



HAL
open science

Multifrontal Methods: Parallelism, Memory Usage and Numerical Aspects

Jean-Yves L'Excellent

► **To cite this version:**

Jean-Yves L'Excellent. Multifrontal Methods: Parallelism, Memory Usage and Numerical Aspects. Modeling and Simulation. Ecole normale supérieure de lyon - ENS LYON, 2012. tel-00737751v1

HAL Id: tel-00737751

<https://theses.hal.science/tel-00737751v1>

Submitted on 2 Oct 2012 (v1), last revised 21 Dec 2012 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ÉCOLE NORMALE SUPÉRIEURE DE LYON, FRANCE

Multifrontal Methods: Parallelism, Memory Usage and Numerical Aspects

Jean-Yves L'Excellent

Chargé de Recherche, Inria

MÉMOIRE D'HABILITATION À DIRIGER DES RECHERCHES

présenté et soutenu publiquement le 25 septembre 2012.

Rapporteurs :

Tim DAVIS,	Professeur des universités,	University of Florida, Gainesville, Etats-Unis
Jocelyne ERHEL,	Directeur de recherche,	Inria Rennes Bretagne Atlantique
John GILBERT,	Professeur des universités,	University of California, Santa Barbara, Etats-Unis

Jury :

Tim DAVIS,	Professeur des universités,	University of Florida, Gainesville, Etats-Unis
Iain DUFF,	Professeur des universités,	Rutherford Appleton Laboratory, Angleterre
John GILBERT,	Professeur des universités,	University of California, Santa Barbara, Etats-Unis
Yves ROBERT,	Professeur des universités,	École Normale Supérieure de Lyon
Jean VIRIEUX,	Professeur des universités,	Université Joseph Fourier, Grenoble I

Résumé :

La résolution de systèmes linéaires creux est critique dans de nombreux domaines de la simulation numérique. Beaucoup d'applications, notamment industrielles, utilisent des méthodes directes en raison de leur précision et de leur robustesse. La qualité du résultat, les fonctionnalités numériques, ainsi que le temps de calcul sont critiques pour les applications. Par ailleurs, les ressources matérielles (nombre de processeurs, mémoire) doivent être utilisées de manière optimale. Dans cette habilitation, nous décrivons des travaux poursuivant ces objectifs dans le cadre de la plateforme logicielle MUMPS, développée à Toulouse, Lyon-Grenoble et Bordeaux depuis une quinzaine d'années. Le cœur de l'approche repose sur une parallélisation originale de la méthode multifrontale : une gestion asynchrone du parallélisme, associée à des ordonnanceurs distribués, permet de traiter des structures de données dynamiques et autorise ainsi le pivotage numérique. Nous nous intéressons à l'ordonnancement des tâches, à l'optimisation de la mémoire et à différentes fonctionnalités numériques. Les travaux en cours et les objectifs futurs visent à résoudre efficacement des problèmes de plus en plus gros, sans perte sur les aspects numériques, et tout en adaptant nos approches aux évolutions rapides des calculateurs. Dans ce contexte, les aspects génie logiciel et transfert deviennent critiques afin de maintenir sur le long terme une plateforme logicielle comme MUMPS. Cette plateforme est à la fois nécessaire à nos travaux de recherche et utilisée en production ; elle maximise ainsi les retours applicatifs qui valident nos travaux et permettent d'orienter nos recherches futures.

Mots-clés :

Matrices creuses, méthodes multifrontales, systèmes linéaires, solveurs directs, ordonnancement.

Abstract:

Direct methods for the solution of sparse systems of linear equations are used in a wide range of numerical simulation applications. Such methods are based on the decomposition of the matrix into the product of triangular factors, followed by triangular solves. In comparison to iterative methods, they are known for their numerical accuracy and robustness. However, they are also characterized by a high memory consumption (especially for 3D problems) and a large amount of computations. The quality of the computed solution, the numerical functionalities and the computation time are essential parameters, while the use of material resources (number of processors and memory usage) must be carefully optimized. In this *habilitation* thesis, we describe some work to pursue these objectives in the context of the sparse direct solver MUMPS, developed in Toulouse, Lyon-Grenoble and Bordeaux. The approach relies on an original parallelization of the multifrontal method for distributed-memory machines, in which an asynchronous management of parallelism associated with distributed scheduling algorithms allows for dynamic datastructures and numerical pivoting. We consider task scheduling, optimization of the memory usage, and various numerical functionalities. On-going and future work aim at efficiently solving problems that are always bigger, while maintaining numerical stability and adapting our approaches to the quick evolutions of computer platforms: increase of the number of computing nodes, increase of the number of cores per node, but decrease of memory per core. In this context, software engineering and technology transfer aspects become critical in order to maintain in the long term a software package like MUMPS. This software is both necessary to our research and widely used in industry, maximizing feedback that validates our work and provides future work directions.

Keywords:

Sparse matrices, multifrontal methods, linear systems, direct solvers.

Acknowledgements

I am grateful to Tim Davis and John Gilbert for having agreed to report on this work, and for having undertaken such a long trip to be part of my jury. It has been a big honour to have you in Lyon. I also thank Jocelyne Erhel for reporting on this work; I am sorry you were not able to attend the defense in the end. Many thanks to Iain Duff, one of the pioneers in this research field, for having accepted to come to Lyon and preside over my jury. Thanks to Jean Virieux for his participation to my jury and for providing the applications' point of view. Thanks to Yves Robert for participating to my jury and helping with the organization of the defense.

I wish to thank my team leaders at Inria, Frédéric Desprez and Frédéric Vivien, who allowed me to pursue this project. I also thank Patrick Amestoy for having given me the motivation for an academic job and for our fruitful and long-term collaboration on this project. Many thanks to all MUMPS contributors. I especially thank my current close “MUMPS” colleagues, Emmanuel Agullo, Maurice Brémond, Alfredo Buttari, Abdou Guermouche, Guillaume Joslin, Chiara Puglisi, François-Henry Rouet, Mohamed Sid-Lakhdar, Bora Uçar and Clément Weisbecker, for their support for this habilitation.

I am grateful to my wife and family for their constant support.

Contents

Introduction	1
1 General Background	5
1.1 A practical example	5
1.1.1 Dense factorizations	5
1.1.2 Sparse factorization and fill-in	6
1.1.3 The multifrontal method	6
1.1.4 Multifrontal solve algorithm	8
1.1.5 Sparse matrices and graphs	9
1.1.6 Supernodes	11
1.1.7 Orderings and permutations	13
1.1.8 Preprocessing with maximum weighted matching and scalings algorithms	14
1.2 Theoretical formalism	16
1.2.1 LU decomposition	17
1.2.2 Fill-in and structure of the factor matrices	17
1.2.3 Elimination graph structures	18
1.2.4 Left-looking, right-looking and multifrontal methods	21
1.3 Practical issues	24
1.3.1 Three-phase approach to solve $Ax = b$	24
1.3.2 Numerical accuracy and pivoting	24
1.3.2.1 Unsymmetric case	25
1.3.2.2 Symmetric case	26
1.3.2.3 LINPACK vs. LAPACK styles of pivoting	28
1.3.2.4 Static pivoting and iterative refinement	28
1.3.3 Memory management in the multifrontal method	28
1.3.4 Out-of-core approaches	31
2 A Parallel Multifrontal Method for Distributed-Memory Environments	33
2.1 Adapting the multifrontal factorization to a parallel distributed environment	33
2.1.1 Sources of parallelism	33
2.1.1.1 Description of type 2 parallelism	34
2.1.1.2 Description of type 3 parallelism	35
2.1.2 Asynchronous communication issues	36
2.1.3 Assembly process	37
2.1.4 Factorization of type 1 nodes	39
2.1.5 Parallel factorization of type 2 nodes	40
2.1.6 Discussion	43
2.2 Pivot selection	44
2.2.1 Pivoting and stability issues in a parallel asynchronous context	44
2.2.2 Detection of null pivots and rank-revealing	46

2.2.3	Pivoting and out-of-core	46
2.2.4	Algorithm for pivot selection (symmetric indefinite case)	46
2.3	Schur complement	47
2.4	Solution phase	49
2.5	Reduced/condensed right-hand side in solve phase	53
2.6	Determinant	55
2.6.1	Numerical aspects: avoiding underflows and overflows	55
2.6.2	Computing the sign of the determinant	56
2.6.3	Special cases	56
2.6.4	Reduction in parallel environments	57
2.6.5	Testing	58
2.6.6	Complex arithmetic	58
2.6.7	Memory aspects	59
2.7	Concluding remarks	59
3	Task Scheduling for the Serial Multifrontal Method	61
3.1	Introduction – Tree traversals and postorders	62
3.2	Models of assembly in the multifrontal method	63
3.3	Postorders to reduce the storage requirements	65
3.3.1	Notations	65
3.3.2	Terminal allocation	66
3.3.2.1	Working storage requirement	66
3.3.2.2	Total storage requirement (including factors)	67
3.3.2.3	Liu’s theorem and its application to reduce storage requirements	67
3.3.3	Early parent allocation	68
3.3.4	Flexible allocation	69
3.3.4.1	Classical (non in-place) assembly scheme	69
3.3.4.1.1	Working storage minimization	69
3.3.4.1.2	Total storage minimization	70
3.3.4.2	In-place assembly scheme	72
3.3.4.2.1	Working storage minimization	72
3.3.4.2.2	Total storage minimization	73
3.3.4.3	Max-in-place assembly scheme	73
3.3.5	Impact and summary of experimental results	73
3.4	Postorders to reduce the volume of I/O	75
3.4.1	Stacks and <i>I/O</i> volumes	76
3.4.2	Notations	77
3.4.3	Terminal allocation of the parent, classical assembly	78
3.4.3.1	Illustrative example and formal expression of the I/O volume	78
3.4.3.2	Minimizing the I/O volume	80
3.4.4	In-place assembly of the last contribution block	81
3.4.5	In-place assembly of the largest contribution block	81
3.4.6	Theoretical comparison of MinMEM and MinIO	82
3.4.7	Flexible parent allocation	83
3.4.8	Experimental results	85
3.4.8.1	Terminal allocation	85
3.4.8.2	Flexible allocation	86
3.5	Memory management algorithms	88
3.5.1	In-core stack memory	88
3.5.1.1	Recalling the classical and last-in-place assembly schemes	89
3.5.1.2	In-place assembly of the largest contribution block	89
3.5.1.3	Flexible allocation of the frontal matrices	90

3.5.2	Out-of-core stacks	91
3.5.2.1	Dynamic cyclic memory management	91
3.5.2.2	Using information from the analysis: static top-down formulation	92
3.5.2.3	Application to the flexible allocation scheme	94
3.5.3	Limits of the models	96
3.6	Concluding remarks	96
4	Task Scheduling in Parallel Distributed-Memory Environments	97
4.1	Obtaining accurate load estimates	97
4.1.1	Maintaining a distributed view of the load	98
4.1.1.1	Naive mechanism	98
4.1.1.2	Mechanism based on load increments	99
4.1.1.3	Reducing the number of messages	99
4.1.2	Exact algorithm	101
4.1.3	Experiments	104
4.1.3.1	Memory-based scheduling strategy	105
4.1.3.2	Workload-based scheduling strategy	106
4.1.4	Concluding remarks	108
4.2	Hybrid static-dynamic mapping and scheduling strategies	109
4.2.1	History – PARASOL project (1996-1999)	109
4.2.2	Improvements (1999-2001)	112
4.2.3	Static tree mapping: candidate processors	112
4.2.3.1	Preamble: proportional mapping	113
4.2.3.2	Main ideas of the mapping algorithm with candidates	113
4.2.4	Scheduling for clusters of SMP nodes	114
4.2.5	Memory-based dynamic scheduling	114
4.2.6	Hybrid scheduling	116
4.3	Memory scalability issues and memory-aware scheduling	117
5	A Parallel Out-of-Core Multifrontal Method	121
5.1	A robust out-of-core code with factors on disks	122
5.1.1	Direct and buffered (at the system level) I/O mechanisms	123
5.1.2	Synchronous and asynchronous approaches (at the application level)	124
5.1.3	Testing environment	124
5.1.4	Sequential performance	125
5.1.5	Parallel performance	128
5.1.6	Discussion	129
5.1.7	Panel version	130
5.2	Description and analysis of models to further reduce the memory requirements	130
5.2.1	Models to manage the contribution blocks on disk	131
5.2.2	Analysis of the memory needs of the different schemes	132
5.2.3	Analysing how the memory peaks are reached	133
5.2.4	Summary	134
5.3	Conclusion	134
6	Solving Increasingly Large Problems	137
6.1	Parallel analysis	137
6.2	64-bit addressing	139
6.3	Forward elimination during factorization	140
6.4	Memory and performance improvements of the solve algorithms (forward and backward substitutions)	142
6.4.1	Reduction of workspace and locality issues	142

6.4.2	Performance	144
6.4.3	Discussion	148
6.5	Communication buffers	151
6.6	Facing the multicore evolutions	153
6.6.1	Multithreaded BLAS libraries and OpenMP: preliminary experiments based on the fork-join model	153
6.6.2	OpenMP: optimization of a symmetric factorization kernel	157
6.6.2.1	Performance bottleneck observed	157
6.6.2.2	Improvements of the factorization algorithm for the pivot block	159
6.6.2.3	Optimization of pivot search algorithm	160
6.6.2.4	Discussion	160
6.6.3	OpenMP: exploiting the parallelism resulting from the assembly tree	161
6.7	Sparse direct solvers and grid computing	162
6.8	Conclusion and other work directions	163
7	Conclusion	167

Introduction

We consider the solution of

$$Ax = b, \tag{1}$$

where A is a large sparse square matrix (typically several millions of equations), and x and b are vectors or matrices. A and b are given and x is the unknown. Such systems of linear equations arise in a wide range of scientific computing applications from various fields, in relation with numerical simulation: finite element methods, finite difference methods, or numerical optimization. There are two main classes of methods to solve such problems: *direct methods*, based on a factorization of A in, for example, the form LU , LDL^T , or QR ; and *iterative methods*, in which sparse matrix-vector products are used to build a series of iterates, hopefully converging to the solution. Direct methods are more robust and are often preferred in industrial applications. Among direct methods, multifrontal methods build the factorization of a sparse matrix by performing partial factorizations of smaller dense matrices. But multifrontal methods, and direct methods in general, require a larger amount of memory than iterative methods, because the factors of a sparse matrix have a higher density of nonzero elements than the original matrix. Furthermore, they result in an additional computational complexity especially in three-dimensional (3D) problems. This can be illustrated by a simple 7-point finite difference stencil for a 3D Laplacian equation. If N is the number of discretization points in each direction, the matrix A has N^3 rows and N^3 columns, with about $7N^3$ nonzero elements. When nested dissections are used to reorder the variables of the problem and limit the size of the factors, the number of entries in the factor matrix is $O(N^4)$ and the number of floating-point operations for the factorization is $O(N^6)$ (see [42], for example). As a comparison, the matrix-vector product performed at each iteration of an iterative method is $O(N^3)$.

However, thanks to the increasing capacity (storage, performance) of modern supercomputers and thanks to significant progresses in direct methods, with robust software packages available ([3],[59],[116],[124], ...), direct methods are often preferred by applications, even for 3D problems of several millions of equations. For such systems, the size of the graph representing the dependencies between the computations can be huge. In this context, it is critical but difficult to maintain a good numerical stability while making use of parallelism and targetting high performance. We focus in this thesis on robust and efficient parallel algorithms, taking into account the following objectives and constraints:

1. Large-scale high performance machines generally have their memory physically distributed. Therefore, the message-passing paradigm should be used. In the case of SMP¹ nodes or multicores, the shared-memory paradigm can also be used at the node level or at the multicore level.
2. Matrix factorizations are much more stable when numerical pivoting is allowed. Therefore, numerical pivoting should be used even in a parallel distributed environment. One difficulty with sparse matrices is that numerical pivoting results in a modification of the task dependency graph, which can then not be fully known beforehand. Furthermore, the stability of a pivot must be checked against other entries in the matrix, which is not trivial because it often requires costly interprocessor communications.
3. Architecture evolutions make it very hard or even impossible to perfectly predict the behaviour of current computing platforms, with several levels of caches, high performance redundant networks with

¹Symmetric MultiProcessors: a uniform memory shared by the different processors in the node.

a complex topology, some degree of heterogeneity between processors or at the network level, varying load in possibly multi-user environments, etc. Furthermore, the characteristics of our graph of tasks are not fully predictable when numerical pivoting is allowed (see previous point). Therefore, it is worth considering adaptive approaches, with so-called *dynamic scheduling*. In practice, we will show that static information could/should be used to reduce the space to be explored by the dynamic schedulers.

4. There are often time constraints in numerical simulation. Therefore, it is critical that the algorithms reduce the time for solution (or *makespan*) as much as possible on a given number of processors.
5. If the problem is large, a special attention must be paid to memory which is, if not correctly used, a very strong limiting factor for parallel direct methods. The order in which the tasks are scheduled has a strong impact not only on the makespan, but also on memory and unfortunately, more parallelism often leads to more global memory used. Therefore, it is crucial to take these two conflicting criteria into account when designing scheduling strategies for the computational tasks.
6. When memory is not large enough on the target machine, *out-of-core* approaches are necessary, in which the disk is used as the next level of memory hierarchy. In that case, the algorithms must decide what to write and when to write it. This generates I/O traffic and a special attention should be paid to limit it.
7. Parallelism from the graph of tasks obtained (a tree in our context) is generally not sufficient, and the largest tasks should themselves be parallelized (this has been sometimes been called mixed parallelism [50, 66]).

1: **Initialization:**

- 2: pool \leftarrow my share of the initial ready tasks
- 3: **while** (Global termination not detected) **do**
- 4: **if** a message of type *state information* is ready to be received **then**
- 5: Receive the message (load information, memory information...);
- 6: Update the load and memory estimates of the relevant processor(s);
- 7: **else if** an application-related message is ready to be received **then**
- 8: Receive and process the message (task, data, ...), typically:
 - reserve some workspace
 - perform some computations
 - decrease a counter keeping track of unmet task dependencies
 - insert a new ready task in the local pool of ready tasks
 - send another application-related message
 - etc.
- 9: **else if** (pool not empty) **then**
- 10: Extract a task T from the pool of ready tasks
- 11: **if** T is big **then**
- 12: Assign parts of T to other processors (sending application-related messages);
- 13: Process T in cooperation with the chosen processors (sending asynchronous application-related messages);
- 14: **end if**
- 15: **end if**
 - {Note that state information might have been sent, depending on local load or local memory variations}
- 16: **end while**

Algorithm 0.1: Asynchronous approach retained (simplified).

With these objectives in mind, we have designed a fully asynchronous approach [24, 26] to process the graph of tasks during the factorization. The approach is based on message passing and the general scheme is

the one of Algorithm 0.1. A pool of ready tasks is maintained on each processor: an initial (static) mapping defines the local pool of tasks for each processor (tasks that can start without dependency), and a new task is inserted in one of the local pools when all its dependencies have been met. When a task is large, some other processors are assigned to help with that task, dynamically, depending on an estimate of the *state* of the other processors. Overall, there are two types of dynamic (distributed) scheduling decisions: (i) the selection of a new ready task from the local pool, line 10; (ii) the subdivision of the task and its mapping on some other processors, line 12. The processor responsible of a task will be called *master* for that task, while the processors chosen dynamically to help are called *slave* processors. In order for this scheme to work efficiently, two main questions must be answered, and will be discussed later in this document:

1. How to map and schedule the tasks and which decisions should be static and dynamic?
2. How to maintain distributed estimates of the state information (memory, workload) of the processors?

Even in serial environments, scheduling is crucial and impacts the amount and locality of the temporary data produced: each task produces temporary data that is consumed only when the task that depends on it is activated. This motivates theoretical studies on the order in which a graph of tasks should be processed to either:

- minimize the memory usage due to these temporary data, or
- minimize the I/O traffic, in the case those temporary data do not fit in the physical memory.

The work presented in this document is motivated by the needs from applications. An external use of our algorithms on large-scale academic and industrial applications is essential to validate our research, and to get feedback, which in turn motivates new research directions. In this context, an important aspect of this activity consists in making available the results of research under the form of a software package, MUMPS², enabling users to solve new problems efficiently and accurately in various fields related to numerical simulation. This involves activities such as software engineering, development, validation, support and maintenance. Furthermore, each new functionality or research work has to be thought in a global software context, and not just for one particular combination of other functionalities. For example, the detection of null pivot rows/columns that we have recently implemented (for automatic detection of *rigid modes* or for FETI-like methods [86]) should work on the internal 1D, possibly out-of-core, parallel asynchronous pipelined factorizations, in symmetric and unsymmetric cases, including for cases where part of the column is not available on the processor in charge of the pivot selection. Therefore, when a new algorithm is designed and validated, a significant amount of work is needed to make it compatible with a large range of functionalities and thus available to a large range of applications.

Historically, this work on parallel multifrontal methods in distributed environments and the associated software developments that led to the successive versions of the research platform MUMPS were initiated in the scope of a European project called PARASOL (Long Term Research, Esprit IV framework, 1996-1999) and was inspired by an experimental prototype of an unsymmetric multifrontal code for distributed-memory machines using PVM [89] developed by Amestoy and Espirat [85]. That experimental prototype was itself inspired by the code MA41, developed by Amestoy during his PhD thesis [17] at CERFACS under the supervision of Duff. The PARASOL project led to a first public version of the software package MUMPS in 1999. Since then, after my arrival at INRIA in 2001, the research and developments have been mainly supported by CERFACS, CNRS, ENS Lyon, INPT(ENSEEIH)-IRIT, INRIA and University of Bordeaux.

This document describes some aspects of the work of an overall project, which is the result of a team work with many contributors over the years, and was the object of many collaborations, PhD thesis and projects among which we can cite:

- a central collaboration with Patrick Amestoy from INPT(ENSEEIH)-IRIT since 1996,

²See <http://graal.ens-lyon.fr/MUMPS> or <http://mumps.enseeiht.fr>.

- the PhD thesis of Abdou Guermouche (ENS Lyon, 2001-2004), Stéphane Pralet (INPT-CERFACS, 2002-2004), Emmanuel Agullo (ENS Lyon, 2005-2008), Mila Slavova (INPT-CERFACS, 2005-2009), François-Henry Rouet (INPT-IRIT, 2009-2012), Clément Weisbecker (INPT-IRIT, 2010-), Mohamed Sid-Lakhdar (ENS Lyon, 2011-)
- an NSF-INRIA project (2001-2004) aiming at mixing direct and iterative methods,
- the Grid TLSE project (2002-), initially funded by the ACI-Grid programme from the French ministry of research, then by ANR³ projects,
- an Egide-Aurora cooperation with Norway (2004),
- the Solstice project (2007-2010) funded by ANR³ programs,
- the Seiscope consortium (2006-) around Geoscience-Azur,
- contracts with industry: Samtech S.A. (2005-2006 and 2008-2010) and CERFACS/CNES (2005),
- two France-Berkeley projects (1999-2000 and 2008-2009),
- a French-Israeli “Multicomputing” project (2008-2010),
- an Action of Technological Development funded by INRIA (2009-2012),
- lots of discussions within the “MUMPS team” and informal collaborations with the academic and industrial users of MUMPS.

The document is organized as follows. Chapter 1 presents some general background on multifrontal methods, and provides the main principles of the approach we rely on. Chapter 2 shows how some algorithms, features and functionalities which appear simple in a sequential environment must be adapted in a parallel asynchronous scheme targeting distributed-memory environments. Chapter 3 discusses the problem of task scheduling when working with a limited memory in a serial environment. Two aspects are considered: (i) how to reduce the memory requirements of multifrontal methods? (ii) how to use the disk as a secondary storage and minimize I/O? Chapter 4 summarizes the major evolutions of the algorithms responsible of task mapping and scheduling in parallel distributed-memory environments. In Chapter 5, the use of disk-storage for problems where memory is not sufficient is discussed, while Chapter 6 shows that several other algorithmic and software bottlenecks also had to be tackled in order to process real-life large-scale problems efficiently. Finally to conclude we give some medium-term perspectives, together with their expected impact on applications.

³Agence Nationale de la Recherche.

Chapter 1

General Background

In this chapter, we introduce some general concepts related to sparse direct solvers, and in particular multi-frontal methods, that will be used or referred to in this document. We first adopt a practical point of view driven by a running example (Section 1.1) before introducing some theoretical formalism in Section 1.2. We discuss other practical related issues in Section 1.3.

1.1 A practical example

1.1.1 Dense factorizations

Suppose that we want to solve a system of linear equations of the form $Ax = b$, where A is a square non-singular matrix of order n , b is the right-hand side, and x is the unknown. We take an example where the structure of the matrix is symmetric, and the numerical values are not, as illustrated by Equation (1.1).

$$\begin{pmatrix} 2 & 0 & 0 & 2 & 1 \\ 0 & 1 & -1 & 0 & 0 \\ 0 & -1 & 0 & -2 & 0 \\ 4 & 0 & 2 & 14 & 0 \\ -6 & 0 & 0 & 0 & -2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} 5 \\ 1 \\ -2 \\ 6 \\ -12 \end{pmatrix} \quad (1.1)$$

In dense linear algebra, this can be done by applying a variant of Gaussian elimination. The method first factorizes the matrix of the system (referred to as A) under the form $A = LU$, where L is lower triangular with ones on the diagonal and U is upper triangular, as shown in Equation 1.2. This can be done using Algorithm 1.1. At each step, a pivot is eliminated, the column of L is computed, and a rank-one update is performed. Such an algorithm is said to be *right-looking* because after the elimination of each pivot, we only modify the right-bottom part of the matrix, without accessing again the already computed factors.

$$\begin{pmatrix} 2 & 0 & 0 & 2 & 1 \\ 0 & 1 & -1 & 0 & 0 \\ 0 & -1 & 0 & -2 & 0 \\ 4 & 0 & 2 & 14 & 0 \\ -6 & 0 & 0 & 0 & -2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 \\ 2 & 0 & -2 & 1 & 0 \\ -3 & 0 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} 2 & 0 & 0 & 2 & 1 \\ 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & -1 & -2 & 0 \\ 0 & 0 & 0 & 6 & -2 \\ 0 & 0 & 0 & 0 & 3 \end{pmatrix} \quad (1.2)$$

Once the LU factorization of A is obtained, one can solve the system $Ax = b$ in two steps:

- The *forward elimination*, consisting in solving $Ly = b$ for y . In our case we obtain $y = (5, 1, -1, -6, 9)^T$.
- The *backward substitution*, consisting in solving $Ux = y$ for x , finally leading to the solution of our system of equations $x = (1, 2, 1, 0, 3)^T$.


```

1: for  $i = 1$  to  $n$  do
2:    $L(i : i) = 1$ 
3:    $L(i + 1 : n, i) = \frac{A(i+1:n,i)}{A(i,i)}$ 
4:    $U(i, i : n) = A(i, i : n)$ 
5:    $A(i + 1 : n, i + 1 : n) = A(i + 1 : n, i + 1 : n) - L(i + 1 : n, i) \times A(i, i + 1 : n)$ 
6: end for

```

Algorithm 1.1: Factorization of a dense matrix A of order n under the form LU .

In dense matrix computations, L and U generally overwrite the matrix A , avoiding the use of additional storage for L and U . This results in Algorithm 1.2, where in the end A has been overwritten by $(L - I) + U$ (the ones on the diagonal of L are not stored). If the matrix is symmetric, an LDL^T factorization is computed instead. More variants can be found in the literature, see for example [96].

```

1: for  $i = 1$  to  $n$  do
2:    $A(i + 1 : n, i) = \frac{A(i+1:n,i)}{A(i,i)}$ 
3:    $A(i + 1 : n, i + 1 : n) = A(i + 1 : n, i + 1 : n) - A(i + 1 : n, i) \times A(i, i + 1 : n)$ 
4: end for

```

Algorithm 1.2: Dense LU factorization overwriting matrix A .

1.1.2 Sparse factorization and fill-in

In order to exploit the sparsity of our initial matrix A , we want to avoid the storage of and computations on zeros. If we want to exploit the zeros of our matrix in the application of Algorithm 1.1, line 3 should only consider the nonzero entries from the lower part of column i , that is $A(i + 1 : n, i)$; furthermore, the rank-one update at line 5 should only use the nonzero entries in vectors $L(i + 1 : n, i)$ and $A(i, i + 1 : n)$. One way to organize the operations only on nonzeros and manage the associated data structures is the so-called *multifrontal method*, which will be central to this thesis and introduced in Section 1.1.3.

Obviously, except in case of numerical cancellation, if an entry a_{ij} is nonzero in the original matrix, l_{ij} (if $i > j$) or u_{ij} (if $j \geq i$) will also be nonzero. However, even if an entry is zero in the original matrix A , the corresponding entry can be nonzero in the factors, and this phenomenon is known as *fill-in*. This is illustrated in the example of Equation (1.2): both a_{45} and a_{54} are 0 in A but both l_{54} and u_{45} are nonzeros in L and U , respectively¹. Those nonzeros first appeared when performing the rank-one update associated to pivot 1. In order to limit the amount of fill-in, the order in which the variables are eliminated is critical, as explained in Section 1.1.7.

1.1.3 The multifrontal method

In the multifrontal method, instead of modifying directly the entries in A , we use dense matrices to perform and store the rank-one updates. Let us illustrate this process for the rank-one update of the first pivot. Starting from A we build a temporary matrix whose structure results from the nonzeros in the first row/column (indices 1, 4 and 5 of the original matrix) as follows:

$$A_1^{(145)} = \begin{pmatrix} 2 & 2 & 1 \\ 4 & 0 & 0 \\ -6 & 0 & 0 \end{pmatrix} \quad (1.3)$$

The numbers in parentheses refer to the indices of the variables in the numbering of the original matrix. We now perform the factorization of column 1 and the subtraction of the rank-one update inside that dense

¹This also occurs for the diagonal element in position (3,3). However, we consider that we will not try to exploit zeros on the diagonal.

matrix, avoiding some indirections that would arise by directly working on the sparse matrix A . In fact, this can be viewed as the application of the first step of Algorithm 1.2 to $A_1^{(145)}$, a dense 3×3 matrix. After this operation, $A_1^{(145)}$ is modified and overwritten by:

$$F_1^{(145)} = \begin{pmatrix} 2 & 2 & 1 \\ 2 & -4 & -2 \\ -3 & 6 & 3 \end{pmatrix} \quad (1.4)$$

The matrix in which $A_1^{(145)}$, and then $F_1^{(145)}$ is stored is called the *frontal matrix*, or *front*, associated with the first pivot (we also call it *pivot 1*): $A_1^{(145)}$ is the assembled front and $F_1^{(145)}$ is the partially factorized front. The Schur complement produced at the bottom-right part of F_1 , $CB_1^{(45)} = \begin{pmatrix} -4 & -2 \\ 6 & 3 \end{pmatrix}$, is called the *contribution block* of $F_1^{(145)}$ associated with variables 4 and 5. It will be used later during the factorization process, in order to update some rows and columns in the matrix before variables 4 and 5 may be eliminated. The first row and the first column of $F_1^{(145)}$ correspond to the factors associated with variable 1: row 1 of U consists of 2, 2, and 1; column 1 of L consists of 1 (implicit), 2 and -3 .

Similarly, we have for pivot 2

$$A_2^{(23)} = \begin{pmatrix} 1 & -1 \\ -1 & 0 \end{pmatrix} \text{ and } F_2^{(23)} = \begin{pmatrix} 1 & -1 \\ -1 & -1 \end{pmatrix} \quad (1.5)$$

involving variables 2 and 3 in the numbering of the original matrix. $CB_2^{(3)} = (-1)$ is the contribution block of F_2 associated with variable 3.

Let us now define the *arrowhead* associated with a pivot p as the set of nonzero entries of the original matrix A that are part of $A(p : n, p) \cup A(p, p + 1 : n)$ (part of the row and column of the pivot that are not factorized yet).

