.18(a)) and with a purely algebraic ordering computed by METIS (Figures 5.17(b) and 5.18(b)) 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 in $O(n^{1.45})$. Moreover, the constant in the big 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 $O(n^{1.39})$, while the UFCS+LUAR reaches the lowest complexity of the variants, in $O(n^{1.29})$. 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 in $O(n^{1.85})$ for UFSC, $O(n^{1.79})$ for UFSC+LUAR, and finally $O(n^{1.74})$ for UFCS+LUAR. Thus, the numerical results are in good agreement with the theoretical bounds reported in Table 5.10.

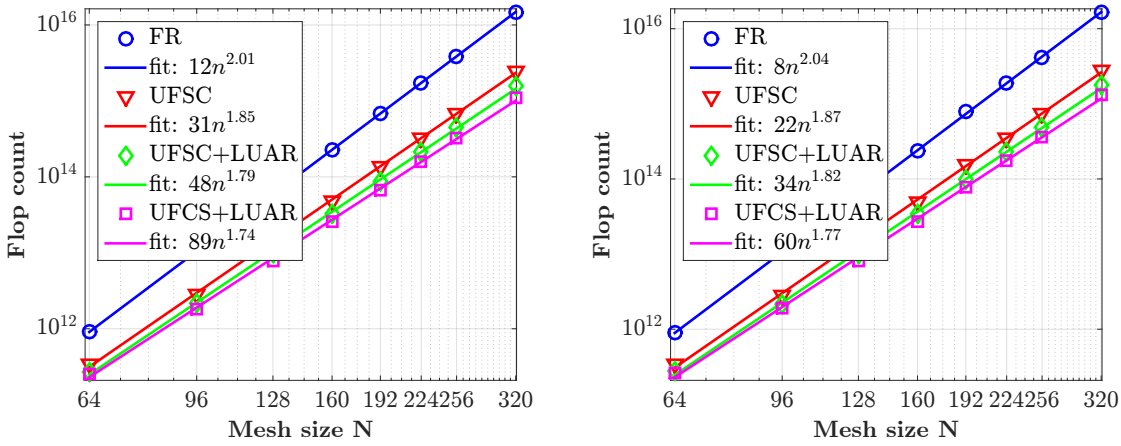
We also analyze the influence of the ordering on the complexity. We observe that even though the METIS ordering slightly degrades the complexity, results remain close to the geometric nested dissection ordering and still in good agreement with the theoretical bounds. This is a very important property 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.



(a) Nested Dissection ordering (geometric)

(b) METIS ordering (purely algebraic)

Figure 5.17: Flop complexity of each BLR variant (Poisson, $\varepsilon = 10^{-10}$).



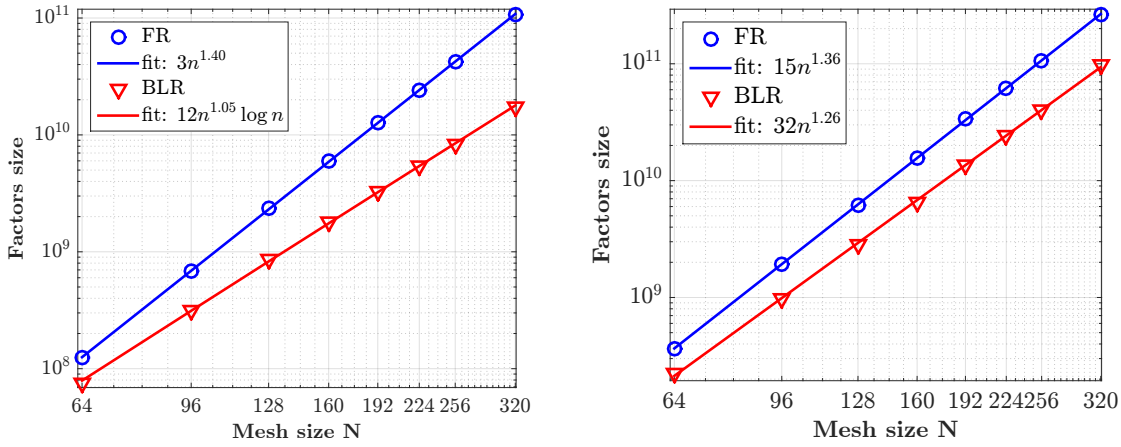
(a) Nested Dissection ordering (geometric)

(b) METIS ordering (purely algebraic)

Figure 5.18: Flop complexity of each BLR variant (Helmholtz, $\varepsilon = 10^{-4}$).

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 5.19, 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 5.10. The complexity is of order $O(n^{1.05} \log n)$ for Poisson and $O(n^{1.26})$ for Helmholtz.



(a) Poisson problem ($\varepsilon = 10^{-10}$)

(b) Helmholtz problem ($\varepsilon = 10^{-4}$)

Figure 5.19: Factor size complexity with METIS ordering.

5.5 Performance and scalability

In this section we address the performance of a BLR-based multifrontal solver. Specifically we will study how the reduction of the complexity of the multifrontal method shown in the previous section can be converted into an actual reduction of the execution time. We discuss what are the main limitations of the the basic FSCU variant and show how the other variants presented in Section 5.3.3 can improve the performance by reducing the complexity of the factorization as well as by improving the efficiency of operations.

Throughout this section we will provide experimental results on matrix S3 (see Appendix A.1) from the electromagnetics application described in Section 5.3.4.2 to illustrate and analyze the behavior of different algorithms. In Section 5.5.4 we will provide experimental results on the complete set of matrices in Table A.2. This includes the matrices described in Sections 5.3.4.1 and 5.3.4.2; coherently with what illustrated in these sections, the BLR approximation threshold for these problems was set to 10^{-3} and 10^{-7}, respectively. In addition to these, we included matrices from an industrial application in 3D structural mechanics provided by Électricité De France (EDF); these are the perf* matrices. 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[3] and require the solution of sparse linear systems such as the ones used in this paper. A previous study [159] showed that using BLR with $\varepsilon = 10^{-9}$ leads to an accurate enough solution for this class of problems.

For all experiments, we have used a right-hand side b such that the solution x is the vector containing only ones.

The experiments were done on the `brunch` machine (see Appendix A.2 for the details); the sustained peak of a core, measured with a dense matrix-matrix product (the BLAS `DGEMM` routine) is 47.1 Gflop/s.

Both the nested-dissection matrix reordering and the BLR clustering of the unknowns

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

are computed with METIS in a purely algebraic way (i.e., without any knowledge of the geometry of the problem domain).

5.5.1 Performance analysis of sequential FSCU algorithm

In this section, we analyze the performance of the FSCU algorithm (described in Algorithm 5.2) in a sequential setting. Our analysis underlines several issues, which will be addressed in subsequent sections.

In Table 5.11, 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 5.11: Sequential run (1 thread) on matrix S3.

To understand why, we report in Table 5.12 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.

step	FR				BLR			
	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 5.12: Performance analysis of sequential run of Table 5.11 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:

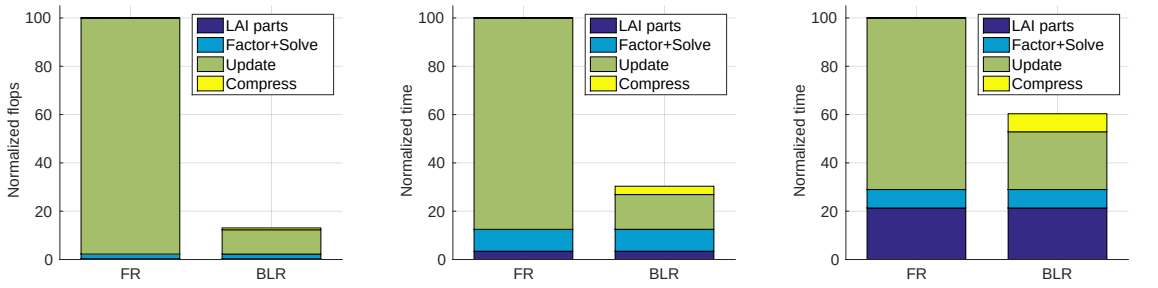
Issue 1 (lower granularity): the flop reduction by a factor 9.7 in the Update is not fully captured, as its execution time is only reduced by a factor 6.1. This is due to the lower granularity of the operations involved in low-rank products, which have thus a lower performance: the speed of the Update 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, we 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 5.20 (compare Figures 5.20(a) and 5.20(b)).

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 5.3.3. **Issue 3** is currently being investigated and we will not address it in this document.



(a) Normalized flops (b) Normalized time (1 thread) (c) Normalized time (24 threads)

Figure 5.20: Normalized (FR = 100%) flops and time on matrix S3.

5.5.2 Multithreading the BLR factorization

In this section, we describe basic shared-memory parallelization of the BLR FSCU factorization (Algorithm 5.2) and present a detailed analysis of its behavior. We will then present techniques that aim at improving the performance in a parallel, shared-memory setting.

5.5.2.1 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 5.2.

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 5.13, 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 5.13: Multithreaded run on matrix S3.

The BLR multithreading is thus less efficient than the FR one. To understand why, we provide in Table 5.14 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 5.14: Performance analysis of multithreaded run (24 threads) of Table 5.13 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, compared to 35.2 GF/s in FR.

A visual representation of this analysis is given on Figure 5.20 (compare Figures 5.20(b) and 5.20(c)).

In the rest of this section, we will revisit our algorithmic choices to address both of these issues.

5.5.2.2 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 et al. [110] and relies on the idea of separating the fronts by a so-called $\mathcal{L}_0$ layer, as illustrated in Figure 5.21. Each subtree rooted at the $\mathcal{L}_0$ layer is treated sequentially by a single thread; therefore, below the $\mathcal{L}_0$ 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 $\mathcal{L}_0$ 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.

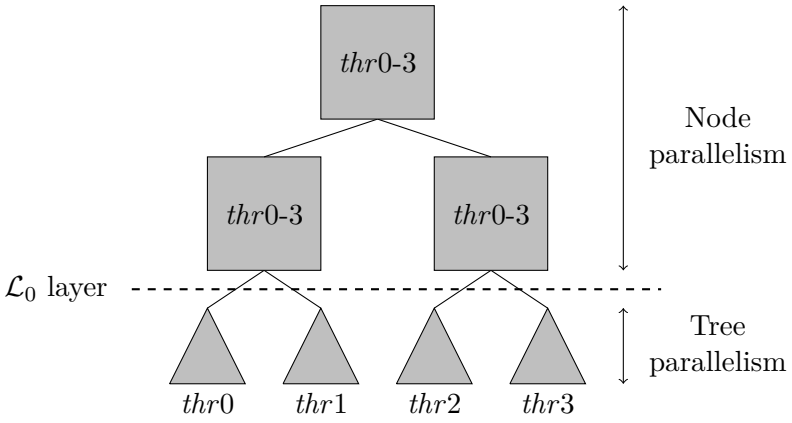


Figure 5.21: Illustration with four threads of how both node and tree multithreading can be exploited.

In Table 5.15, 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 $\mathcal{L}_0$ 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 $\mathcal{L}_0$ 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 $\mathcal{L}_0$ layer which do not expose much node 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 5.5.2.1.

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.

step	FR time	FR %	FR speedup	BLR time	BLR %	BLR 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 5.15: Execution time of FR and BLR factorizations on matrix S3 on 24 threads, exploiting both node and tree parallelism.