We consider the elimination of pivot 3 and extract the arrowhead of 3 to obtain the submatrix $\begin{pmatrix} 0 & -2 \\ 2 & 0 \end{pmatrix}$ associated with variables 3 and 4. Because the elimination of pivot 2 has an impact on the value of pivot 3, we add into this matrix the update $CB_2^{(3)}$ due to the elimination of pivot 2 and computed at step 2 and obtain the assembled front:

$$A_3^{(34)} = \begin{pmatrix} -1 & -2 \\ 2 & 0 \end{pmatrix}. \quad (1.6)$$

The first row and first column of $A_3^{(34)}$ are said to be *fully-summed* because all possible updates from previous pivot eliminations have been incorporated into them. Therefore, pivot 3 can be factorized, leading to the factorized frontal matrix

$$F_3^{(34)} = \begin{pmatrix} -1 & -2 \\ -2 & -4 \end{pmatrix} \text{ and } CB_3^{(4)} = (-4). \quad (1.7)$$

Then, notice that the elimination of pivot 4 involves variable 5 because of the fill-in coming from the elimination of pivot 1 and available in the contribution $CB_1^{(45)}$:

$$A_4^{(45)} = \begin{pmatrix} 14 & 0 \\ 0 & 0 \end{pmatrix} + CB_1^{(45)} + CB_3^{(4)} = \begin{pmatrix} 6 & -2 \\ 6 & 3 \end{pmatrix} \quad (1.8)$$

In the summation above, $CB_3^{(4)}$ is a 1×1 matrix only concerned by variable 4. Since variable 4 corresponds to the first variable of the resulting matrix, the unique element of $CB_3^{(4)}$ is summed at position (1,1) of that resulting matrix.

It follows after factorization of pivot 4 in $A_4^{(45)}$:

$$F_4^{(45)} = \begin{pmatrix} 6 & -2 \\ 1 & 5 \end{pmatrix} \text{ and } CB_4^{(5)} = (5) \quad (1.9)$$

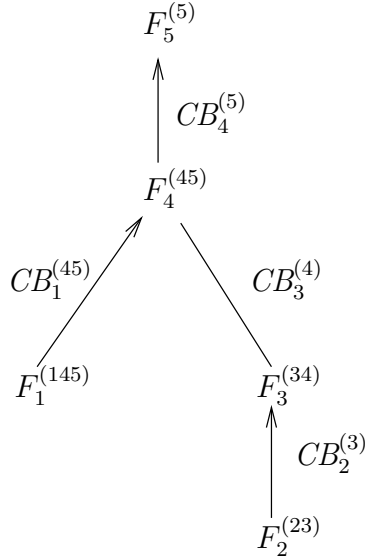


Figure 1.1: Dependencies between pivot eliminations when applying the multifrontal method to the 5×5 matrix of Equation (1.1). F_i represents the i th frontal matrix and the i th node.

Finally, $A_5^{(5)} = -2 + 5 = 3 = F_5^{(5)}$ and the factorization is complete. A closer look at the dependencies during this factorization shows the order of computations follows the tree of Figure 1.1. This tree is called *elimination tree* [128]; we will come back to this notion later. For the moment, note that at each node of the tree, we have computed one column of L (the diagonal element is equal to 1 and is not stored explicitly) and one row of U . In comparison to Equation (1.1) and Algorithm 1.1, zeros are not stored and operations on zeros have not been performed. This is at the cost of indirections during the assembly of each frontal matrix, where for each pivot p , we had to:

- build the symbolic structure of the frontal matrix associated with p ,
- assemble nonzero entries from row $A(p, p : n)$ and column $A(p + 1 : n, p)$ of the initial matrix, forming the *arrowhead* of variable p , into F_p ,
- assemble the contribution blocks CB_j into F_p , for all children j of p ; this operation is also called an *extend-add* operation, and it requires indirections²,
- eliminate pivot p , building the p^{th} row and p^{th} column of the factors and (except for the last pivot – or root) a contribution CB_p used at the parent node.

1.1.4 Multifrontal solve algorithm

The solve algorithm consists in a forward elimination, where the triangular system $Ly = b$ is solved to obtain an intermediate right-hand side, followed by a backward substitution, where the triangular system $Ux = y$ is then solved to obtain the solution x . In the multifrontal method, the columns of L and the rows of U are scattered in the elimination tree and are parts of the matrices F_i (see Figure 1.1). During the forward elimination, the tree is processed from bottom to top. At each step, part of the solution y is computed and the right-hand side b is then modified using the partial computed solution, as shown in Algorithm 1.3.

Applying this algorithm to our example can be decomposed as below, following the nodes of the tree:

²For example, in Equation (1.8), the $+$ operator must be considered as an extend-add operation rather than a simple summation.

```

for  $i = 1$  to  $n$  do
  {Work on front  $i$ }
  Solve  $l_{ii}y_i = b_i$  for  $y_i$ 
  for all nonzeros  $l_{ki}$  in column  $i$  of  $L$ ,  $k > i$  do
    {Update right-hand side of trailing subsystem: }
     $b_k \leftarrow b_k - l_{ki}y_i$ 
  end for
end for

```

Algorithm 1.3: Algorithm for the forward elimination. Remark that, in our case, l_{ii} is equal to one for all i .

- Node 1: $y_1 = b_1 = 5$, $b_4 \leftarrow b_4 - l_{41}y_1 = -4$, $b_5 \leftarrow b_5 - l_{51}y_1 = 3$
- Node 2: $y_2 = b_2 = 1$, $b_3 \leftarrow b_3 - l_{32}y_2 = -1$
- Node 3: $y_3 = b_3 = -1$, $b_4 \leftarrow b_4 - l_{43}y_3 = -6$
- Node 4: $y_4 = b_4 = -6$, $b_5 \leftarrow b_5 - l_{54}y_4 = 9$
- Node 5: $y_5 = b_5 = 9$.

Concerning the backward substitution $Ux = y$, the solution is first computed at the root node: $x_5 = u_{55}^{-1}y_5 = 3$, then equation $u_{44}x_4 + u_{45}x_5 = y_4$ results in $x_4 = 0$ at node 4. x_1 and x_3 can then be computed independently using the equations $u_{11}x_1 + u_{14}x_4 + u_{15}x_5 = y_1$ and $u_{33}x_3 + u_{34}x_4 = y_3$, respectively associated with nodes 1 and 3 of the tree. Finally, x_2 is obtained from x_3 using the equation $u_{22}x_2 + u_{23}x_3 = y_2$ (node 2), leading to $x = (1 \ 2 \ 1 \ 0 \ 3)^T$. The process is summarized in Algorithm 1.4.

```

for  $i = n$  downto  $1$  do
  {Compute known part of equation  $i$ }
   $\alpha \leftarrow 0$ 
  for all nonzeros  $u_{ik}$  in row  $i$  of  $U$ ,  $k > i$  do
     $\alpha \leftarrow \alpha + u_{ik}x_k$ 
  end for
  Solve  $u_{ii}x_i = y_i - \alpha$  for  $x_i$ 
end for

```

Algorithm 1.4: Algorithm for the backward substitution.

It is important to notice that, at each step of the algorithm, only the factors at each node are used, preserving the data organization from the factorization. The algorithm naturally extends to multiple right-hand sides.

1.1.5 Sparse matrices and graphs

A graph $G = (V, E)$ is associated with a matrix A with symmetric pattern in such a way that

- $V = 1, \dots, n$ represents the list of variables of the graph, where n is the number of variables of the matrix;
- an edge (i, j) , $i \neq j$ from i to j belongs to E if a_{ij} is nonzero. Note that edges (i, i) are not considered.

In general, such a graph is *directed*. However, for matrices with a symmetric structure, one may consider undirected edges $\{i, j\}$, together with an *undirected* graph. In Figure 1.2, we represent the undirected graph associated with the matrix of Equation (1.1), together with the graph associated with the matrix of factors $(L + U)$.

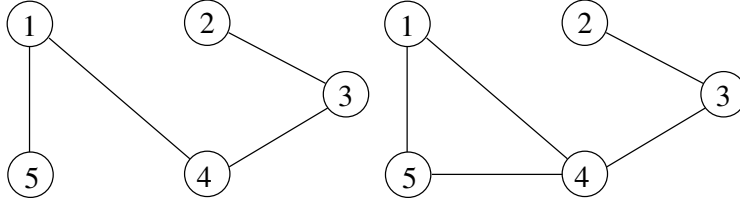


Figure 1.2: Graph of the example matrix A (left) and graph of the filled matrix $L + U$ (right). Fill-in has appeared between variables 4 and 5, corresponding to the elimination of pivot 1, then no other fill occurs during the elimination of pivots 2, 3, 4 and 5.

Initialization: $G_0 = (E_0, V_0)$ is the graph associated with A

for $i = 1$ **to** n **do**

$E_i \leftarrow E_{i-1}$

Add edges to E_i to make all neighbours of i in G_{i-1} pairwise adjacent

Remove edges adjacent to i from E_i

$V_i \leftarrow V_{i-1} \setminus \{i\} = \{i + 1 \dots n\}$

$G_i \stackrel{def}{=} (E_i, V_i)$ is the elimination graph after step i

end for

Algorithm 1.5: Elimination graphs associated with the factorization of a matrix A .

Following the concept of *elimination graphs* introduced in [138] and starting from the graph associated with A , the rank-one update associated with each pivot results in the addition of a clique between the nodes that were adjacent to the eliminated pivot: every two vertices among the neighbours of the pivot become connected by an edge. In our example, the clique introduced by the elimination of pivot 1 is a simple edge between nodes 4 and 5, corresponding to the fill between variables 4 and 5 in the factors. This is followed by the elimination of the current pivot and its adjacent edges. Starting from the graph of the original matrix $G_0 = (E_0, V_0)$, the process is described by Algorithm 1.5, where at each step, a graph G_i is built in which variable i has been eliminated. (Remark that G_n is empty). The *filled graph* is the graph $G^+ = (V_0, \bigcup_{i=0}^n E_i)$ resulting from all the fill-in that appeared during the factorization. It is also the graph associated with the matrix of factors $L + U$ (see Figure 1.2, right).

The notion of *elimination tree* can be deduced from the graph of the filled matrix (Figure 1.2, right) by:

- replacing undirected edges by directed edges following the order of elimination of the variables (1 2 3 4 5), and
- suppressing unnecessary dependencies; in the example, edge $1 \rightarrow 5$ is suppressed because of the existence of the edges $1 \rightarrow 4$ and $4 \rightarrow 5$. We keep the edges $1 \rightarrow 4$, $2 \rightarrow 3$, $3 \rightarrow 4$, $1 \rightarrow 4$ and $4 \rightarrow 5$.

In terms of graphs, remark that this corresponds to computing a so called *transitive reduction* of the directed graph associated to U (or L^T). In our example, the elimination tree defined this way exactly corresponds to the tree of Figure 1.1. In the multifrontal approach, the fact that edge $1 \rightarrow 5$ is suppressed corresponds to the fact that the contributions of variable 1 on variable 5, available in $CB_1^{(45)}$ are passed from node 1 to node 5 not directly but *via* node 4. In practice, computing the full filled graph and its transitive reduction would be too costly so that other techniques must be applied (see for example [128]).

Finally, the notion of *quotient graph model*, first introduced for sparse symmetric matrix factorizations by [91] is very useful to both save storage and limit the computational complexity associated with the construction of elimination graphs. The sequence of elimination graphs is replaced by a sequence of quotient graphs. With quotient graphs, the clique information is implicitly represented by the pivot, which, instead of being eliminated and suppressed from the graph, becomes a special node sometimes called *element* (see [19]).

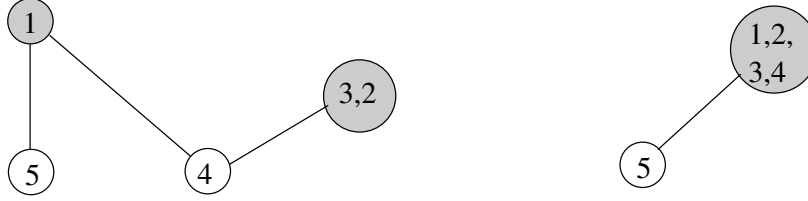


Figure 1.3: Quotient graphs after elimination of pivots 1, 2, and 3 (left) and after elimination of pivots 1, 2, 3, and 4 (right). On the left, the fill (clique) between 4 and 5 is implicitly represented by element 1.

Each new element absorbs the already formed elements adjacent to it: the collapsed elements together with each not yet eliminated variable form a partition of the original graph, to which a quotient graph can thus be associated (duplicate edges are suppressed). In our example, and starting from the graph of Figure 1.2 (left), the sequence of quotient graphs corresponding to the pivot eliminations is obtained as follows:

1. After elimination of pivot 1, $\{1\}$ becomes an element.
2. After elimination of pivot 2, $\{2\}$ becomes an element.
3. After elimination of pivot 3, $\{3\}$ becomes an element and absorbs $\{2\}$ because there was an edge between element $\{2\}$ and variable 3; one obtains the quotient graph of Figure 1.3 (left).
4. After elimination of pivot 4, $\{4\}$ becomes an element which absorbs element $\{3, 2\}$ and element $\{1\}$, that were adjacent to 4 (see Figure 1.3, right).
5. After elimination of pivot 5, $\{5\}$ absorbs element $\{4, 1, 3, 2\}$.

The quotient graph model allows an efficient computation of the tree structure: each time an element i is absorbed by an element j , j becomes the parent of i . For example, after the elimination of pivot 4, because $\{4\}$ absorbs elements $\{3, 2\}$ and $\{1\}$, 4 is the parent of 1 and 3 (where 3 became the parent of 2 at the moment of building element $\{3, 2\}$).

1.1.6 Supernodes

Still considering symmetric matrices, a *supernode* [41] is a contiguous range of columns in the matrix of factors having the same lower diagonal nonzero structure. More precisely, in the graph associated with the matrix of factors (directed graph associated with L^T , or U), a supernode is such that the nodes associated with the matrix variables form a clique and have the same outgoing edges outside the clique. In our example, variables 4 and 5 can be amalgamated into a single supernode, leading to a 2×2 frontal matrix $F_{45}^{(45)}$ instead of the last two frontal matrices of the tree in Figure 1.1. The multifrontal method applies as before except that:

- both the arrowheads associated with variables 4 and 5 must be copied into the frontal matrix F_{45} ;
- both variables 4 and 5 are eliminated in the frontal matrix associated with the supernode (4, 5), instead of one variable for the frontal matrix of 4 and one variable for the frontal matrix of 5.

The notion of elimination tree is replaced by a so called *assembly tree*, as shown in Figure 1.4. Remark that we could also have defined a larger supernode by amalgamating variables 1, 4, and 5, suppressing another frontal matrix. This also illustrates that there is not a unique way to perform amalgamation and that this process might be different to enhance vectorization and/or parallelism. At each supernode, instead of eliminating just one variable, all the variables defined by the supernode are eliminated. In our example, the amalgamated node was the root node, but more generally, a frontal matrix has the shape of Figure 1.5. At

each frontal matrix, fully-summed block is factorized and the non fully-summed block (contribution block or Schur complement) is updated. The algorithms for the solve phase also naturally extend to supernodes, working on blocks of columns of L (resp. blocks of rows of U) during the forward (respectively backward) substitution.

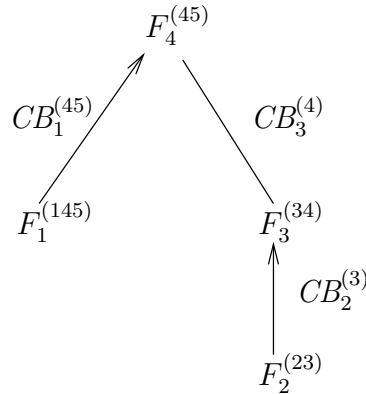


Figure 1.4: Dependencies between pivot eliminations when applying the multifrontal method to the 5×5 matrix of Equation (1.1). Variables 4 and 5 have been amalgamated into a single supernode.

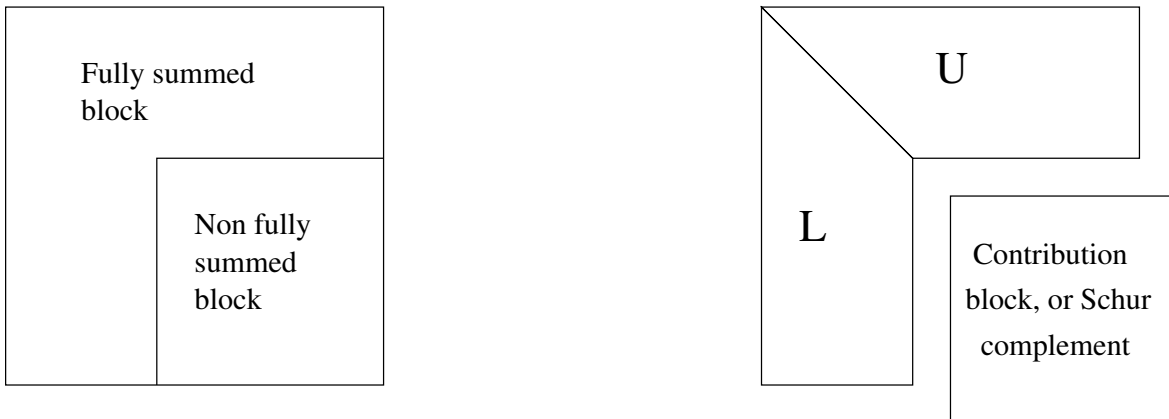


Figure 1.5: Frontal matrix at a node of the tree before (left) and after (right) the partial factorization of the fully-summed block. Unsymmetric case (LU factorization).

Working on supernodes instead of individual variables is essential in order to speed-up the computations and use high-level BLAS [72] (Basic Linear Algebra Subprograms): supernodes lead to a higher flops to memory access ratio, and this allows a better usage of memory hierarchy and better performance thanks to the blocking techniques used in BLAS routines. Typically, BLAS 3 routines (TRSM, GEMM) will be used during the factorization. During the solve stage, BLAS 2 instead of BLAS 1 (respectively BLAS 3 instead of BLAS 2) routines will be used for a single (respectively multiple) right-hand-side vector(s). In practice, it is also worth relaxing the notion of supernodes by amalgamating nodes that introduce some extra computations on zeros: amalgamation reduces the amount of indirections and increases the sizes of the matrices used in BLAS.

1.1.7 Orderings and permutations

Different strategies exist to minimize the amount of fill-in, and thus decrease the amount of computation. Consider a permutation matrix P . The initial system $Ax = b$ and the modified system $(PAP^T)(Px) = (Pb)$ have the same solution x . Consider the permutation matrix P such that PAP^T leads to pivots in the order $(2, 3, 4, 5, 1)$ (second pivot of the original matrix first, third pivot second, etc.). We thus have:

$$P = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}, \text{ and} \tag{1.10}$$

$$PAP^T = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 0 & -2 & 0 & 0 \\ 0 & 2 & 14 & 4 & 0 \\ 0 & 0 & 2 & 2 & 1 \\ 0 & 0 & 0 & -6 & -2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 \\ 0 & -2 & 1 & 0 & 0 \\ 0 & 0 & 0.2 & 1 & 0 \\ 0 & 0 & 0 & -5 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 0 & -1 & -2 & 0 & 0 \\ 0 & 0 & 10 & 4 & 0 \\ 0 & 0 & 0 & 1.2 & 1 \\ 0 & 0 & 0 & 0 & 3 \end{pmatrix} \tag{1.11}$$

Notice that in that case, no fill-in occurs: the structures of L and U are included in the structure of A (not taking into account the diagonal, for which the sparsity will never be exploited). The objective of *reordering techniques* (or *orderings*) is precisely to find a permutation P that limits the amount of fill-in (and number of operations). In general, minimizing the fill-in is NP-complete [166] so heuristics have to be defined. The problem is easier seen in terms of graph, where we associate a graph to a matrix A as explained in Section 1.1.5.

An example of local heuristic to find such a permutation, or *ordering*, is the minimum degree, where at each step of Algorithm 1.5, we eliminate the pivot with smallest degree (or smallest adjacency size) and update the current elimination graph accordingly. The use of quotient graphs allows to run in-place, in the sense that quotient graphs do not require more storage than the initial matrix. The approximate minimum degree (AMD [19]) and the multiple minimum degree (MMD [125]) are variants of the minimum degree that use such an approach to be computationally efficient. The minimum fill algorithm [133, 147] is another local approach which attempts at eliminating at each step the pivot leading to the smallest fill, taking into account already existing edges in the elimination graph at the current step.

For large matrices, nested dissections [92] give better orderings, because they have a more global view of the graph. The way they work is the following: given a node separator S of the graph, and two disconnected domains D_1 and D_2 , the order of the variables in the permuted matrix is chosen to be D_1, D_2, S . No fill can appear between D_1 and D_2 and the zero blocks remain during the factorization. D_1 and D_2 are partitioned recursively leading to a recursive block-bordered structure. Typically, once the subgraphs are small enough, a switch to a local heuristic becomes interesting again. After the switch to a local heuristic, the adjacency with halo variables may be taken into account [140].

There exist other heuristics. For example, the Cuthill-McKee algorithm consists in a breadth-first search traversal of the matrix graph that limits the envelope and bandwidth of the matrix, restricting the fill to this structure. In the breadth-first search, layers are built in such a way that there are no connections between layer i and layer $i + 2$. This way, the band structure can be seen as a tridiagonal block structure where entries in block $(i, i + 1)$ represent the connections between layer i and layer $i + 1$. No fill-in can occur outside this tridiagonal block structure.

As indicated by their name, fill-reducing heuristics target the reduction of fill-in and thus the size of the factors. We indicate in Table 1.1 the size of the factors obtained with the following heuristics, that will be used as representatives of the different classes (local, global, or hybrid) of orderings all along this document:

- AMD: the Approximate Minimum Degree [19];
- AMF: the Approximate Minimum Fill, as implemented in MUMPS by P. Amestoy;

	METIS	SCOTCH	PORD	AMF	AMD
GUPTA2	8.55	<i>12.97</i>	9.77	7.96	8.08
SHIP_003	73.34	79.80	73.57	68.52	<i>91.42</i>
TWOTONE	25.04	25.64	<i>28.38</i>	22.65	22.12
WANG3	7.65	9.74	7.99	8.90	<i>11.48</i>
XENON2	94.93	100.87	107.20	144.32	<i>159.74</i>

Table 1.1: Size of factors (millions of entries) as a function of the ordering heuristic applied. Bold and italic correspond to best and worse orderings, respectively, for each matrix.

	METIS	SCOTCH	PORD	AMF	AMD
GUPTA2	2757.8	4510.7	<i>4993.3</i>	2790.3	2663.9
SHIP_003	83828.2	92614.0	112519.6	96445.2	<i>155725.5</i>
TWOTONE	29120.3	27764.7	<i>37167.4</i>	29847.5	29552.9
WANG3	4313.1	5801.7	5009.9	6318.0	<i>10492.2</i>
XENON2	99273.1	112213.4	126349.7	237451.3	<i>298363.5</i>

Table 1.2: Number of floating-point operations (millions) as a function of the ordering heuristic applied. Bold and italic correspond to best and worse orderings, respectively, for each matrix.

- PORD: a tight coupling of bottom-up and top-down sparse reordering methods [156];
- METIS: we use here the routine `METIS_NODEND` or `METIS_NODEWND` from the METIS package [120], which is an hybrid approach based on multilevel nested dissection and multiple minimum degree;
- SCOTCH: we use a version of `SCOTCH` [139] provided by the author that couples nested dissection and (halo) Approximate Minimum Fill (HAMF), in a way very similar to [141].

Another goal of fill-reducing orderings is to reduce the number of operations and the effect of the above heuristics on the number of floating-point operations is reported in Table 1.2. Let us take the example of an LDL^T factorization with D diagonal and L lower triangular. Noting n_i the number of nonzero elements in column i of L , the size of the factors is $\sum n_i$ and the number of floating-point operations is $\sum(n_i + \frac{n_i \times (n_i + 1)}{2})$, so that the objectives of floating-point operations and factor size reductions are somewhat related, as can be seen in Tables 1.1 and 1.2 for matrices WANG3 and TWOTONE for example, where METIS minimizes both the number of operations and the size of the factors. This is not always the case, for example AMD minimizes the size of the factors for matrix TWOTONE but not the number of operations (minimized with SCOTCH).

Note that although these heuristics mainly target the reduction of fill-in (and thus size of the factors) and the number of operations – see Tables 1.2 and 1.1, they also have a significant impact on the shape of the assembly tree and the parallelism (see, for example, [24]). Figure 1.6 summarizes the characteristics of the trees resulting from different orderings.

1.1.8 Preprocessing with maximum weighted matching and scalings algorithms

Let us assume that A is a nonsingular matrix with an unsymmetric nonzero pattern. Let us now consider a representation of A by a bipartite graph, where one set of nodes represents the rows and the other set represents the columns. An edge between a *row* node and a *column* node exists only if the corresponding entry in A is nonzero.

In that case, finding a maximum cardinality matching in the bipartite graph and permuting the row (or column) nodes accordingly corresponds to permuting the matrix to a zero-free diagonal. Using weighted matching algorithms allows one to obtain large values on the diagonal (by minimizing the sum or the product of diagonal entries). This has several advantages:

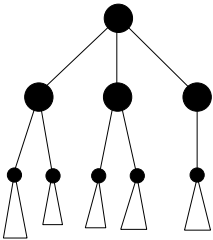
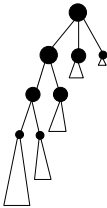
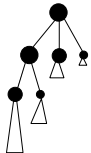
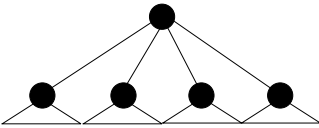
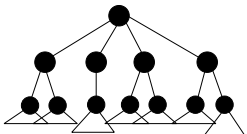
Reordering technique	Shape of the tree	observations
AMD		<ul style="list-style-type: none"> • Deep well-balanced tree • Large frontal matrices on top of the tree
AMF		<ul style="list-style-type: none"> • Very deep unbalanced tree • Small frontal matrices • Very large number of nodes
PORD		<ul style="list-style-type: none"> • deep unbalanced tree • Small frontal matrices • Large number of nodes
SCOTCH		<ul style="list-style-type: none"> • Very wide well-balanced tree • Large frontal matrices • Small number of nodes
METIS		<ul style="list-style-type: none"> • Wide well-balanced tree • Large number of nodes • Smaller frontal matrices (than SCOTCH)

Figure 1.6: Shape of the trees resulting from various reordering techniques.

Matrix		Structural symmetry	$ LU $ (10^6)	Flops (10^9)	Backwd Error
TWO-TONE	OFF	28	235	1221	10^{-6}
	ON	43	22	29	10^{-12}
FIDAPM11	OFF	100	16	10	10^{-10}
	ON	46	28	29	10^{-11}

Table 1.3: Effect of weighted matching algorithms on the factorization of two unsymmetric matrices with MUMPS.

- better numerical properties and improved numerical behaviour: the linear system is easier to solve and the amount of numerical pivoting and of row/column interchanges is limited;
- more predictable data structures; in the extreme case of codes using static data structures, where run-time pivoting is very limited or even not done, many problems cannot be solved without preprocessing based on such column permutations;
- improved reliability of memory estimates;
- reduced amount of computations and fill-in in the factors, especially for approaches working on a symmetrized structure, that is where an explicit zero is introduced at position i, j in cases where a_{ij} is zero and a_{ji} is nonzero.

Duff and Koster [76, 77] provide effective algorithms together with more details on the application and effects of such techniques to sparse Gaussian elimination. Scaling arrays can also be provided on output of the weighted matching algorithm so that the diagonal entries of the permuted scaled matrix are all one, and off-diagonal entries are smaller, also improving the numerical properties. The linear system of Equation (1) becomes:

$$(D_r A Q D_c)(D_c^{-1} Q^T x) = D_r b,$$

where D_r and D_c are diagonal scaling matrices, and Q is a permutation matrix. Fill-reducing heuristics of Section 1.1.7 can then be applied, leading to:

$$(P D_r A Q D_c P^T)(P D_c^{-1} Q^T x) = P D_r b.$$

We illustrate in Table 1.3 the importance of weighted matching algorithms with such scaling on the behaviour of the MUMPS solver on two matrices. In this table, the symmetry is defined as the percentage of nonzero elements (i, j) in the (possibly permuted) matrix for which the element (j, i) is also nonzero. Clearly, the effect is negative when the matrix has a symmetric structure. In [78], Duff and Pralet show how maximum weighted matching algorithms can be useful to symmetric indefinite matrices: on top of the scaling arrays (see Table 1.4), matched entries are candidates for numerically good 2×2 pivots – see Section 1.3.2.2, which the fill-reducing heuristics may then take into account structurally. One way to do that is to force those 2×2 pivots to become supervariables. Significant improvements can be obtained, as shown in Table 1.5 for augmented matrices of the form $K = \begin{pmatrix} H & A \\ A^T & 0 \end{pmatrix}$.

1.2 Theoretical formalism

This section introduces some theoretical formalism and is complementary to the previous section. Reading it is not strictly necessary for the understanding of the rest of this document.

Scaling :	Total time (seconds)		Nb of entries in factors (millions)			
	OFF	ON	(estimated)		(effective)	
			OFF	ON	OFF	ON
CONT-300	45	5	12.2	12.2	32.0	12.4
CVXQP3	1816	28	3.9	3.9	62.4	9.3
STOKES128	3	2	3.0	3.0	5.5	3.3

Table 1.4: Influence of scaling from [78] on augmented matrices. The scaled matrix is DAD , where $D = \sqrt{D_r D_c}$.

Compression :	Total time (seconds)		Nb of entries in factors in Millions			
	OFF	ON	(estimated)		(effective)	
			OFF	ON	OFF	ON
CONT-300	5	4	12.3	11.2	32.0	12.4
CVXQP3	28	11	3.9	7.1	9.3	8.5
STOKES128	1	2	3.0	5.7	3.4	5.7

Table 1.5: Influence of preselecting 2×2 pivots (with scaling).

1.2.1 LU decomposition

Given a sparse system of linear equations $Ax = b$, where $A = (a_{ij})_{1 \leq i, j \leq n}$ is a sparse matrix of order n , and x and b are column vectors, we admit that a decomposition $A = LU$ exists if the matrix is invertible (non-singular) even if in general it means swapping some columns. In this decomposition L is a lower triangular matrix whose diagonal values are equal to 1 and U is an upper triangular matrix.

Matrices L and U verify: $a_{ij} = (LU)_{ij}$, $1 \leq i, j \leq n$. Considering the respective triangular structures of L and U , we can write: $a_{ij} = \sum_{k=1}^{\min(i,j)} l_{ik} u_{kj}$. Because $l_{ii} \equiv 1$, we have:

$$\begin{cases} a_{ij} = \sum_{k=1}^{i-1} l_{ik} u_{kj} + u_{ij} & \text{if } i \leq j \\ a_{ij} = \sum_{k=1}^{j-1} l_{ik} u_{kj} + l_{ij} u_{jj} & \text{if } i > j \end{cases} \quad (1.12)$$

We deduce the following expression of the factors (remark that the notations i and k have been swapped):

$$I_j \begin{cases} u_{kj} = a_{kj} - \sum_{i=1}^{k-1} l_{ki} u_{ij} & \text{for } k = 1, \dots, j \\ l_{kj} = \frac{1}{u_{jj}} (a_{kj} - \sum_{i=1}^{j-1} l_{ki} u_{ij}) & \text{for } k = j+1, \dots, n \end{cases} \quad (1.13)$$

which allows their computation by iteratively applying I_j for $j = 1$ to n . We present in Figure 1.7 the data involved during an iteration. Computations are performed column by column. Each column j (terms u_{kj} or l_{kj} in Formula (1.13)) depends on the columns i , $i = 1, \dots, j-1$ (term l_{ki} in Formula (1.13)).

Depending on the properties of the matrix, the decomposition can be simplified: an LDL^T decomposition can be performed with a symmetric matrix and a LL^T decomposition (or Cholesky decomposition) with a symmetric positive-definite matrix. In this thesis, we will focus on the factorization step. Although we will present results related to LU , LDL^T and LL^T factorizations, we will use the LU formulation (except when stated otherwise) to present the concepts that are common to the three methods.

1.2.2 Fill-in and structure of the factor matrices

Usually the L and U factors have more nonzero values than the original matrix A : expression (1.13) shows that a nonzero entry l_{ij} (or u_{ij}) in the factors can appear even if a_{ij} is equal to 0, when there exists k in

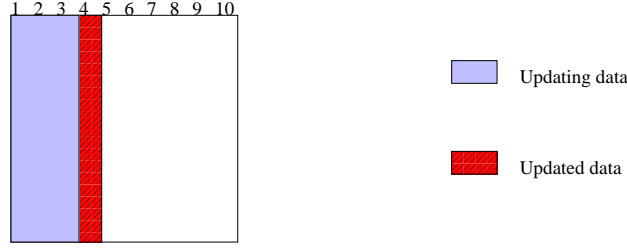


Figure 1.7: Iteration I_4 : the elements of column 4 are updated using the elements of columns 1, 2 and 3.

$\{1, \dots, \min\{i, j\}\}$ such that l_{ik} and u_{kj} are nonzeros. This phenomenon is known as *fill-in* and is illustrated by Figure 1.8.

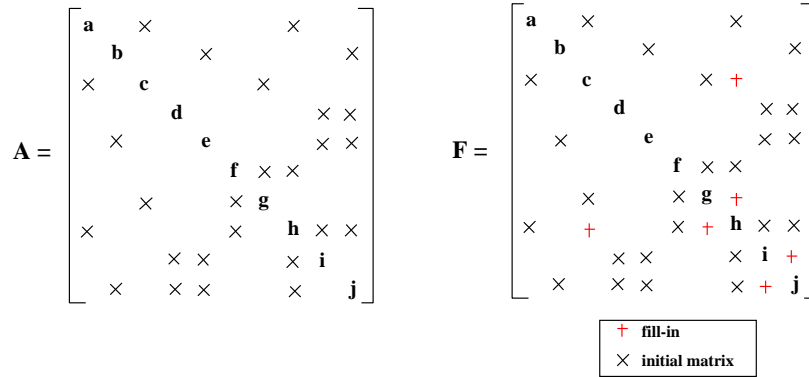


Figure 1.8: Fill-in. After factorization, matrix F represents the nonzero values l_{ij} (for $i > j$) and u_{ij} values (for $i \leq j$). Formally, and since $l_{ii} = 1$, $F = L + U - I$.

1.2.3 Elimination graph structures

Algorithms on the structure of sparse matrices can be viewed as operations on graphs since the structure of a general sparse matrix is equivalent to a graph. Let $\mathcal{G}(A)$ be the directed graph of a sparse matrix A (with nonzero diagonal entries) as follows. The vertex set of $\mathcal{G}(A)$, the graph associated with A , is $V = \{1, 2, \dots, n\}$ and there is an edge (i, j) from i to j (for $i \neq j$) if and only if the entry a_{ij} is nonzero.

In Gaussian elimination, the sparse structure of the factors depends on the order of elimination of the variables. However the elimination of a column does not impact all the following columns but only part of them, depending on their respective sparse structures. Said differently, the computation of some columns may be independent of the computation of some other columns. The study of these dependencies between columns is essential in sparse direct factorization as they are used to manage several phases of the factorization [128, 93]. Formula (1.13) provides these dependencies that we express as a binary relation \rightarrow on the set of columns $\{1, \dots, n\}$ in Definition 1.1:

Definition 1.1. *Column j explicitly depends on column i (noted $i \rightarrow j$) if and only if there exists some k in $\{i + 1, \dots, n\}$ such that $l_{ki}u_{kj} \neq 0$.*

The transitive closure $\overset{\pm}{\rightarrow}$ of \rightarrow expresses whether a column i must be computed before a column j : $i \overset{\pm}{\rightarrow} j$ if and only if column i must be computed before column j . This information can be synthesized with a transitive reduction $\overset{\pm}{\rightarrow}$ of \rightarrow (or of $\overset{\pm}{\rightarrow}$): column i must be computed before column j if and only

if there is a path in the directed graph associated with $\bar{\rightarrow}$ from i to j . This statement would be true for any of the relations \rightarrow , $\overset{+}{\rightarrow}$ or $\bar{\rightarrow}$, but $\bar{\rightarrow}$ presents the advantage to be the most compact way to code this information [14].

The graph associated with $\bar{\rightarrow}$ reflects some available freedom to reorder the variables without changing the sparsity structure of the factors. Because the dependencies respect the initial ordering ($i \rightarrow j$ implies that $i < j$), there is no directed cycle in the graph of dependencies. A directed graph without directed cycle is called a *directed acyclic graph*, or, *dag* for short [15]. We can thus introduce the notion of *descendant* and *ancestor* between columns as follows: i descendant of $j \Leftrightarrow j$ ancestor of $i \Leftrightarrow i \overset{+}{\rightarrow} j$. Although an arbitrary directed graph may have many different transitive reductions, a *dag* only has one. Thus the transitive reduction of the graph of dependencies is unique [14].

Symmetric elimination tree

In the symmetric (LL^T or LDL^T) case, the transitive reduction of the graph of explicit dependencies is a tree and is called *symmetric elimination tree* [155, 128]. As we will heavily rely on this property in this thesis, we briefly provide a proof.

Lemma 1.1. For $i > j$, $i \rightarrow j$ if and only if $l_{ji} \neq 0$.

Proof. According to Definition 1.1, in the symmetric case, $i \rightarrow j$ if and only if there exists some k in $\{i + 1, \dots, n\}$ such that $l_{ki}u_{ij} \neq 0$. Thus, because of symmetry, $l_{ji} \neq 0$ so that $i \rightarrow j$ implies $l_{ji} \neq 0$. Conversely, if $l_{ji} \neq 0$, then $l_{ji}l_{ji} \neq 0$ and so $l_{ki}l_{ji} \neq 0$ with $k = j$. \square

Lemma 1.2. Let be $i < j < k$. The statements $i \rightarrow j$ and $i \rightarrow k$ imply $j \rightarrow k$.

Proof. From Lemma 1.1, we have $l_{ji} \neq 0$ and $l_{ki} \neq 0$, which imply that $l_{ki}l_{ji} \neq 0$. Thus formula (1.13) states that $l_{kj} \neq 0$ (we omit the possible cases of numerical cancellation). \square

This lemma is illustrated in terms of matrix structure by Figure 1.9. Because of symmetry, there is a nonzero at position i, j .

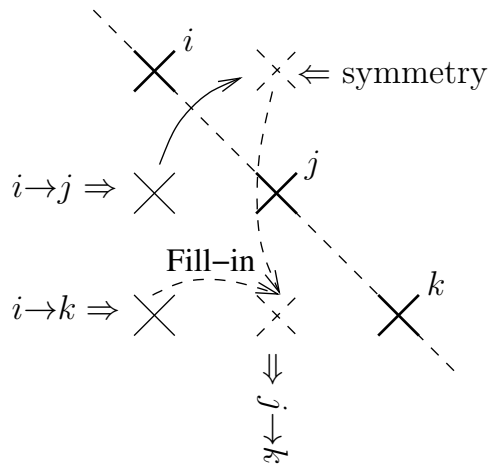


Figure 1.9: Illustration of Lemma 1.2. $i \rightarrow j$ corresponds to a nonzero element at position (j, i) (see Lemma 1.1), or (i, j) thanks to symmetry. $i \rightarrow k$ corresponds to a nonzero at position (k, i) . Then at the moment of eliminating pivot i , the fact that both elements at position (k, i) and (i, j) are nonzero lead to fill-in at position (k, j) . Hence $j \rightarrow k$.

Property 1.1. The transitive reduction of the graph of dependencies is a tree (if the matrix is irreducible, a forest in general).

tree for unsymmetric matrices [83], where the notion of edge is replaced by the notion of path to define a parent node, leading to a general framework.

Column elimination tree

Some methods aim at anticipating possible structural change due to numerical pivoting. They are based on a so-called *column elimination tree* which is the appropriate analogue of the symmetric elimination tree that takes into account all possible partial pivoting [94]. The column elimination tree is the symmetric elimination tree of $A^T A$ (provided that there is no cancellation in computing $A^T A$). Note that $A^T A$ does not need to be explicitly formed and that the column elimination tree can be computed in time almost linear in the number of nonzero values of the original matrix A [65, 94]. For instance, the serial version of SuperLU [65] is based on this approach.

In this thesis, we will not discuss further methods based on elimination dags or column elimination trees but only methods based on an elimination tree defined as follows:

Definition 1.2. *The elimination tree - or etree for short - will implicitly refer to:*

- *the symmetric elimination tree for symmetric direct methods;*
- *the symmetric elimination tree of the symmetrized matrix for unsymmetric direct methods with symmetric structure.*

1.2.4 Left-looking, right-looking and multifrontal methods

There are two main types of operations occurring during the factorization algorithm. Using the notations of [69], we will call the first one FACTO. It divides the part of the column below the diagonal by a scalar. In the second one, a column updates another column. We will call this operation UPDATE. Considering that A is overwritten by the factors so that eventually, $A = L + U - I$, we more formally have the following definitions (that stand thanks to Formula (1.13)):

- $\text{FACTO}(A_j): A_j(j+1:n) \leftarrow A_j(j+1:n)/a_{jj};$
- $\text{UPDATE}(A_i, A_j): A_j(i+1:n) \leftarrow A_j(i+1:n) - a_{ij} \cdot A_i(i+1:n);$

where A_j denotes column j of A .

There are n operations of type FACTO during the whole factorization, where n is the order of the matrix. These operations have to be performed according to the dependencies of the elimination tree: the parent node has to be processed after all its children. Said differently, $\text{FACTO}(A_j)$ has to be performed after $\text{FACTO}(A_i)$ if j is the parent of i (*i.e.*, if $i \overrightarrow{\text{---}} j$). And there is an effective $\text{UPDATE}(A_i, A_j)$ operation between any pair of columns (i, j) such that column j explicitly depends on column i (*i.e.*, such that $i \rightarrow j$). Any $\text{UPDATE}(A_i, A_j)$ operation has to be performed after $\text{FACTO}(A_i)$ and before $\text{FACTO}(A_j)$. We will note $\text{UPDATE}(*, A_j)$ an update *of* column j and $\text{UPDATE}(A_i, *)$ an update *from* column i .

In spite of these constraints of dependency, the structure of the elimination tree still provides some flexibility and freedom to schedule the computations, and we will see the interest of exploiting this freedom in Chapter 3. Moreover, once the scheduling of the FACTO operations is fixed, there is still some flexibility to schedule the UPDATE operations. Among all their possible schedules, there are two main types of algorithms: *left-looking* and *right-looking* methods. *Left-looking* algorithms delay the UPDATE operations as late as possible: all the $\text{UPDATE}(*, A_j)$ are performed just before $\text{FACTO}(A_j)$, looking to the **left** to nonzero entries in row j . On the opposite, *right-looking* algorithms perform the UPDATE operations as soon as possible: all the $\text{UPDATE}(A_i, *)$ operations are performed right after $\text{FACTO}(A_i)$, looking **right** to all columns that need be updated. Algorithms 1.6 and 1.7 respectively illustrate *left-looking* and *right-looking* factorizations. Note that Algorithm 1.6 exactly corresponds to applying iteration I_j from Formula (1.13) for $j = 1$ to n .


```

for  $j = 1$  to  $n$  do
  foreach  $i$  such that  $i \rightarrow j$  ( $j$  explicitly depends on  $i$ ) do
     $\lfloor$  UPDATE( $A_i, A_j$ ) ;
  FACTO( $A_j$ ) ;

```

Algorithm 1.6: General *left-looking* factorization algorithm.

```

for  $i = 1$  to  $n$  do
  FACTO( $A_i$ ) ;
  foreach  $j$  such that  $i \rightarrow j$  ( $j$  explicitly depends on  $i$ ) do
     $\lfloor$  UPDATE( $A_i, A_j$ ) ;

```

Algorithm 1.7: General *right-looking* factorization algorithm.

The *multifrontal* method [80, 81, 155, 129] is a variant of the *right-looking* method. The columns are still processed one after another but the UPDATE operations are not directly performed between the columns of the matrix. Instead, the contribution of a column i to a column j (j having to be updated by i) is carried through the path from i to j in the elimination tree. To do so, an UPDATE operation is performed in several steps and temporary columns are used to carry the contributions. This mechanism makes the *multifrontal* method slightly more complex than the previous ones. This is why we restrict the presentation of the method to the symmetric case. When processing a node i , some temporary columns are used on top of A_i . These temporary columns store the contributions from the descendants of column i and from column i itself to the ancestors. In general, not all the ancestors of column i will have to receive a contribution but only the ones that explicitly depend on column i (columns j such that $i \rightarrow j$). With each such ancestor j is associated a temporary column T_j^i that is used when processing column i . These columns are set to zero (INIT(T_j^i)) at the beginning of the process of i . Then the contribution stored in the temporary columns associated with any child k of i is carried into A_i and the different temporary columns associated with i . This operation is called ASSEMBLE. If the destination column is i , then ASSEMBLE is of the form ASSEMBLE(T_k^i, A_i) and consists in adding the temporary column T_k^i associated with child k of i from A_i . Otherwise, the destination column is a temporary column T_j^i associated with i ; the ASSEMBLE operation is of the form ASSEMBLE(T_j^k, T_j^i) and consists in adding T_j^k to T_j^i . Algorithm 1.8 describes the whole algorithm.

```

for  $i = 1$  to  $n$  do
  foreach  $j$  such that  $i \rightarrow j$  ( $j$  explicitly depends on  $i$ ) do
     $\lfloor$  INIT( $T_j^i$ ) ;
  foreach  $k$  such that  $k \rightarrow i$  ( $k$  child of  $i$ ) do
    ASSEMBLE( $T_k^i, A_i$ ) ;
    foreach  $j$  such that  $j > i$  and  $k \rightarrow j$  ( $j$  explicitly depends on  $k$ ) do
       $\lfloor$  ASSEMBLE( $T_j^k, T_j^i$ ) ;
  FACTO( $A_i$ ) ;
  foreach  $j$  such that  $i \rightarrow j$  ( $j$  explicitly depends on  $i$ ) do
     $\lfloor$  UPDATE( $A_i, T_j^i$ ) ;

```

Algorithm 1.8: General *multifrontal* factorization algorithm for symmetric matrices.

The symmetric multifrontal method can be described in terms of operations on dense matrices. With each node (column) i of the elimination tree is associated a dense matrix, called *frontal matrix* or *front*, that is usually square and that contains the union of the column A_i and the temporary columns T_j^i updated by A_i . Column A_i is the *factor block* of frontal matrix i ; the temporary columns constitute a *contribution block* that will be passed to the parent. The following tasks are performed at each node i of the tree:

- (MF-1) allocation of the frontal matrix in memory; gather entries of column i of matrix A into the first column of the front;
- (MF-2) assembly of contribution blocks coming from the child nodes into that frontal matrix;
- (MF-3) partial factorization of the factor block of the frontal matrix, and update of the remaining part.

This algorithm generalizes to the unsymmetric factorization of symmetrized matrices as we now explain. The factor associated with node i is then the *arrowhead* constituted by the union of column i and row i of the frontal matrix; the contribution block is the remaining square part. Figure 1.11(a) illustrates the association of the frontal matrices with the nodes of the elimination tree on a symmetrized matrix. For unsymmetric multifrontal factorizations, we refer the reader to [61, 37, 84].

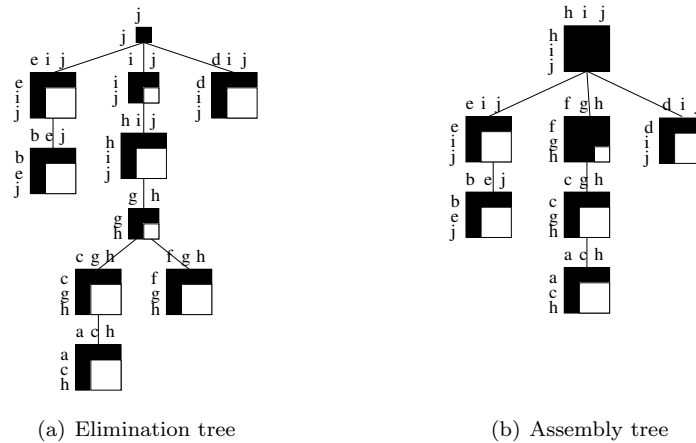


Figure 1.11: Frontal matrices associated with the elimination tree (left) or to the assembly tree (right) of the matrix presented in Figure 1.8. The black part of the frontal matrices corresponds to their factor block and the white part to their contribution block (that has to be assembled into the parent).

Let us reconsider the three algorithms presented above (*left-looking*, *right-looking* and *multifrontal* methods) according to their data access pattern. We illustrate their behaviour with the elimination tree presented in Figure 1.12. In all three methods, the nodes are processed one after the other, following a so called *topological ordering*³ In the case of the *left-looking* method, when the current node (circled in the figure) is processed, all its descendants (the nodes of the subtree rooted at the current node) are possibly accessed. More accurately, the descendants that have an explicit dependency on the current node update it. In the *right-looking* method, on the contrary, all its ancestors (the nodes along the path from the current node to the root of the tree) are possibly accessed. Again, only the nodes which explicitly depend on the current node are actually updated. In the *multifrontal* method, only the children nodes are accessed (to assemble the contributions blocks).

The three methods (*left-looking*, *right-looking*, *multifrontal*) naturally generalize to supernodes. However, the term *supernodal method* is commonly used to refer to *left-looking* and *right-looking* methods, rather than multifrontal methods.

Parallelism in the multifrontal method will be discussed in Chapter 4. In parallel, the left-looking and right-looking methods generalize to the so called *fan-in* and *fan-out* approaches [112], respectively. Early implementations of parallel multifrontal methods are discussed by Duff [74, 75].

³A *topological ordering* – in the case of a tree – is an order in which parents are ordered after their children.

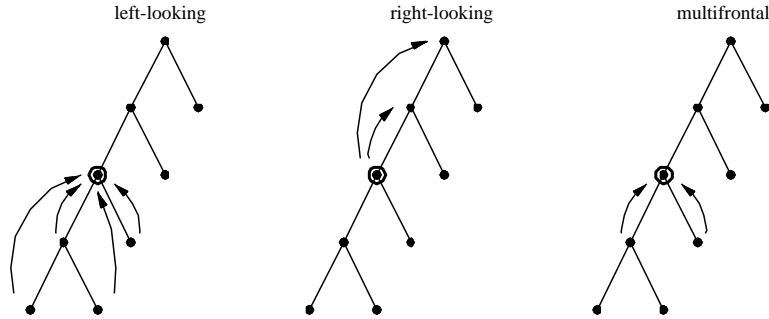


Figure 1.12: Data access pattern for the *left-looking*, *right-looking* and *multifrontal* methods.

1.3 Practical issues

1.3.1 Three-phase approach to solve $Ax = b$

Multifrontal methods, similar to direct solvers in general, generally use a three-phase approach to solve $Ax = b$:

1. Analysis: the graph of the matrix is analysed, so that a fill-reducing permutation (see Section 1.1.7) is obtained. Before that, a non symmetric permutation of the columns may also be computed, in order to put large values onto the diagonal (see Section 1.1.8). The analysis also forecasts the datastructures for the factorization. This requires performing a symbolic factorization, determining (relaxed) supernodes, computing an assembly tree, and partially mapping the tasks onto the processors.
2. Factorization: the permuted matrix is factorized, under the form LU , LDL^t , or LL^T , depending on the properties of the matrix; in case of numerical pivoting, the datastructures forecasted during the analysis may be modified at runtime. The factorization is usually the most computationally-intensive phase.
3. Solve: Triangular systems are solved to obtain a solution. Post-processing may also be applied, such as iterative refinement and error analysis.

1.3.2 Numerical accuracy and pivoting

Because we use floating-point arithmetic, the representation of numbers is not exact and rounding errors occur. In order to test the stability of an algorithm, the backward error analysis shows that in finite-precision arithmetic, the relative forward error is bounded by the condition number of the linear system, multiplied by the backward-error. It allows ([165]) to distinguish between

- an ill-posed problem, in which case the backward error can be small even if the solution is far from the exact solution; in that case, the condition number of the system is large;
- an unstable algorithm, leading to a large backward error compared to the machine precision, even when the condition number of the linear system is small.

When solving a linear system $Ax = b$, let \tilde{x} be the calculated solution; the backward error corresponds to the smallest perturbations ΔA and Δb on A and b such that \tilde{x} is the exact solution of a such a perturbed

system [146]:

$$\begin{aligned} \text{err} &= \min \{ \epsilon > 0 \text{ such that } \|\Delta A\| \leq \epsilon \|A\|, \|\Delta b\| \leq \epsilon \|b\|, \\ &\quad (A + \Delta A)\tilde{x} = b + \Delta b \} \\ &= \frac{\|A\tilde{x} - b\|}{\|A\|\|\tilde{x}\| + \|b\|}. \end{aligned}$$

Because we are dealing with sparse matrices, we know that the zeros in A are exact zeros. In other words, the structure of ΔA (resp. Δb if b is sparse) is known to be the same as that of A (resp. b). Therefore, it is possible to use the *component-wise* backward error defined by

$$\omega = \max_i \left(\frac{|A\tilde{x} - b|_i}{(|A|\|\tilde{x}\| + |b|)_i} \right)$$

(see, for example, [40, 114]) together with the condition number

$$\frac{\| |A^{-1}| |A| |\tilde{x}| + |A^{-1}| |b| \|_\infty}{\|\tilde{x}\|_\infty},$$

allowing for a better approach and possibly better bounds on the forward error.

In order to avoid large backward errors in the multifrontal factorization algorithm, too small pivots should be avoided; this is because the division by a small pivot will lead to large elements, leading to significant rounding errors when added to smaller numbers. The *growth factor* can be defined as $\rho = \frac{\max_{i,j,k} |a_{ij}^{(k)}|}{\max_{i,j} |a_{ij}|}$ where $a_{ij}^{(k)} = a_{ij}^{(k-1)} - \frac{a_{ik}^{(k-1)} a_{kj}^{(k-1)}}{a_{kk}^{(k-1)}}$, $i, j > k$, is the value of $A(i, j)$ at step k of Gaussian elimination: ρ is the largest $|A(i, j)|$ obtained during the application of Algorithm 1.2 divided by the largest $|A(i, j)|$ term in the original matrix. It gives an idea of the numerical problems one can expect.

In order to limit errors due to round-off, a first approach that can be applied is scaling (for example [32, 78, 151]) because it improves the numerical properties of the matrix. If D_r and D_c are diagonal matrices providing a row and a column scaling (respectively), the initial system $Ax = b$ is replaced by $(D_r A D_c)(D_c^{-1} x) = D_r b$ and matrix $D_r A D_c$ is factorized instead of A . However, dynamic numerical pivoting during the factorization is also crucial to limit the growth factor and obtain a stable algorithm.

1.3.2.1 Unsymmetric case

The goal of pivoting is to ensure a good numerical accuracy during Gaussian elimination. A widely used technique is known as *partial pivoting*: at step i of the factorization (see Algorithm 1.2), we first determine k such that $|A(k, i)| = \max_{l=i:n} |A(l, i)|$. Rows i and k are swapped in A (and the permutation information is stored in order to apply it to the right-hand side b) before dividing the column by the pivot and performing the rank-one update. The advantage of this approach is that it bounds the growth factor and improves the numerical stability.

Unfortunately, in the case of sparse matrices, numerical pivoting prevents a full static prediction of the structure of the factors: it dynamically modifies the structure of the factors, thus forcing the use of dynamic data structures. Numerical pivoting can thus have a significant impact on the fill-in and on the amount of floating-point operations. To limit the amount of numerical pivoting, and stick better to the sparsity predictions done during the symbolic factorization, partial pivoting can be relaxed, leading to the *partial threshold pivoting* strategy:

Strategy 1.1. *In the Partial threshold pivoting strategy, a pivot $a_{i,i}$ is accepted if it satisfies:*

$$|a_{i,i}| \geq u \times \max_{k=i:n} |a_{k,i}|, \quad (1.14)$$

for a given value of u , $0 \leq u \leq 1$. This ensures a growth factor limited to $1 + 1/u$ for the corresponding step of Gaussian elimination. In practice, one often chooses $u = 0.1$ or $u = 0.01$ as a default threshold and this generally leads to a stable factorization.

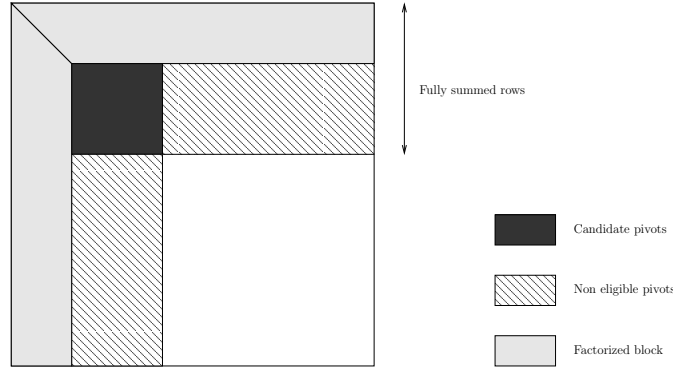


Figure 1.13: Frontal matrix during its factorization. After a given number of steps in the factorization, eligible pivots are restricted to the square block on the left of the fully-summed rows.

It is possible to perform the pivot search on the row rather than on the column with similar stability. This can be useful when, for example, frontal matrices are stored in row-major order (*i.e.*, two consecutive elements in a row are contiguous in memory). In that case, the stability check of a pivot $a_{i,i}$ is obtained by replacing Equation (1.14) by:

$$|a_{i,i}| \geq u \times \max_{k=i:n} |a_{i,k}|. \quad (1.15)$$

Furthermore, in the multifrontal method, once a frontal matrix is formed, we cannot choose a pivot outside the square top-left part of the fully-summed block, because the corresponding rows are not fully-summed. Figure 1.13 illustrates the candidate pivots at a given step of the factorization. If the current pivot candidate is not large enough, other pivots should be tried in the block of candidate pivots; sometimes, priority is given to the diagonal elements. Once all possible pivots in the block of candidate pivots have been eliminated, if no other pivot satisfies the partial pivoting threshold, some rows and columns remain unfactored in the front. Those are then delayed to the frontal matrix of the parent, as part of the Schur complement. This is shown in Figure 1.14. Thus, because of numerical pivoting, the frontal matrix of the parent becomes bigger than predicted. Furthermore, fill-in occurs because the frontal matrix of the parent involves variables that were not in the structure of the child. A column can be delayed several times, but the more we go up in the tree, the more it has chances to become stable because some non fully-summed parts of that column in the child become fully-summed in the parent.

1.3.2.2 Symmetric case

The same type of approach is applied to the symmetric case. One important issue is that we want to maintain the symmetry of the frontal matrices and of the factorization. Similarly to what is done in dense linear algebra, *two-by-two pivots* are used [45, 46], so that the matrix D in the symmetric factorization LDL^T is composed of symmetric and 1×1 and 2×2 blocks. For example, in the factorization

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

D is composed of a 2×2 pivot $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$, and a 1×1 pivot, -2 .

For symmetric matrices, partial threshold pivoting may be defined as follows:

- A 1×1 diagonal pivot can be selected if it satisfies (1.14).

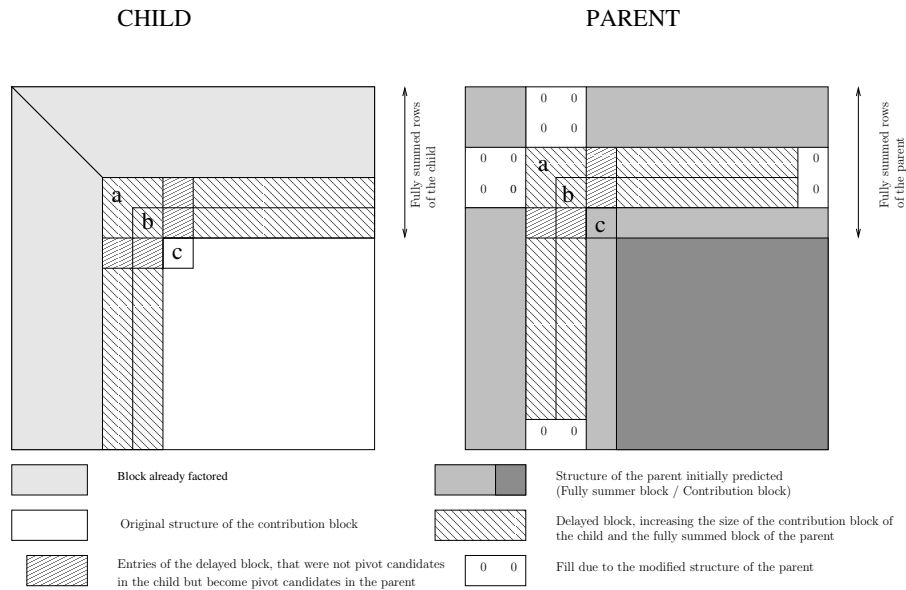


Figure 1.14: Frontal matrix of a child (left) and of a parent (right) when two pivots — here a and b — are delayed from the child to the parent. Fill-in occurs in the parent. Because some non fully-summed variables in the child — here variable c — may become fully-summed in the parent, the possibilities to find a good pivot in the parent are increased. For example, the entry at the intersection of row c and column a may be large enough to swap the corresponding row with row a, or the elimination of c before a and b may result in a and b being stable pivots with respect to the pivoting criterion.

- After the adequate permutations of rows and columns, a 2×2 pivot $P = \begin{pmatrix} a_{i,i} & a_{i,i+1} \\ a_{i+1,i} & a_{i+1,i+1} \end{pmatrix}$ is accepted at step i if it satisfies

$$|P^{-1}| \begin{pmatrix} \max_{k \geq i+2} |a_{i,k}| \\ \max_{k \geq i+2} |a_{i+1,k}| \end{pmatrix} \leq \begin{pmatrix} 1/u \\ 1/u \end{pmatrix}. \quad (1.16)$$

This approach is the one from the Duff-Reid pivot selection algorithm [80, 82]. It ensures a growth factor limited to $1 + 1/u$ at each step of the factorization, and is as good as rook pivoting [132]. 2×2 pivots are often explicitly inverted during the factorization, in order to simplify the update of the columns of the L factor.

1.3.2.3 LINPACK vs. LAPACK styles of pivoting

There are two ways to perform the row/column exchanges during the factorization, known as the LINPACK style of pivoting and the LAPACK style of pivoting. Consider the example of a right-looking LU factorization with (threshold) partial pivoting, where pivots are chosen in the column. In the LINPACK [70] style of pivoting, when a pivot in the column has been chosen, the part of the rows in the already computed L factors are not exchanged, only the rows in the current submatrix are permuted. One obtains a series of Gauss transformations interleaved with matrix permutations that must be applied similarly during the solution stage. In the LAPACK [38] style of pivoting, all rows including the already computed factors are permuted, so that the series of transformations can be expressed in the form $PA = LU$. A clear description of those two styles of pivoting is available in [144], in section 2.1.

1.3.2.4 Static pivoting and iterative refinement

In order to avoid the complications due to numerical pivoting, perturbation techniques can be applied. This is typically the case in approaches that require static data structures during the factorization, *i.e.*, where the data structures are entirely predicted. In particular, SuperLU_{dist} uses such an approach; at each step, a pivot smaller than $\epsilon \|A\|$ in absolute value is replaced by $\epsilon \|A\|$, where ϵ is the machine precision, see [123]. In practice, *iterative refinement* (see Algorithm 1.9) can help obtaining a solution to the original system from the solution of the perturbed system.

$$r = b - Ax$$

repeat

Solve $A\Delta x = r$ using the approximate factorization relying on static pivoting

$$x \leftarrow x + \Delta x$$

$$r = b - Ax$$

$$\omega = \max_i \frac{|r_i|}{(|A||x|+|b|)_i}$$

until $\omega \leq \epsilon$ or convergence is not obtained or is too slow

Algorithm 1.9: Iterative refinement. At each step, the component-wise backward error ω is computed and checked.

A comparison of approaches based on static pivoting with approaches based on numerical pivoting in the context of high-performance distributed solvers can be found in [28]. Remark that static pivoting and numerical pivoting at runtime can be combined, as proposed in [78], also discussed in Section 2.2.4.