5.5.2.3 Right-looking vs. Left-looking

Algorithm 5.2 has been presented in its Right-looking (RL) version. In Table 5.16, we compare it to its Left-looking (LL) equivalent, referred to as UFSC (Algorithm 5.3). The RL and LL variants perform the same operations but in a different order, which results in a different memory access pattern [63].

parallelism	step	FR RL	FR LL	BLR RL	BLR 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

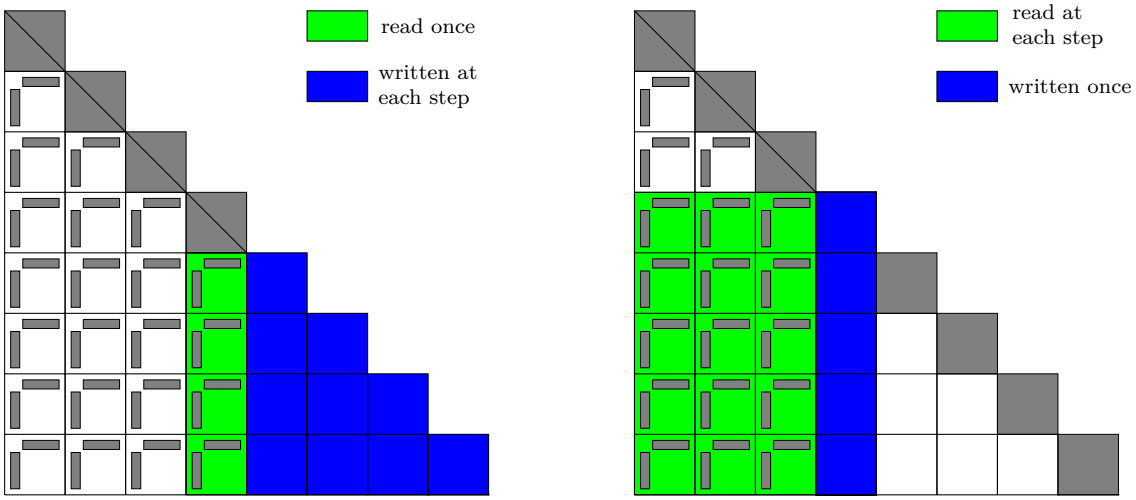
Table 5.16: Execution time of Right- and Left-looking factorizations on matrix S3.

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 5.22, 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 5.22(a)), 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 5.22(b)), 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:



(a) RL factorization

(b) LL factorization

Figure 5.22: 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 $k = 4$).

- Under the assumption that one FR block and two LR blocks fit in cache, the LL BLR factorization loads $O(p^2)$ FR blocks and $O(p^3)$ 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 be even worse), the RL BLR factorization loads $O(p^3)$ FR blocks and $O(p^2)$ 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 Section 5.5.2.1.

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 5.13) to 2.4 (Table 5.16).

Next, we show how the algorithmic variants of the BLR factorization can further improve its performance.

5.5.3 BLR factorization variants

In this section, we study the UFSC+LUAR and UFCS+LUAR BLR factorization variants. In Section 5.4, 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.

5.5.3.1 LUAR: Low-rank Updates Accumulation and Recompression

We begin by the UFSC+LUAR variant, i.e. Algorithm 5.3 with the modified LUAR-Update of Algorithm 5.5.

			UFSC	+LUA	+LUAR
average size of Outer Product			16.5	61.0	32.8
flops	($\times 10^{12}$)	Outer Product	3.76	3.76	1.59
	($\times 10^{9}$)	Recompress	0.00	0.00	5.39
	($\times 10^{12}$)	Total	10.19	10.19	8.15
time (s)		Outer Product	21.4	14.0	6.0
		Recompress	0.0	0.0	1.2
		Total	174.7	167.1	160.0
speed (GF/s)		Outer Product	29.3	44.7	44.4
		Recompress			0.7
		Total	9.7	10.2	8.5

Table 5.17: Performance analysis of the UFSC+LUAR factorization on matrix S3 on 24 threads

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 5.5), which is thus performed more efficiently. This should address **Issue 1**, identified in Section 5.5.1. Second, it allows for additional compression, as explained in Section 5.4.4.1.

In Table 5.17, 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). 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 $\widetilde{C}_{ik}^{(acc)}$ on line 8 in Algorithm 5.5. It also corresponds to the number of columns of X_S and Y_S in Figure 5.11.

Thanks to the accumulation, the average size of the Outer Product increases from 16.5 to 61.0. As illustrated by Figure 5.23, 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 5.5.4.

Next, we analyze the gain obtained by recompressing the accumulated low-rank updates (Figure 5.11). 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 5.23, 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.

5.5.3.2 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

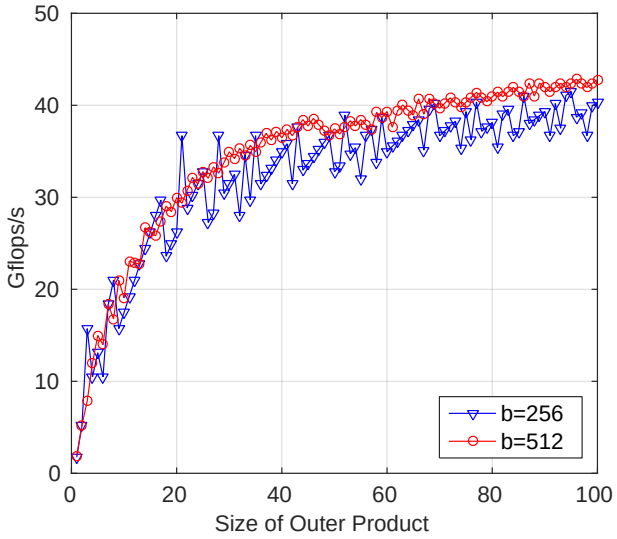


Figure 5.23: Performance benchmark of the Outer Product step on `brunch`. Please note that the average sizes (first line) and speed values (eighth line) of Table 5.17 cannot be directly linked using this figure because the average size would need to be weighted by its number of flops.

as described in Section 5.3.3.1. 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 16 in Algorithm 5.4.

In Table 5.18, 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. We refer the reader to the PhD thesis of Theo Mary [125] for a larger set of experiments showing how the scaled residual is affected by the use of the BLR factorization variants.

	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 5.18: 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.

	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 5.19: Detailed analysis of UFSC and UFCS results of Table 5.18 on matrix S3.

To explain the relatively high gain observed in Table 5.18, we analyze in detail the difference between UFSC and UFCS in Table 5.19. 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 to 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 Table 5.17) 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 5.17).

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 5.16) to 3.6 (Table 5.18).

5.5.4 Complete set of results

The results on the matrices coming from the three real-life applications from SEISCOPE, EMGS and EDF are reported in Table 5.20. We recall that the test matrices are described and assigned an ID in Table A.2. A richer set of experiments on matrices from the SuiteSparse Matrix Collection (SSMC) [56] can be found in our report from which this section is extracted [B1].

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. One can see in Table 5.20 that in BLR the scaled residual correctly reflects the influence of the low-rank approximations with threshold ε on the FR precision.

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.

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" line in Tables 5.20 corresponds 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 problem perf009ar (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.

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. Section 5.4).

We also analyze the parallel efficiency of the FR and BLR factorization by reporting the speedup on 24 threads. The speedup achieved by the FR factorization is of 17.1 on average and goes up to 18.8. 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 Table 5.20. 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 11.7 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.

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 threads)	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 threads)	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 5.20: Experimental results on real-life matrices from SEISCOPE, EMGS, and EDF.

Chapter 6

Conclusions and future work

The solution of large scale, sparse linear systems of equations lies at the heart of numerous applications from science and academia as well as industry. Among the most popular techniques used to solve these problems, sparse direct methods are appreciated for their ease of use and robustness. These favorable properties come at price of a significant memory and time consumption. As a consequence, the use of parallel computers is unavoidable.

The work described in this thesis is concerned with the scalability of sparse, direct solvers where, by scalability, we intend their ability to solve problems of very large size (currently, up to 10^7 unknowns) and/or make good use of the computing power and memory of modern, large-scale, heterogeneous computing platforms.

In Chapter 3 we have addressed the implementation of sparse, direct methods on modern computing systems. These are typically equipped with multi or manycore processors and, possibly, multiple accelerators, such as GPUs. The high number of working cores and the fact that they all share the same memory demand for the development of algorithms that can achieve high degrees of concurrency and do not suffer for excessive synchronization. The *tiled* (also referred to as *communication avoiding*) algorithms for the factorization of dense matrices presented in Section 3.4.1 respond to this necessity. As discussed in Section 3.4, these methods can effectively be used for the factorization of frontal matrices within a multifrontal method for the factorization of sparse matrices.

These algorithms have complex data access pattern, especially when used within the multifrontal method where multiple matrices can be assembled and factorized concurrently. Therefore, traditional parallelization techniques such as the fork-join one cannot take full advantage of the concurrency delivered by tiled or communication avoiding algorithms. For this reason we have investigated the use of a task-based parallel computing paradigm combined with an asynchronous and fully dynamic execution pattern as described in Chapter 3. This approach proved to be capable of achieving high performance and scalability on shared memory systems with high core counts. This method was achieved "by hand", that is, by manually implementing (either using Pthreads or a minimal subset of the OpenMP standard) our own tasking systems, described in Sections 3.4.1 and 3.1. Later we have investigated the use of modern runtime engines, such as StarPU, that rely on task-based parallelism as described in Sections 3.3, 3.4 and 3.5. Besides being very efficient, these tools provide a very wide set of features like the support for architectures equipped with accelerators or the possibility to implement and integrate task scheduling policies.

The achieved results, reported in Chapter 3 allow us to conclude that the use of task-based parallelism (where possible) combined with the use of modern runtime engines, delivers code that is capable of achieving high performance and portability on single

node, manycore systems equipped with multiple GPUs. Whether this approach can deliver the same good results on larger scale, distributed memory computers remains to be investigated and is the subject of ongoing and future research as described below. In addition, the use of task-based parallel programming models such as the Sequential Task Flow, eases the development of some algorithms and renders the code easier to develop and maintain.

Sparse, direct methods are generally very demanding in terms of memory. Additionally, when executed in parallel, their memory consumption can be (much) higher than in a sequential setting. Consequently, reducing, if possible, and controlling the memory consumption is extremely important to make these solvers scalable on large size supercomputers. In Chapter 4 we have presented techniques for controlling the memory footprint of a parallel sparse direct method. These basically consist in ensuring that the factorization is achieved within a prescribed memory envelope which has to be greater than or equal to the sequential memory consumption. This is achieved by means of careful scheduling or mapping policies that trade parallelism for memory and therefore the stricter the memory constraint will be, the higher the factorization execution time. Experimental results show that the proposed techniques allow for reliably controlling the memory consumption and for processing problems that would not fit in memory with traditional and commonly used mapping techniques such as the proportional mapping; the performance penalty is acceptable and, for some specific problems and settings, very small.

Finally, in Chapter 5 we investigated the use of low-rank approximation techniques within multifrontal solvers; these allow for reducing asymptotically the complexity (both in terms of memory and operations) of sparse, direct solvers by giving up some accuracy. This compromise between cost and accuracy can be safely and reliably chosen through a threshold that the application expert can set based on her/his needs. We have proposed a novel low-rank format called Block Low-Rank (BLR); unlike other commonly used low-rank formats, BLR does not use a hierarchical partitioning of data but, rather, a flat one which makes it very convenient for integration within a complex, parallel, algebraic multifrontal solver. We have discussed the details of the implementation of a BLR multifrontal solver and presented different factorization variants that aim reducing the cost or improving the performance of operations. We have presented a detailed theoretical analysis that shows that BLR can actually reduce the complexity of the factorization asymptotically and make it comparable to that of complex hierarchical formats. We have studied the parallel implementation of a BLR multifrontal solver on shared memory systems and presented techniques to improve its performance and scalability. The correctness of all theoretical results and the effectiveness of all the proposed techniques have been assessed on real life, large scale problems. The results presented in Chapter 4 lead us to conclude that the use of low-rank approximation techniques, although not suited for all kinds of problems, can considerably reduce the complexity of sparse, direct solvers and make them competitive even on those problems or setting where they are commonly regarded as too resource consuming. We can, moreover, conclude that, despite its slightly higher asymptotic complexity, the BLR format can provide considerable time and memory gains that are on par with those offered by more complex hierarchical formats [125].

6.1 Future work in sparse direct methods

Although sparse, direct solvers have been the object of a vast research effort, there are still numerous directions left to explore and new ones will emerge due to the evolution of supercomputer architectures and of the ever increasing need of applications.

First of all, it must be noted that the three main contributions presented in this thesis, i.e., task-based parallelism with runtimes, memory aware scheduling and mapping and the use of low-rank approximations, were achieved separately. Combining these methods opens up numerous opportunities for research.

It must be noted that the parallelization of the BLR factorization discussed in Section 5.5 essentially relies on a fork-join approach achieved through simple constructs like OpenMP parallel loops. The BLR format lends itself very naturally to task-based parallelism because of the decomposition into blocks of homogeneous sizes (within each front). The use of task based parallelism can further improve the performance and scalability of BLR, multifrontal solvers beyond the already good results presented in Section 5.5. This, moreover, could ease the porting of BLR solvers onto machines equipped with accelerators. Besides the technical issues related to implementation on heterogeneous platforms, this poses a number of algorithmic challenges. As explained in Section 5.5, the performance of BLR factorization suffers from the smaller granularity of operations with respect to the full-rank case; if the performance penalty is already considerable on standard multicore platforms, it can be catastrophic of devices such as GPUs that require operations of larger granularity in order to achieve a satisfactory performance. One possible way of addressing this issue is to use batched routines [41, 96] which allow for executing multiple operations of the same type at once within the same kernel; as a result a higher occupancy of the GPU is achieved which ultimately leads to higher performance. This, however, necessarily introduces synchronizations which may harm the scalability of the code. This topic has been partially investigated by Akbudak et al. [9] and Sergent et al. [145].

The unpredictability of the BLR workload and memory consumption also makes the development of reliable memory-aware mapping algorithms challenging. First of all, it is not possible to compute beforehand the sequential peak memory which basically makes the techniques presented in Chapter 4 unusable without major modifications. Another issue comes from the fact that the branches of the assembly tree may have different compression rates which makes it impossible to compute static mappings that achieve a good memory balance. As a result, it may be necessary to develop (partially) novel dynamic mapping and scheduling techniques even in distributed memory environments which is rather complex to achieve or may suffer from an excessive overhead. The implementation of such techniques may be eased by the use of a task-based parallel programming paradigm and, possibly, a runtime system.

Performance and scalability on large scale heterogeneous platforms The work described in Chapter 3, achieved, for the most part, in the context of the SOLHAR project, targets single node multi or manycore systems possibly equipped with multiple accelerators. Typical supercomputers, however, include many such nodes connected through a fast network. Achieving high performance and scalability on such architectures through the use of task-based parallelism and runtime systems is one of the research topics that we intend to investigate in the forthcoming years. This is, however, a challenging task that requires to address several issues that include (but are not limited to):

- Dynamic generation or update of the computational workload. In modern, task-based runtime systems such as StarPU, the DAG of tasks is, often, defined statically. In a large scale, distributed memory system, the tasks of the DAG, as well as the related data, are also statically mapped onto the available resources. This static DAG generation and mapping, which is hard to achieve because of the irregular nature of the target algorithms and because of the hierarchical and heterogeneous nature of supercomputers, may lead to scalability issues due to load imbalance. One

way to address this issue is to dynamically generate the DAG. This allows for an up-to-date view of the load of the computing system, but may lead to suboptimal allocation decisions due to the short-sightedness of the scheduler. Alternatively, a statically generated DAG and its associated mapping can be modified at run-time. Hybrid options are also possible, where a high-level DAG containing macro-tasks can be statically generated, whose tasks dynamically generate actual computational tasks.

- Partitioning of data and workload. Parallelization often implies partitioning data and operations. Based on how this partitioning is done, more or less concurrency becomes available, more or less communications are done and different levels of granularity of operations can be achieved. Because of heterogeneity, uniform partitioning schemes may lead to a suboptimal use of available resources. One way to overcome this issue is to design specific heterogeneous allocation schemes, to merge resources together into homogeneous groups and assign macro-tasks or sub-DAGs to each group. Alternatively, the partitioning of data and operations can be done dynamically.

- DAG generation and pruning. In very large-size problems, the DAG of tasks can become extremely big and costly to handle. However, in a distributed memory context, each compute node only needs to have a partial view of the entire DAG. In other words, the DAG can be pruned locally, which was shown in previous work [3] to be effective for large-scale scalability. How to achieve this efficiently and automatically remains an open challenge.

- Memory scalability. As explained above, the use of **memory aware** scheduling and mapping techniques is necessary when targeting very large scale systems. This requires the runtimes to provide methods and APIs that allow for reliably controlling the memory consumption of tasks.

- Scheduling and resource allocation. Static offline allocation and scheduling problems are known to be computationally hard, and it is extremely difficult to come up with a model able to estimate communication and computation times and to reflect co-scheduling and co-allocation effects. On the other hand, purely dynamic runtime strategies may take inefficient resource allocation decisions and lead to unbalanced computations, especially in a distributed setting. It is therefore crucial to design intermediate strategies based on good offline allocation decisions that may be revisited with fast runtime corrections, based on possibly partial information on the state of the platform.

- Communication avoiding algorithms. These algorithms are extremely important to achieve scalability especially on distributed memory systems. Previous studies on dense linear algebra methods [64] suggest that hierarchical communication-avoiding methods can be conceived to better match the hierarchical and heterogeneous nature of modern supercomputing platforms. These techniques must be extended to the case of sparse computations or, more generally, to algorithms that have a less regular communication pattern.

  Other techniques, instead, achieve lower communication overhead at the cost of a higher memory consumption. One such example is the supernodal fan-both method proposed by Ashcraft [24] which is a variant of the better known fan-in and fan-out ones. The use of a runtime can allow for developing a general method than

seamlessly switch between these three variants and that can, moreover, selectively do so only when more concurrency is needed. Another option is to investigate the use, within sparse factorizations, of methods that replicate some data in order to reduce the volume of communications such as the one proposed by Solomonik et al. [150] for dense matrix factorizations.

Block Low-Rank The use of low-rank approximation within sparse direct solvers is a relatively new subject and, although some solvers are already routinely used in production environments (including MUMPS, which provides the BLR feature since version 5.1 released in February 2017), numerous related research topics are still to be explored.

Among the things that we plan to investigate is the Multilevel BLR. In Section 5.4 we have showed that the BLR format can asymptotically reduce the cost (both in terms of memory and operations) of dense and sparse factorizations. Although the provided gain is considerable, the BLR format does not achieve the same asymptotic complexity as hierarchical ones. It is possible to extend the BLR format with multiple levels. This basically amounts to having nested BLR representations with a fixed number of levels. Through this technique it is possible to further reduce the cost of the storage and operations with respect to the single-level BLR. Although the multilevel BLR format can reach the same complexity as hierarchical formats with an infinite number of levels, its use is of great interest in the context of sparse factorizations where linear complexity can be achieved if the cost of the dense fronts factorizations is $O(m^{1.5})$ (see Section 5.4.3). This can be obtained with the Multilevel BLR using only a few levels. This results have been theoretically assessed in our recent work [B2] but the practical interest of the Multilevel BLR format still has to be validated.

A better understanding of the accuracy and stability of BLR-based solvers is necessary to develop methods that are not only faster and with a lower memory consumption but also reliable and roust from a numerical point of view. We have already started addressing this topic [159] for the case of the basic FSCU factorization but our analysis has to be improved and extended to the other variants taking into account different pivoting policies.

Finally, although the advantages of using low-rank compression can be clearly shown in theory, their practical use opens a number of challenges, especially related to the efficiency of the implementation, that are hard to address. Scalability in a distributed memory parallel setting, for example, is difficult to achieve because the workload cannot be accurately modeled prior to the factorization itself and because the number of operations is reduced more than the volume of communications. In such a parallel context, we also plan to achieve the implementation of forward elimination and backward substitution operation that can make an effective use of the BLR format; this can be difficult to do, especial in the case of a single right-hand side, because of the unfavorable ratio between communications and computations.

Rank-deficient QR In numerous applications, especially from mathematical optimization problems, it is required to solve least-squares problems with rank deficient matrices. In such a case, the least-squares problem admits an infinite solutions; among these, the one of minimum norm is required. When the problem matrix is dense, the canonical approach relies on the use of a QR factorization wit column pivoting, as explained in Section 2.1.4.3. Although this technique can also be applied to the sparse QR factorization [135], it is very hard to implement in an efficient way and may destroy the staircase structure of fronts which results in an increased cost. Davis [55] recently adapted to the case of the Householder multifrontal QR factorization a technique which was originally

proposed by Heath [98] for sparse QR factorizations based on Givens transformations: when a diagonal entry of R is found to be lower than a prescribed threshold in the course of the factorization, the corresponding column is skipped which leads to an R factor which is not triangular but can be permuted into a trapezoid. This approach, unfortunately is totally incompatible with the 2D partitioning of fronts into tiles that we rely on (see Section 3.4.1). Therefore, we have already started working on a different variant of the Heath method which achieves the factorization of a rank deficient matrix in two stages. In a first stage a regular, non rank-revealing, factorization is computed; in a second stage the R factor is inspected and, for all its diagonal entries smaller than a prescribed threshold, the corresponding rows are zeroed-out by means of Householder or Givens transformations. This second stage is achieved in a topological order assembly tree traversal and can, thus, be pipelined with the previous one which allows for an efficient use of parallelism. This technique can be possibly numerically improved by means of regularization techniques such as the Riley-Golub iteration [1] or the one proposed by Avron et al. [27].

Bibliography

Published work

Journal Articles

[J1] E. Agullo, P. R. Amestoy, A. Buttari, A. Guermouche, J.-Y. L'Excellent, and F.-H. Rouet. "Robust Memory-Aware Mappings for Parallel Multifrontal Factorizations". In: *SIAM Journal on Scientific Computing* 38.3 (2016), pp. C256–C279. DOI: 10.1137/130938505. eprint: http://dx.doi.org/10.1137/130938505. URL: http://dx.doi.org/10.1137/130938505.

[J2] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez. "Implementing Multifrontal Sparse Solvers for Multicore Architectures with Sequential Task Flow Runtime Systems". In: *ACM Trans. Math. Softw.* 43.2 (Aug. 2016), 13:1–13:22. ISSN: 0098-3500. DOI: 10.1145/2898348. URL: http://doi.acm.org/10.1145/2898348.

[J3] P. R. Amestoy, R. Brossier, A. Buttari, J.-Y. L'Excellent, T. Mary, L. Métivier, A. Miniussi, and S. Operto. "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 (2016), R363–R383.

[J4] P. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L'Excellent, and C. Weisbecker. "Improving Multifrontal Methods by Means of Block Low-Rank Representations". In: *SIAM Journal on Scientific Computing* 37.3 (2015), A1451–A1474. DOI: 10.1137/120903476. eprint: http://dx.doi.org/10.1137/120903476. URL: http://dx.doi.org/10.1137/120903476.

[J5] P. Amestoy, A. Buttari, G. Joslin, J.-Y. L'Excellent, M. Sid-Lakhdar, C. Weisbecker, M. Forzan, C. Pozza, R. Perrin, and V. Pellissier. "Shared-Memory Parallelism and Low-Rank Approximation Techniques Applied to Direct Solvers in FEM Simulation". In: *IEEE Transactions on Magnetics* 50.2 (Feb. 2014), pp. 517–520. ISSN: 0018-9464. DOI: 10.1109/TMAG.2013.2284024.

[J6] P. Amestoy, A. Buttari, J.-Y. L'Excellent, and T. Mary. "On the Complexity of the Block Low-Rank Multifrontal Factorization". In: *SIAM Journal on Scientific Computing* 39.4 (2017), A1710–A1740. DOI: 10.1137/16M1077192. eprint: https://doi.org/10.1137/16M1077192. URL: https://doi.org/10.1137/16M1077192.

[J7] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov. "Accelerating scientific computations with mixed precision algorithms". In: *Computer Physics Communications* 180.12 (2009), pp. 2526–2533. DOI: 10.1016/j.cpc.2008.11.005.

[J8] L. Bouchet, P. Amestoy, A. Buttari, F.-H. Rouet, and M. Chauvin. "INTEGRAL/SPI data segmentation to retrieve sources intensity variations". In: *Astronomy & Astrophysics* A52 (July 2013), (on line). DOI: 10.1051/0004-6361/201219605.

[J9] L. Bouchet, P. Amestoy, A. Buttari, F.-H. Rouet, and M. Chauvin. "Simultaneous analysis of large INTEGRAL/SPI datasets: optimizing the computation of the solution and its variance using sparse matrix algorithms". In: *Astronomy and Computing* 1 (2013), pp. 59–69. DOI: 10.1016/j.ascom.2013.03.002.

[J10] A. Buttari. "Fine-Grained Multithreading for the Multifrontal QR Factorization of Sparse Matrices". In: *SIAM Journal on Scientific Computing* 35.4 (2013), pp. C323–C345. eprint: http://epubs.siam.org/doi/pdf/10.1137/110846427. URL: http://epubs.siam.org/doi/abs/10.1137/110846427.