1.3.3 Memory management in the multifrontal method

Because of the dependencies of the method, the assembly tree must be processed from leaves to roots during the factorization process, enforcing a topological order. Traditionally, the multifrontal method uses three areas of storage, one for the factors, one to store the contribution blocks waiting to be assembled, and one for the current frontal matrix. During the factorization process, the memory space required for the factors

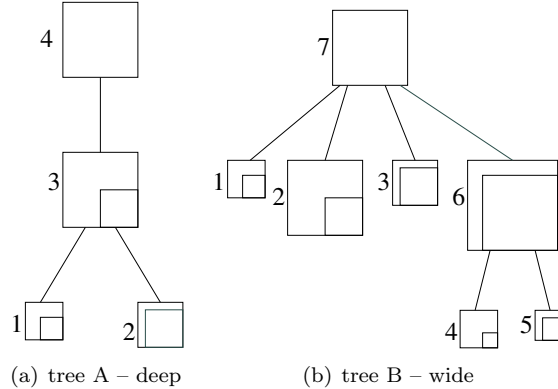


Figure 1.15: Examples of assembly trees.

always grows but the memory for the contribution blocks (we name it *CB memory*) varies depending on the operations performed. When the partial factorization of a frontal matrix is finished, a contribution block is stored and this increases the size of the CB memory; on the other hand, when the frontal matrix of a parent is formed and assembled, the contribution blocks of the children are used and discarded, freeing some memory. It is important to note that in a sequential environment, the CB memory forms a stack when a postorder traversal of the assembly tree is used to visit the nodes; a postorder is a particular topological order where the nodes in any subtree are numbered consecutively: each time the last sibling of a family is processed, the parent node is activated, consuming the contribution blocks on the top of the stack. To illustrate this remark, we give in Figure 1.15 two examples of assembly trees. As is classical in many multifrontal codes, rather than relying on dynamic allocation for each small dense matrix generated during the traversal of the tree (frontal matrix, contribution block, generated factors), we assume here that a preallocated workspace is used, allowing for an explicit management of memory. For more discussions on the use of such a preallocated workspace and possible optimizations, we refer the reader to Chapter 3, Section 3.5. For the trees of Figure 1.15, the memory evolution for the factors, the stack of contribution blocks and the current frontal matrix is given in Figure 1.16. First, storage for the current frontal matrix is reserved (see “Allocation of 3” in Figure 1.16(a)); then, the frontal matrix is assembled using values from the original matrix and contribution blocks from the children nodes, and those can be freed (“Assembly step for 3” in 1.16(a)); the frontal matrix is factorized (“Factorization step for 3” in 1.16(a)). Factors are stored in the factor area on the left in our figure and the contribution block is stacked (“Stack step for 3”). The process continues until the complete factorization of the root node(s). We can observe the different memory behaviours between the wide tree (Figure 1.16(b)) and the deep tree (1.16(a)): the peak of active memory (see Figure 1.16(b)) is significantly larger for the wide tree.

This shows that the shapes of the trees resulting from different orderings, (see Figure 1.6) will have a strong impact on the behaviour of the stack memory and on its maximum size. Further details can be found in [107]. Furthermore, the postorder is not unique since given a parent node, there is some freedom to order its child subtrees. This is illustrated by the simple tree from Figure 1.17. In [126], Liu suggested a postorder which minimizes (among the possible postorders) the working storage of the multifrontal method; we will propose variations and extensions of his work for various objectives and models of multifrontal methods in Chapter 3.

In parallel environments, the memory for contribution blocks no longer exactly behaves as a stack (see [21]), because of the difficulty of ensuring a postorder while taking advantage of tree parallelism. Attempts to better take memory into account in dynamic schedulers are proposed in Chapter 4. In Section 4.3, the impact of the static mapping on the memory scalability is also discussed.

In the rest of this thesis, we call *active storage* or *working storage* the storage corresponding to both the stack and the current active frontal matrix.

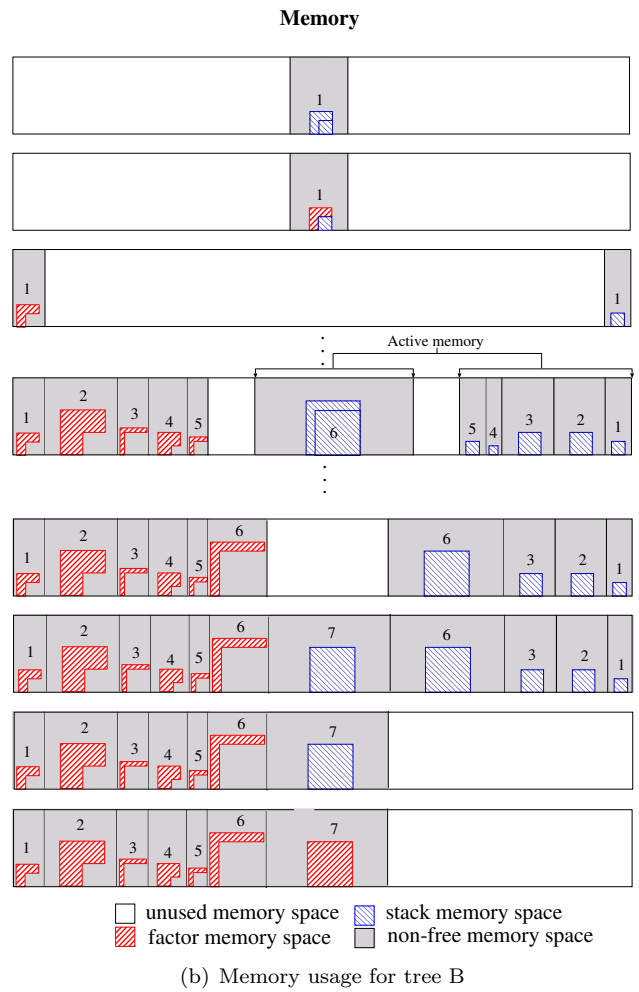
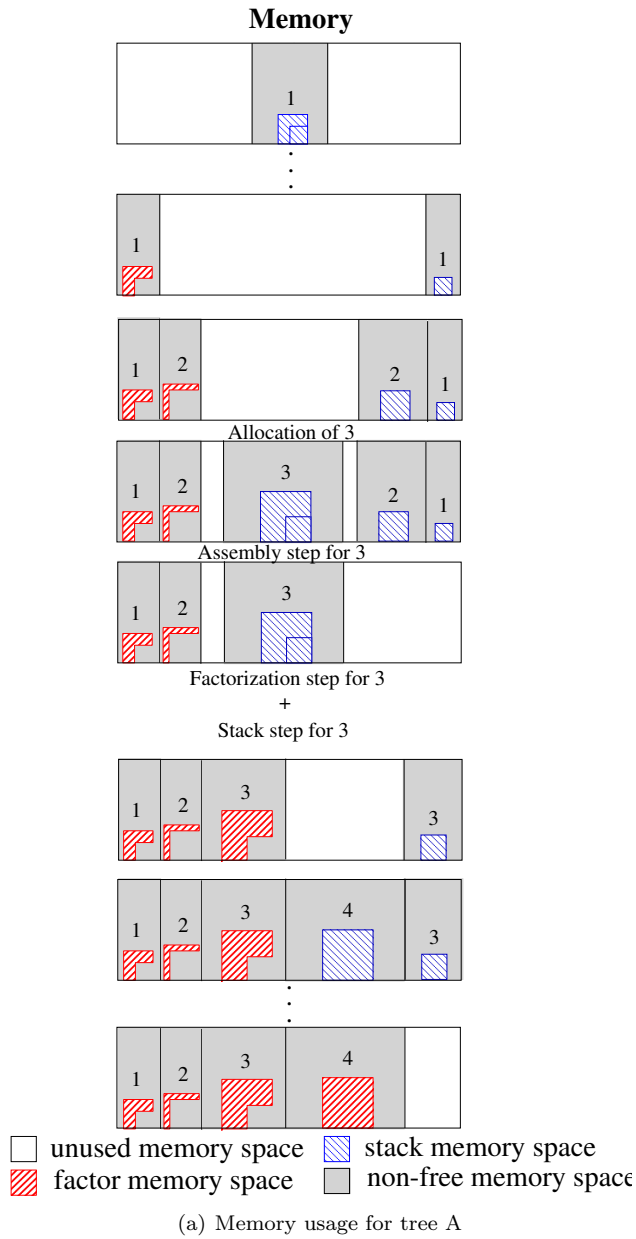


Figure 1.16: Memory evolution for the trees given in Figure 1.15.

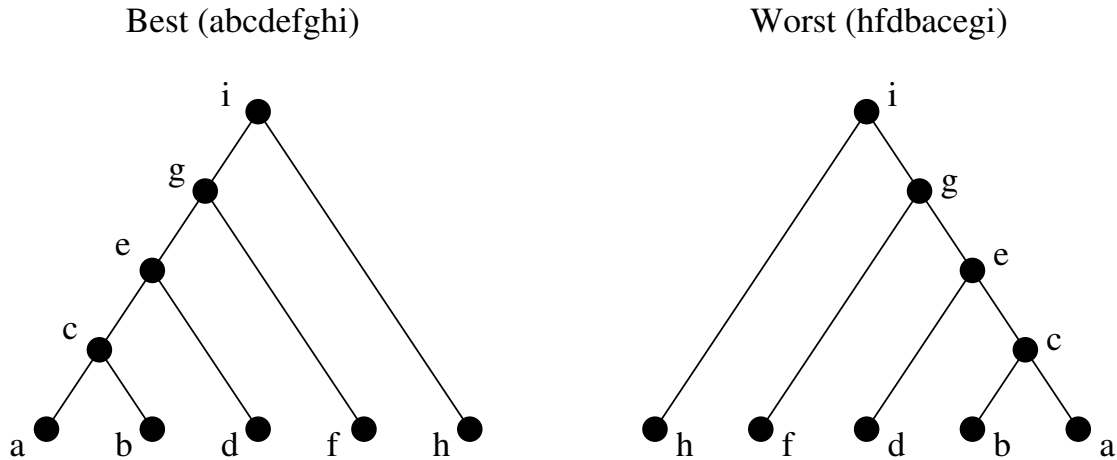


Figure 1.17: Importance of the tree traversal: assuming that the postorder traverses the nodes left-to-right (a parent is activated as soon as possible), the tree on the left (resp. right) leads to a maximum number contribution blocks stored in the stack of two (resp. five).

1.3.4 Out-of-core approaches

Some applications require huge amounts of memory that sometimes exceed the physical memory available. This may happen when large problems are factorized via direct methods (among which multifrontal methods) on a server, on a parallel cluster or a high performance computer. Even when the initial matrix holds in physical memory, the factors or the active storage are often orders of magnitude larger than the initial matrix and may not fit in physical memory. If nothing is done, the application runs *out-of-memory*. However, this limit can be overcome by using other units of storage like disks to extend the main memory. *Out-of-core* approaches are then used, motivated by the fact that disks are much cheaper than core memory. In a way, disks are just part of a hierarchical set of units of storage (registers, L1 cache, L2 cache, L3 cache, physical memory, disks, tapes). In fact, memory can be seen as a cache for the disks and one may therefore wonder why the out-of-core question should be any different from the locality questions related to the use of memory hierarchies and caches. However the differences are:

- Contrary to the data traffic between main memory and cache (except when considering special cases such as GPGPU or special accelerators), it is possible to control the data movements between disks and main memory.
- The amount of cache memory (usually several gigabytes) is much larger than the amount of cache memory (in the order of megabytes). The ratio between disk storage and physical memory is often much smaller than the ratio between physical memory and cache size.
- Letting the system use swapping mechanisms and disks usually kills performance, whereas it performs a good job at managing caches.

As said in the previous section, the multifrontal approach uses several areas of storage, one of which is constituted by the factors computed at each node of the tree. Because the factors produced are only accessed at the solve stage, it makes sense to write them to disk first. This is the first approach that we will study in Chapter 5 (Section 5.1), before assessing the volume of *I/O* necessary for the active storage in a parallel environment (Section 5.2), as a function of the available core memory. Furthermore, in the case of serial environments, tree traversals and postorders have a strong impact on the memory usage (see Section 1.3.3); however they also have an impact on the *I/O* traffic and we will see in Chapter 3 that minimizing memory is intrinsically different from minimizing *I/O*.

Chapter 2

A Parallel Multifrontal Method for Distributed-Memory Environments

In this chapter, we present an original implementation of the multifrontal method and show the impact of parallelism on numerical functionalities. In particular, we show how some algorithms that appear relatively simple in a sequential context need be extended to address parallel, dynamic and asynchronous environments. We focus on parallel multifrontal methods, and refer the reader to [112] for a more general presentation of parallelism in other sparse direct methods. We first show in Section 2.1 how we adapted the multifrontal method to a parallel environment. In Sections 2.2 to 2.6 we then explain how some features or functionalities can be adapted or developed in this parallel context; we discuss the pivot selection, the solution phase, the Schur complement and the computation of the determinant. Note that out-of-core issues will be the object of a specific chapter (Chapter 5).

2.1 Adapting the multifrontal factorization to a parallel distributed environment

As explained in the previous chapter, the multifrontal method factorizes a matrix by performing a succession of partial factorizations of small dense matrices called frontal matrices, associated with the nodes of an assembly tree. A crucial aspect of the assembly tree is that it defines only a partial order for the factorization since the only requirement is that a child must complete its elimination operations before the parent can be fully processed. This gives a first source of parallelism, where independent branches are allowed to be processed in parallel. Then each partial factorization is a task that can itself be parallelized; a Schur complement (possibly distributed over the processors) is produced, that must be assembled (assembly tasks) into the processors participating to the parent node. In our approach, all these tasks are managed asynchronously, according to the central Algorithm 0.1, presented in the main introduction of this thesis page 2. Furthermore, the mapping of most of the computations onto the processors is dynamic, with distributed scheduling decisions. This allows to deal with the dynamic nature of our tasks graphs (due to numerical pivoting), as well as with the possible load variations of the platform.

2.1.1 Sources of parallelism

We consider the condensed assembly tree of Figure 2.1, where the leaves are subtrees of the assembly tree. We will in general define more leaf subtrees than processors and map them onto the processors in order to obtain a good overall load balance of the computation at the bottom of the tree. However, if we only exploit the tree parallelism, the speed-up is usually low: the actual speed-up from this parallelism depends on the problem but is typically only 2 to 4 irrespective of the number of processors. This poor efficiency is caused by the fact that the tree parallelism decreases while going towards the root of the tree. Moreover,

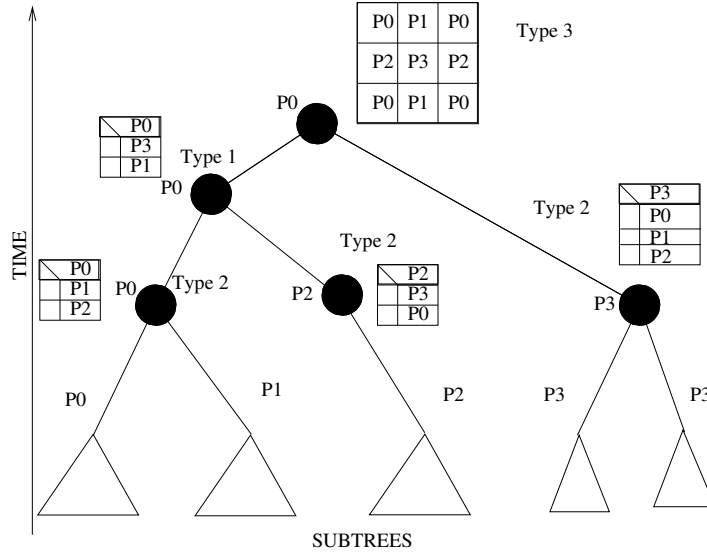


Figure 2.1: Distribution of the computations of a multifrontal assembly tree on four processors P0, P1, P2, and P3.

it has been observed (see for example [21]) that often more than 75% of the computations are performed in the top three levels of the assembly tree. It is thus necessary to obtain further parallelism within the large nodes near the root of the tree. The additional parallelism will be based on parallel versions of the blocked algorithms used in the factorization of the frontal matrices.

Nodes of the tree processed by only one processor will be referred to as nodes of *type 1* and the parallelism of the assembly tree will be referred to as *type 1 parallelism*. Further parallelism is obtained by doing a 1D block partitioning of the rows of the frontal matrix for nodes with a large contribution block. Such nodes will be referred to as nodes of *type 2* and the corresponding parallelism as *type 2 parallelism*. Finally, if the root node is large enough, then 2D block cyclic partitioning of the frontal matrix is performed. The parallel root node will be referred to as a node of *type 3* and the corresponding parallelism as *type 3 parallelism*.

2.1.1.1 Description of type 2 parallelism

If a node is of type 2, one processor (called the master of the node) holds all the fully-summed rows and performs the pivoting and the factorization on this block while other processors (so called slaves) perform the updates on the contribution rows (see Figure 2.2).

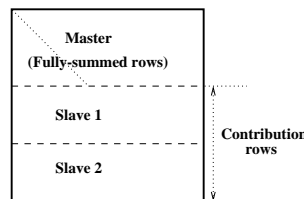


Figure 2.2: Type 2 nodes: partitioning of frontal matrix.

Macro-pipelining based on a blocked factorization of the fully-summed rows is used to overlap communication with computation. The efficiency of the algorithm thus depends both on the block size used to factor the fully-summed rows and on the number of rows allocated to a slave process. During the analysis phase,

based on the structure of the assembly tree, a node is determined to be of type 2 if its frontal matrix is sufficiently large. We assume that the master processor holding the fully-summed rows has been mapped during the analysis phase and that any other processors might be selected as slave processors. As a consequence, part of the initial matrix is duplicated onto all the processors to enable efficient dynamic scheduling of the corresponding computational tasks. At execution time, the master then first receives symbolic information describing the structure of the contribution blocks sent by its children. Based on this information, the master determines the exact structure of its frontal matrix and decides which slave processors will participate in the factorization of the node. Figure 2.3 illustrates the dynamic subdivision and mapping of type 2 nodes in the tree, where the subgraph corresponding to the pipelined factorization is only defined at runtime. The overall graph of tasks we manage is therefore a dynamic, distributed tasks graph.

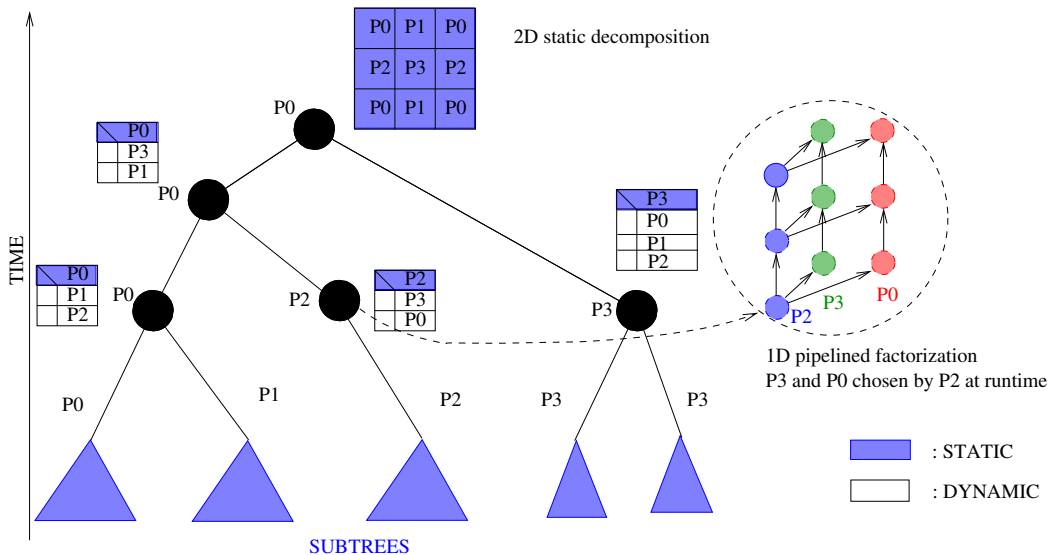


Figure 2.3: Illustration of dynamic definition and mapping of tasks in type 2 nodes. The tasks graph in the middle node is a dag defined and mapped dynamically.

Further details on the implementation of type 2 nodes depend on whether the initial matrix is symmetric or not and will be given in Section 2.1.5.

2.1.1.2 Description of type 3 parallelism

In order to have good scalability, we perform a 2D block cyclic distribution of the root node, on which a standard dense factorization is required. We use ScaLAPACK [44] or the vendor equivalent implementation (PDGETRF for unsymmetric matrices and PDPOTRF for symmetric positive matrices¹).

Currently, a maximum of one root node, chosen during the analysis, is processed in parallel. This node is of type 3. The node chosen will be the largest root provided its size is larger than a computer dependent parameter. One processor, the so-called master of the root, holds all indices describing the frontal matrix.

We define the root node as determined by the analysis phase, the *estimated* root node. Before factorization, the estimated root node frontal matrix is statically mapped onto a 2D grid of processors. We use a static distribution and mapping for those variables known by the analysis to be in the root node so that, for an entry in the estimated root node, we know where to send it and assemble it using functions involving integer divisions, moduli, ...

¹For symmetric indefinite matrices, because no LDL^T kernel is available in ScaLAPACK, we use the LU decomposition computed by PDGETRF.

In the factorization phase, the original matrix entries and the part of the contribution blocks from the children corresponding to the estimated root can be assembled as soon as they are available. The master of the root node then collects the index information for all the uneliminated (or delayed, see Figure 1.14) variables of its children and builds the structure of the frontal matrix. This symbolic information is broadcast to all participating processors. The contributions corresponding to uneliminated variables can then be sent by the children to the appropriate processors in the 2D grid for assembly, or directly assembled locally if the destination is the same processor. Note that, because of the requirements of ScaLAPACK, local copying of the root node is performed and the leading dimension of the local array changes.

2.1.2 Asynchronous communication issues

To enable automatic overlapping between computation and communication, we have chosen to use fully asynchronous communications, and we rely on the Message Passing Interface (MPI [71]). For flexibility and efficiency, explicit buffering in the user space has been implemented. The size of the buffer is equal to a relaxed estimation of the maximum message size, computed by each processor prior to factorization². This estimation is based on the partial static mapping of the assembly tree and takes into account the three types of parallelism used during the factorization. A software layer (Fortran 90 module) takes care of sending asynchronous messages, based on immediate sends (MPI_ISEND [159]). Note that messages are never sent when the destination is identical to the source; in that case the associated action is performed locally directly, in place of the send, slightly modifying synchronization issues: instead of sending the message and continuing the current action, we need to perform the action that would be done on reception straight away, which is much earlier.

When trying to send contribution blocks, factorized blocks, ... we first check whether there is room in the send buffer. If there is room in the send buffer, the message is packed into the buffer and an asynchronous communication is posted. Otherwise, the procedure requesting the send is informed that the send cannot be done. In such cases, to avoid deadlock, the corresponding processor will try to receive messages (and will perform the associated action) until space becomes available in its local send buffer. Let us take a simple illustrative example. Processor *A* has filled-up its buffer doing an asynchronous send of a large message to processor *B*. Processor *B* has done the same to processor *A*. The next messages sent by both processors *A* and *B* will then be blocked until the other processor has received the first message. More complicated situations involving more processors can occur, but in all cases the key issue for avoiding deadlock is that each processor tries not to be the blocking processor. The way to allow message reception at the moment of performing a send is implemented through calls to the function that receives and processes messages. Notice that when a message is received, the associated action can induce another send. If the buffer is still full, we may enter deeper and deeper levels of recursive calls until the situation stabilizes. Recursivity is a source of complication that makes the code complex. In particular, if we detect at a deep level of recursivity that all messages from a given processor have been received, we may need to delay some action if one of these messages is only going to be processed at the top level of recursivity.

Another issue is that MPI only ensures that messages are non-overtaking, that is if a processor sends two messages to the same destination, then the receiver will receive them in the same order. For synchronous algorithms the non-overtaking property is often enough to ensure that messages are received in the correct order. With a fully asynchronous algorithm, based on dynamic scheduling of the computational tasks, it can happen that messages arrive “too early”. In this case, it is crucial to ensure that the “missing” messages have already been sent so that blocking receives can be performed to process all messages that should have already been processed at this stage of the computation. As a consequence, the order used for sending messages is critical. To summarize, the properties that we must ensure to avoid deadlocks in our asynchronous approach are the following:

Property 2.1. *When trying to send a message, if the asynchronous send buffer is full, one must try to receive and process any message before a new attempt to send. (If processing a message involves a send, this implies recursivity.)*

²We will see in Section 6.5 how the buffer sizes can be reduced.

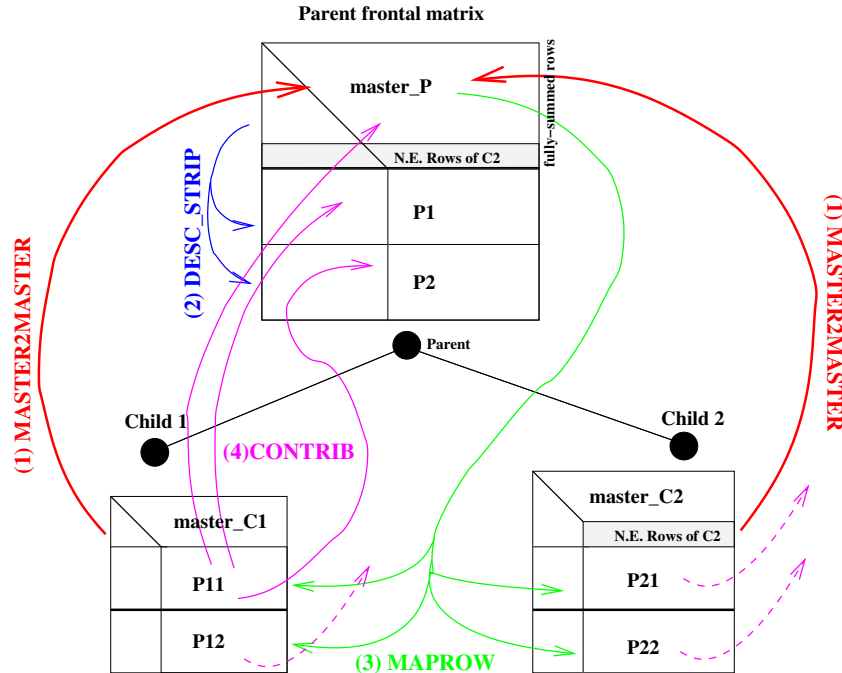


Figure 2.4: Messages involved in the assembly of a type 2 node. Although the shape of the tree is static, the sizes of the frontal matrices depends on numerical issues (delayed rows).

Property 2.2. *If a processor P has to process messages M_1 and M_2 , in that order, one must ensure that M_1 was sent before M_2 , even if M_1 and M_2 are sent by two distinct processors. In general, this means that the processor sending M_1 to P should do so before performing any action that may result in M_2 to be sent.*

Their impact on the algorithm design will be illustrated in Sections 2.1.3 and 2.1.5 during the detailed description of type 2 parallelism for LDL^T factorization.

Remark that in Algorithm 0.1, priority is given to message reception and a new local task is extracted from the pool of ready tasks only when no message is available on reception. The main reasons for this choice are first that the message received might be a source of additional work and parallelism and second that the sending processor might be blocked because its send buffer is full, in which case it is even more critical to receive the available messages as soon as possible.

2.1.3 Assembly process

An estimation of the frontal matrix structure (size, number of fully-summed variables) is computed during the analysis phase. Because of numerical pivoting, the final structure and the list of indices in the front is however unpredictable and it is only computed during the assembly process of the factorization phase. This requires a message from each child node to the master of a parent node. The list of indices of a front is then the result of a merge of the index lists of the contribution blocks of the children with the list of indices in the arrowheads (corresponding to entries of the original matrix, see the definition of that term in Section 1.1.3) associated with all the fully-summed variables of the front. Once the index list of the front is computed, the assembly of numerical values can be performed efficiently.

We describe the assembly process in Figure 2.4. Let $Parent$ be a node of type 2 with two children. Assume that the master $master_P$ has received all symbolic information from its children: those messages are tagged MASTER2MASTER because they are sent from a master of a child to the master of the parent. Remark that such messages also include delayed or non-eliminated rows. Since $master_P$ has received all its

MASTER2MASTER messages for node *Parent*, the assembly task for *Parent* was inserted in the local pool of tasks of *master_P*. When that task is extracted (see Algorithm 0.1, the structure of the parent is built by *master_P* and the non-eliminated rows from the children, if any, are ordered and assembled. Processor *master_P* then defines the partition of non-fully summed rows of the frontal matrix into blocks, and chooses a set of slave processors that will participate in the parallel assembly and factorization of *Parent*. This choice is based on an estimate of the workload and memory load of the other processors. In order to inform other processors that they have been chosen, *Master_P* sends a message (identified by the tag DESC_STRIP) describing the work to be done on each slave processor. It then sends messages (with tag MAPROW) to all processors involved in a type 1 child (none in the example of the figure) and to all slave processors involved in type 2 children, providing them symbolic information on where to send the rows of their contribution blocks for the assembly process. Thanks to that information, messages containing contributions rows (tagged CONTRIB) are sent from children processors P11, P12, P21, P22, to processors involved in the parent: P1, P2, master_P. The numerical assembly (extend-add operations) can then be performed, row by row.

As already mentioned in Section 2.1.2, the order in which messages are sent is important (see Property 2.2). For example, a slave of *master_P* may receive a contribution block (message CONTRIB) before receiving the message with tag DESC_STRIP from its master. To allow this slave processor to safely perform a blocking receive on the missing DESC_STRIP message, we must ensure that the master of the node has sent DESC_STRIP before sending MAPROW. Otherwise we cannot guarantee that DESC_STRIP has actually been sent (for example, the send buffer might be full).

After the message MAPROW has been sent, the factorization of the master of the parent cannot start before all processes involved in the children of the parent have sent their contributions. Therefore, once the messages DESC_STRIP and MAPROW have been sent, the master of *inode* returns in the main loop of Algorithm 0.1 and checks for receptions or extracts new nodes from the pool of tasks. When it detects after a reception that all CONTRIB messages it depends on have been received, the factorization task for *Parent* is inserted into the pool. The pool is managed using a LIFO (or stack) mechanism, in order to keep locality of computations and avoid too many simultaneous active tasks that would cause memory difficulties.

The main difference between the symmetric and the unsymmetric cases is due to the fact that a global ordering of the indices in the frontal matrices is necessary for efficiency in the symmetric case to guarantee that all lower triangular entries in a contribution row of a child belong to the corresponding row in the parent. We use the global ordering obtained during analysis for that, with one exception: the delayed variables coming from the children because of numerical difficulties are assembled last in the fully summed part of the parent in order to avoid them being used as first pivot candidates in the front of the parent. Because of that, special care has to be taken when assembling rows/columns corresponding to delayed variables.

Moreover, it is quite easy to perform a merge of sorted lists efficiently. If we assume that the list of indices of the contribution block of each child is sorted then the sorted merge algorithm will be efficient if the indices associated with the arrowheads are also sorted. Unfortunately, sorting all the arrowheads can be costly. Furthermore, the number of fully-summed variables (or number of arrowheads) in a front might be quite large and the efficiency of the merging algorithm might be affected by the large number of sorted lists to merge. Based on experimental results, we have observed that it is enough to sort only the arrowhead associated with the first fully-summed variable of each frontal matrix. The assembly process for the list of indices of the node is described in Algorithm 2.1, whose key issue for efficiency is the fact that only a small number of variables are found at step 4. For example, on matrix WANG3 with default amalgamation parameter, the average number of indices found at step 4 was 0.3. During this algorithm, a flag array is necessary; it gives for each global variable its position in the front and allows to detect whether a variable has already been encountered at earlier steps and is zero for variables not yet encountered. The position in the parent contained in this array is also used to inform the children of the positions where to assemble their rows, in the “MAPROW” messages discussed earlier. In practice, children lists of row indices are overwritten with their position in the parent. The flag array can be reset to zero when it has been used (only the known nonzero entries are reset to zero in order to limit the associated cost).

At the end of the algorithm, the index list of the parent front contains, in that order:

1. the fully-summed variables known statically;

2. the delayed variables coming from the children;
3. the variables in the contribution block.

Since the delayed variables may correspond to numerical difficulties, it makes sense to order them last in the fully-summed part to avoid starting with the numerical difficulties. Referring to Figure 1.14 where no particular order was defined, this means that we prefer to put variables a and b last in the fully-summed block of the parent, after variable c, although this means that the global ordering is not respected on the parent (all delayed variables should otherwise be ordered first in the front).

Step 1: Get the list of fully summed variables of the parent supernode known statically (and sorted according to a global ordering).

Step 2: Sort the indices of the first arrowhead, *i.e.*, the one associated with the first fully summed variable of the front.

Step 3: Merge the sorted lists of indices from the children and from the first arrowhead, excluding variables from step 1.

Step 4: Build and sort variables belonging only to the other arrowheads (and not found in steps 1 or 3).

Step 5: Merge the sorted lists built at steps 3 and 4, append the resulting list to the list of fully summed variables from step 1.

Algorithm 2.1: Construction of the indices of the frontal matrix of a parent node, symmetric case.

2.1.4 Factorization of type 1 nodes

Blocked algorithms are used during the factorization of type 1 nodes and, for both the LU and the LDL^T factorization algorithms, we want to keep the possibility of postponing the elimination of fully-summed variables. Note that classical blocked algorithms for the LU and LL^T factorizations of full matrices [38] are quite efficient, but it is not the case for the LDL^T factorization.

We will briefly compare kernels involved in the blocked algorithms. We then show how we have exploited the frontal matrix structure to design an efficient blocked algorithm for the LDL^T factorization.

Let us suppose that the frontal matrix has the structure of Figure 2.5, where A is the block of fully-summed variables available for elimination. Note that, in the code, the frontal matrix is stored by rows.

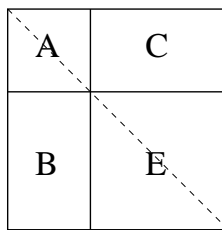


Figure 2.5: Structure of a type 1 node.

During LU factorization, an efficient right-looking blocked algorithm [20, 62, 73] is used to compute the LU factor associated with the block of fully-summed rows (matrices A and C). The Level 3 BLAS kernel DTRSM is used to compute the off-diagonal block of L (overwriting matrix B). Updating the matrix E is then a simple call to the Level 3 BLAS kernel, DGEMM.

During LDL^T factorization, a right-looking blocked algorithm (see Chapter 5 of [73]) is first used to factor the block column of the fully-summed variables. Let L_{off} be the off diagonal block of L stored in place of the matrix B and D_A be the diagonal matrix associated with the LDL^T factorization of the matrix A . The updating operation of the matrix E is then of the form $E \leftarrow E - L_{off}D_AL_{off}^T$ where only the lower triangular part of E needs to be computed. No Level 3 BLAS kernel is available to perform this type of operation which corresponds to a generalized DSYRK kernel.

Note that, when we know that no pivoting will occur (symmetric positive definite matrices), L_{off} is computed in one step using the Level 3 BLAS kernel DTRSM. Otherwise, the trailing part of L_{off} has to be updated after each step of the blocked factorization, to allow for a stability test for choosing the pivot.

To update the matrix E , we have applied the ideas used by [63] to design efficient and portable Level 3 BLAS kernels. Blocking of the updating is done in the following way. At each step, a block of columns of E (E_k in Figure 2.6) is updated. In our first implementation of the algorithm, we stored the scaled matrix

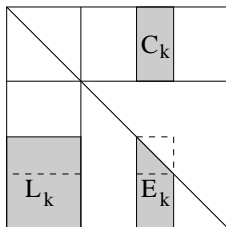


Figure 2.6: Blocks used for updates of the contribution part of a type 1 node.

$D_A L_{off}^T$ in matrix C , used here as workspace. Because of cache locality issues, the Megaflops/s rate was still much lower than that of the LU or Cholesky factorizations. In the current version of the algorithm, we compute the block of columns of $D_A L_{off}^T$ (C_k in Figure 2.6) only when it will be used to update E_k . Furthermore, to increase cache locality, the same working area is used to store all C_k matrices. This was possible because C_k matrices are never reused in the algorithm. Finally, the Level 3 BLAS kernel DGEMM is used to update the rectangular matrix E_k . This implies more operations but is more efficient on many platforms than the updates of the shaded trapezoidal submatrix of E_k using a combination of DGEMV and DGEMM kernels. Our blocked algorithm is summarized in Algorithm 2.2. In practice, a second level of blocking for the diagonal block of E_k is applied, avoiding most unnecessary operations in the upper-triangular diagonal block of E_k .

```

do k = 1, nb_blocks
  Compute  $C_k$  (block of columns of  $D_A L_{off}^T$ )
   $E_k \leftarrow E_k - L_k C_k$ 
end do

```

Algorithm 2.2: LDL^T factorization of type 1 nodes, Blocked factorization of the fully-summed columns.

2.1.5 Parallel factorization of type 2 nodes

Figure 2.7 illustrates the structure of a frontal matrix for the unsymmetric and symmetric cases. In both algorithms, the master processor is in charge of all the fully-summed rows and the blocked algorithms used to factor the block of fully-summed rows are the ones described in the previous subsection. In the unsymmetric case, remember that partial pivoting is here done with column interchanges. This is why it makes sense to use a partitioning by rows. In the symmetric case, a partitioning by columns could have been envisaged in order to simplify pivoting issues; however, the load of the master processor would then have been very large in comparison to the one of the slaves so that such a scheme would probably be less scalable.

In the unsymmetric case, at each block step, the master processor sends the factorized block of rows to its slave processors and then updates its trailing submatrix. The behaviour of the algorithm is illustrated in Figure 2.8, where program activity is represented in black, inactivity in grey, and messages by lines between processes. The figure is a trace record generated by the VAMPIR [131] package. We see that, on this example, the master processor is relatively more loaded than the slaves.

In the symmetric case, a different parallel algorithm has been implemented. The master of the node performs a blocked factorization of only the diagonal block of fully-summed rows. At each block step, its part of the factored block of columns is broadcast to all slaves ((1) in Figure 2.7). Each slave can then use

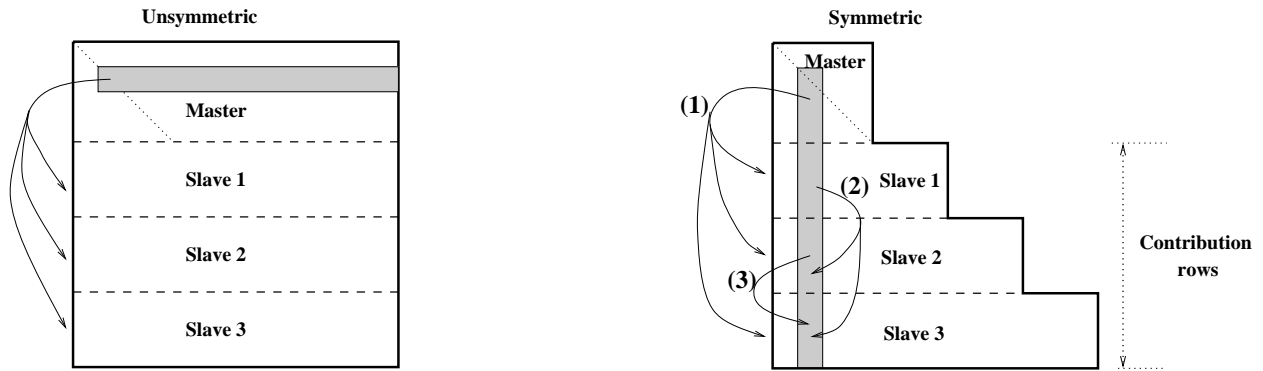


Figure 2.7: Structure of a type 2 node.

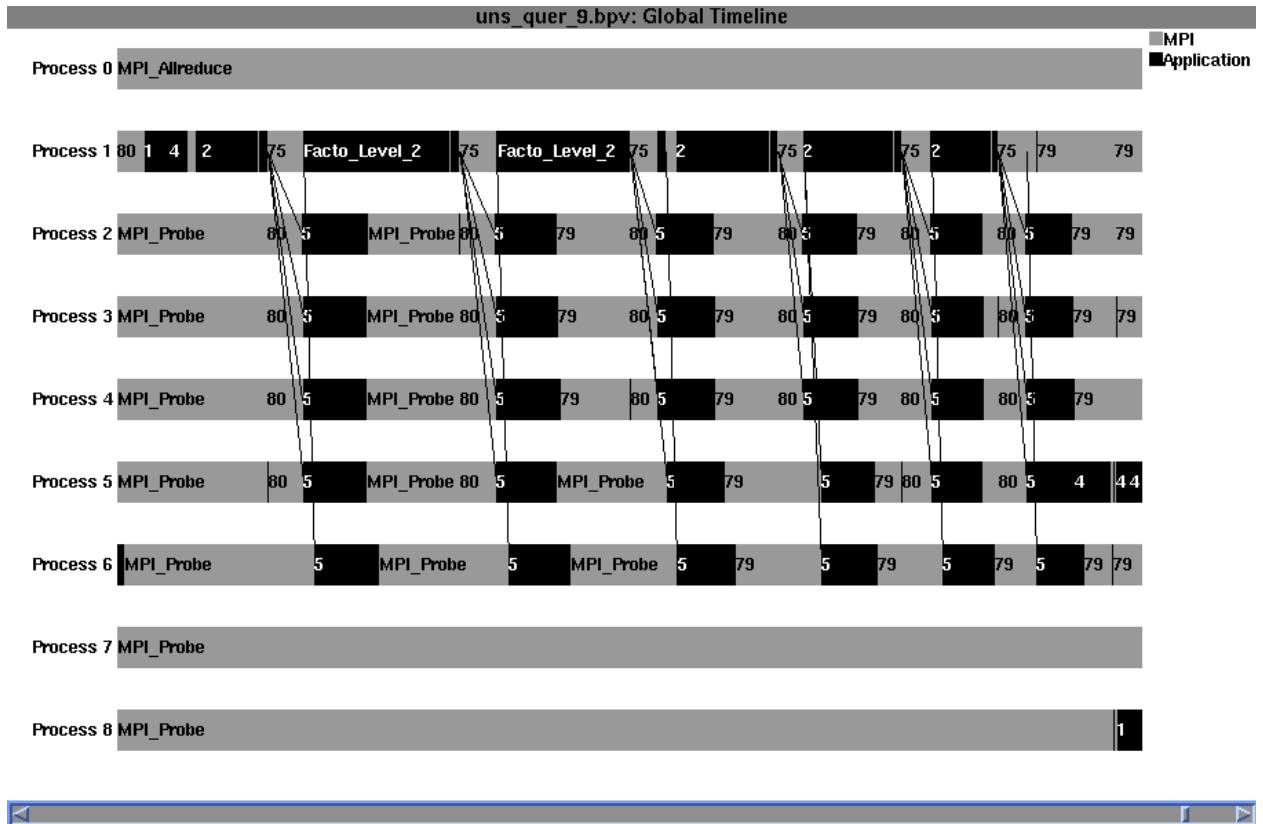


Figure 2.8: VAMPIR trace of an isolated type 2 unsymmetric factorization (Master is Process 1).

this information to compute its part of the block column of L and to update part of the trailing matrix. Each slave, apart from the last one, then broadcasts its just computed part of the block of column of L to the following slaves (illustrated by messages (2) and (3) in Figure 2.7). Note that, in order to process messages (2) or (3) at step k of the blocked factorization, the corresponding message (1) at step k must have been received and processed.

Since we have chosen a fully asynchronous approach, messages (1) and (2) might arrive in any order. The only property that MPI guarantees is that messages of type (1) will be received in the correct order because they come from the same source processor. When a message (2) at step k arrives too early, we have then to force the reception of all the pending messages of type (1) for steps smaller than or equal to k . Property 2.2 induces a necessary constraint in the broadcast process of messages (1): if at step k , message (1) is sent to slave 1, we must be sure that it will also be sent to the subsequent slaves. In our implementation of the broadcast, we first check availability of memory in the send buffer (with no duplication of data to be sent) before starting the actual send operations. Thus, if the asynchronous broadcast starts, it will complete. Another possibility, more specific to the described algorithm, would be to implement the broadcast in reverse order, starting with the last slaves (Property 2.2 would then be strictly respected).

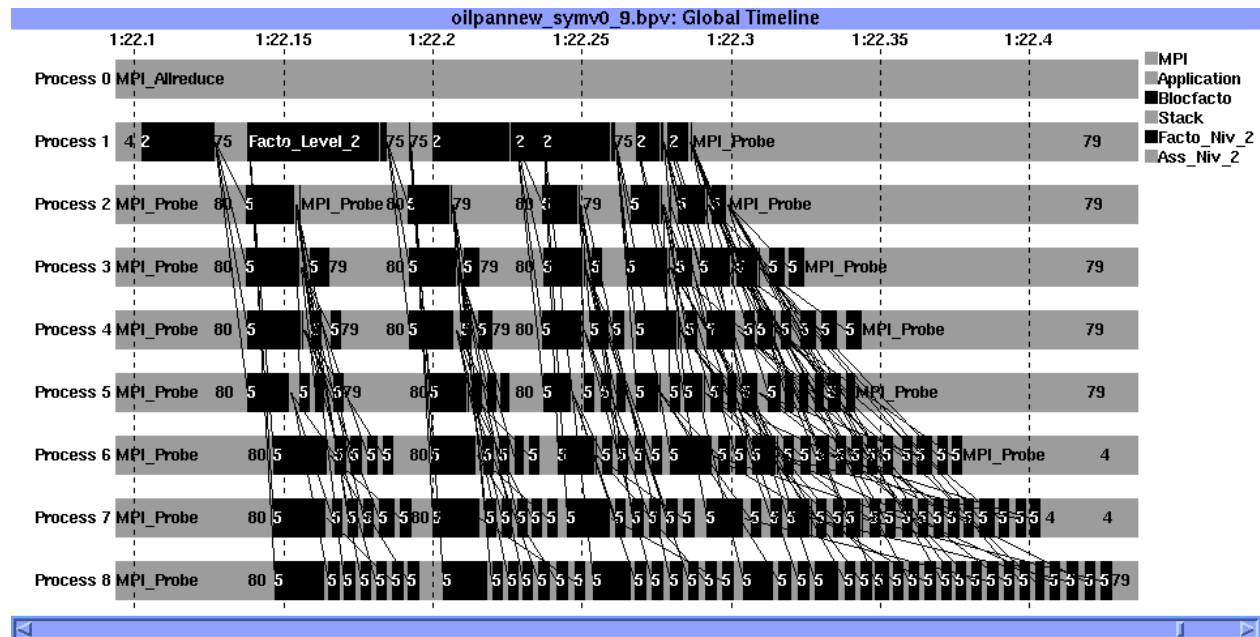


Figure 2.9: VAMPIR trace of an isolated type 2 symmetric factorization; constant row block sizes. (Master is Process 1).

Similarly to the unsymmetric case, our first implementation of the algorithm is based on constant row block size. We can clearly observe from the corresponding execution trace in Figure 2.9 that the later slaves have much more work to perform than the others. To balance work between slaves, later slaves should hold less rows. This has been implemented using a heuristic that aims at balancing the total number of floating-point operations involved in the type 2 node factorization on each slave. As a consequence, the number of rows treated varies from slave to slave. The corresponding execution trace is shown in Figure 2.10. We can observe that work on the slaves is much better balanced and both the difference between the termination times of the slaves and the elapsed time for factorization are reduced.

However, the comparison of Figures 2.8 and 2.10 shows that firstly the number of messages involved in the symmetric algorithm is much larger than in the unsymmetric case; secondly, that the master processor performs relatively less work than in the parallel algorithm for unsymmetric matrices.

To finish this section, let us remark that the elimination of the fully-summed rows can represent a

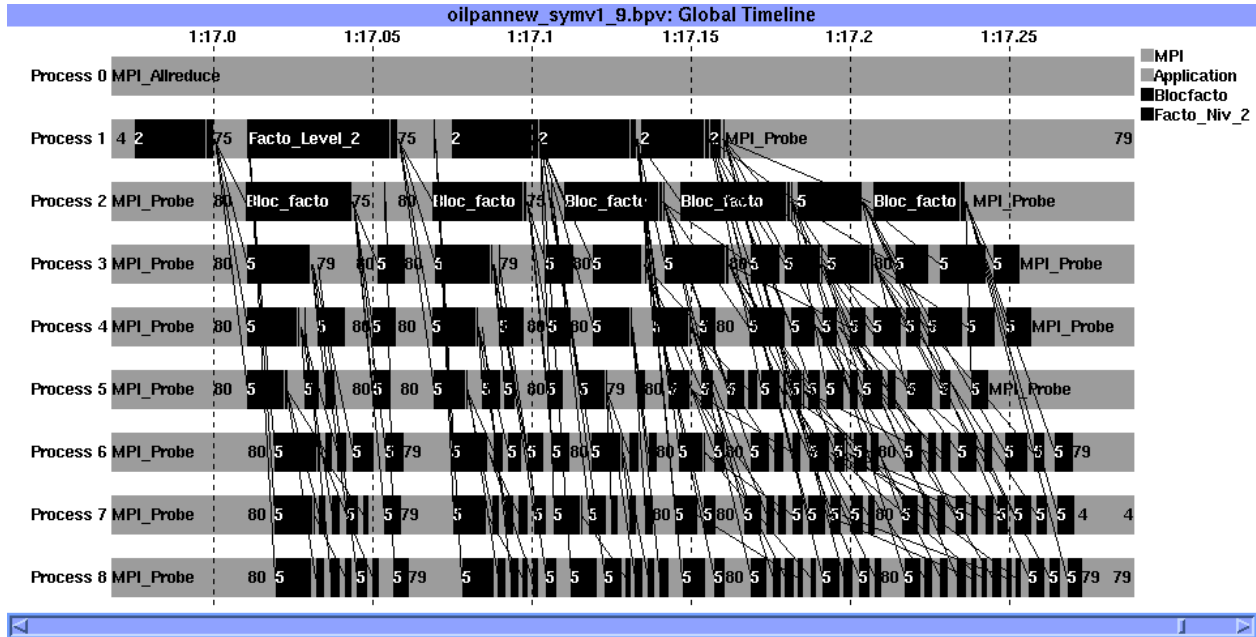


Figure 2.10: VAMPIR trace of an isolated type 2 symmetric factorization; variable row block sizes. (Master is Process 1).

potential bottleneck for scalability, especially for frontal matrices with a large fully-summed block near the root of the tree. To overcome this problem, we subdivide such nodes with large fully-summed blocks, as illustrated in Figure 2.11. In practice, a node is split recursively as much as needed, although this increases the number of assembly operations and volume of communication. Nevertheless, we benefit from splitting by increasing the amount of parallelism (the amount of work performed by the master decreases). Forcing the mapping of the rows in a way that limits the volume of communications during the processing of the chain is then useful, although it slightly limits the amount of dynamic scheduling decisions. Furthermore, the assembly operations can be optimized to avoid indirections and could even be avoided when the mapping of the rows is constrained. Another approach would consist in defining a new type of node (type 2b, say), and design a factorization algorithm that exploits several processors assigned to the fully summed rows of the frontal matrix. However, such a factorization algorithm may be complicated to implement in an asynchronous environment, or would require a strong synchronization among the processors participating to the factorization of the fully summed rows. Currently, this synchronization is done by using the parent-child dependency of the elimination tree.

2.1.6 Discussion

We aimed in this section at providing the basis of the asynchronous approach we proposed to implement parallel multifrontal methods. We refer the reader to [27, 24] for performance analysis, more detailed descriptions, and presentation of other features related to the software that implements this approach. For example, it is possible to use asynchronous (or immediate) communications not only on the sender side but also on the receiver side, significantly improving the performance [29], at the cost of slightly complexifying the asynchronous algorithms.

Various approaches defining the way the tree is mapped onto the processors together with the dynamic scheduling heuristics that can be used are discussed further in Chapter 4. Even with a relatively basic partial static mapping of the master processors for each node, dynamic decisions based solely on estimates of the workload of other processors allowed to reach a very competitive performance, as was shown in [110] and [28].

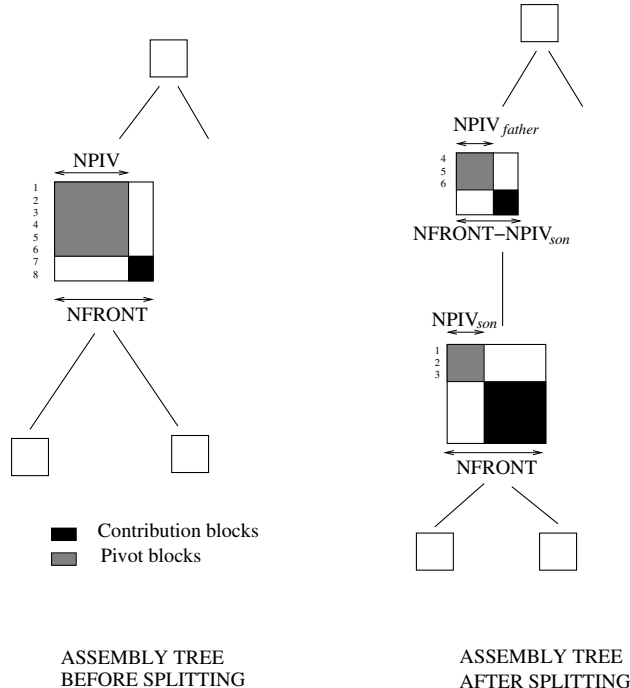


Figure 2.11: Assembly tree before and after the subdivision of a frontal matrix with a large pivot block.

The latter reference ([28]) includes comparisons with SuperLU_{dist} [124], and involves its main author.

2.2 Pivot selection

In the sequential algorithm, we use the pivot selection algorithm described in Sections 1.3.2.1 and 1.3.2.2. In parallel distributed environments, the main difficulty comes from the fact that, because data are distributed over the processors, it is not always possible to check the relative magnitude of a candidate pivot against the largest value on the row or the column, which may be on distant processors, or not even computed yet. In order to avoid this problem, and in order to avoid the dynamic data structures possibly resulting from pivoting, many approaches rely on *static pivoting* (see Section 1.3.2.4).

In the following subsections, we present the approach we have retained to handle pivoting in a parallel or out-of-core environment, and also how singular matrices can be handled.

2.2.1 Pivoting and stability issues in a parallel asynchronous context

We explained earlier how our parallel approach is capable of delaying pivots and managing dynamic data structures. Still, the decision of delaying a pivot is more difficult to take in parallel because part of the pivot column may reside on a different processor. In the unsymmetric version, because we use a 1D decomposition, we can access the complete row and check the stability of a pivot against the row. In the symmetric case (where only the lower triangular part of the matrix is defined), some columns must be distributed in order to have enough parallelism. This is done by what we call *type 2* parallelism, and we have to perform the partial factorization of a matrix that has the shape of Figure 2.7 (right).

Thus, neither the complete row nor the complete column are available to check for pivot stability. In the case of a synchronous approach, a reduction operation could inform the processor responsible for the pivot selection of the largest entry in the column. In an asynchronous approach, waiting for the information would require a synchronization which is not affordable: in our approach, a processor may start the partial

Matrix Name (origin)	Number of MPI processes	Number of type 2 nodes	Backward error and estimate of max in distant part of column	
			without	with
CONESHL (Samtech)	1	0	6.5×10^{-13}	
	4	5	2.1×10^{-11}	5.5×10^{-13}
CONT-201 (Uf1 collec.)	1	0	3.2×10^{-9}	
	8	6	1.0	3.1×10^{-9}
CONESHL2 (Samtech)	1	0	9.4×10^{-11}	
	2	1	1.3×10^{-6}	1.4×10^{-10}
	8	14	4.1×10^{-2}	1.3×10^{-10}

Table 2.1: Effect on the backward error (value of $RINFOG(7)$ in MUMPS) of taking into account estimates of the magnitude of elements of columns mapped on other processors. The mechanism computing such estimates is necessary (and sufficient) to stabilize the numerical behaviour of the parallel code. No iterative refinement. Except on problem CONESHL2 where the right-hand side was available, we used a right-hand side vector whose entries are all equal to one.

factorization of a frontal matrix even before the other processors have finished to assemble their share of the matrix. When the data we are interested in are assembled, it is too late to send the information to the processor in charge of the pivot selection. An alternative approach would consist in forcing the entire fully summed columns to be on a single processor (instead of the shape of type 2 nodes from Figure 2.7), but this would seriously limit parallelism by forcing a too large granularity of the task associated with the master process.

We have introduced an approach to address this issue. For each column, the idea consists in checking the pivot against the maximal magnitudes of the contributions assembled into the column, instead of the magnitude of the column after it is assembled and fully updated. A first study was done in [78] and showed that this should result in a reasonable numerical behaviour on the problems tested. In our distributed-memory approach, each child (more precisely each processor involved in a child) of the considered node sends information about the largest magnitude of each of its columns to the processor in charge of the pivot selection at the parent level. Note that a message had to be sent anyway, containing the contributions. Therefore, the contribution information and magnitude information are sent in the same message. Furthermore, only the information corresponding to columns that will be fully-summed in the parent are sent. We use an array *PARPIV* to store those estimates at the parent. The size of *PARPIV* is *NFS*, the number of fully summed variables in the front. At the construction of the front, *PARPIV*(*i*) ($1 \leq i \leq NFS$) is initialized to the maximum arrowhead value to be assembled in column *i* of the front, excluding those assembled in the master part; then each time information on a given column *i* is received from a child process, *PARPIV*(*i*) is updated. When the master of the parent node factors its frontal matrix, each pivot is checked against those maximal magnitudes before being accepted (see the details in the algorithm of Section 2.2.4).

Table 2.1 reports on the effect of this strategy on three example problems. Serial executions are stable numerically thanks to the availability of the full column on the processor, but we observe that parallel executions with type 2 nodes require the mechanism that builds estimates of the magnitudes from the contributions of the children to obtain a reasonable backward error.

Recently, this approach showed some limitations on two matrices from a French industrial group with whom we collaborate. We have modified it in order to take into account modifications of the estimates of the magnitude of the columns during the factorization. Those modifications can for example be based on the growth factor observed in the pivot block of the master part. Another approach which seems promising consists in updating *PARPIV* as if it was an extra row in the front: at each factorization of a pivot $F(k, k)$ in the frontal matrix *F*, the estimates can be updated as follows:

$$PARPIV(k+1 : NFS) \leftarrow PARPIV(k+1 : NFS) + PARPIV(k) / |F(k, k)| \times |F(k, k+1 : NFS)|. \quad (2.1)$$

In the case of 2×2 pivots, a (slightly more complicated) formula can also be obtained.

2.2.2 Detection of null pivots and rank-revealing

When the input matrix is close to singularity, it is often useful from an application’s point of view to detect null or tiny pivots during the factorization process. The global variable associated with the null pivot is often of interest to the user because it may correspond to special physical properties or regions of the mesh where there is a difficulty (*e.g.*, too many constraints). In FETI-like methods, domains are not always fixed so that the corresponding matrices passed to the direct solver are usually singular (with typical deficiency 6). In some applications, the deficiency may be much bigger. The definition of a quasi-null pivot is subjective. In practice we will say that a pivot is quasi-null if it is smaller than a given tiny threshold, α , and if its whole row (or column) is also smaller than α . In that case, we want to isolate, or exclude that pivot from the matrix. This can be done by setting the pivot to a huge value: after dividing the column by this value, rank 1 updates will have no influence on the rest of the computations; or by setting the pivot to 1 and forcing the column entries to 0. Furthermore, some pivot rows/columns may be small enough that we can suspect a rank deficiency but large enough to have doubts. In that case, it is usually better to avoid factoring the pivot and delaying it to the parent node (or higher in the tree) where it will be checked again. We note β this second criterion. In the multifrontal method, pivots smaller than β may be postponed until the root, where a more reliable rank-revealing QR factorization can be performed. Once null pivots have been detected, one may also build a null space basis thanks to backward solve algorithms. The right-hand side vectors are either null vectors of the root node (for deficiencies detected with QR), or vectors composed of zeros except for a 1 at the position corresponding to a null pivot. Exploiting sparsity during the solve phase may in that case is then of interest to reduce the amount of computation [158].

2.2.3 Pivoting and out-of-core

We will discuss out-of-core in a Chapter 5, but assume here that we want to perform the factorization of a frontal matrix F , writing blocks of L or U to disk in a pipelined manner, as soon as those blocks are computed. In the LAPACK-style of pivoting (see remark of Section 1.3.2.3), one needs to permute rows and/or columns that are already on disk. Since this is not convenient, we store this symbolic pivoting information (of small size compared to the factors of the frontal matrix) during the factorization and will use it later at the solution phase, after reading factors from disk. We consider that performing the permutation explicitly is cheap compared to the cost of I/O , and this avoids complexifying too much the solve algorithm.

2.2.4 Algorithm for pivot selection (symmetric indefinite case)

A simplified version of the pivot selection algorithm is given by Algorithm 2.3. We illustrate the pivot selection by considering here the case of type 2 nodes involved in symmetric indefinite matrices. In type 2 nodes, only the fully summed part of the symmetric front is available on the local master processor, and we assume that for each column i , $PARPIV(i)$ contains an estimate of the maximum magnitude in $F(1 : NFS, i)$ using the mechanism described in Section 2.2.1. The unblocked version of the algorithm is here provided³. The selection of the 2×2 pivots at lines 20–24 corresponds to the Duff-Reid algorithm [80]. Remark the search for the maximum element at line 20: part of the search is on the column $F(i + 1 : NFS, i)$ below the candidate pivot i , and part is on the row $F(i, k + 1 : i - 1)$ on the left of $F(i, i)$ because that part of the row would become part of the column after a potential swap. The candidate pivot in this Duff-Reid algorithm is then (i, j) .

If one wants to avoid dynamic data structures, it is possible to switch to static pivoting (see Section 1.3.2.4). In our context, static pivoting can for example be useful when we want to precisely respect the

³In the blocked version of the algorithm, the pivot selection is limited to the variables in the current block: authorized pivots must be in the square diagonal block of the current panel instead of the range $k : NFS$ of the loop at line 1 and instead of the range $i + 1 : NFS$ of the max at line 19. Furthermore, when numerical difficulties are encountered, the block size dynamically increases in order to avoid the situation where only unstable pivots would be available in the current block. Thanks to this mechanism, all stable pivots in the front are eventually factorized.

memory estimates forecasted during analysis. However, even then, it makes sense to apply standard partial threshold pivoting with 1×1 and 2×2 pivots as much as possible, so that we only switch to static pivoting (replacing small pivots by larger ones or accepting pivots which do not satisfy the pivoting threshold) when no more stable pivots are available in the current front.

In case of real arithmetic, the count of the number of negative pivots (which is also equal to the number of negative eigenvalues) can be computed by checking the sign of the chosen pivots. If a 1×1 pivot is negative or if the determinant of a 2×2 pivot is negative, then the number of negative pivots is increased by one. If the determinant of a 2×2 pivot is positive, then the inertia is increased by 2 if the sign of the diagonal elements (both necessarily have the same sign) is negative, and the inertia is not modified otherwise.

2.3 Schur complement

In several types of applications (domain decomposition, coupled problems, reduction of the problem on an interface), it is useful to factorize only part of the matrix and return a Schur complement matrix corresponding to the non-factorized part. Consider the following 2×2 block factorization, where variables defining the Schur complement are ordered last.

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{pmatrix} = \begin{pmatrix} \mathbf{L}_{1,1} & 0 \\ \mathbf{L}_{2,1} & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{U}_{1,1} & \mathbf{U}_{1,2} \\ 0 & \mathbf{S} \end{pmatrix} \quad (2.2)$$

The Schur complement is $\mathbf{S} = \mathbf{A}_{2,2} - \mathbf{A}_{2,1}\mathbf{A}_{1,1}^{-1}\mathbf{A}_{1,2}$. Computing a Schur complement in a multifrontal approach can be done in the following way:

1. The ordering should force the variables of the Schur complement to be ordered last (constrained ordering).
2. The structure of the assembly tree should be such that the variables of the Schur complement form the last node.
3. Delayed pivots are forbidden in the children of the root, in order to preserve the structure of the Schur.
4. The root node is assembled normally, but not factorized.
5. The root node should be returned to the user.