[J11] A. Buttari, P. D'Ambra, D. Di Serafino, and S. Filippone. "2LEV-D2P4: a package of high-performance preconditioners for scientific and engineering applications". In: *Appl. Algebra Eng., Commun. Comput.* 18.3 (2007), pp. 223–239. ISSN: 0938-1279. DOI: 10.1007/s00200-007-0035-z.

[J12] A. Buttari, J. Dongarra, J. Kurzak, P. Luszczek, and S. Tomov. "Using Mixed Precision for Sparse Matrix Computations to Enhance the Performance while Achieving 64-bit Accuracy". In: *ACM Trans. Math. Softw.* 34.4 (2008), pp. 1–22. ISSN: 0098-3500. DOI: 10.1145/1377596.1377597.

[J13] A. Buttari, J. Dongarra, J. Langou, J. Langou, P. Luszczek, and J. Kurzak. "Mixed Precision Iterative Refinement Techniques for the Solution of Dense Linear Systems". In: *Int. J. High Perform. Comput. Appl.* 21.4 (2007), pp. 457–466. ISSN: 1094-3420. DOI: 10.1177/1094342007084026.

[J14] A. Buttari, V. Eijkhout, J. Langou, and S. Filippone. "Performance Optimization and Modeling of Blocked Sparse Kernels". In: *Int. J. High Perform. Comput. Appl.* 21.4 (2007), pp. 467–484. ISSN: 1094-3420. DOI: 10.1177/1094342007083801.

[J15] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. "A class of parallel tiled linear algebra algorithms for multicore architectures". In: *Parallel Comput.* 35 (1 Jan. 2009), pp. 38–53. ISSN: 0167-8191. DOI: 10.1016/j.parco.2008.10.002. URL: http://dl.acm.org/citation.cfm?id=1486274.1486415.

[J16] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. "Parallel tiled QR factorization for multicore architectures". In: *Concurr. Comput. : Pract. Exper.* 20.13 (2008), pp. 1573–1590. ISSN: 1532-0626. DOI: 10.1002/cpe.v20:13.

[J17] S. Filippone and A. Buttari. "Object-Oriented Techniques for Sparse Matrix Computations in Fortran 2003". In: *ACM Transactions on Mathematical Software* 38.4 (Aug. 2012), 23:1–23:20. DOI: 10.1145/2331130.2331131.

[J18] J. Kurzak, A. Buttari, and J. Dongarra. "Solving Systems of Linear Equations on the CELL Processor Using Cholesky Factorization". In: *IEEE Trans. Parallel Distrib. Syst.* 19.9 (2008), pp. 1175–1186. ISSN: 1045-9219. DOI: 10.1109/TPDS.2007.70813.

[J19] J. Kurzak, A. Buttari, P. Luszczek, and J. Dongarra. "The PlayStation 3 for High-Performance Scientific Computing". In: *Computing in Science and Eng.* 10.3 (2008), pp. 84–87. ISSN: 1521-9615. DOI: 10.1109/MCSE.2008.85.

[J20] D. Shantsev, P. Jaysaval, S. de la Kethulle de Ryhove, P. Amestoy, A. Buttari, J.-Y. L'Excellent, and T. Mary. "Large-scale 3D EM modeling with a Block Low-Rank multifrontal direct solver". In: *Geophysical Journal International* (Mar. 2017). DOI: 10.1093/gji/ggx106.

Conference Proceedings

[C1] E. Agullo, P. R. Amestoy, A. Buttari, A. Guermouche, G. Joslin, J.-Y. L'Excellent, X. S. Li, A. Napov, F.-H. Rouet, M. Sid-Lakhdar, et al. "Recent advances in sparse direct solvers". In: *Conference on Structural Mechanicsin Reactor Technology*. 2013.

[C2] E. Agullo, G. Bosilca, A. Buttari, A. Guermouche, and F. Lopez. "Exploiting a Parametrized Task Graph Model for the Parallelization of a Sparse Direct Multifrontal Solver". In: *Euro-Par 2016: Parallel Processing Workshops: Euro-Par 2016 International Workshops, Grenoble, France, August 24-26, 2016, Revised Selected Papers*. Ed. by F. Desprez et al. Cham: Springer International Publishing, 2017, pp. 175–186. ISBN: 978-3-319-58943-5. DOI: 10.1007/978-3-319-58943-5_14. URL: http://dx.doi.org/10.1007/978-3-319-58943-5_14.

[C3] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez. "Multifrontal QR Factorization for Multicore Architectures over Runtime Systems". In: *Euro-Par 2013 Parallel Processing*. Springer Berlin Heidelberg, 2013, pp. 521–532. ISBN: 978-3-642-40046-9. URL: http://dx.doi.org/10.1007/978-3-642-40047-6_53.

[C4] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez. "Task-Based Multifrontal QR Solver for GPU-Accelerated Multicore Architectures." In: *HiPC*. IEEE Computer Society, 2015, pp. 54–63. ISBN: 978-1-4673-8488-9.

[C5] P. R. Amestoy, R. Brossier, A. Buttari, J.-Y. L'Excellent, T. Mary, L. Métivier, A. Miniussi, S. Operto, J. Virieux, and C. Weisbecker. "3D frequency-domain seismic modeling with a Parallel BLR multifrontal direct solver". In: *SEG Technical Program Expanded Abstracts 2015*. 2015. Chap. 692, pp. 3606–3611. DOI: 10.1190/segam2015-5811693.1. eprint: http://library.seg.org/doi/pdf/10.1190/segam2015-5811693.1. URL: http://library.seg.org/doi/abs/10.1190/segam2015-5811693.1.

[C6] P. R. Amestoy et al. "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*. 2015. Chap. 251, pp. 1303–1308. DOI: 10.1190/segam2015-5713962.1. eprint: http://library.seg.org/doi/pdf/10.1190/segam2015-5713962.1. URL: http://library.seg.org/doi/abs/10.1190/segam2015-5713962.1.

[C7] P. Amestoy, A. Buttari, G. Joslin, J.-Y. L'Excellent, M. Sid-Lakhdar, C. Weisbecker, M. Forzan, C. Pozza, R. Perrin, and V. Pellissier. "Shared memory parallelism and low-rank approximation techniques applied to direct solvers in FEM simulation". In: *IEEE International Conference on the Computation of Electromagnetic Fields (COMPUMAG), Budapest, Hungary, 30/06/2013-04/07/2013*. IEEE, June 2013. DOI: 10.1109/TMAG.2013.2284024.

[C8] G. Antoniu et al. "Towards exascale with the ANR-JST Japanese-French Project FP3C". In: *Ninth International Conference on Computer Science and Information Technologies Revised Selected Papers*. Sept. 2013, pp. 1–10. DOI: 10.1109/CSITechnol.2013.6710357.

[C9] G. Bella, A. Buttari, A. De Maio, F. Del Citto, S. Filippone, and F. Gasperini. "FAST-EVP: an Engine Simulation Tool". In: *High Perfromance Computing and Communications. First International Conference, HPCC 2005, Proceedings*. Ed. by Springer. Vol. 3726. Lecture Notes in Computer Science. [doi:10.1007/11557654_108]. Sept. 2005, pp. 976–986.

[C10] L. Bouchet, P. Amestoy, A. Buttari, F.-H. Rouet, and M. Chauvin. "INTEGRAL/SPI data segmentation to retrieve sources intensity variations (regular paper)". In: *An INTEGRAL view of the high-energy sky (the first 10 years), Paris, France, 15/10/2012-19/10/2012*. Ed. by A. Goldwurm, F. Lebrun, and C. Winkler. 2013.

[C11] A. Buttari. "Fine granularity sparse QR factorization for multicore based systems". In: *Proceedings of the 10th international conference on Applied Parallel and Scientific Computing - Volume 2*. PARA'10. Reykjavik, Iceland: Springer-Verlag, 2012, pp. 226–236. ISBN: 978-3-642-28144-0. URL: http://dx.doi.org/10.1007/978-3-642-28145-7_23.

[C12] A. Buttari, P. D'Ambra, D. S. Di Serafino, and S. Filippone. "Extending PSBLAS to Build Parallel Schwarz Preconditioners". In: *Applied Parallel Computing. State of the Art in Scientific Computing: 7th International Conference, PARA 2004, Lyngby, Denmark, June 20-23, 2004*. Ed. by Springer. Vol. 3732. Lecture Notes in Computer Science. Feb. 2006, pp. 593–602. DOI: 10.1007/11558958_71.

[C13] A. Buttari, J. Dongarra, P. Husbands, J. Kurzak, and K. Yelick. "Multithreading for Synchronization Tolerance in Matrix Factorization". In: *Proceedings of the SciDAC 2007 Conference*. Boston, Massachusetts: Journal of Physics: Conference Series, 2007. DOI: 10.1088/1742-6596/78/1/012028.

[C14] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov. "The impact of multicore on math software". In: *Proceedings of the 8th international conference on Applied parallel computing: state of the art in scientific computing*. PARA'06. Umeå;, Sweden: Springer-Verlag, 2007, pp. 1–10. ISBN: 3-540-75754-6, 978-3-540-75754-2. URL: http://dl.acm.org/citation.cfm?id=1775059.1775061.

[C15] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. "Parallel tiled QR factorization for multicore architectures". In: *PPAM'07: Proceedings of the 7th international conference on Parallel processing and applied mathematics*. Gdansk, Poland: Springer-Verlag, 2008, pp. 639–648. ISBN: 3-540-68105-1, 978-3-540-68105-2. DOI: 10.1007/978-3-540-68111-3_67.

[C16] J. Demmel et al. "Prospectus for the Next LAPACK and ScaLAPACK Libraries". In: *PARA'06: State-of-the-Art in Scientific and Parallel Computing*. High Performance Computing Center North (HPC2N) and the Department of Computing Science, UmeåUniversity. Umeå, Sweden: Springer, June 2006. DOI: 10.1007/978-3-540-75755-9_2.

[C17] G. Hautreux et al. "Pre-exascale Architectures: OpenPOWER Performance and Usability Assessment for French Scientific Community". In: *High Performance Computing: ISC High Performance 2017 International Workshops, DRBSD, ExaComm, HCPM, HPC-IODC, IWOPH, IXPUG, P3MA, VHPC, Visualization at Scale, WOPSSS, Frankfurt, Germany, June 18-22, 2017, Revised Selected Papers*. Ed. by J. M. Kunkel, R. Yokota, M. Taufer, and J. Shalf. Cham: Springer International Publishing, 2017, pp. 309–324. ISBN: 978-3-319-67630-2. DOI: 10.1007/978-3-319-67630-2_23. URL: https://doi.org/10.1007/978-3-319-67630-2_23.

[C18] J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, and J. Dongarra. "Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems)". In: *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. Tampa, Florida: ACM, 2006, p. 113. ISBN: 0-7695-2700-0. DOI: 10.1145/1188455.1188573.

[C19] L. Stanisic, E. Agullo, A. Buttari, A. Guermouche, A. Legrand, F. Lopez, and B. Videau. "Fast and Accurate Simulation of Multithreaded Sparse Linear Algebra Solvers". In: *Parallel and Distributed Systems (ICPADS), 2015 IEEE 21st International Conference on*. Dec. 2015, pp. 481–490. DOI: 10.1109/ICPADS.2015.67.

[C20] C. Weisbecker, P. Amestoy, O. Boiteau, R. Brossier, A. Buttari, J.-Y. L'Excellent, S. Operto, and J. Virieux. "3D frequency-domain seismic modeling with a block low-rank algebraic multifrontal direct solver". In: *SEG Technical Program Expanded Abstracts 2013*. 2013. Chap. 662, pp. 3411–3416. DOI: 10.1190/segam2013-0603.1.

Book Chapters

[B1] P. Amestoy, A. Buttari, I. Duff, A. Guermouche, J.-Y. L'Excellent, and B. Uçar. "MUMPS". In: *Encyclopedia of Parallel Computing*. Ed. by D. Padua. Springer Verlag, 2011.

[B2] P. Amestoy, A. Buttari, I. Duff, A. Guermouche, J.-Y. L'Excellent, and B. Uçar. "The Multifrontal Method". In: *Encyclopedia of Parallel Computing*. Ed. by D. Padua. Springer Verlag, 2011.

[B3] A. Buttari, J. Dongarra, J. Kurzak, and J. Langou. "Parallel Dense Linear Algebra Software in the Multicore Era". In: *Cyberinfrastructure Technologies and Applications*. Ed. by J. Cao. Nova Science Publishers, 2007.