In certain cases, the Schur complement can be big, and may not hold in the memory of a single processor. Even if it holds in the memory of a single processor, it makes sense to have it distributed on the processors so that it can be efficiently used in parallel computations. For that, the Schur complement must be assembled on the processors directly in a distributed manner. This work was first done in the context of a collaboration, for an application involving a linear system coupling finite elements and integral equations, and in which part of the linear system is sparse, whereas another part is dense. The Schur complement of the sparse linear system is then used as a submatrix of the dense linear system that can be solved using, for example, ScaLAPACK [53]. The main difficulties concern the interface, the constraints on the various orderings, the need to write directly into user-allocated workspaces, not counting the 2D block cyclic assembly of children contributions that are communicated from processes using 1D data distributions. Remark that (see point 3 above), because the Schur complement memory is provided by the application, the possibility to delay pivots is switched off in the children of the root because there would be no place to store them. However, this is mainly an interface problem because such mechanisms are available for type 3 nodes when the root node must be factorized by the solver, instead of being returned in the interface. Concerning the solve phase, only the solution on the internal problem is available (see Section 2.5 for more advanced functionalities), that is, the root node is excluded from the forward elimination algorithm, and a 0 vector is used on entry to the backward substitution algorithm as the solution on the root node.

```

1: for  $i = k$  to  $NFS$  do
2:    $Fmax = \max(F(i, k : i - 1), F(i + 1 : NFS, i), PARPIV(i))$ 
3:   if  $Fmax < \alpha$  then
4:     A quasi-null pivot was found
5:      $PIVNUL\_LIST = PIVNUL\_LIST \cup$  global variable associated with  $i$ 
6:     Symmetric swap of variables  $i$  and  $k$ 
7:     if  $\delta > 0$  then
8:        $F(k,k) = \text{sign}(F(k,k)) \times \delta \|A\|$ 
9:     else
10:       $F(k,k) = 1$ 
11:      Force  $F(k:NFS,k) = 0$ 
12:    end if
13:    Exit loop: a quasi-null pivot has been found.
14:  else if  $Fmax > \beta$  and  $F(i,i) < u \times Fmax$  then
15:    Symmetric swap of variables  $i$  and  $k$  in  $F$ 
16:    Swap  $PARPIV(i)$  and  $PARPIV(k)$ 
17:    Exit loop: a new pivot can now be eliminated
18:  else
19:    {Pivot  $i$  cannot be eliminated alone; try to find a  $2 \times 2$  pivot}
20:    Let  $j$  be such that  $F(j, i) = \max_{j=i+1:NFS} F(j, i)$  (or  $F(i, j) = \max_{j=k+1:i-1} F(i, j)$  if larger)
21:    if  $(i, j)$  forms a stable  $2 \times 2$  pivot (see condition (1.16)) then
22:      Symmetric swap moving variables  $i, j$  at positions  $k, k + 1$  in  $F$ 
23:      Swap  $PARPIV(i)$  and  $PARPIV(k)$ ,  $PARPIV(j)$  and  $PARPIV(k + 1)$ 
24:      Exit loop: a new  $2 \times 2$  pivot can be eliminated
25:    else
26:      No pivot was found including  $i$ ; try next iterate
27:    end if
28:  end if
29: end for
30: if a quasi-null pivot was found or a  $1 \times 1$  pivot can be eliminated then
31:   Eliminate  $1 \times 1$  pivot  $i$  (now at position  $k$ )
32:   Update  $PARPIV(k + 1 : NFS)$  (see Equation (2.1))
33:    $k \leftarrow k + 1$ 
34: else if a  $2 \times 2$  pivot can be eliminated then
35:   Eliminate  $2 \times 2$  pivot  $i, j$  (now at position  $k, k + 1$ )
36:   Update  $PARPIV(k + 2 : NFS)$ 
37:    $k \leftarrow k + 2$ 
38: else
39:   No stable pivot was found in entire loop. Delay variables  $k : NFS$  to parent node or enable static pivoting to pursue the factorization.
40: end if

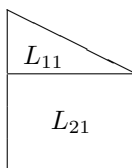
```

Algorithm 2.3: Selection of a new pivot k in a symmetric front F of size $N \times N$ with NFS fully-summed variables, unblocked algorithm. The front is assumed to be distributed in such a way that the $NFS \times NFS$ first block is on the local processor, and the $N - NFS$ last rows are on other processors. $PARPIV(1 : NFS)$ contains an estimate of the maximum magnitudes of the columns of $F(NFS + 1 : N, 1 : NFS)$ the columns of the front, as explained in Section 2.2.1. We assume that $k - 1$ pivots have been eliminated already and that we are looking for the k^{th} one. Static pivoting is assumed to be off. u is the threshold for relaxed partial pivoting, α and β are the criteria defined in Section 2.2.2 for small pivot detection and for delaying pivots, respectively.

2.4 Solution phase

Given a linear system $Ax = b$ and an LU decomposition of the matrix A , the solution phase consists of a forward elimination followed by a backward substitution, involving matrices L and U , respectively: The forward elimination consists in a triangular solve on L , processing the nodes of the tree from bottom to top, and the backward substitution consists in a triangular solve on U , starting from the solution on the root node, and passing it to all children nodes until the bottom of the tree to compute the solution corresponding to the fully summed variables at each node. The multifrontal triangular solution algorithms were presented in Section 1.1.4. In this section, we generalize those algorithms to supernodes and to a parallel distributed environment using an asynchronous approach to parallelism. The new algorithms take advantage of both tree and node parallelism by inheriting the mapping and the Type 1, Type 2 and Type 3 nodes from the factorization phase.

Let us first consider the **forward elimination**. A lower triangular factor at a generic node of the tree has the following shape:



A solve on L_{11} is first performed, and the result is multiplied on the left by L_{21} and used as a contribution for the parent node. This can be simply interpreted as a generalization of Algorithm 1.3 for supernodes. In Algorithm 1.3, L_{11} was just the scalar l_{11} , and L_{21} was just one column. In our parallel distributed-memory environment, the algorithm for the forward elimination becomes Algorithm 2.4. This algorithm was first described in [26], and was initially inspired by [85]; we also recommend [158] which contains more explanations and illustrations. For the sake of clarity, Algorithm 2.4 is simplified compared to actual implementations. For example, when *Myid* and *Pparent* are the same processor, no message is actually sent so that the actions that would be performed on reception are performed locally, directly. Also, in the case of delayed pivots, the master of a node owns some rows of L_{21} corresponding to the pivots that are delayed to the parent, implying some computations and communications related to *Wtmp2*. We have omitted those, and also do not discuss the fact that the root node may be processed using ScaLAPACK. Finally, the algorithm is presented in the case of a single right-hand side vector.

The main loop of the forward elimination algorithm (lines 6 to 13 is asynchronous, similar to the factorization algorithm (Algorithm 0.1). All sends are asynchronous (use of the `MPI_ISEND` routine) and a cyclic communication buffer is also used. When a communication buffer is full, no sends are possible, and the associated processor can receive any message before trying to send again, avoiding deadlocks (see also Property 2.1).

The main workarray in the algorithm is Wb , of size n , the order of the matrix. At line 5 of the algorithm, the right-hand side b is distributed into the Wb local vectors of size n (the order of the matrix), which are such that:

$$\begin{cases} (1) & Wb(i) = b(i), \text{ if variable } i \text{ is part of the pivot block of a front mapped on Myid;} \\ (2) & Wb(i) = 0, \text{ otherwise.} \end{cases} \quad (2.3)$$

Wb is then used to store temporary data (accumulated contributions of the form $-l_{ij}y_j$) during the forward elimination, avoiding the use of a stack as was done in the factorization algorithm. At line 20 of the algorithm, Wb is also used to store the solution vector $Wtmp1$ associated with the fully summed variables of the current node, so that after the forward elimination is complete, the (distributed) $Wb(i)$ entries corresponding to condition (1) above hold the solution y of $Ly = b$.

In case of type 2 nodes, the L_{21} factor is distributed over the slave processors, so that the master first sends (line 28) the solution $Wtmp1$ together with partial contributions for the slaves to perform the matrix-vector product (line 33). The slaves do not answer to the master but send the contribution directly to the

```

1: Main Algorithm (forward elimination):
2: {Input: the right-hand side  $b$ , on processor 0}
3: {Output:  $Wb$ , on all processors}
4: Initialize a pool with the leaf nodes mapped on  $Myid$ 
5: Communicate and store into  $Wb$  the entries of the right-hand side  $b$  corresponding to variables in the
   pivot block of nodes mapped on  $Myid$  (scatter)
6: while Termination not detected do
7:   if message is available then
8:     Process the message
9:   else if pool is not empty then
10:    Extract a node  $\mathcal{N}$  from the pool
11:    Fwd_Process_node( $\mathcal{N}$ )
12:   end if
13: end while
14:
15: Fwd_Process_node( $\mathcal{N}$ )
16: { $L_{11}$  and  $L_{21}$  are the  $L$  factors of  $\mathcal{N}$  }
17: { $Pparent$  is the process owning the master of the parent of  $\mathcal{N}$  }
18:  $Wtmp1 \leftarrow$  Entries of  $Wb$  corresponding to fully summed variables of  $\mathcal{N}$ 
19:  $Wtmp1 \leftarrow L_{11}^{-1} \times Wtmp1$  (or  $U_{11}^T \times Wtmp1$ ).
20: Store entries of  $Wtmp1$  back into  $Wb$  (scatter).
21: Gather in  $Wtmp2$  entries of  $Wb$  corresponding to row indices of  $L_{21}$ 
22: Reset the corresponding entries of  $Wb$  to zero.
23: if  $\mathcal{N}$  is of Type 1 then
24:    $Wtmp2 = Wtmp2 - L_{21} \times Wtmp1$ 
25:   Send the resulting contribution ( $Wtmp2$ ) to  $Pparent$ 
26: else if  $\mathcal{N}$  is of Type 2 then
27:   for all slave  $Islave$  of  $\mathcal{N}$  do
28:     Send  $Wtmp1$  together with the rows of  $Wtmp2$  corresponding to rows of  $L_{21}$  owned by  $Islave$  to
       the process in charge of  $Islave$ 
29:   end for
30: end if
31:
32: On reception of  $Wtmp1 +$  rows of  $Wtmp2$  by a slave
33: Multiply rows of  $L_{21}$  owned by the slave by  $Wtmp1$  and subtract the result from the received rows of
    $Wtmp2$ 
34: Send the resulting contribution to  $Pparent$ 
35:
36: On reception of a contribution corresponding to  $\mathcal{N}$  by  $Pparent$ 
37: Assemble the contribution into  $Wb$  (Scatter)
38: if all contributions for node  $\mathcal{N}$  have been received by  $Pparent$  then
39:   Insert parent of  $\mathcal{N}$  into the pool
40: end if

```

Algorithm 2.4: Forward elimination algorithms.

parent node (line 34), avoiding an extra message and the need for the master to *wait* for an answer from its slave.

Coming back to Wb , it is important to note that when the contributions stored in Wb are consumed and sent to a parent node (via array $Wtmp2$), the corresponding entries must necessarily be reset to zero (line 22 of the algorithm). Otherwise, some contributions might be sent a second time, at a different node, leading to a wrong algorithm.

Let us now consider the **backward substitution**. At each node, factors have the shape:

$$\left[\begin{array}{c|c} U_{11} & U_{12} \end{array} \right],$$

where U_{12} might be distributed over several processes in the case of a type 2 node⁴. During the backward substitution algorithm, the tree is processed from top to bottom and each node requires the entries of the solution vector corresponding to column indices of U_{12} . Since those are included in the structure of the parent node, a sufficient condition to know the required entries is to inherit from the entries of the solution vector corresponding to the entire set of column indices of the parent node.

At each step of Algorithm 2.5, the local solution x_2 corresponding to columns of U_{12} is thus available and the solution x_1 corresponding to column variables of U_{11} are computed; y_1 is the part of the right-hand side corresponding to variables of U_{11} and comes from the Wb array computed during the forward elimination phase (line 18 of the algorithm). With these notations, the system that must be solved at each node is $U_{11}x_1 = y_1 - U_{12}x_2$ (at the root, U_{12} and x_2 are empty), where both x_1 and x_2 are then sent to the children nodes, although each child only requires parts of them.

Throughout the algorithm, $Wsol$ is used to save parts of the solution, with the property that the solution for variable i will at least be available in $Wsol(i)$ on the processor in charge of the pivot block containing i . For type 2 nodes, we see in the algorithm that the slave does a matrix-vector product (matrix-matrix in case of multiple right-hand sides) and sends the result back to the sender, implying more communications than in the forward elimination algorithm where slaves do not have to send anything back to their master.

Algorithms 2.4 and 2.5 can be adapted to different contexts:

- When there are multiple right-hand sides, they can be processed by blocks (block size typically between 16 and 128), so that Wb and $Wsol$ are allocated once and with a number of columns equal to the block size. This allows for a good efficiency of the level 3 BLAS routines TRSM and GEMM, while avoiding a huge, possibly unaffordable, workspace for Wb and $Wsol$ if only one block were used. The sketch of the approach is given by Algorithm 2.6, where the loop on the blocks is external to both the forward and backward elimination algorithms.
- In an out-of-core context, factors must be read from disk with prefetching mechanisms and an adapted scheduling [23].
- In the case of sparse right-hand sides, entries must be distributed into Wb while respecting the condition 2.3 above; furthermore, tree pruning can be used to avoid computations on zeros during the forward elimination. For example, if all variables of the right-hand side corresponding to fully summed variables in a given subtree are zero, that subtree can be excluded from the computations in Algorithm 2.4. Furthermore, if the matrix is reducible, and if the right-hand side is 0 for all entries corresponding to fully summed variables in one of the blocks, the forest can be pruned in both the forward and the backward elimination.
- Similar to the sparse right-hand side case, if only selected entries of the solution are requested, the complexity of Algorithm 2.5 can be reduced by excluding computations on subtrees where none of the variables of the solution are of interest to the user.

⁴Actually, during an unsymmetric factorization, U_{21} is always on the master and is not distributed. However, if $A^T x$ is to be solved, or in case of an LDL^T factorization, L_{21}^T can be distributed over several slave processors. To simplify the presentation, we only mention U_{12} and consider that U_{12} may be distributed on the slaves in the backward substitution.

```

1: Main Algorithm (backward substitution):
2: {On input:  $Wb$  is the vector obtained on output from Algorithm 2.4}
3: {Output:  $Wsol$ }
4: Initialize the pool with the roots mapped on  $Myid$ 
5: while Termination not detected do
6:   if message is available then
7:     Process the message
8:   else if pool is not empty then
9:     Extract a node  $\mathcal{N}$  from the pool
10:    Bwd_Process_node( $\mathcal{N}$ )
11:   end if
12: end while
13: Gather solution from  $Wsol$  arrays to the host (or keep it distributed) in order to return it to the user
14:
15: Bwd_Process_node( $\mathcal{N}$ )
16:  $x_2 \leftarrow$  known entries of solution corresponding to columns of  $U_{12}$  (gather from  $Wsol$ )
17: if  $\mathcal{N}$  is of type 1 then
18:    $y_1 \leftarrow$  entries of  $Wb$  corresponding to variables in the pivot block  $U_{11}$  (gather, row indices)
19:   Solve  $U_{11}x_1 = y_1 - U_{12}x_2$  for  $x_1$ 
20:   Save  $x_1$  in  $Wsol$  (scatter)
21:   Send partial solution  $x_1, x_2$  to masters of children nodes (only one send per destination process)
22: else if  $\mathcal{N}$  is of type 2 then
23:   Send (distribute) entries of  $x_2$  to the slaves, according to their structure
24: end if
25:
26: On reception of  $x_1, x_2$ , sent by the master of node  $\mathcal{N}$ 
27: Update my view of the solution (scatter into  $Wsol$ )
28: Insert children of  $\mathcal{N}$  mapped on  $Myid$  into the local pool
29:
30: On reception of parts of  $x_2$  by a slave of  $\mathcal{N}$ 
31: Multiply the part of  $U_{12}$  mapped on  $Myid$  by the piece of  $x_2$  just received
32: Send the negative of the result back to the master process of  $\mathcal{N}$ 
33:
34: On reception of a portion of  $-U_{12}x_2$  from a slave by a master for node  $\mathcal{N}$ 
35: Scatter and add it into  $Wb$ 
36: if this is the last update (all slaves sent their part) then
37:    $y_1 \leftarrow$  entries of  $Wb$  corresponding to variables in the pivot block  $U_{11}$  (gather, row indices)
38:   Solve  $U_{11}x_1 = y_1$  for  $x_1$ 
39:   Save  $x_1$  in  $Wsol$  (scatter, using column indices of  $U_{11}$ )
40:   Send partial solution  $x_1, x_2$  to masters of children nodes
41: end if

```

Algorithm 2.5: Backward substitution algorithms.

Allocate Wb and $Wsol$ of size $b \times n$

for $i=1$ **to** nb_{rhs} **by steps of** b **do**

$ibeg \leftarrow i$

$iend \leftarrow \min(ibeg + b - 1, nb_{rhs})$

 On each processor, initialize Wb with entries of the right-hand sides in columns $ibeg : iend$ (one-to-all communications to ensure property 2.3)

$Wb \leftarrow$ result of forward elimination (Algorithm 2.4)

$Wsol \leftarrow$ result of backward elimination (Algorithm 2.5)

 Store $Wsol$ back into user workspace in columns $ibeg : iend$

end for

Algorithm 2.6: Forward and backward eliminations by blocks. b is the block size, n is the order of the matrix, nb_{rhs} is the total number of right-hand sides.

- For applications requiring the computation of a set of entries of the inverse, the tree can also be pruned, both during forward and backward elimination: $(A^{-1})_{ij}$ is obtained by computing $x = L^{-1}e_j$, exploiting the sparsity of the j^{th} canonical vector e_j , then obtaining the i^{th} component of $U^{-1}x$, exploiting the sparsity of requested entries of the solution. Pruning the tree is even more important in an out-of-core context, where the access to the factors is far more critical than in an in-core context. When several entries of the inverse are requested, it is interesting to group them in blocks that will follow similar paths in the tree. A detailed study is available in [30].
- In case null pivots have been isolated during the factorization, a null space basis can also be computed with the backward substitution algorithm 2.5. Wb is initialized directly on entry to the backward step with one nonzero element per column, which can be the value δ of Algorithm 2.3, or simply 1 if in Algorithm 2.3, it was decided to replace the pivot by 1 and set the column to 0. Algorithm 2.5 is applied with those vectors on input. Remark that this strategy works for symmetric matrices, but not for unsymmetric matrices, where null pivots would need to be at the end for this approach to provide a correct null space basis.
- In case of Schur complement, it can be useful to return an intermediate solution corresponding to the variables of the Schur complement between the forward and the backward substitutions. This will be discussed in Section 2.5.
- When solving large problems with multiple right-hand side vectors, the workspace for Wb and $Wsol$ may become problematic in terms of memory. A modified algorithm that builds over the one of Section 2.5 together with several optimizations is discussed in Section 6.4.

2.5 Reduced/condensed right-hand side in solve phase

In this section, we explain the work done to provide a new *reduced-right-hand side* functionality and present some modifications to the solve algorithm which, although motivated by this new functionality, are useful in a more general context.

Related to the Schur complement functionality, a strong need appeared from the applications which consisted in separating the forward and backward solution phases, in order to return an intermediate right-hand side vector of smaller size (*reduced right-hand side*, or *condensed right-hand side*) in-between. The solution on the interface is computed externally and may be injected back in the backward solution step of the solver to obtain the solution on the complete problem:

- Reduction/condensation phase: An intermediate vector $y = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$ is computed, such that

$$\begin{pmatrix} \mathbf{L}_{1,1} & 0 \\ \mathbf{L}_{2,1} & \mathbf{I} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}. \quad (2.4)$$

y_2 , the so called *Reduced right-hand side*, or *Condensed right-hand side*, is returned to the user application.

- The linear system involving the Schur complement must then be solved. In the simplest case, this is just $Sx_2 = y_2$ but in general, x_2 and y_2 are just part of much larger problem. For example, in domain decomposition methods, x_2 and y_2 correspond to a subset of variables of a much larger interface problem because each domain leads to a local Schur complement; the interface problem indeed looks like $\sum S^k X = \sum y_2^k$, y_2 is one of the y_2^k , S is the corresponding S^k and x_2 is the part of X concerned by the interface variables of domain k .
- Expansion phase:
Given x_2 , compute $x_1 = U_{11}^{-1}(y_1 - U_{12}x_2)$: the solution x_2 is expanded to obtain the solution on the *internal* variables.

The condensation and expansion phases are performed using Algorithms 2.4 and 2.5, respectively. As for the factorization with a Schur complement, the root node which corresponds to the Schur complement receives a special treatment. During the forward elimination, **Fwd_Process_node** in Algorithm 2.4 is avoided for the root node corresponding to the Schur complement and at the end, Wb contains the vectors y_1 and y_2 . On entry to the backward substitution, x_2 which is built externally must be provided, and most of the treatment from the routine **Bwd_Process_node** in Algorithm 2.5 must be avoided: x_2 will be sent to the children of the root. At the end of the backward substitution, $Wsol$ contains both x_1 and x_2 .

However, other modifications are required to manage workspace efficiently in a parallel distributed environment. First, remark that in the previous algorithms, Wb and $Wsol$ are temporary workspaces needed⁵ on all processes. They are of size n (multiplied by the blocking factor for multiple right-hand sides, in case of multiple right-hand sides), where n is the size of the initial matrix A . On exit from the solve phase, both Wb and $Wsol$ are freed but in case of a separate forward and backward step, all the Wb workarrays from the forward eliminations (see Algorithm 2.6) would have to be kept between the forward and backward stages.

In case of multiple right-hand sides, keeping Wb in memory for all the columns and all the blocks becomes problematic from the memory point of view, as illustrated by the following example: let us consider a problem with 1 million equations and 10000 right-hand sides processed by blocks of 16 columns, using 100 processors. Normally, Wb and $Wsol$ are the same workspace for all the blocks so that in double precision arithmetic, the workspace for Wb represents $100 \times 16 \times 1 \text{ million} \times 8 \text{ bytes per entry} = 12.8 \text{ GigaBytes}$, or 128 MB per processor, which remains reasonably small. In order to keep Wb for all 10000 columns between the reduction and the expansion phases, that is, in order for Algorithm 2.5 to apply with 10000 intermediate right-hand side vectors y on input, the required workspace for Wb would be $100 \times 10000 \times 1 \text{ million} \times 8 = 8 \text{ TeraBytes}$, which is clearly less affordable! This means that the workspace for y_1 must remain much smaller. One solution would consist in centralizing the useful information of Wb on just one processor after each block of right-hand sides, but this would require lots of communications and could still be a memory bottleneck for that processor. A more natural solution consists in keeping the intermediate right-hand sides distributed.

For that, Algorithm 2.4 has been modified as follows:

- We introduce a new workspace **WRHS** which scales with the number of processes and contains, for each process, only the entries corresponding to the pivot blocks of the fronts mapped on this process. In case of type 2 node, only the master is concerned with those variables and is in charge of it: slave processors never update **WRHS**. In case of multiple right-hand sides, the number of columns of **WRHS** is either equal to the block size (if the blocking is outside both the forward and the backward stages), or equal to the total number of columns, if all intermediate right-hand sides must be kept between the forward and backward stages, as required by this reduced-right-hand side functionality.
- On each process, an indirection array **POSinWRHS** is precomputed to obtain, for each node of the tree, its position in **WRHS**. By *its position*, we mean the location of the variables of the pivot block associated

⁵In the symmetric case, Wb and $Wsol$ could be the same workspace, however, in the unsymmetric case, because of off-diagonal pivoting issues possibly leading to unsymmetric lists of indices inside each front, it is not immediate to have a single workspace for both Wb and $Wsol$.

with the node. The size of `POSinWRHS` is the number of nodes in the tree: we do not need an indirection for each single variable. Instead, for a node \mathcal{N} , `POSinWRHS(\mathcal{N})` gives the position of the first variable in the pivot block of node \mathcal{N} , and we know that the other ones are stored contiguously. Notice that in case of multiple right-hand-sides, processed by blocks, `POSinWRHS` needs to be computed only once for all blocks.

- The workarray Wb is still there to hold the intermediate contributions. The same workarray Wb can be reused for each block, in case of multiple right-hand sides.
- At line 19, instead of storing the result in $Wtmp1$, the result of $L_{11}^{-1} \times Wtmp1$ is stored in the appropriate contiguous locations of `WRHS`, starting at position `POSinWRHS(\mathcal{N})`.
- On exit from the forward stage, Wb is freed.

Concerning the backward substitution, Wb is not used anymore. Instead, the intermediate solution y is obtained directly from `WRHS` at each node. This involves the following modifications:

- At lines 18 and 37, y_1 is obtained using the entries (or rows) of the right-hand side available in `WRHS` (instead of Wb), starting at position `POSinWRHS(\mathcal{N})`. Remember that the entries (or rows, in case of multiple right-hand side) corresponding to the pivot block variables are contiguous in `WRHS`.
- Line 35 becomes “Add it into `WRHS`, starting at position `POSinWRHS(\mathcal{N})`”.

Remark that the above modifications (introduced in MUMPS 4.7) are useful not only for reduced right-hand functionality, but also for the general solve algorithm: the backward solve no more requires Wb and only one workspace of size $n \times 32$ is needed in each phase (Wb for the forward elimination, $Wsol$ for the backward substitution), while the average size of the new workspace `WRHS` scales perfectly with the number of processors. Furthermore, a good locality is ensured in `WRHS` because fully-summed variables are contiguous. We will see in Section 6.4 other optimizations of the solve phase to allow processing larger problems, suppressing completely the use of the workarrays Wb and $Wsol$, which do not scale with the number of processors and limit locality aspects.

2.6 Determinant

Some applications require computing the determinant of a sparse matrix. In electrostatics for instance, the determinant is related to electric potentials or electric charges. Such a feature has been requested by several users of the MUMPS package and an implementation was done in collaboration with one of them, A. Salzmann, who actually suggested a partial patch to version 4.8.3 of our package. Given an LU (or LDL^T) factorization, the determinant is simply the product of the diagonal elements of D (or U). In theory, given a code that is able to factorize a sparse matrix, its computation should therefore be trivial. In this section, we show that a careful implementation is however needed, especially when considering parallel distributed environments.

2.6.1 Numerical aspects: avoiding underflows and overflows

When multiplying a large amount of diagonal floating-point numbers (matrices with several million rows/columns), underflows and overflows may easily occur. Depending on the setting of floating-point exception flags, overflows may become equal to Infinity and underflows, after some loss of accuracy due to subnormal numbers, may become 0. To avoid such situations, either the logarithms of diagonal values should be accumulated, or a mantissa and exponent should be maintained explicitly. Because computing and accumulating logarithms is prone to numerical errors, we prefer the approach consisting in computing a normalized mantissa and keeping track of the exponent. Although a library like LINPACK[70] uses radix (or base) 10 in the computation of determinants, it seems better to use the natural radix (which is 2 for most processors). In Fortran (similar

functions exist in C), the exponent of a floating-point can be directly extracted from its floating-point representation thanks to the intrinsic function *EXPONENT* and the mantissa can be obtained with the intrinsic function *FRACTION*. Those functions are such that $x = SIGN(x) \times FRACTION(x) \times 2^{EXPONENT(x)}$. For example, $FRACTION(2.0) = 0.5$ and $EXPONENT(2.0) = 1$. Constructing the determinant using a normalized product can then be done as shown in Algorithm 2.7.

```

1: m ← 1; e ← 0
2: for all diagonal pivots with floating-point value f do
3:   m ← m × FRACTION(f)
4:   e ← e + EXPONENT(f) + EXPONENT(m)
5:   m ← FRACTION(m)
6: end for

```

Algorithm 2.7: Computing a normalized determinant as a normalized floating-point mantissa m and an integer exponent e by multiplying diagonal pivots f .

2.6.2 Computing the sign of the determinant

For symmetric matrices, because the factorization aims at maintaining the symmetry, only symmetric permutations are applied (both for reducing the fill or for numerical pivoting issues). Given a permutation matrix P , the determinant of $PAP^t = LDL^T$ is the same as that of A , so that the sign of the determinant is maintained. For unsymmetric matrices, two issues must be considered:

- Numerical pivoting. If a pivot is chosen on the diagonal the sign of the determinant is not modified. Otherwise, the sign depends on the total number of row and column exchanges. If this number is odd, the sign is modified. In particular, assuming a right-looking factorization of the frontal matrices, choosing a non-diagonal pivot in the current column or in the current row modifies the sign, choosing it somewhere else (including on the diagonal) keeps the sign unchanged.
- Unsymmetric preprocessings. Given an initial matrix and an unsymmetric permutation Q of the columns (see Section 1.1.8), we work on AQ instead of A . To obtain the determinant of A , the *parity*, or *signature*, or sign, of Q has to be computed. This is something relatively classical that can be done by Algorithm 2.8, which follows the cycles in the permutation to determine the number of corresponding exchanges. In practice initialization of flag arrays (flag arrays are often used in sparse matrix codes) is avoided, see line 1. In fact, each time an algorithm requires a flag array, it is generally easy to reset it to its initial value as done at line 1 of Algorithm 2.8. Furthermore, any integer array of size n whose contents has a special property could be used.

2.6.3 Special cases

In LDL^T factorizations of symmetric indefinite matrices, two-by-two pivots may be necessary (see Section 1.3.2.2). If D contains a two-by-two pivot $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ the determinant $f = ad - bc$ is computed and simply used as one of the diagonal values in Algorithm 2.7.

Another special case is the one of singular matrices. Factorization algorithms can often detect quasi-null pivots (which may for example correspond to the so-called *rigid body modes* in structural mechanics) and isolate them (see Algorithm 2.3). It makes sense to exclude those quasi-null pivots in the computation of the determinant, which would otherwise be equal to 0. The determinant computed is then the one of the rest of the matrix⁶. Finally, if static pivoting is used (see Section 1.3.2.4), static pivots are also excluded.

As said in Section 2.1.1.2, the factorization of the last separator of a sparse matrix, corresponding to the root of its assembly tree may use dense factorization kernels on a matrix built with a 2D block cyclic

⁶Remark that, in case of scaling, the corresponding entries of the scaling arrays must also be excluded from the computation of the determinant.

```

1: Assumption: visited(1 : n) = false
2: k ← 0
3: for i = 1 to n do
4:   if visited(i) then
5:     visited(i) ← false {(reset in case of further use)}
6:   else
7:     j ← σ(i) {Start following a new cycle}
8:     while j ≠ i do
9:       visited(j) ← true
10:      Accumulate number of swaps in cycle:
11:      k ← k + 1 {Follow cycle}
12:      j ← σ(j)
13:    end while
14:    After first cycle, k contains number of swaps in first cycle
15:  end if
16: end for {k now contains number of swaps in whole permutation}
17: if k is odd then
18:   Change the sign of determinant
19: end if

```

Algorithm 2.8: Computation of the parity of a permutation $\sigma(1 : n)$.

distributed format. ScaLAPACK is used for that purpose. In that case, each MPI process finds its diagonal pivots in the factorized dense root node in 2D block cyclic format and multiplies them with the current value of the determinant as in Algorithm 2.7.

Finally, scaling vectors should be taken care of. Given a scaled matrix $D_r A D_c$, the product of the diagonal entries of D_r and D_c is obtained, then we take the inverse of the result by negating the exponent and taking the inverse of the mantissa (rather than multiplying together all the inverses of the scaling values).

2.6.4 Reduction in parallel environments

In parallel, each processor i computes the product of the pivots it owns. We note m_i the resulting mantissa on processor i and e_i the corresponding exponent, using the default radix of the machine (usually 2). A reduction operation must then be performed to obtain the final determinant. Assuming we use MPI, reduction operators `MPI_PROD` and `MPI_SUM` can be used to perform the reduction (with `MPI_REDUCE` and obtain the determinant $(m, e) = (\prod_i m_i, \sum_i e_i)$ which can then be normalized thanks to the `FRACTION` and `EXPONENT` intrinsic functions. Let us now estimate the risk of overflow/underflow when computing $\prod_i m_i$.