[B4] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov. "Exploiting Mixed Precision Floating Point Hardware in Scientific Computations". In: *High Performance Computing and Grids in Action*. Ed. by L. Grandinetti. IOS Press, 2007.

[B5] J. Demmel et al. "Prospectus for a Linear Algebra Software Library for Dense Matrix Problems". In: *Handbook of Parallel Computing: Models, Algorithms and Applications*. Ed. by S. Rajasekaran and J. Reif. 1st ed. Vol. 17. Chapman & HallCRC Computer & Information Science. ISBN: 9781584886235. CRC Press, Dec. 2007.

To appear or submitted

[B1] P. R. Amestoy, A. Buttari, J.-Y. L 'excellent, and T. Mary. *Performance and Scalability of the Block Low-Rank Multifrontal Factorization on Multicore Architectures*. Research Report. submitted to ACM TOMS. INPT-IRIT ; CNRS-IRIT ; INRIA-LIP ; UPS-IRIT, Apr. 2017. URL: https://hal.archives-ouvertes.fr/hal-01505070.

[B2] P. Amestoy, A. Buttari, J.-Y. L'Excellent, and T. Mary. *Bridging the gap between flat and hierarchical low-rank matrix formats: the multilevel BLR format*. Tech. rep. Submitted to the SIAM Journal on Scientific Computing. IRIT, 2018.

References

[1] D. Achiya and E. Lars. "Approximating minimum norm solutions of rank-deficient least squares problems". In: *Numerical Linear Algebra with Applications* 5.2 (1998), pp. 79–99. DOI: 10.1002/(SICI)1099-1506(199803/04)5:2<79::AID-NLA126>3.0.CO;2-4.

[2] E. Agullo. "On the out-of-core factorization of large sparse matrices". PhD thesis. École Normale Supérieure de Lyon, Nov. 2008.

[3] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, and S. Thibault. *Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model*. Research Report RR-8927. June 2016, p. 27. URL: https://hal.inria.fr/hal-01332774.

[4] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi. *Task-based FMM for heterogeneous architectures*. Research Report RR-8513. Inria, Apr. 2014, p. 29. URL: https://hal.inria.fr/hal-00974674.

[5] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects". In: *Journal of Physics: Conference Series* 180.1 (2009), p. 012037. URL: http://stacks.iop.org/1742-6596/180/i=1/a=012037.

[6] E. Agullo, J. Dongarra, R. Nath, and S. Tomov. "Fully Empirical Autotuned QR Factorization For Multicore Architectures". In: *CoRR* abs/1102.5328 (2011).

[7] E. Agullo, L. Giraud, S. Nakov, and J. Roman. *Hierarchical hybrid sparse linear solver for multicore platforms*. Research Report RR-8960. INRIA Bordeaux, Oct. 2016, p. 25. URL: https://hal.inria.fr/hal-01379227.

[8] E. Agullo, A. Guermouche, and J.-Y. L'Excellent. "Reducing the I/O Volume in Sparse Out-of-core Multifrontal Methods". In: *SIAM Journal on Scientific Computing* 31.6 (2010), pp. 4774–4794.

[9] K. Akbudak, H. Ltaief, A. Mikhalev, A. Charara, and D. E. Keyes. *Exploiting Data Sparsity for Large-Scale Matrix Computations*. Tech. rep. KAUST, 2018. URL: http://hdl.handle.net/10754/627403.

[10] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann, 2002.

[11] P. R. Amestoy, T. A. Davis, and I. S. Duff. "An Approximate Minimum Degree Ordering Algorithm". In: *SIAM J. Matrix Anal. Appl.* 17.4 (Oct. 1996), pp. 886–905. ISSN: 0895-4798. DOI: 10.1137/S0895479894278952. URL: http://dx.doi.org/10.1137/S0895479894278952.

[12] P. R. Amestoy and I. S. Duff. "Memory Management Issues in Sparse Multifrontal Methods On Multiprocessors". In: *The International Journal of Supercomputing Applications* 7.1 (1993), pp. 64–82. DOI: 10.1177/109434209300700105. eprint: https://doi.org/10.1177/109434209300700105. URL: https://doi.org/10.1177/109434209300700105.

[13] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. "A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling". In: *SIAM Journal on Matrix Analysis and Applications* 23.1 (2001), pp. 15–41.

[14] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, Y. Robert, F.-H. Rouet, and B. Uçar. "On computing inverse entries of a sparse matrix in an out-of-core environment". In: *SIAM Journal on Scientific Computing* 34.4 (2012), A1975–A1999.

[15] P. R. Amestoy, I. S. Duff, and C. Puglisi. "Multifrontal QR factorization in a multiprocessor environment". In: *Int. Journal of Num. Linear Alg. and Appl.* 3(4) (1996), pp. 275–300.

[16] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet. "Hybrid scheduling for the parallel solution of linear systems". In: *Parallel Computing* 32.2 (2006), pp. 136–156.

[17] P. R. Amestoy and C. Puglisi. "An unsymmetrized multifrontal LU factorization". In: *SIAM Journal on Matrix Analysis and Applications* 24 (2002), pp. 553–569.

[18] P. Amestoy, J.-Y. L'Excellent, and G. Moreau. *On Exploiting Sparsity of Multiple Right-Hand Sides in Sparse Direct Solvers*. Research Report RR-9122. Submitted to SIAM SISC. ENS de Lyon ; INRIA Grenoble - Rhone-Alpes, Dec. 2017, pp. 1–28. URL: https://hal.inria.fr/hal-01649244.

[19] A. Aminfar, S. Ambikasaran, and E. Darve. "A fast block low-rank dense solver with applications to finite-element matrices". In: *Journal of Computational Physics* 304 (2016), pp. 170–188.

[20] E. Anderson et al. *LAPACK Users' Guide*. Third. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999. ISBN: 0-89871-447-8 (paperback).

[21] J. Anton, C. Ashcraft, and C. Weisbecker. "A Block Low-Rank multithreaded factorization for dense BEM operators". In: *SIAM Conference on Parallel Processing (SIAM PP16)*. Paris, France, Apr. 2016.

[22] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. "Thread Scheduling for Multiprogrammed Multiprocessors". In: *Theory Comput. Syst.* 34.2 (2001), pp. 115–144. DOI: 10.1007/s00224-001-0004-z.

[23] K. Asanovic et al. *The Landscape of Parallel Computing Research: A View from Berkeley*. Tech. rep. UCB/EECS-2006-183. TECHNICAL REPORT, UC BERKELEY, 2006.

[24] C. Ashcraft. "The Fan-Both Family of Column-Based Distributed Cholesky Factorization Algorithms". In: *Graph Theory and Sparse Matrix Computation*. Ed. by A. George, J. R. Gilbert, and J. W. H. Liu. New York, NY: Springer New York, 1993, pp. 159–190. ISBN: 978-1-4613-8369-7.

[25] C. Ashcraft and R. G. Grimes. "SPOOLES: An object oriented sparse matrix library". In: *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*. San Antonio, Texas, Mar. 1999.

[26] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures". In: *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009* 23 (2 Feb. 2011), pp. 187–198. DOI: 10.1002/cpe.1631. URL: http://hal.inria.fr/inria-00550877.

[27] H. Avron, E. Ng, and S. Toledo. "Using Perturbed *QR* Factorizations to Solve Linear Least-Squares Problems". In: *SIAM Journal on Matrix Analysis and Applications* 31.2 (2009), pp. 674–693. DOI: 10.1137/070698725. eprint: https://doi.org/10.1137/070698725. URL: https://doi.org/10.1137/070698725.

[28] R. M. Badia, J. R. Herrero, J. Labarta, J. M. Pérez, E. S. Quintana-Ortí, and G. Quintana-Ortí. "Parallelizing dense and banded linear algebra libraries using SMPSs". In: *Concurrency and Computation: Practice and Experience* 21.18 (2009), pp. 2438–2456.

[29] G. Ballard, E. Carson, J. Demmel, M. Hoemmen, N. Knight, and O. Schwartz. "Communication lower bounds and optimal algorithms for numerical linear algebra". In: *Acta Numerica* 23 (May 2014), pp. 1–155. ISSN: 1474-0508. DOI: 10.1017/S0962492914000038. URL: http://journals.cambridge.org/article_S0962492914000038.

[30] O. Beaumont and A. Guermouche. "Task scheduling for parallel multifrontal methods". In: *Euro-Par 2007 Parallel Processing* (2007), pp. 758–766.

[31] M. Bebendorf. *Hierarchical Matrices: A Means to Efficiently Solve Elliptic Boundary Value Problems (Lecture Notes in Computational Science and Engineering)*. 1st ed. Springer, 2008. ISBN: 3540771468.

[32] M. Bebendorf. "Approximation of boundary element matrices". In: *Numerische Mathematik* 86.4 (2000), pp. 565–589. ISSN: 0945-3245. DOI: 10.1007/PL00005410. URL: http://dx.doi.org/10.1007/PL00005410.

[33] M. Bebendorf. "Efficient inversion of Galerkin matrices of general second-order elliptic differential operators with nonsmooth coefficients". In: *Mathematics of Computation* 74 (2005), pp. 1179–1199.

[34] M. Bebendorf. "Why finite element discretizations can be factored by triangular hierarchical matrices". In: *SIAM Journal on Numerical Analysis* 45 (2007), p. 1472.

[35] M. Bebendorf and W. Hackbusch. "Existence of $\mathcal{H}$-matrix approximants to the inverse FE-matrix of elliptic operators with L^∞-coefficients". In: *Numerische Mathematik* 95.1 (2003), pp. 1–28.

[36] Å. Björck. *Numerical methods for Least Squares Problems*. Philadelphia: SIAM, 1996.

[37] S. Börm, L. Grasedyck, and W. Hackbusch. "Introduction to hierarchical matrices with applications". In: *Engineering analysis with boundary elements* 27.5 (2003), pp. 405–422. ISSN: 0955-7997. DOI: 10.1016/S0955-7997(02)00152-2. URL: http://www.mis.mpg.de/de/publications/preprints/2002/prepr2002-18.html.

[38] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. J. Dongarra. "PaRSEC: Exploiting Heterogeneity to Enhance Scalability". In: *Computing in Science and Engineering* 15.6 (2013), pp. 36–45. DOI: 10.1109/MCSE.2013.98. URL: http://dx.doi.org/10.1109/MCSE.2013.98.

[39] H. Bouwmeester, M. Jacquelin, J. Langou, and Y. Robert. "Tiled QR Factorization Algorithms". In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '11. Seattle, Washington: ACM, 2011, 7:1–7:11. ISBN: 978-1-4503-0771-0. DOI: 10.1145/2063384.2063393. URL: http://doi.acm.org/10.1145/2063384.2063393.

[40] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. "hwloc: A Generic Framework for Managing Hardware Affinities in HPC Application". In: *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference*. Feb. 2010, pp. 180–186. DOI: 10.1109/PDP.2010.67.

[41] W. J. Brouwer and P.-Y. Taunay. "Efficient Batch LU and QR Decomposition on GPU". In: *Numerical Computations with GPUs*. Ed. by V. Kindratenko. Cham: Springer International Publishing, 2014, pp. 69–86. ISBN: 978-3-319-06548-9. DOI: 10.1007/978-3-319-06548-9_4. URL: https://doi.org/10.1007/978-3-319-06548-9_4.

[42] R. A. Brualdi and H. J. Ryser. *Combinatorial Matrix Theory*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1991. DOI: 10.1017/CBO9781107325708.

[43] R. A. Brualdi and B. L. Shader. "Strong Hall Matrices". In: *SIAM J. Matrix Anal. Appl.* 15.2 (Apr. 1994), pp. 359–365. ISSN: 0895-4798. DOI: 10.1137/S0895479892225142. URL: http://dx.doi.org/10.1137/S0895479892225142.

[44] J. R. Bunch and L. Kaufman. "Some Stable Methods for Calculating Inertia and Solving Symmetric Linear Systems". In: *Mathematics of Computation* 31 (1977), pp. 162–179.

[45] P. Businger and G. H. Golub. "Linear Least Squares Solutions by Householder Transformations". In: *Numer. Math.* 7.3 (June 1965), pp. 269–276. ISSN: 0029-599X. DOI: 10.1007/BF01436084. URL: http://dx.doi.org/10.1007/BF01436084.