Except for subnormal numbers, the mantissa in radix 2, as returned by the intrinsic function `FRACTION`, is in the range $[0.100\dots00, 0.11\dots11]$, where the first 1 after the dot is implicit in the floating-point representation. This corresponds to the range $[0.5, 0.99\dots99]$ in radix 10. Assuming a uniform distribution of the mantissas m_i , the average mantissa is 0.75 (in an extreme case, all mantissas might be equal to 0.5). Noting $fmin$ the smallest non-subnormal number of the floating-point arithmetic used, and p the number of processors, an underflow occurs if $0.75^p < fmin$, or $p > \frac{\log(fmin)}{\log(0.75)}$. In single precision arithmetic, this gives an underflow when p exceeds 303 processors (126 if all mantissas m_i were equal to 0.5). In double precision arithmetic, an underflow occurs for $p = 2463$ processors (1022 processors if all $m_i = 0.5$).

Given the number of processors/cores of modern high performance computers, there is thus a real risk of underflow, which should be taken care of. This can be done using an MPI operator to normalize the product and keep the exponent separately. This operator is defined by Algorithm 2.9, where the MPI implementation is allowed to call this operator on an arbitrary number nel of elements. Note that the implementation also requires MPI derived datatypes in order to define a new type consisting of a mantissa and an exponent.

```

1: for all  $i = 1$  to  $nel$  do
2:   Compute normalized product  $m'_i, e'_i \leftarrow (m'_i, e'_i) \times (m_i, e_i)$ 
3:    $m'_i \leftarrow m'_i \times m_i$ 
4:    $e'_i \leftarrow e'_i + e_i + EXPONENT(m'_i)$ 
5:    $m'_i \leftarrow FRACTION(m'_i)$ 
6: end for

```

Algorithm 2.9: MPI operator used for the reduction. Input: $(m_i, e_i)_{i=1:nel}$, Input and output: $(m'_i, e'_i)_{i=1:nel}$.

2.6.5 Testing

Validation was done by inserting some new tests in non-regression tests executed nightly on different machines. For each tested matrix, a reference determinant is computed, then the determinant is computed with various options of the solver (scaling on or off, different orderings, preprocessings, weighted matching algorithms) and compared to the reference determinant.

In single precision arithmetic, some relative differences higher than expected (10^{-3}) were observed on harder problems. However, by comparing with double precision calculations, it appeared that the numerical error was due to the accuracy of pivots rather than to the accuracy of computing the product of the diagonal values. If the accuracy of computing the product of pivots becomes an issue, one could use the compensated product techniques, see for example the discussions on this in [99]; a compensated product of n floating-point numbers can be done in $19n - 18$ operations, or $3n - 2$ if a fused-multiply-and-add (FMA) operator is available. Techniques by Kahan also exist to compute the determinants of 2×2 pivots more accurately in case there are risks of cancellation (see analysis by [119]), but we expect this not to be the priority knowing that such pivots are, by choice, far from singularity.

2.6.6 Complex arithmetic

In complex arithmetic, rather than maintaining an exponent for the real part and another one for the imaginary part of the determinant, we use a single exponent. This allows to keep performing operations on complex numbers without reimplementing them. When multiplying the current determinant by a new complex number we simply replace the normalization of Algorithm 2.7 by the one of Algorithm 2.10. Although this approach is debatable and may have some limits if the real and imaginary parts have very different magnitudes, the strategy here is typical in our work where there is always so much more to do than doable: wait for applications or users to show such limits before going for something more sophisticated. The issues of the previous section (reduction, two-by-two pivots, ...) are similar with complex arithmetic; because scaling arrays are real even in complex arithmetic, the multiplication of scaling entries are done in real arithmetic, still using Algorithm 2.7.

```

1:  $r \leftarrow 1; c \leftarrow 0; e \leftarrow 0$ 
2: for all diagonal pivots with complex value  $f$  do
3:    $(r,c) \leftarrow (r,c) \times f$ 
4:    $e_{loc} \leftarrow EXPONENT(|r| + |c|)$ 
5:    $r \leftarrow r \times 2^{-e_{loc}}$ 
6:    $c \leftarrow c \times 2^{-e_{loc}}$ 
7:    $e \leftarrow e + e_{loc}$ 
8: end for

```

Algorithm 2.10: Computation of a normalized complex determinant (r, c, e) as a floating-point real part r , a floating-point imaginary part c and an integer exponent e by multiplying diagonal complex pivots f . The operations at lines 5 and 6 do not use floating-point operations; instead, the exponent is just overwritten by a new one (in Fortran, using the *SCALE* intrinsic function).

2.6.7 Memory aspects

In several physics and electrostatics applications, only the determinant of a matrix A is needed, because it has an intrinsic physical meaning, and solving systems of equations of the form $Ax = b$ is not required. In that case, it can be interesting to discard the computed factors right after computing them, significantly reducing the storage requirements and data movements. In the multifrontal method, only the active storage (see end of Section 1.3.3) remains and at each node of the tree, the determinant is updated and factors are discarded. In that case, it may be interesting to use orderings and postorders (see Section 1.1.7) that minimize the active storage rather than the size of the factors or even the number of floating-point operations. While we illustrated the impact of the ordering on the size of the factors in Table 1.1, the impact on the active storage is given in Table 2.2. We observe that AMD, PORD and AMF are much more competitive at reducing the number of operations than they were at reducing the factor size of floating-point count. Remark that the memory behaviour when computing only the determinant is indeed identical to the out-of-core situation when the factors are written to disk: in that case, active storage should be minimized. In the next chapter (Chapter 3), we will see how an adequate choice of the tree traversal can minimize different metrics, in particular the active storage.

	METIS	SCOTCH	PORD	AMF	AMD
GUPTA2	58.33	<i>289.67</i>	78.13	33.61	52.09
SHIP_003	<i>25.09</i>	23.06	20.86	20.77	32.02
TWOTONE	13.24	13.54	11.80	11.63	<i>17.59</i>
WANG3	3.28	3.84	2.75	3.62	<i>6.14</i>
XENON2	14.89	15.21	13.14	23.82	<i>37.82</i>

Table 2.2: Peak of active memory for the multifrontal approach ($\times 10^6$ entries), as a function of the ordering heuristic applied.

2.7 Concluding remarks

In this chapter, we have seen that functionalities and algorithms that appear to be relatively simple in a serial environment may become much more complex in a parallel distributed environment. This is the case of pivoting issues, general management of parallelism with dynamic data structures and asynchronous communications, as well as mapping and dynamic scheduling issues. As another example, we showed in Section 2.6 that even a functionality apparently simple like the computation of the determinant of a sparse matrix (product of the diagonal elements of the factorized matrix), is not so immediate to implement and requires some care.

In the next chapter (Chapter 3), we come back to sequential aspects of multifrontal methods and show how different schedules for the tasks of the assembly tree can reduce metrics like the memory usage or the I/O traffic. We will come back to scheduling aspects of multifrontal methods in parallel environments in Chapter 4. Performance on shared-memory or multicore environments will be discussed in Section 6.6.

Chapter 3

Task Scheduling for the Serial Multifrontal Method

The objective of this chapter is to discuss the impact of tree traversals and multifrontal models on memory usage and I/O volumes. More precisely, the chapter aims at presenting in a single document and with coherent notations all the multifrontal variants discussed in [8, 9, 11, 13, 106, 105], corresponding to some work accomplished in the context of the PhD thesis [5] and [102]. This chapter can be skipped by the reader mainly interested in getting a general view of the type of work done, to whom we advise to read references [106, 13, 11] (in that order) instead. Because of the high number of variants and combinations, there is a deep level of sections/subsections in this chapter, which we summarize below:

Contents

2.1	Adapting the multifrontal factorization to a parallel distributed environment	33
2.1.1	Sources of parallelism	33
2.1.1.1	Description of type 2 parallelism	34
2.1.1.2	Description of type 3 parallelism	35
2.1.2	Asynchronous communication issues	36
2.1.3	Assembly process	37
2.1.4	Factorization of type 1 nodes	39
2.1.5	Parallel factorization of type 2 nodes	40
2.1.6	Discussion	43
2.2	Pivot selection	44
2.2.1	Pivoting and stability issues in a parallel asynchronous context	44
2.2.2	Detection of null pivots and rank-revealing	46
2.2.3	Pivoting and out-of-core	46
2.2.4	Algorithm for pivot selection (symmetric indefinite case)	46
2.3	Schur complement	47
2.4	Solution phase	49
2.5	Reduced/condensed right-hand side in solve phase	53
2.6	Determinant	55
2.6.1	Numerical aspects: avoiding underflows and overflows	55
2.6.2	Computing the sign of the determinant	56
2.6.3	Special cases	56
2.6.4	Reduction in parallel environments	57
2.6.5	Testing	58
2.6.6	Complex arithmetic	58

3.1 Introduction – Tree traversals and postorders

As explained in Section 1.3.3 (please refer to that section for further details) of Chapter 1, the multifrontal tree should be processed using a topological ordering, that is, an ordering such that children nodes are processed before their parents. We also recall that a postorder is a particular topological order where the nodes in any subtree are numbered consecutively: each time the last sibling of a family is processed, the parent node is activated, consuming the contribution blocks available. In multifrontal methods, the use of postorders allows the storage of the contribution blocks produced at each step of the multifrontal method to be accessed using a stack mechanism, significantly simplifying the memory management. However, one should note that it is possible to build cases where the topological order that best minimizes memory usage is not a postorder. For that, let us start this chapter with a simple example. We assume that factors can be stored to disk as soon as they have been computed, and we focus on the working storage of the multifrontal method, that is, the storage for the contribution blocks and for the current frontal matrix. We consider the example of Figure 3.1, where the frontal matrices associated with nodes a and b require a storage $m_a = m_b = 1000$ (MB, say), produce contribution blocks requiring a storage $cb_a = cb_b = 10$ MB consumed by their respective parents c and d , which in turn have a frontal matrix of size $m_c = m_d = 100$, producing contribution blocks of size $cb_c = cb_d = 90$ for the root node e , whose frontal matrix is of size $m_e = 100$.

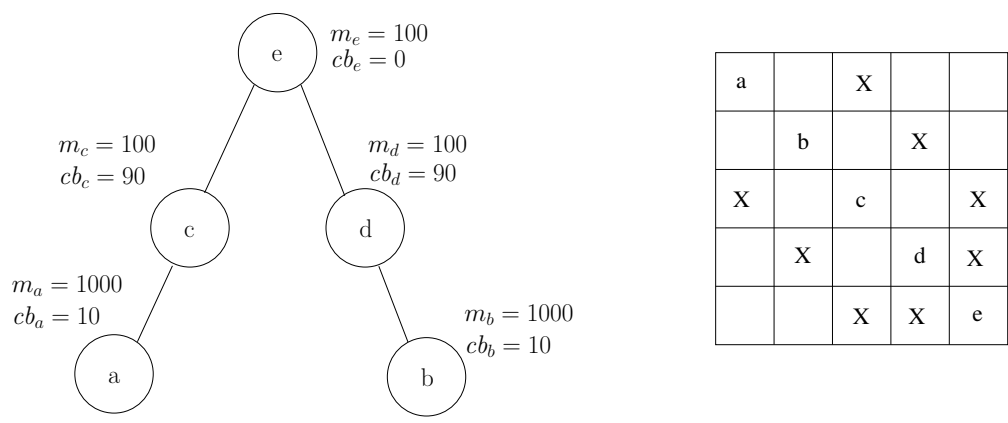


Figure 3.1: Example tree with 5 nodes a, b, c, d, e , and possible associated matrix. m_i represents the storage associated with the frontal matrix of node i and cb_i the storage for the contribution block produced by node i , which will be consumed by the parent of node i . A,B,C,D,E are square matrices corresponding to the range of variables that must be eliminated at nodes a, b, c, d, e , respectively.

The possible topological orders are $a - b - c - d - e$, $a - b - d - c - e$, $a - c - b - d - e$, $b - a - c - d - e$, $b - a - d - c - e$, $b - d - c - a - e$, two of which are postorders: $a - c - b - d - e$ and $b - d - a - c - e$. Given the symmetry of the tree, the two postorders result in the same amount of core memory requirement. Consider the postorder $a - c - b - d - e$ and the successive steps of the multifrontal method. Remember that factors are stored on disk right after they are computed. For each step, we give in parenthesis the core storage requirement and the contents of the memory.

1. a is allocated ($m_a = 1000$) and factored,
2. factors of a are moved to secondary storage – disk ($cb_a = 10$),
3. c is allocated ($cb_a + m_c = 110$),

4. c consumes the contribution block of a ($m_c = 100$) and is factored,
5. factors of c are discarded ($cb_c = 100$),
6. b is allocated ($m_b + cb_a + cb_c = 1100$) and factored,
7. ...

We see that at this stage, the best postorder (the other postorder $b - d - a - c - e$ is equivalent to the one above) has a peak of working storage at least equal to 1100. Let us now consider the topological order defined by the sequence $a - b - c - d - e$, which is not a postorder. The working storage takes the successive values: $m_a = 1000$, $cb_a = 10$, $cb_a + m_b = 1010$, $cb_a + cb_b = 20$, $cb_a + cb_b + m_c = 120$, $cb_b + m_c = 110$, $cb_b + cb_c = 100$, $cb_b + cb_c + m_d = 200$, $cb_c + m_d = 190$, $cb_c + cb_d = 180$, $cb_c + cb_d + m_e = 280$, $m_e = 100$, $cb_e = 0$. The storage requirement associated with $a - b - c - d - e$ is only equal to 1010.

Thus, forcing the use of a postorder instead of a general topological order may lead to a larger memory usage. This was already observed by Liu [127], who proposed an algorithm to find a memory-minimizing topological ordering. In [118], the authors show that it is possible to build trees for which postorders are arbitrarily bad in terms of working storage compared to the best postorder. In codes that rely on dynamic allocation and do not use in-place assemblies, using general topological order would make sense and an algorithm to find an optimal topological order faster than the one from [127] has recently been proposed by Jacquelin et al. [118]. In the rest of this chapter, we still restrict our study to the case of postorders, for the following reasons:

- (i) postorders allow for a more friendly memory management (stack mechanism with a good locality of reference);
- (ii) gains from using more general topological orderings do not seem that big on many practical problems [127];
- (iii) postorders allow for in-place assemblies (see below), which lead to significant gains ($\approx 30\%$) in terms of working storage; it is not clear whether/how general topological orders would allow this type of assembly.

Even in the case of postorders, minimizing the working storage has been studied by Liu [126]. In this chapter, we generalize this work and also study the case of minimizing the I/O volume, in case not only the factors but also the working storage must go to disk. The chapter is organized as follows. First, we introduce some variants of the multifrontal method depending on (i) the way, and (ii) the moment when, contribution blocks are assembled into the frontal matrix of the parent node (Section 3.2). Although this still represents a limited set of variants, ideas of this chapter are more general and could be applied or adapted to different variants of memory management. We focus on the minimization of the working storage requirement in Section 3.3 (either when factors are kept in memory, or when they are stored on disk) before considering the minimization of the I/O traffic in an out-of-core context where contribution blocks are also written to disk (Section 3.4). In doing so, we consider a significant combination of situations and models. Finally we provide some algorithms that demonstrate that an efficient memory management can be obtained for those different models in Section 3.5, and give some concluding remarks in Section 3.6.

3.2 Models of assembly in the multifrontal method

We introduce several variants of the multifrontal method, which correspond to different existing and/or possible implementations of the consumption of contribution blocks in the method. A first set of variants comes from the possible overlap between the memory for the frontal matrix of the parent with the memory for the contribution block that is first assembled into it. Here are the possible corresponding schemes:

- The *classical* assembly scheme: the memory for the frontal matrix cannot overlap with the one of the stack of contribution blocks at a given instant. This is illustrated in Figure 3.2(b): the frontal matrix f is allocated in a memory space different from its children. All elements of f are first initialized to 0, then the contributions e , d , c , b , a , possibly also some entries of the original matrix, are assembled one by one in the memory reserved for f . This scheme is implemented for instance in the MA41 solver [37, 116].
- The *in-place* or *last-in-place* assembly scheme: the memory for the frontal matrix at the parent node is allowed to overlap with the contribution block at the top of the stack, as illustrated in Figure 3.2(c). Thanks to the postorder property, this contribution is the last one that was computed and is the first one assembled into the parent. It is assembled “in-place”, in the sense that it is expanded in-place to form the frontal matrix. This can be done if the order of the variables in the child and in the parent are compatible:
 - First, the entries in f that are not in e must be set to 0.
 - Second, the entries in e are moved to their final position in f , one-by-one. Each time an entry in e is moved to its position in f , if the original entry was in both e and f , it is reset to 0. This must be done starting with the top-left corner of e , row-by-row (assuming a row-major storage) and it is for this step that the order of the variables in the contribution block of e and in f have to be compatible, which requires some care when part of the variables from f are computed dynamically (delayed pivots).
 - Third, entries from the contribution blocks of other children are assembled using classical *extend-add* operations. Entries from the initial matrix corresponding to fully summed rows or columns in the front of the parent are also assembled.

To summarize, we save space by not consuming both the memory of the contribution block of that child and the memory of the frontal matrix of the parent. Instead, only the maximum between those two quantities is required. This scheme is available in a code like MA27 [79] and in MUMPS, in the case of serial executions and in parts of the tree that are processed by a single processor.

In practice, if an overlap is detected between the memory of the parent and the memory of the child assembled in-place, the assembly will be done in-place. In that case, Algorithm 2.1 must be slightly modified because a global compatible ordering between parent and child variables has to be strictly respected. Because of that, delayed variables from a child assembled *in-place* must appear at the beginning of the index list of the parent (they were expected to be eliminated first in the global ordering).

- The *max-in-place* assembly scheme is a natural extension of the *last-in-place* assembly scheme. It was suggested by [13]. In this approach, we overlap the memory for the frontal matrix of the parent with the memory of the child producing the largest contribution block, even if that child is not processed last. This new variant of *in-place* assembly requires a slightly different memory management which ensures that some space for the parent is available contiguous to the memory of the largest child. We will discuss a possible memory management algorithm corresponding to this scheme in Section 3.5.

Another possible variation (that combines to the previous one) is related to the moment when the frontal matrix of the parent is allocated. We will discuss three such cases in this thesis:

- The *terminal* allocation scheme: the memory for the frontal matrix is allocated after all the children have been processed.
- The *early* allocation scheme: the memory for the frontal matrix of the parent is allocated right after the first child subtree has been processed.
- The *flexible* allocation scheme: the memory of the frontal matrix can be allocated earlier in order to assemble (and thus consume) the contribution blocks of some children on the fly. By adequately choosing the tree traversal and the position at which the frontal matrix of the parent is allocated, one can significantly reduce the memory requirements, as was proposed by [106].

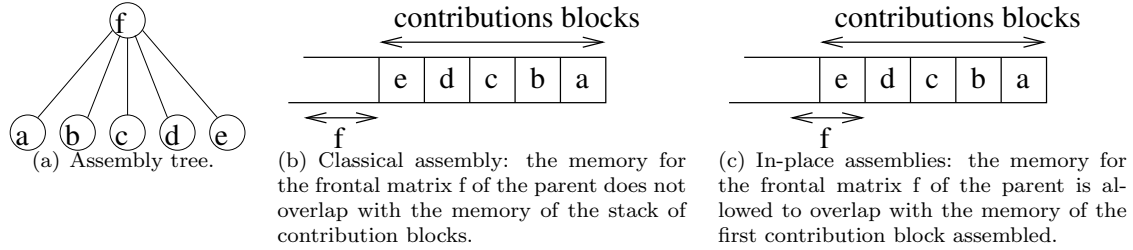


Figure 3.2: An assembly tree and the corresponding memory state at the moment of the allocation of the frontal matrix of its root node depending on the assembly scheme.

3.3 Postorders to reduce the storage requirements

3.3.1 Notations

We now introduce some notations that will be used in the rest of this chapter. We consider a generic parent node and its n ordered children numbered $j = 1, \dots, n$ and we note:

- cb / cb_j , the storage for the contribution block of the parent node / of child j (remark that $cb = 0$ for the root of the tree);
- m / m_j , the storage of the frontal matrix associated with the parent node / with its j^{th} child (remark that $m \geq cb_j$, $m_j \geq cb_j$, and that $m_j - cb_j$ typically represents the size of the factors produced by child j);
- S / S_j , the working storage (or active storage) required to process the subtree rooted at the parent / at child j , when factors are written to disk as soon as they are computed;
- T / T_j , the total storage (including in-core factors) to process the subtree rooted at the parent / at child j .
- F / F_j , the storage for all the factor matrices inside the subtree rooted at the parent / at child j .

Any convenient unit can be used for the above quantities, such as bytes, GB (gigabytes), or number of scalar entries. Furthermore, we note that every tree whose corresponding nodes respect the constraints above can be associated with a matrix: one can build the structure of a frontal matrix associated with each node, and from the structure of each frontal matrix, it is easy to find a corresponding original sparse matrix.

In the case of flexible allocation (see last bullet of Section 3.2), we note p the position of the child after which the frontal matrix of the parent is allocated and the contribution blocks of the already processed children are assembled. With respect to all the variants of Section 3.2, we will use the following superscripts for the above quantities:

term: terminal allocation of the parent, no in-place assembly.

term-ip: terminal allocation of the parent, in-place assembly of the last child, available on the top of the stack.

term-maxip: terminal allocation of the parent, in-place assembly of the child with the largest contribution block, assuming that some contiguous memory is available for the parent next to that contribution block.

flex: flexible parent allocation, no in-place assembly.

flex-ip: flexible parent allocation, in-place assembly of child p .

flex-maxip: flexible parent allocation, in-place assembly of the child with largest contribution block.

early: early parent allocation, first child assembled without an in-place scheme.

early-ip: early parent allocation, in-place assembly of first child.

(*early-maxip* is not applicable: since there is only one child before the parent allocation, this scheme is just a particular case of – and can be no better than – *early-ip*.)

3.3.2 Terminal allocation

In this section, we focus on the multifrontal method with a *terminal* allocation of each frontal matrix: the memory for a frontal matrix is only reserved after all child subtrees have been processed. This is the situation that we have assumed in the previous chapters (for example, in the discussion on memory usage from Section 1.3.3); furthermore, we have considered a *classical* assembly scheme (as opposed to an *in-place* assembly scheme).

3.3.2.1 Working storage requirement

The working storage covers the storage for the factors and for the stack of contribution blocks, excluding the storage for already computed factors. When processing a child j , the contribution blocks of all previously processed children have to be stored. Their memory size sums up with the storage requirements S_j of the considered child, leading to a global storage equal to $S_j + \sum_{k=1}^{j-1} cb_k$. After all the children have been processed, the frontal matrix (of size m) of the parent is allocated, requiring a storage equal to $m + \sum_{k=1}^n cb_k$. Therefore, the storage required to process the complete subtree rooted at the parent node is given by the maximum of all these values, that is:

$$S^{term} = \max \left(\max_{j=1,n} (S_j^{term} + \sum_{k=1}^{j-1} cb_k), m + \sum_{k=1}^n cb_k \right) \quad (3.1)$$

Knowing that the storage requirement S for a leaf node is equal to the size of its frontal matrix m , applying this formula recursively (as done in [126]), allows to determine the storage requirement for the complete tree, in a bottom-up process.

In case of *last-in-place* assembly, the contribution block of child n overlaps with that of the frontal matrix of the parent, so that the cb_n term can be suppressed from the right part of Formula (3.1). We obtain:

$$S^{term-ip} = \max \left(\max_{j=1,n} (S_j^{term-ip} + \sum_{k=1}^{j-1} cb_k), m + \sum_{k=1}^{n-1} cb_k \right), \quad (3.2)$$

which corresponds to the situation considered in [126].

Finally, in case we are able to build the frontal matrix of the parent at a memory location which overlaps with the memory of the largest contribution block (corresponding to a child we note k_{max}), we now obtain the working storage in the *max-in-place* assembly scheme:

$$S^{term-maxip} = \max \left(\max_{j=1,n} (S_j^{term-maxip} + \sum_{k=1}^{j-1} cb_k), m + \sum_{k=1, k \neq k_{max}}^n cb_k \right) \quad (3.3)$$

Compared to Equation (3.1) corresponding to the *classical* scheme, $cb_{k_{max}}$ must simply be subtracted from the term $m + \sum_j cb_j$.

3.3.2.2 Total storage requirement (including factors)

Here we consider that the factors are kept in memory. As introduced in Section 3.3.1, F/F_i represents the sum of the memory requirements for all factor matrices in the subtree rooted at a parent node $/$ in the subtree rooted at child i . This corresponds to the sum of the factors produced at each frontal matrix in the considered subtree. Starting from Formula (3.1) and taking into account the fact that factors must be kept in memory, we obtain the total storage requirement at a parent subtree:

$$T^{term} = \max \left(\max_{j=1,n} (T_j^{term} + \sum_{k=1}^{j-1} (cb_k + F_k)), m + \sum_{k=1}^n (cb_k + F_k) \right). \quad (3.4)$$

The modified formulas for the *last-in-place* and *max-in-place* schemes are, respectively:

$$T^{term-ip} = \max \left(\max_{j=1,n} (T_j^{term-ip} + \sum_{k=1}^{j-1} (cb_k + F_k)), m + \sum_{k=1}^{n-1} cb_k + \sum_{k=1}^n F_k \right), \quad (3.5)$$

and

$$T^{term-maxip} = \max \left(\max_{j=1,n} (T_j^{term-maxip} + \sum_{k=1}^{j-1} (cb_k + F_k)), m + \sum_{k=1, k \neq k_{max}}^n cb_k + \sum_{k=1}^n F_k \right). \quad (3.6)$$

All these formulas (including F/F_k) can be evaluated using a natural bottom-up traversal of the tree.

3.3.2.3 Liu's theorem and its application to reduce storage requirements

Depending on the assembly scheme and objective, we wish to find, at each level of the tree, a permutation of the children which minimizes one of the above formulas. The total storage requirement T should be minimized when everything is in-core, whereas the working storage S should be minimized when the factors are stored to disk after they are computed.

We now state a fundamental theorem that will use many times in this chapter.

Theorem 3.1. (Liu [126, Theorem 3.2]) *Given a set of values $(x_i, y_i)_{i=1, \dots, n}$, the minimal value of $\max_{i=1, \dots, n} (x_i + \sum_{j=1}^{i-1} y_j)$ is obtained by sorting the sequence (x_i, y_i) in decreasing order of $x_i - y_i$, that is, $x_1 - y_1 \geq x_2 - y_2 \geq \dots \geq x_n - y_n$.*

Thanks to Theorem 3.1, one can minimize either the total storage requirement T or the working storage requirement S . Suppressing the constant terms, x_i and y_i correspond to different quantities depending on the formulas, as indicated in Table 3.1.

Quantity to minimize	Assembly scheme	Reference formula	x_i	y_i
Working storage (without factors)	<i>classical</i>	3.1	S_i	cb_i
	<i>last-in-place</i>	3.2	$\max(S_i, m)$	cb_i
	<i>max-in-place</i>	3.3	S_i	cb_i
Total storage (with factors)	<i>classical</i>	3.4	T_i	$cb_i + F_i$
	<i>last-in-place</i>	3.5	$\max(T_i, m + F_i)$	$cb_i + F_i$

Table 3.1: Optimal order of children for the *terminal* parent allocation, for the working storage (factors written on disk) and for the total storage (when both the factors and active storage remain in core memory). For each family of the tree, following a bottom-up process, child subtrees must be ordered in decreasing order of $x_i - y_i$.

In order to better see x_i and y_i , Formula (3.2) is rewritten as:

$$S^{term-ip} = \max_{j=1,n} \left(\max(S_j^{term-ip}, m) + \sum_{k=1}^{j-1} cb_k \right),$$

and Formula (3.5) as:

$$T^{term-ip} = \max_{j=1,n} \left(\max(T_j^{term-ip}, m + F_j) + \sum_{k=1}^{j-1} (cb_k + F_k) \right).$$

The total memory with a *max-in-place* assembly scheme does not make so much sense from an implementation point of view because it is difficult to imagine a memory management scheme that applies to that case, as will be seen in Section 3.5. It was therefore excluded from the table.

3.3.3 Early parent allocation

Because on wide trees, the quantity of contribution blocks to store can be large, and can be larger than the frontal matrix of the parent, Liu experimented in [126] a strategy consisting in preallocating the frontal matrix of the parent node before children are processed and before associated contributions are formed. Each contribution block from each new child is then assembled directly into the structure of the parent, thus avoiding a possibly large collection of contribution blocks in stack memory. The storage requirement with this approach is simply:

$$S = m + \max_{j=1,n} (S_j).$$

Liu noticed that rather than pre-allocating the parent, the above scheme can be slightly improved by allocating the parent right after the first child has been processed. This allows one to process a large first subtree which would not fit in memory together with the frontal matrix of the parent. Using the notations above, we obtain a peak of active storage equal to:

$$S^{early} = \max(S_1^{early}, m + cb_1, m + \max_{j=2,n} (S_j^{early})) \quad (3.7)$$

if the assembly of the first child into the parent is not in-place (aka classical assembly), and

$$S^{early-ip} = \max(S_1^{early-ip}, m + \max_{j=2,n} (S_j^{early-ip})) \quad (3.8)$$

if the assembly of the contribution of the first child into the parent is done in-place. Both Equations (3.7) and (3.8) are minimized by processing the child with largest S_i first (at each level of the tree in a bottom-up process). Unfortunately, a chain of parent nodes must be kept in memory, possibly also leading to a large memory requirement. Therefore, Liu found this approach somewhat disappointing, even if applied only partially, to levels of the tree where it is beneficial.

Concerning the total memory (including in-core factors), we similarly obtain:

$$T^{early} = \max(T_1^{early}, F_1 + cb_1 + m, m + \max_{j=2,n} (T_j^{early} + \sum_{k=1}^{j-1} F_k)) \quad (3.9)$$

for the classical assembly, and

$$T^{early-ip} = \max(T_1^{early-ip}, F_1 + m, m + \max_{j=2,n} (T_j^{early-ip} + \sum_{k=1}^{j-1} F_k)) \quad (3.10)$$

if the assembly of the contribution from the first child into the parent can be done in-place¹. In that case, it could for example make sense to try to order the children in decreasing order of their $T_j - F_j$ as this will at

¹Remark that the max-in-place allocation would be the same as the normal *in-place* case because only one child has been processed at the moment of allocating the parent.

least minimize the term $\max_j(T_j + \sum_{k=1}^{j-1} F_k)$. It could also make sense to process the child with the largest contribution first in the *in-place* case. However, because this early parent allocation is a special case of the more general flexible allocation described in the next subsection, we delay the obtainment of an optimal to the next section.

3.3.4 Flexible allocation

As explained in the previous section, neither the systematic pre-allocation of the parent node, nor its allocation right after the last child (terminal allocation), provide an optimal storage requirement. However, it is possible to allocate the parent node at an arbitrary position, somewhere in-between those two extremes. Suppose that the frontal matrix of the parent node is allocated in memory right after the p^{th} child has been treated. We define $\mathcal{S}_1 = \{1, \dots, p\}$ as the set of children processed before the allocation of the parent and

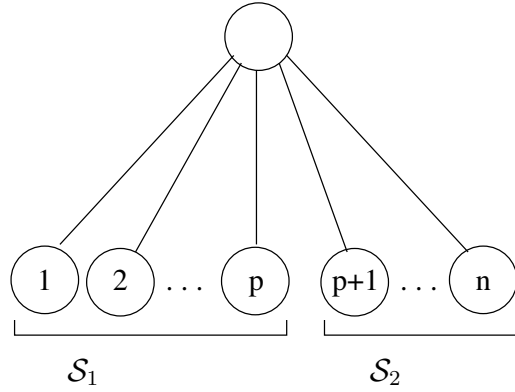


Figure 3.3: Example of a parent node and its children.

$\mathcal{S}_2 = \{p+1, \dots, n\}$ as the set of children processed after the allocation (see Figure 3.3).

In the presentation of this flexible allocation, we first consider that a “classical” assembly scheme is used in Section 3.3.4.1 (in other words cb_p and m do not overlap) and we then consider the “last-in-place” assembly scheme in Section 3.3.4.2 (where cb_p and m overlap). In both cases we study both the working and total storage requirements. We finally consider the “max-in-place” assembly scheme in Section 3.3.4.3.