[46] S. Chandrasekaran, P. Dewilde, M. Gu, and N. Somasunderam. "On the Numerical Rank of the Off-Diagonal Blocks of Schur Complements of Discretized Elliptic PDEs". In: *SIAM Journal on Matrix Analysis and Applications* 31.5 (2010), pp. 2261–2290. DOI: 10.1137/090775932. URL: http://link.aip.org/link/?SML/31/2261/1.

[47] S. Chandrasekaran, M. Gu, and T. Pals. "A fast ULV decomposition solver for hierarchically semiseparable representations". In: *SIAM Journal on Matrix Analysis and Applications* 28.3 (2006), pp. 603–622.

[48] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam. "Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate". In: *ACM Trans. Math. Softw.* 35.3 (Oct. 2008), 22:1–22:14. ISSN: 0098-3500. DOI: 10.1145/1391989.1391995. URL: http://doi.acm.org/10.1145/1391989.1391995.

[49] H. Cheng, Z. Gimbutas, P. G. Martinsson, and V. Rokhlin. "On the Compression of Low Rank Matrices". In: *SIAM Journal on Scientific Computing* 26.4 (2005), pp. 1389–1404. DOI: 10.1137/030602678. eprint: http://dx.doi.org/10.1137/030602678. URL: http://dx.doi.org/10.1137/030602678.

[50] T. F. Coleman, A. Edenbrandt, and J. R. Gilbert. "Predicting Fill for Sparse Orthogonal Factorization". In: *J. ACM* 33.3 (May 1986), pp. 517–532. ISSN: 0004-5411. DOI: 10.1145/5925.5932. URL: http://doi.acm.org/10.1145/5925.5932.

[51] S. Constable. "Ten years of marine CSEM for hydrocarbon exploration". In: *Geophysics* 75.5 (2010), 75A67–75A81.

[52] P. Coulier, H. Pouransari, and E. Darve. "The inverse fast multipole method: using a fast approximate direct solver as a preconditioner for dense linear systems". In: *ArXiv e-prints* (Aug. 2015). arXiv: 1508.01835 [math.NA].

[53] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng. "A column approximate minimum degree ordering algorithm". In: *ACM Trans. Math. Softw.* 30.3 (Sept. 2004), pp. 353–376. ISSN: 0098-3500. DOI: 10.1145/1024074.1024079. URL: http://doi.acm.org/10.1145/1024074.1024079.

[54] T. A. Davis. "Algorithm 832: UMFPACK V4.3 — an unsymmetric-pattern multifrontal method". In: *ACM Transactions On Mathematical Software* 30.2 (2004), pp. 196–199.

[55] T. A. Davis. "Algorithm 915, SuiteSparseQR: Multifrontal multithreaded rank-revealing sparse QR factorization". In: *ACM Trans. Math. Softw.* 38.1 (Dec. 2011), 8:1–8:22. ISSN: 0098-3500. DOI: 10.1145/2049662.2049670. URL: http://doi.acm.org/10.1145/2049662.2049670.

[56] T. A. Davis and Y. Hu. "The university of Florida sparse matrix collection". In: *ACM Trans. Math. Softw.* 38.1 (Dec. 2011), 1:1–1:25. ISSN: 0098-3500. URL: http://doi.acm.org/10.1145/2049662.2049663.

[57] J. Demmel. *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, 1997. DOI: 10.1137/1.9781611971446. eprint: http://epubs.siam.org/doi/pdf/10.1137/1.9781611971446. URL: http://epubs.siam.org/doi/abs/10.1137/1.9781611971446.

[58] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. "Communication-optimal Parallel and Sequential QR and LU Factorizations". In: *SIAM J. Sci. Comput.* 34.1 (Feb. 2012), pp. 206–239. ISSN: 1064-8275. URL: http://dx.doi.org/10.1137/080731992.

[59] E. W. Dijkstra. "Een algorithme ter voorkoming van de dodelijke omarming". circulated privately. 1965. URL: http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD108.PDF.

[60] E. W. Dijkstra. "The Mathematics Behind the Banker's Algorithm". English. In: *Selected Writings on Computing: A personal Perspective*. Texts and Monographs in Computer Science. Springer New York, 1982, pp. 308–312. ISBN: 978-1-4612-5697-7. DOI: 10.1007/978-1-4612-5695-3_54. URL: http://dx.doi.org/10.1007/978-1-4612-5695-3_54.

[61] E. D. Dolan and J. J. Moré. "Benchmarking Optimization Software with Performance Profiles". In: *Mathematical Programming* 91 (2002), pp. 201–213.

[62] V. Dolean, P. Jolivet, and F. Nataf. *An Introduction to Domain Decomposition Methods*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2015. DOI: 10.1137/1.9781611974065. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611974065. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611974065.

[63] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. Van der Vorst. *Numerical Linear Algebra for High-Performance Computers*. Philadelphia: SIAM Press, 1998.

[64] J. Dongarra, M. Faverge, T. Hérault, M. Jacquelin, J. Langou, and Y. Robert. "Hierarchical QR factorization algorithms for multi-core clusters". In: *Parallel Computing* 39.4-5 (2013), pp. 212–232. URL: http://hal.inria.fr/hal-00809770.

[65] I. S. Duff, R. Guivarch, D. Ruiz, and M. Zenadi. "The Augmented Block Cimmino Distributed Method". In: *SIAM Journal on Scientific Computing* 37.3 (2015), A1248–A1269. DOI: 10.1137/140961444. eprint: https://doi.org/10.1137/140961444. URL: https://doi.org/10.1137/140961444.

[66] I. S. Duff and J. K. Reid. "The multifrontal solution of indefinite sparse symmetric linear systems". In: *ACM Transactions On Mathematical Software* 9 (1983), pp. 302–325.

[67] C. Eckart and G. Young. "The approximation of one matrix by another of lower rank". In: *Psychometrika* 1.3 (Sept. 1936), pp. 211–218. ISSN: 1860-0980. DOI: 10.1007/BF02288367. URL: https://doi.org/10.1007/BF02288367.

[68] S. C. Eisenstat and J. W. H. Liu. “A tree-based dataflow model for the unsymmetric multifrontal method”. In: *Electronic Transactions on Numerical Analysis* 21 (2005), pp. 1–19.

[69] S. C. Eisenstat and J. W. H. Liu. “Algorithmic Aspects of Elimination Trees for Sparse Unsymmetric Matrices”. In: *SIAM Journal on Matrix Analysis and Applications* 29.4 (2008), pp. 1363–1381.

[70] S. Ellingsrud, T. Eidesmo, S. Johansen, M. C. Sinha, L. M. MacGregor, and S. Constable. “Remote sensing of hydrocarbon layers by seabed logging (SBL): Results from a cruise offshore Angola”. In: *The Leading Edge* 21.10 (2002), pp. 972–982. DOI: 10.1190/1.1518433. eprint: https://doi.org/10.1190/1.1518433. URL: https://doi.org/10.1190/1.1518433.

[71] B. Engquist and L. Ying. “Sweeping preconditioner for the Helmholtz equation: Hierarchical matrix representation”. In: *Communications on Pure and Applied Mathematics* 64.5 (2011), pp. 697–735. ISSN: 1097-0312. DOI: 10.1002/cpa.20358. URL: http://dx.doi.org/10.1002/cpa.20358.

[72] L. Eyraud-Dubois, L. Marchal, O. Sinnen, and F. Vivien. “Parallel Scheduling of Task Trees with Limited Memory”. In: *ACM Trans. Parallel Comput.* 2.2 (June 2015), 13:1–13:37. ISSN: 2329-4949. DOI: 10.1145/2779052. URL: http://doi.acm.org/10.1145/2779052.

[73] A. Geist and E. G. Ng. “Task scheduling for parallel sparse Cholesky factorization”. In: *Int J. Parallel Programming* 18 (1989), pp. 291–314.

[74] A. J. George. “Nested dissection of a regular finite-element mesh”. In: *SIAM J. Numer. Anal.* 10.2 (1973), pp. 345–363.

[75] A. J. George and M. T. Heath. “Solution of Sparse Linear Least Squares Problems Using Givens Rotations”. In: *Linear Algebra and its Applications* 34 (1980), pp. 69–83.

[76] A. George, J. Liu, and E. Ng. “A Data Structure for Sparse QR and LU Factorizations”. In: *SIAM Journal on Scientific and Statistical Computing* 9.1 (1988), pp. 100–121. DOI: 10.1137/0909008. eprint: https://doi.org/10.1137/0909008. URL: https://doi.org/10.1137/0909008.

[77] A. George and E. Ng. “On the Complexity of Sparse QR and LU Factorization of Finite-Element Matrices”. In: *SIAM Journal on Scientific and Statistical Computing* 9.5 (1988), pp. 849–861. DOI: 10.1137/0909057. eprint: https://doi.org/10.1137/0909057. URL: https://doi.org/10.1137/0909057.

[78] P. Ghysels, X. S. Li, F.-H. Rouet, S. Williams, and A. Napov. “An Efficient Multicore Implementation of a Novel HSS-Structured Multifrontal Solver Using Randomized Sampling”. In: *SIAM Journal on Scientific Computing* 38.5 (2016), S358–S384. DOI: 10.1137/15M1010117. eprint: https://doi.org/10.1137/15M1010117. URL: https://doi.org/10.1137/15M1010117.

[79] J. R. Gilbert and E. G. Ng. “Predicting structure in nonsymmetric sparse matrix factorizations”. In: *Graph Theory and Sparse Matrix Computations*. Ed. by J. G. A. George and J. Liu. Springer-Verlag NY, 1993, pp. 107–140.

[80] J. R. Gilbert, X. S. Li, E. G. Ng, and B. W. Peyton. “Computing Row and Column Counts for Sparse QR and LU Factorization”. In: *BIT Numerical Mathematics* 41.4 (2001), pp. 693–710. ISSN: 1572-9125. DOI: 10.1023/A:1021943902025. URL: http://dx.doi.org/10.1023/A:1021943902025.

[81] J. R. Gilbert and J. W. H. Liu. "Elimination Structures for Unsymmetric Sparse LU Factors". In: *SIAM Journal on Matrix Analysis and Applications* 14.2 (1993), pp. 334–352. DOI: 10.1137/0614024. eprint: https://doi.org/10.1137/0614024. URL: https://doi.org/10.1137/0614024.

[82] A. Gillman, P. Young, and P.-G. Martinsson. "A direct solver with $\mathcal{O}(N)$ complexity for integral equations on one-dimensional domains". In: *Frontiers of Mathematics in China* 7 (2 2012), pp. 217–247. ISSN: 1673-3452.

[83] L. Giraud and A. Haidar. "Parallel algebraic hybrid solvers for large 3D convection-diffusion problems". In: *Numerical Algorithms* 51.2 (June 2009), pp. 151–177. ISSN: 1572-9265. DOI: 10.1007/s11075-008-9248-x. URL: https://doi.org/10.1007/s11075-008-9248-x.

[84] W. Givens. "Computation of Plane Unitary Rotations Transforming a General Matrix to Triangular Form". English. In: *Journal of the Society for Industrial and Applied Mathematics* 6.1 (1958), pp. 26–50. ISSN: 03684245. URL: http://www.jstor.org/stable/2098861.

[85] G. Golub. "Numerical methods for solving linear least squares problems". English. In: *Numerische Mathematik* 7.3 (1965), pp. 206–216. ISSN: 0029-599X. DOI: 10.1007/BF01436075. URL: http://dx.doi.org/10.1007/BF01436075.

[86] G. H. Golub and C. F. Van Loan. *Matrix Computations. 4th ed.* Baltimore, MD.: Johns Hopkins Press, 2012.

[87] L. Grasedyck, R. Kriemann, and S. Le Borne. "Parallel black box $\mathcal{H}$-LU preconditioning for elliptic boundary value problems". In: *Computing and Visualization in Science* 11.4 (2008), pp. 273–291. ISSN: 1433-0369. DOI: 10.1007/s00791-008-0098-9. URL: http://dx.doi.org/10.1007/s00791-008-0098-9.

[88] A. Guermouche and J. Y. L'excellent. "Constructing memory-minimizing schedules for multifrontal methods". In: *ACM Transactions on Mathematical Software (TOMS)* 32.1 (2006), pp. 17–32.

[89] A. Guermouche, J.-Y. L'Excellent, and G. Utard. "Impact of Reordering on the Memory of a Multifrontal Solver". In: *Parallel Computing* 29.9 (2003), pp. 1191–1218.

[90] A. Gupta. "A Shared- and distributed-memory parallel general sparse direct solver". In: *Appl. Algebra Eng. Commun. Comput.* 18.3 (2007), pp. 263–277.

[91] M. H. Gutknecht. "Variants of BICGSTAB for matrices with complex spectrum". In: *SIAM Journal on Scientific Computing* 14.5 (1993), pp. 1020–1033.

[92] T. M. Habashy and A. Abubakar. "A general framework for constraint minimization for the inversion of electromagnetic measurements". In: *Progress in electromagnetics Research* 46 (2004), pp. 265–312.

[93] W. Hackbusch, B. N. Khoromskij, and R. Kriemann. "Hierarchical Matrices Based on a Weak Admissibility Criterion". English. In: *Computing* 73.3 (2004), pp. 207–243. ISSN: 0010-485X. DOI: 10.1007/s00607-004-0080-4. URL: http://dx.doi.org/10.1007/s00607-004-0080-4.

[94] W. Hackbusch. "A sparse matrix arithmetic based on H-matrices. Part I: introduction to H-matrices". In: *Computing* 62.2 (1999), pp. 89–108. ISSN: 0010-485X. DOI: http://dx.doi.org/10.1007/s006070050015.

[95] W. Hackbusch. *Hierarchical matrices : algorithms and analysis*. Vol. 49. Springer series in computational mathematics. Berlin: Springer, 2015, pp. xxv, 511. DOI: 10.1007/978-3-662-47324-5.

[96] A. Haidar, A. Abdelfattah, M. Zounon, S. Tomov, and J. Dongarra. "A Guide For Achieving High Performance With Very Small Matrices On GPU: A case Study of Batched LU and Cholesky Factorizations". In: *IEEE Transactions on Parallel and Distributed Systems* PP.99 (2017), pp. 1–1. ISSN: 1045-9219. DOI: 10.1109/TPDS.2017.2783929.

[97] N. Halko, P.-G. Martinsson, and J. A. Tropp. "Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions". In: *SIAM Review* 53.2 (2011), pp. 217–288. DOI: 10.1137/090771806. eprint: http://dx.doi.org/10.1137/090771806. URL: http://dx.doi.org/10.1137/090771806.

[98] M. T. Heath. "Some Extensions of an Algorithm for Sparse Linear Least Squares Problems". In: *SIAM Journal on Scientific and Statistical Computing* 3.2 (1982), pp. 223–237. DOI: 10.1137/0903014. eprint: http://dx.doi.org/10.1137/0903014. URL: http://dx.doi.org/10.1137/0903014.

[99] P. Hénon, P. Ramet, and J. Roman. "PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems". In: *Parallel Computing* 28.2 (Jan. 2002), pp. 301–321.

[100] H. P. Hofstee. "Power Efficient Processor Architecture and The Cell Processor". In: *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*. HPCA '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 258–262. ISBN: 0-7695-2275-0. DOI: 10.1109/HPCA.2005.26. URL: http://dx.doi.org/10.1109/HPCA.2005.26.

[101] J. Hogg, J. K. Reid, and J. A. Scott. "Design of a Multicore Sparse Cholesky Factorization Using DAGs". In: *SIAM J. Scientific Computing* 32.6 (2010), pp. 3627–3649.

[102] A. S. Householder. "Unitary Triangularization of a Nonsymmetric Matrix". In: *J. ACM* 5.4 (Oct. 1958), pp. 339–342. ISSN: 0004-5411. DOI: 10.1145/320941.320947. URL: http://doi.acm.org/10.1145/320941.320947.

[103] M. Jacquelin, L. Marchal, Y. Robert, and B. Uçar. "On Optimal Tree Traversals for Sparse Matrix Factorization". In: *Proceedings of 25th International Parallel and Distributed Processing Symposium (IPDPS'11)*. IEEE Computer Society, 2011.

[104] P. Jaysaval, D. Shantsev, and S. de la Kethulle de Ryhove. "Fast multimodel finite-difference controlled-source electromagnetic simulations based on a Schur complement approach". In: *Geophysics* 79.6 (2014), E315–E327.

[105] G. Karypis and V. Kumar. "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs". In: *SIAM J. Sci. Comput.* 20 (1 Dec. 1998), pp. 359–392. ISSN: 1064-8275. DOI: http://dx.doi.org/10.1137/S1064827595287997. URL: http://dx.doi.org/10.1137/S1064827595287997.

[106] K. Key. "Marine Electromagnetic Studies of Seafloor Resources and Tectonics". In: *Surveys in Geophysics* 33.1 (2012), pp. 135–167. ISSN: 1573-0956. DOI: 10.1007/s10712-011-9139-x. URL: http://dx.doi.org/10.1007/s10712-011-9139-x.

[107] A. Kleen. *An NUMA API for Linux*. Tech. rep. SUSE Labs, 2004.

[108] J. Kurzak and J. Dongarra. "Fully Dynamic Scheduler for Numerical Computing on Multicore Processors". In: *LAPACK working note* lawn220 (2009).

[109] J.-Y. L'Excellent. "Multifrontal methods for large sparse systems of linear equations: parallelism, memory usage, performance optimization and numerical issues". Habilitation. École Normale Supérieure de Lyon, 2012.

[110] J.-Y. L'Excellent and M. W. Sid-Lakhdar. "A study of shared-memory parallelism in a multifrontal solver". In: *Parallel Computing* 40.3-4 (2014), pp. 34–46.

[111] X. Lacoste. "Scheduling and memory optimizations for sparse direct solver on multi-core/multi-GPU cluster systems". PhD thesis. Talence, France: LaBRI, Université Bordeaux, Feb. 2015. URL: http://www.labri.fr/~ramet/restricted/these_lacoste_preprint.pdf.

[112] X. S. Li and J. W. Demmel. "Making Sparse Gaussian Elimination Scalable by Static Pivoting". In: *Supercomputing, 1998.SC98. IEEE/ACM Conference on*. Nov. 1998, pp. 34–34. DOI: 10.1109/SC.1998.10030.

[113] X. S. Li and J. W. Demmel. "SuperLU_DIST: A Scalable Distributed-memory Sparse Direct Solver for Unsymmetric Linear Systems". In: *ACM Trans. Math. Softw.* 29.2 (June 2003), pp. 110–140. ISSN: 0098-3500. DOI: 10.1145/779359.779361. URL: http://doi.acm.org/10.1145/779359.779361.

[114] E. Liberty, F. Woolfe, P.-G. Martinsson, V. Rokhlin, and M. Tygert. "Randomized algorithms for the low-rank approximation of matrices". In: *Proceedings of the National Academy of Sciences* 104.51 (2007), pp. 20167–20172.

[115] J. W. H. Liu, A. George, and E. Ng. "Communication results for parallel sparse Cholesky factorization on a hypercube". In: *Parallel Computing* 10 (1989), pp. 287–298.

[116] J. W. H. Liu. "An Application of Generalized Tree Pebbling to Sparse Matrix Factorization". In: *SIAM J. Algebraic Discrete Methods* 8.3 (July 1987), pp. 375–395. ISSN: 0196-5212. DOI: 10.1137/0608031. URL: http://dx.doi.org/10.1137/0608031.

[117] J. W. H. Liu. "Modification of the Minimum-degree Algorithm by Multiple Elimination". In: *ACM Trans. Math. Softw.* 11.2 (June 1985), pp. 141–153. ISSN: 0098-3500. DOI: 10.1145/214392.214398. URL: http://doi.acm.org/10.1145/214392.214398.

[118] J. W. H. Liu. "On the storage requirement in the out-of-core multifrontal method for sparse factorization". In: *ACM Transactions On Mathematical Software* 12 (1986), pp. 127–148.

[119] J. W. H. Liu. "The Multifrontal Method for Sparse Matrix Solution: Theory and Practice". In: *SIAM Review* 34.1 (1992), pp. 82–109. DOI: 10.1137/1034004. URL: https://doi.org/10.1137/1034004.

[120] F. Lopez. "Task-based multifrontal QR solver for heterogeneous architectures". PhD thesis. Université Paul Sabatier, 2015. URL: http://www.theses.fr/2015TOU30303/document.

[121] R. Luce and E. G. Ng. "On the Minimum FLOPs Problem in the Sparse Cholesky Factorization". In: *SIAM Journal on Matrix Analysis and Applications* 35.1 (2014), pp. 1–21. DOI: 10.1137/130912438. eprint: https://doi.org/10.1137/130912438. URL: https://doi.org/10.1137/130912438.

[122] K. Marfurt. "Accuracy of finite-difference and finite-element modeling of the scalar and elastic wave equations". In: *Geophysics* 49 (1984), pp. 533–549.

[123] P. G. Martinsson. "A Fast Randomized Algorithm for Computing a Hierarchically Semiseparable Representation of a Matrix". In: *SIAM Journal on Matrix Analysis and Applications* 32.4 (2011), pp. 1251–1274. DOI: 10.1137/100786617. eprint: https://doi.org/10.1137/100786617. URL: https://doi.org/10.1137/100786617.

[124] P. G. Martinsson. "Compressing Rank-Structured Matrices via Randomized Sampling". In: *SIAM Journal on Scientific Computing* 38.4 (2016), A1959–A1986. DOI: 10.1137/15M1016679. eprint: https://doi.org/10.1137/15M1016679. URL: https://doi.org/10.1137/15M1016679.

[125] T. Mary. "Block Low-Rank multifrontal solvers: complexity, performance, and scalability". PhD thesis. Toulouse, France: EDMITT, Université Paul Sabatier, Nov. 2017.

[126] P. Matstoms. *Parallel Sparse QR factorization on shared memory architectures*. Tech. rep. LiTH-MAT-R-1993-18. Department of Mathematics, 1993.

[127] J. D. McCalpin. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. Tech. rep. Charlottesville, Virginia: University of Virginia, 1991-2007. URL: http://www.cs.virginia.edu/stream/.

[128] P. J. Mucci, S. Browne, C. Deane, and G. Ho. *PAPI: A Portable Interface to Hardware Performance Counters*. Proceedings of Department of Defense HPCMP Users Group Conference. 1999.

[129] W. Mulder. "A multigrid solver for 3D electromagnetic diffusion". In: *Geophysical prospecting* 54.5 (2006), pp. 633–649.

[130] S. Operto, A. Miniussi, R. Brossier, L. Combe, L. Métivier, V. Monteiller, A. Ribodetti, and J. Virieux. "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 (2015), pp. 1362–1391.

[131] S. Operto, J. Virieux, P. R. Amestoy, J.-Y. L'Excellent, L. Giraud, and H. Ben Hadj Ali. "3D finite-difference frequency-domain modeling of visco-acoustic wave propagation using a massively parallel direct solver: A feasibility study". In: *Geophysics* 72.5 (2007), pp. 195–211. DOI: 10.1190/1.2759835. URL: http://link.aip.org/link/?GPY/72/SM195/1.

[132] J. Peiró and S. Sherwin. "Finite difference, finite element and finite volume methods for partial differential equations". In: *Handbook of materials modeling*. Springer, 2005, pp. 2415–2446.

[133] F. Pellegrini and J. Roman. "Scotch: A Software Package for Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs". In: *Proceedings of HPCN'96, Brussels,* LNCS 1067. Apr. 1996, pp. 493–498.

[134] G. Pichon, E. Darve, M. Faverge, P. Ramet, and J. Roman. "Sparse Supernodal Solver Using Block Low-Rank Compression". In: *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. May 2017, pp. 1138–1147. DOI: 10.1109/IPDPSW.2017.86.

[135] D. J. Pierce and J. G. Lewis. "Sparse Multifrontal Rank Revealing QR Factorization". In: *SIAM J. Matrix Anal. Appl.* 18.1 (1997), pp. 159–180. ISSN: 0895-4798. DOI: http://dx.doi.org/10.1137/S0895479893244353.

[136] A. Pothen and C. Sun. "A mapping algorithm for parallel sparse Cholesky factorization". In: *SIAM Journal on Scientific Computing* 14 (1993), pp. 1253–1253.

[137] S. G. N. Prasanna and B. R. Musicus. "Generalized multiprocessor scheduling and applications to matrix computations". In: *IEEE Transactions on Parallel and Distributed Systems* 7.6 (1996), pp. 650–664.

[138] J. Rigal and J. Gaches. "On the compatibility of a given solution with the data of a linear system". In: *J. Assoc. Comput. Mach.* 14 (1967), pp. 526–543.

[139] D. J. Rose, R. E. Tarjan, and G. S. Lueker. "Algorithmic Aspects of Vertex Elimination on Graphs". In: *SIAM Journal on Computing* 5.2 (1976), pp. 266–283. DOI: 10.1137/0205021. eprint: http://dx.doi.org/10.1137/0205021. URL: http://dx.doi.org/10.1137/0205021.

[140] F.-H. Rouet. "Memory and performance issues in parallel multifrontal factorizations and triangular solutions with sparse right-hand sides". anglais. Thèse de doctorat. Toulouse, France: Institut National Polytechnique de Toulouse, Oct. 2012. URL: http://tel.archives-ouvertes.fr/tel-00785748.

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

[142] E. Schmidt. "Über die Auflösung linearer Gleichungen mit Unendlich vielen unbekannten". German. In: *Rendiconti del Circolo Matematico di Palermo (1884-1940)* 25.1 (1908), pp. 53–77. DOI: 10.1007/BF03029116. URL: http://dx.doi.org/10.1007/BF03029116.

[143] R. Schreiber and C. Van Loan. "A storage-efficient WY representation for products of Householder transformations". In: *SIAM J. Sci. Stat. Comput.* 10 (1989), pp. 52–57.

[144] R. Schreiber. "A new implementation of sparse Gaussian elimination". In: *ACM Transactions On Mathematical Software* 8 (1982), pp. 256–276.

[145] M. Sergent, D. Goudin, S. Thibault, and O. Aumage. "Controlling the Memory Subscription of Distributed Applications with a Task-Based Runtime System". In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. May 2016, pp. 318–327. DOI: 10.1109/IPDPSW.2016.105.

[146] R. Sethi. "Complete Register Allocation Problems". In: *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*. STOC '73. Austin, Texas, USA: ACM, 1973, pp. 182–195. DOI: 10.1145/800125.804049. URL: http://doi.acm.org/10.1145/800125.804049.

[147] R. Sethi and J. D. Ullman. "The Generation of Optimal Code for Arithmetic Expressions". In: *J. ACM* 17.4 (Oct. 1970), pp. 715–728. ISSN: 0004-5411. DOI: 10.1145/321607.321620. URL: http://doi.acm.org/10.1145/321607.321620.

[148] W. M. Sid-Lakhdar. "Scaling multifrontal methods for the solution of large sparse linear systems on hybrid shared-distributed memory architectures". Ph.D. dissertation. ENS Lyon, Dec. 2014.

[149] T. Slavova. "Parallel triangular solution in the out-of-core multifrontal approach for solving large sparse linear systems". Available as CERFACS Report TH/-PA/09/59. Ph.D. dissertation. Institut National Polytechnique de Toulouse, Apr. 2009.

[150] E. Solomonik and J. Demmel. "Communication-optimal Parallel 2.5D Matrix Multiplication and LU Factorization Algorithms". In: *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*. Euro-Par'11. Bordeaux, France: Springer-Verlag, 2011, pp. 90–109. ISBN: 978-3-642-23396-8. URL: http://dl.acm.org/citation.cfm?id=2033408.2033420.

[151] A. Tarantola. "Inversion of seismic reflection data in the acoustic approximation". In: *Geophysics* 49.8 (1984), pp. 1259–1266.

[152] A. N. Tikhonov. "Regularization of incorrectly posed problems". In: *Soviet Math. Dokl.* 4 (1963), pp. 1624–1627.

[153] H. Topcuouglu, S. Hariri, and M.-Y. Wu. "Performance-effective and low-complexity task scheduling for heterogeneous computing". In: *IEEE Transactions on Parallel and Distributed Systems* 13.3 (Mar. 2002), pp. 260–274. ISSN: 1045-9219. DOI: 10.1109/71.993206.

[154] L. N. Trefethen and D. Bau. *Numerical Linear Algebra*. SIAM, 1997. ISBN: 0898713617.

[155] H. A. Van der Vorst. "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 (1992), pp. 631–644.

[156] J. Virieux and S. Operto. "An overview of full waveform inversion in exploration geophysics". In: *Geophysics* 74.6 (2009), WCC1–WCC26.

[157] S. Wang, X. S. Li, F.-H. Rouet, J. Xia, and M. V. De Hoop. "A Parallel Geometric Multifrontal Solver Using Hierarchically Semiseparable Structure". In: *ACM Transactions on Mathematical Software* 42.3 (May 2016), 21:1–21:21. ISSN: 0098-3500. DOI: 10.1145/2830569. URL: http://doi.acm.org/10.1145/2830569.

[158] S. Wang, X. S. Li, F.-H. Rouet, J. Xia, and M. Van de Hoop. "A Parallel Geometric Multifrontal Solver Using Hierarchically Semiseparable Structure". In: *Submitted to ACM Trans. Math. Softw.* (2013).

[159] C. Weisbecker. "Improving multifrontal solvers by means of algebraic Block Low-Rank representations". PhD thesis. Toulouse, France: Institut National Polytechnique de Toulouse, Oct. 2013. URL: http://ethesis.inp-toulouse.fr/archive/00002471/01/thesis.pdf.

[160] J. H. Wilkinson. *Rounding Errors in Algebraic Processes*. Englewood Cliffs, New Jersey: Prentice-Hall, 1963.

[161] S. Williams, A. Waterman, and D. Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Commun. ACM* 52.4 (Apr. 2009), pp. 65–76. ISSN: 0001-0782. DOI: 10.1145/1498765.1498785. URL: http://doi.acm.org/10.1145/1498765.1498785.

[162] W. Wu, A. Bouteiller, G. Bosilca, M. Faverge, and J. Dongarra. "Hierarchical DAG scheduling for Hybrid Distributed Systems". In: *29th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. Hyderabad, India, May 2015, pp. 156–165.

[163] J. Xia. "Efficient Structured Multifrontal Factorization for General Large Sparse Matrices". In: *SIAM Journal on Scientific Computing* 35.2 (2013), A832–A860. DOI: 10.1137/120867032. eprint: http://dx.doi.org/10.1137/120867032. URL: http://dx.doi.org/10.1137/120867032.

[164] J. Xia, S. Chandrasekaran, M. Gu, and X. S. Li. "Superfast Multifrontal Method for Large Structured Linear Systems of Equations". In: *SIAM Journal on Matrix Analysis and Applications* 31.3 (2009), pp. 1382–1411. DOI: 10.1137/09074543X. URL: http://link.aip.org/link/?SML/31/1382/1.

[165] M. Yannakakis. "Computing the Minimum Fill-In is NP-Complete". In: *SIAM Journal on Algebraic Discrete Methods* 2.1 (1981), pp. 77–79. DOI: 10.1137/0602010. eprint: https://doi.org/10.1137/0602010. URL: https://doi.org/10.1137/0602010.

[166] A. YarKhan, J. Kurzak, and J. Dongarra. *QUARK Users' Guide: QUeueing And Runtime for Kernels*. Tech. rep. Innovative Computing Laboratory, University of Tennessee, 2011.

Appendix A

Experimental setup

A.1 Matrices

Tables A.1 and A.2 list the matrices used for the experiments presented in Chapters 3, 4 and 5.

#	Mat. name	Source	m	n	nz	Ordering	op. count (Gflops)
0	image_interp	UFMC	240000	120000	711683	COLAMD	30.2
1	LargeRegFile	UFMC	2111154	801374	4944201	COLAMD	84.7
2	cont11_l	UFMC	1468599	1961395	5382999	COLAMD	184.5
3	EternityII_E	UFMC	11077	262144	1572792	COLAMD	544.0
4	degme	UFMC	185,501	659415	8127528	COLAMD	591.9
5	cat_ears_4_4	UFMC	19020	44448	132888	COLAMD	716.1
6	Hirlam	UFMC	1385270	452200	2713200	COLAMD	2339.9
7	e18	UFMC	24617	38602	156466	COLAMD	3399.5
8	flower_7_4	UFMC	27693	67,593	202218	COLAMD	4261.2
9	Rucci1	UFMC	1977885	109900	7791168	COLAMD	12768.5
10	sls	UFMC	1748122	62729	6804304	COLAMD	22716.2
11	TF17	UFMC	38132	48630	586218	COLAMD	38203.1
12	hirlam	Hirlam	1385270	452200	2713200	SCOTCH	1384
13	flower_8_4	UFMC	55081	125361	375266	SCOTCH	2851
14	Rucci1	UFMC	1977885	109900	7791168	SCOTCH	5671
15	ch8-8-b3	UFMC	117600	18816	470400	SCOTCH	10709
16	GL7d24	UFMC	21074	105054	593892	SCOTCH	16467
17	neos2	UFMC	132568	134128	685087	SCOTCH	20170
18	spal_004	UFMC	10203	321696	46168124	SCOTCH	30335
19	n4c6-b6	UFMC	104115	51813	728805	SCOTCH	62245
20	sls	UFMC	1748122	62729	6804304	SCOTCH	65607
21	TF18	UFMC	95368	123867	1597545	SCOTCH	194472
22	lp_nug30	UFMC	95368	123867	1597545	SCOTCH	221644
23	mk13-b5	UFMC	135135	270270	810810	SCOTCH	259751

Table A.1: Complete set of matrices for the experiments in Chapter 3.

application	matrix	ID	arith.	fact.	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 (SSMC)	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 (SSMC)	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 (SSMC)	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 (SSMC)	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 A.2: Complete set of matrices for the experiments in Section 5.5 and their Full-Rank statistics: order (n), number of nonzeros (nnz), number of operations for the factorization (flops), memory required to store the factor entries (factor size), and arithmetic (c=single complex, z=double complex, d=double real).

A.2 Computers

The following list describes the systems used for the experiment presented in Chapters 3, 4 and 5:

- `brutus`: this system is equipped with two octo-core AMD Opteron 8214 processors clocked at 2.2 GHz for a total of 16 cores and a peak performance of 70.4 Gflop/s for real, double precision computations. It has 65 GB of main memory.
- `dude`: this system is equipped with four hexa-core AMD Istanbul processors clocked at 2.4 GHz. Each of these CPUs has six cores and is attached to a DRAM module through two DRAM controllers and the CPUs are connected to each other through HyperTransport links in a ring layout.
- `vargas`: this machine is made of IBM Power6 p575 nodes. One node of the Vargas supercomputer installed at the IDRIS supercomputing institute (grant x2012065063) is equipped with 16 dual-core Power6 processors clocked at 4.7 GHz. Processors are grouped in sets of four called MCMs (Multi Chip Module) and each node has four MCMs for a total of 32 cores. Processors in an MCM are fully connected as well as MCMs in a node.
- `ada`: This is an IBM x3750-M4 system installed at the IDRIS supercomputing center (grant 2014-i2014065063) equipped with four Intel Sandy Bridge E5-4650 (eight

cores) processors and 128 GB of memory per node. The cores are clocked at 2.7 GHz and are equipped with Intel AVX SIMD units; the peak performance is of 21.6 Gflop/s per core and thus 691.2 Gflop/s per node for real, double precision computations.

- `sirocco`: This is a five nodes cluster part of the PlaFRIM center. Each node is equipped with two Haswell Intel Xeon E5-2680 (twelve cores) processors and 124 GB of memory per node. The cores are clocked at 2.5 GHz and are equipped with Intel AVX SIMD units. In addition, each node is accelerated with four Nvidia K40M GPUs; the peak performance is of 40.0 Gflop/s per core, 1.4 Tflop/s per GPU and thus 6.0 Tflop/s per node for real, double precision computations.
- `brunch`: a shared-memory machine installed at the LIP laboratory of ENS-Lyon equipped with 1.5 TB of memory and four Intel 24-cores Broadwell E7-8890v4 processors running at a frequency varying between 2.2 and 3.4 GHz, due to the turbo technology.
- `eos`: the supercomputer of the Calcul en Midi-Pyrénées (CALMIP) center (grant P0989, since 2008). 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.
- `licallo`: the supercomputer of the SIGAMM mesocenter in 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.
- `farad`: a shared-memory machine equipped with 264 GB of memory and two Intel 16-cores Sandy Bridge processors running at 2.9 GHz.