3.3.4.1 Classical (non in-place) assembly scheme

3.3.4.1.1 Working storage minimization The working storage (excluding factors) needed to process a subtree rooted at a child j is denoted by S_j^{flex} . Considering only the children nodes in \mathcal{S}_1 , the peak of storage is obtained similarly to the case of the *terminal* allocation, (see Formula (3.1) in Section 3.3.2.1), and is equal to $\max\left(\max_{j=1,p}(S_j^{\text{flex}} + \sum_{k=1}^{j-1} cb_k), m + \sum_{k=1}^p cb_k\right)$. Then, the amount of memory needed to process each child j in \mathcal{S}_2 (after the parent is allocated) is $m + S_j^{\text{flex}}$. Thus, the storage requirement, S^{flex} , to process the subtree rooted at the parent is:

$$S^{\text{flex}} = \max\left(\max_{j=1,p}\left(S_j^{\text{flex}} + \sum_{k=1}^{j-1} cb_k\right), m + \sum_{k=1}^p cb_k, m + \max_{j=p+1,n} S_j^{\text{flex}}\right). \quad (3.11)$$

We now look for an optimal schedule (that is, an order of the children and a position p to allocate the parent) that minimizes (3.11).

Lemma 3.1. *Let j be a child node belonging to \mathcal{S}_1 . If S_j^{flex} is smaller than $\max_{i \in \mathcal{S}_2}(S_i^{\text{flex}})$, then j can be moved to \mathcal{S}_2 without increasing the peak.*

Proof. Removing a node from \mathcal{S}_1 does not increase the peak in this set. Furthermore since S_j^{flex} is smaller than $\max_{i \in \mathcal{S}_2}(S_i^{flex})$, the peak for \mathcal{S}_2 , $m + \max_{i \in \mathcal{S}_2}(S_i^{flex})$, will not increase. (Note that the order of the children in \mathcal{S}_2 has no impact on the memory peak.) \square

Theorem 3.2. *Considering a parent node and its n children, an optimal peak of working storage and its corresponding schedule are obtained by applying the following algorithm:*

```

Set  $\mathcal{S}_1 = \{1, \dots, n\}$ ,  $\mathcal{S}_2 = \emptyset$  and  $p = n$ ;
Find the schedule providing an optimal  $S^{flex}$  value for partition  $(\mathcal{S}_1, \mathcal{S}_2)$ ;
repeat
  Find  $j$  such that  $S_j^{flex} = \min_{k \in \mathcal{S}_1} S_k^{flex}$ ;
  Set  $\mathcal{S}_1 = \mathcal{S}_1 \setminus \{j\}$ ,  $\mathcal{S}_2 = \mathcal{S}_2 \cup \{j\}$ , and  $p = p - 1$ ;
  Find the schedule providing an optimal  $S'^{flex}$  value for partition  $(\mathcal{S}_1, \mathcal{S}_2)$ ;
  if  $S'^{flex} \leq S^{flex}$  then
    Keep the value of  $p$ , and the schedule of children in  $\mathcal{S}_1$  and  $\mathcal{S}_2$  corresponding to  $S'^{flex}$ ;
    Set  $S^{flex} = S'^{flex}$ ;
  end if
until  $p == 1$  or  $S'^{flex} > S^{flex}$ 

```

Proof. Let σ be an optimal schedule of the children (defining the partition $(\mathcal{S}_1, \mathcal{S}_2)$ as well as the order of the nodes in \mathcal{S}_1 and \mathcal{S}_2). If $\exists k \in \mathcal{S}_1$ such that $S_k^{flex} \leq \max_{j \in \mathcal{S}_2}(S_j^{flex})$, the schedule σ' obtained by moving k to \mathcal{S}_2 is still optimal (Lemma 3.1). Thus, there exists an optimal schedule σ'' such that $\min_{k \in \mathcal{S}_1}(S_k^{flex}) > \max_{k \in \mathcal{S}_2}(S_k^{flex})$ obtained by repeating the previous operation.

As a consequence, an optimal schedule can be computed by starting with all nodes in \mathcal{S}_1 and by trying to move the child node j having the smallest S_j^{flex} from \mathcal{S}_1 to \mathcal{S}_2 . The latter operation is repeated until the minimal value of S^{flex} is obtained. For a given schedule, removing a node in \mathcal{S}_1 will not increase the peak of memory in \mathcal{S}_1 (Lemma 3.1). Then, finding the optimal schedule on the new set cannot increase the peak. Thus, the peak corresponding to \mathcal{S}_1 (resp. \mathcal{S}_2) can only decrease (resp. increase) or be stable from one step to the next, and we can stop iterating when $S'^{flex} > S^{flex}$ (the peak is then due to \mathcal{S}_2 and will not decrease again). \square

At each step of the algorithm above, the schedule providing the optimal S^{flex} value for a partition $(\mathcal{S}_1, \mathcal{S}_2)$ is obtained by sorting the nodes in \mathcal{S}_1 in decreasing order of $S_k^{flex} - cb_k$ (see Section 3.3.2.3 and remark that only the first term of Formula (3.11) is impacted by the order of the nodes), while the order in \mathcal{S}_2 has no effect. There is no need to sort the nodes in \mathcal{S}_1 again at each step of the algorithm (when a child is moved from \mathcal{S}_1 to \mathcal{S}_2), because the sequence of nodes resulting from the previous step of the algorithm is still sorted correctly: only the new S^{flex} value has to be computed at each step, using Formula (3.11).

The determination of the best position for the allocation of the parent is thus done in a maximum of n steps. As explained earlier, the schedule on the complete tree is then obtained by applying the algorithm at each level of the tree, recursively.

3.3.4.1.2 Total storage minimization We use the same definition as before for the sets $\mathcal{S}_1, \mathcal{S}_2$ and for the position to allocate the parent, p . The peak of storage for \mathcal{S}_1 , including the allocation of the parent node, is obtained by applying the reasoning that led to Formula (3.4), on one hand, and (3.11), on the other hand:

$$\mathcal{P}_1 = \max \left(\max_{j=1,p} (T_j^{flex} + \sum_{k=1}^{j-1} (cb_k + F_k)), m + \sum_{k=1}^p (cb_k + F_k) \right). \quad (3.12)$$

Furthermore, the amount of memory needed to process the children nodes of \mathcal{S}_2 is:

$$\mathcal{P}_2 = m + \sum_{k=1}^p F_k + \max_{j=p+1, n} (T_j^{flex} + \sum_{k=p+1}^{j-1} F_k). \quad (3.13)$$

Indeed, when treating a node, the memory will contain the factors corresponding to all already processed sibling subtrees. In the formulas above, note also that T_j^{flex} includes F_j so that the factors for the last child are effectively taken into account. Finally, the amount of memory needed to process the subtree rooted at the parent is:

$$T^{flex} = \max(\mathcal{P}_1, \mathcal{P}_2) \quad (3.14)$$

Lemma 3.2. *Suppose that the position p to allocate the parent, the set of children nodes in \mathcal{S}_1 and the set of children nodes in \mathcal{S}_2 are given. Then, sorting the children nodes in \mathcal{S}_1 in decreasing order of $T_j^{flex} - (cb_j + F_j)$ and the children nodes in \mathcal{S}_2 in decreasing order of $T_j^{flex} - F_j$ provides an optimal peak of memory on \mathcal{S}_1 and an optimal peak of memory on \mathcal{S}_2 .*

Proof. For \mathcal{S}_1 , the order is the one of the classical assembly scheme, terminal allocation (see Table 3.1, minimization of Formula (3.4)). For \mathcal{S}_2 , this results from Theorem 3.1 (see Section 3.3.2.3) applied to the right-hand side of Formula (3.13), with $x_k = T_k^{flex}$ and $y_k = F_k$. \square

Lemma 3.3. *Suppose that the max in \mathcal{S}_2 is obtained for child j_0 , $p+1 \leq j_0 \leq n$ and that the children in \mathcal{S}_1 and \mathcal{S}_2 are ordered according to Lemma 3.2. In other words, suppose that $\mathcal{P}_2 = \mathcal{P}_2(j_0)$, where we define*

$$\begin{aligned} \mathcal{P}_2(j_0) &= m + \sum_{k=1}^p F_k + (T_{j_0}^{flex} + \sum_{k=p+1}^{j_0-1} F_k) \\ &= m + \sum_{k=1}^{j_0-1} F_k + T_{j_0}^{flex} \end{aligned} \quad (3.15)$$

Then, any partition $(\mathcal{S}'_1, \mathcal{S}'_2)$ such that $\mathcal{S}_1 \subset \mathcal{S}'_1$ and $j_0 \in \mathcal{S}'_2$ leads to a storage requirement larger or equal to the value $\mathcal{P}_2(j_0)$ above. In other words, it is not possible to decrease the storage by moving elements from \mathcal{S}_2 to \mathcal{S}_2 and keeping j_0 in \mathcal{S}_2 .

Proof. (i) Changing the order of the children cannot improve the peak (Lemma 3.2). (ii) If we move nodes j_1 from \mathcal{S}_2 to \mathcal{S}_1 that are before j_0 (that is, $j_1 < j_0$), the second line in (3.15) will not change. (iii) If we move nodes j_1 from \mathcal{S}_2 to \mathcal{S}_1 that are after j_0 (that is, $j_1 > j_0$), $\mathcal{P}_2(j_0)$ will increase (since F_{j_1} adds up to the sum), and thus the peak on \mathcal{S}_2 . \square

Lemma 3.4. *Given a set \mathcal{S}_1 ordered according to Lemma 3.2, inserting a new element j_0 into \mathcal{S}_1 cannot decrease the peak on \mathcal{S}_1 : if we define \mathcal{P}'_1 to be the peak of total memory including the allocation of the parent using $\mathcal{S}'_1 = \mathcal{S}_1 \cup \{j_0\}$, we have $\mathcal{P}'_1 \geq \mathcal{P}_1$.*

Proof. See Formula (3.12). \square

Theorem 3.3. *Given a parent node and its n children, an optimal peak of total memory T_i^{flex} is obtained by applying the following algorithm:*

Set $\mathcal{S}_1 = \emptyset$, $\mathcal{S}_2 = \{1, \dots, n\}$ and $p = 0$;
Sort \mathcal{S}_2 according to Lemma 3.2;
Compute $T^{flex} = \mathcal{P}_2$ according to Formula (3.13);
repeat
 Find $j_0 \in \mathcal{S}_2$ such that $\mathcal{P}_2 = m + \sum_{k=1}^{j_0} F_k + T_{j_0}^{flex}$ (Formula (3.15));
 Set $\mathcal{S}_1 = \mathcal{S}_1 \cup \{j_0\}$, $\mathcal{S}_2 = \mathcal{S}_2 \setminus \{j_0\}$, and $p = p + 1$;

(Remark: j_0 is inserted at the appropriate position in \mathcal{S}_1 so that the order of Lemma 3.2 is respected.)
 Compute $\mathcal{P}_1, \mathcal{P}_2$, and $T^{flex} = \max(\mathcal{P}_1, \mathcal{P}_2)$;
if $T'^{flex} \leq T^{flex}$ **then**
 Keep the values of p, \mathcal{S}_1 and \mathcal{S}_2 and set $T^{flex} = T'^{flex}$;
end if
until $p = n$ or $\mathcal{P}_1 \geq \mathcal{P}_2$

Proof. Starting from a configuration where $\mathcal{P}_2 > \mathcal{P}_1$, it results from Lemma 3.3 that the only way to possibly decrease the peak is by moving j_0 from \mathcal{S}_2 to \mathcal{S}_1 . Thus, at each iteration either we have obtained the optimal peak T^{flex} , or the solution with the optimal peak is such that j_0 (which was responsible for the peak in \mathcal{S}_2) belongs to \mathcal{S}_1 . Since we start with $\mathcal{S}_1 = \emptyset$, we are sure to reach the optimal configuration after a maximum of n iterations. (At each iteration, the order from Lemma 3.2 is respected by inserting j_0 at the appropriate position in \mathcal{S}_1 .)

For the termination criterion, we know that the optimal peak has been obtained when \mathcal{P}_1 becomes larger or equal than \mathcal{P}_2 , since in that case the memory peak $T^{flex} = \mathcal{P}_1$ will only increase if the algorithm is pursued further (Lemma 3.4). \square

Remark that we use the stopping criterion $\mathcal{P}_1 \geq \mathcal{P}_2$. The condition $T'^{flex} > T^{flex}$ is not sufficient to ensure that the optimal peak has been reached: it may happen that the global peak will increase by moving an element j_0 from \mathcal{S}_2 to \mathcal{S}_1 , and decrease again at a further iteration to reach the optimal. An example producing such a situation is the one below:

$$\begin{cases} m = 100 & n = 3 \\ T_1^{flex} = 160, & F_1 = 100, \\ T_2^{flex} = 140, & F_2 = 120, \\ T_3^{flex} = 10, & F_3 = 5, \\ cb_1 = cb_2 = cb_3 = 5 \end{cases}$$

Initially all three children are in \mathcal{S}_2 , sorted according to Lemma 3.2 and $T^{flex} = 340$ is reached for child 2, that the algorithm tries to move to \mathcal{S}_1 : $\mathcal{S}_1 = \{2\}$ and $\mathcal{S}_2 = \{1, 3\}$, leading to $T'^{flex} = 380 > 340$. Then, moving child 3 from \mathcal{S}_2 to \mathcal{S}_1 leads to a peak of total memory equal to $T^{flex} = 330$, which is the optimal since $\mathcal{P}_1 = \mathcal{P}_2$.

3.3.4.2 In-place assembly scheme

In this section we describe how the flexible allocation adapts to an *in-place* (more precisely *last-in-place*) assembly scheme. We assume that the assembly of the contribution block corresponding to the last child treated before the allocation of the parent, the p^{th} child using the notations introduced before, is done in-place into the frontal matrix of the parent: the memory of the contribution block of the last child overlaps with the memory of the parent.

3.3.4.2.1 Working storage minimization In the flexible scheme, if the memory for the p^{th} child overlaps with the memory for the parent node, the storage required at the parent becomes:

$$S^{flex-ip} = \max\left(\max_{j=1,p}(S_j^{flex-ip} + \sum_{k=1}^{j-1} cb_k), m + \sum_{k=1}^{p-1} cb_k, m + \max_{j=p+1,n}(S_j^{flex-ip})\right) \quad (3.16)$$

Note the difference with Formula (3.11): the memory for cb_p does not appear here since it is now included in the memory for the parent, m . Finding a schedule that minimizes the working storage is equivalent to the problem presented in Section 3.3.4.1.1. The only difference comes from the computation/processing of the optimal order inside the set \mathcal{S}_1 . Indeed, inside this set, we have to use the schedule proposed by Liu [126] that ensures an optimal memory occupation with the assumption that the last child is assembled in-place

into the parent: this is done by sorting the children nodes in descending order of $\max(S_j^{flex-ip}, m) - cb_j$. Thus one can simply use a variant of Theorem 3.2 where children nodes inside \mathcal{S}_1 are sorted in that order.

3.3.4.2 Total storage minimization In the flexible allocation scheme, if the memory of the contribution block of the p^{th} child overlaps with the memory of the parent (in-place assembly), the total memory is $T^{flex-ip} = \max(\mathcal{P}_1, \mathcal{P}_2)$ (as in Formula (3.14)), where \mathcal{P}_2 is defined by Formula (3.13). The difference with the non in-place case comes from the peak in \mathcal{S}_1 , where the contribution block for child p is not taken into account when assembling the parent, leading to:

$$\begin{aligned} \mathcal{P}_1 &= \max\left(\max_{j=1,p}(T_j^{flex-ip} + \sum_{k=1}^{j-1} (cb_k + F_k)), m + \max_{j=1,p}\left(\sum_{k=1}^{j-1} cb_k + \sum_{k=1}^j F_k\right)\right) \\ &= \max_{j=1,p}\left(\sum_{k=1}^{j-1} (cb_k + F_k) + \max(T_j^{flex-ip}, m + F_j)\right) \end{aligned} \quad (3.17)$$

By applying Theorem 3.1 again, the smallest peak in \mathcal{S}_1 is obtained when the children nodes are sorted in decreasing order of $\max(T_j^{flex-ip}, m + F_j) - (cb_j + F_j)$. An optimal schedule for the total memory in the in-place case is then obtained by applying Theorem 3.3 with children nodes in \mathcal{S}_1 in that order instead of the order from Lemma 3.2.

3.3.4.3 Max-in-place assembly scheme

We can also use a *max-in-place* assembly scheme together with a flexible parent allocation (and will discuss a possible memory management in Section 3.5). In that case, noting k_{max} the index of the child with the largest contribution block in \mathcal{S}_1 , and assuming that there is a contiguous space next to that contribution at the moment of allocating the parent, the working storage can be expressed as:

$$S^{flex-maxip} = \max\left(\max_{j=1,p}(S_j^{flex-maxip} + \sum_{k=1}^{j-1} cb_k), m + \sum_{k=1, k \neq k_{max}}^p cb_k, m + \max_{j=p+1,n}(S_j^{flex-ip})\right) \quad (3.18)$$

Similar to the *terminal* allocation scheme, we do not discuss the association of *max-in-place* with total memory minimization.

To conclude this section on flexible allocation, we summarize in Table 3.2 the order in \mathcal{S}_1 and provide the references to the formulas in the text.

Quantity to minimize	Assembly scheme	Reference formula	Order in \mathcal{S}_1		Theorem to obtain assembly position p
			x_i	y_i	
Working storage (without factors)	<i>classical</i>	3.11	S_i	cb_i	3.2
	<i>last-in-place</i>	3.16	$\max(S_i, m)$	cb_i	3.2
	<i>max-in-place</i>	3.18	S_i	cb_i	3.2
Total storage (with factors)	<i>classical</i>	3.12, 3.13, 3.14	T_i	$cb_i + F_i$	3.3
	<i>last-in-place</i>	3.17	$\max(T_i, m + F_i)$	$cb_i + F_i$	3.3

Table 3.2: Summary of the results for the *flexible* parent allocation, for the working storage (factors written on disk) and for the total storage (when both the factors and working storage remain in core memory). In \mathcal{S}_1 , child subtrees are sorted in decreasing order of $x_i - y_i$.

3.3.5 Impact and summary of experimental results

Rather than presenting experimental results for all variants, we summarize the results by the following general observations and will only detail results for one of the cases. We suggest [102] and the references cited below for additional results.

- Sorting the children is not costly and should always be done [126].
- The impact on memory of using good tree traversal varies depending on the ordering [107].
- The relative gains of a good strategy on the total memory (including factors) is smaller than the relative gains on the working memory, depending on the size of the stack.
- The *last-in-place* assembly scheme allows gains on the working storage requirements between 20% and 30%. The total storage requirements (including factors) are also reduced significantly, depending on the volume of stack memory compared to factors.
- The new *max-in-place* assembly scheme, although slightly more complex to implement can be very efficient in terms of working storage; this is illustrated in Figure 3.4, where the x -axis corresponds to the matrices and orderings used in [13].
- The flexible allocation scheme can be very effective in many cases [104].

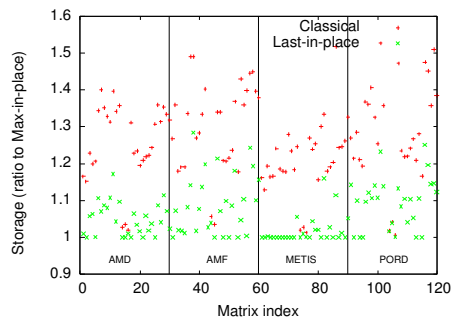


Figure 3.4: Working storage requirements of the *max-in-place* assembly scheme, compared to the *classical* and *last-in-place* assembly schemes. A *terminal* allocation of the frontal matrix of the parent is used.

Related to this last point, let us focus on gains one can get using the *flexible* allocation scheme with an *in-place* allocation scheme. We use a set of 44 matrices, listed in [106] and we present in Figure 3.5 the working storage requirement obtained with the flexible *in-place* schedule (“Flexible parent allocation, in-place”) compared to:

- the terminal allocation scheme, with in-place assembly of the last child;
- the early parent allocation scheme, with in-place assembly of the first child into the parent; as in Section 3.3.3, the child with largest peak $S_j^{early-ip}$ is ordered first to minimize the memory usage.
- the situation of Section 3.3.4.1.1 (“Flexible allocation”, “Classical assembly”), where we measure S^{flex} (Formula (3.11)) rather than $S^{flex-ip}$, and order the children correspondingly.

In this figure, METIS is used to illustrate the behaviour of the in-place algorithms, but results with other orderings would lead to the same type of remarks. (Results with other orderings are available in [104]). We observe that significant gains can be obtained compared to the non in-place case. Indeed, with the new algorithm, the gains obtained at each level of the tree modify the global traversal and often allow a better allocation position for the parent node. The difference between the flexible in-place and non in-place schemes comes also from the fact that the order in \mathcal{S}_1 is different for the two schemes. This explains that we gain more than just the memory of the largest contribution block of the complete assembly tree. Comparing in-place approaches, we also remark that memory ratios of up to 2 may be obtained over the case where the parent is allocated after the first child, and that huge gains can still be obtained over the *terminal* parent allocation scheme for very wide trees (GUPTA matrices).

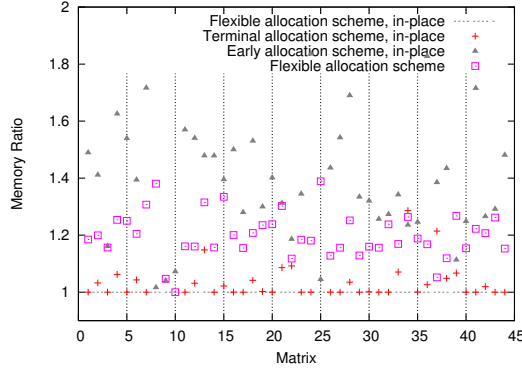


Figure 3.5: Comparison of the working storage with in-place and non in-place schemes, using METIS. Memory is normalized with respect to the in-place version of the flexible allocation scheme (with its associated schedule). Gains relative to the *terminal allocation scheme, in-place assembly* are equal to 60.03, 36.44, 19.37 and 3.03 for matrices 8, 9, 10 and 25, respectively.

Percentage of total memory for factors	Number of test cases	
	Flexible scheme	Flexible scheme+in-place
< 50 %	2	0
50%-60%	2	3
60%-70%	9	6
70%-80%	22	22
80%-100%	141	145
Total	176	176

Table 3.3: Number of combinations of test cases (matrix/ordering) for different ranges of the percentage of total memory used by the final factors.

We have also experimented the strategy consisting in minimizing the total memory with a *flexible, in-place assembly scheme* on the same range of test problems (see [104] for a full set of results). We observed significant gains compared to a *terminal allocation scheme, in-place* and an *early allocation scheme, in-place*, with their corresponding schedules (see [104] for more details). Furthermore, the memory ratio between the optimal in-place mechanism and the optimal non in-place mechanism can still reach 10% or 20% in some of the cases, so that it may be worth performing the assembly in-place if the implementation allows it: the percentage of total memory used by the factors has significantly increased compared to the case of the non in-place assembly (see Table 3.3), and most of the memory is now used by the factors; clearly, this was often not the case with the classical multifrontal scheme.

3.4 Postorders to reduce the volume of I/O

In this section, we assume that the factors are written to disk as soon as they are computed; thus, the corresponding I/O traffic is known and independent from the tree traversal. Assuming that the physical memory is insufficient for the working storage (contribution blocks and current frontal matrix), we therefore focus on the volume of I/O related to the stack of contribution blocks. We also assume that the current frontal matrix alone holds in core memory.

3.4.1 Stacks and I/O volumes

Because the contribution blocks are produced once and accessed once, they will be written to / read from disk at most once. This property gives an upper bound on the I/O volume equal to the sum of sizes of all the contribution blocks. However, we wish to limit this amount (that may be huge) by using as much of the available core memory as possible and performing I/O only when necessary. Said differently, we want to reach Objective 3.1:

Objective 3.1. *Given a postorder of the elimination tree and an amount of available core memory M_0 , our purpose is to find the I/O sequence that minimizes the I/O volume on the contribution blocks (the I/O volume on the factors being constant).*

The amount of core memory and the I/O volume thus appear to be related one to the other. To go further in the understanding of the notion of I/O volume, it is thus appealing to relate the evolution of the I/O volume to the evolution of the core memory. Said differently:

Objective 3.2. *Can we characterize the (optimum) volume of I/O as a function of the available core memory M_0 ?*

Actually, Objective 3.1 is easy to reach. Indeed, as we have mentioned, the contribution blocks are managed with a stack mechanism. In this context, a minimum I/O volume on the contribution blocks is obtained by writing the bottom of the stack *first* since the application will need it *last*. Property 3.1 states this result in other words:

Property 3.1. *For a given postorder of the elimination tree and a given amount of available core memory M_0 , the bottom of the stack should be written first when some I/O is necessary and this results in an optimum volume of I/O.*

Therefore, we can assume in the rest of the thesis (in the context of the multifrontal method) that the I/O's on the stack of contribution blocks are performed with respect to Property 3.1.

In particular, we can deduce the following result that aims at answering to Objective 3.2:

Property 3.2. *For a given postorder of the elimination tree, the (optimum) volume of I/O on the contribution blocks as a function of the available memory M_0 ($V^{I/O} = f(M_0)$) is a piece-wise affine function; the steepness of each piece is an integer multiple of -1 whose absolute value decreases when the value of M_0 increases.*

The proof of this property is technical and can be found in the appendix of [5]. We illustrate it on simple examples.

In Figure 3.6(a), the storage requirement for the application increases from $S = 0$ to $S = 4$ (GB, say), which corresponds to a total amount of *push* operations of 4, followed by a total amount of *pop* operations of 4. We use the notation $(push, 4), (pop, 4)$ to describe this sequence of memory accesses. If $M_0 > 4$ (for example, $M_0 = 4.5$) no I/O is necessary. If $M_0 = 2$, the storage increases from $S = 0$ to $S = 2$ without I/O, then the bottom of the stack is written to disk (2 units of I/O) in order to free space in memory for the 2 GB produced when S increases from 2 to 4. The storage then decreases to 2 when the top of the stack is accessed, and the 2 units of data that were written to disk have to be read again when the storage decreases from 2 to 0. Counting only write operations, the volume of I/O obtained for $M_0 = 2$ is 2. When M_0 further decreases, the volume of I/O will increase from 2 to a maximum value of 4. We see that on such a sequence, the volume of I/O will be equal to $\max(4 - M_0, 0)$, which corresponds to an affine function of steepness -1 .

If we now consider the sequence of Figure 3.6(b), which can be represented as $(push, 4); (pop, 4); (push, 4); (pop, 4)$, there are two peaks of stack storage, with no common data between the two peaks. Therefore, for $M_0 = 2$ (say), we will perform 2 units of I/O for the first peak, and 2 units of I/O for the second peak. Overall, the volume of I/O obtained is $2 \times \max(4 - M_0, 0)$ (piecewise affine function of steepness -2).

Let us now take a slightly more complex example: sequence $(push, 4); (pop, 2); (push, 1); (pop, 3)$ from Figure 3.6(c). We start performing I/O when the physical memory available M_0 becomes smaller than the

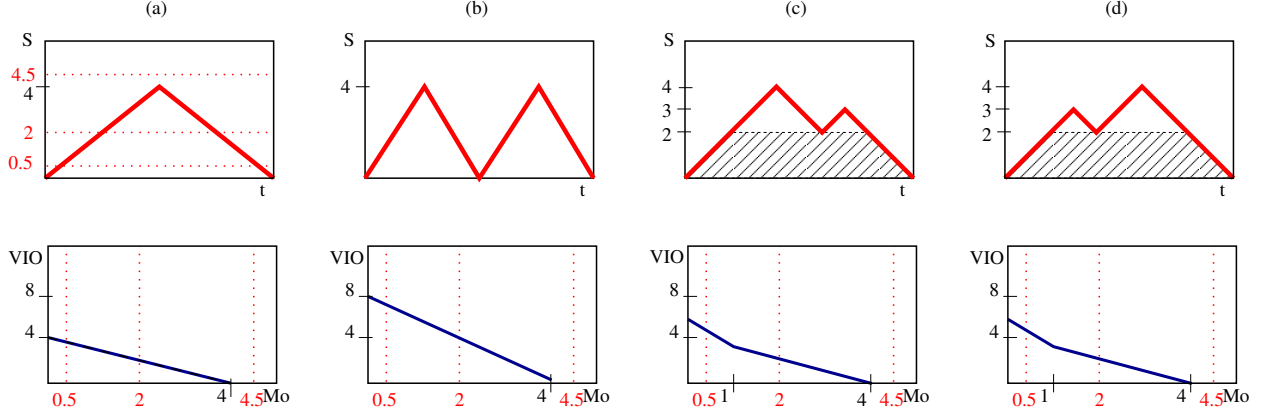


Figure 3.6: Evolution of the storage requirement of a stack (top) and I/O volume as a function of the available memory (bottom) on four examples (a, b, c and d).

storage requirement, equal to 4. If $M_0 = 2$, then the first peak of storage $S = 4$ will force us to write 2 GB from the bottom of the stack. Then the storage requirement decreases until $S = 2$. When S increases again and reaches the second peak $S = 3$, the bottom of the stack is still on disk and no supplementary I/O is necessary. Finally S decreases to 0 and the bottom of the stack (2 GB) that was written will be read from disk and consumed by the application. For this value of M_0 (2), the volume of I/O (written) is only equal to 2. In fact if $M_0 > 1$ the second peak has no impact on the volume of I/O . Said differently, even if there are two peaks of storage equal to 4 GB and 3 GB, 2 GB are shared by these two peaks and this common amount of data can be processed out-of-core only once. By trying other values of M_0 , one can observe that the volume of I/O , $V^{I/O}(M_0)$, is equal to $\max(4 - M_0, 0) + \max(1 - M_0, 0)$: we first count the volume of I/O resulting from the largest peak ($\max(4 - M_0, 0)$) and then only count new additional I/O resulting from the second peak ($\max(1 - M_0, 0)$). Note that the value 1 in the latter formula is obtained by subtracting 2 (volume of storage common to both peaks) to 3 (value of the peak). Again we have a piecewise affine function; its steepness is -1 when $M_0 > 1$ and -2 when $M_0 \leq 1$. We finally consider Figure 3.6(d). In that case, we obtain exactly the same result as in the previous case, with a volume of I/O equal to $\max(4 - M_0, 0)$ when $M_0 \geq 1$ to which we must add $\max(1 - M_0, 0)$ when $M_0 < 1$ for the I/O corresponding to data only involved in the first peak.

We summarize this behaviour. When the available memory M_0 becomes slightly smaller than the in-core threshold, if the available memory decreases by 1 GB (say), the volume of I/O will increase by 1 GB (steepness -1). This corresponds to a line of equation in this context $y(M_0) = \text{peak storage} - M_0$, which represents a lower bound for the actual volume of I/O . For smaller values of the available memory, reducing the available memory of 1 GB may increase the volume of I/O by 2 GB, 3 GB or more.

In the next subsection, we introduce some notations that we use next to give a formal way of forecasting the volume of I/O in the multifrontal method. Experiments on real matrices will then be discussed in Section 3.4.8.

3.4.2 Notations

We need some additional notations, on top of the ones introduced in Section 3.3.1. We still consider a generic family with a parent and n children numbered $j = 1, \dots, n$ and define:

- M_0 , the amount of available core memory;
- A / A_j , the core memory effectively used to process the subtree rooted at the parent / at child j (note that $A_j = \min(S_j, M_0)$).

- $V^{I/O} / V_j^{I/O}$ the volume of I/O required to process the subtree rooted at the parent $/$ at child j given an available memory of size M_0 . When needed, we use $V^{I/O,term}$ and $V^{I/O,flex}$ to design the volumes of I/O with the flexible and *terminal* allocation schemes, respectively. We also use $V^{I/O,term,ip}$ and $V^{I/O,flex,ip}$ in case of *in-place* assembly.

Clearly when the storage requirement S exceeds M_0 at the root of a given subtree, $V^{I/O}$ will be positive for that subtree.

3.4.3 Terminal allocation of the parent, classical assembly

In this Section, we consider that the allocation of the parent is done after all child subtrees are processed (*terminal* allocation) and cannot be done *in-place*. Focusing on memory-handling issues, the multifrontal method with terminal allocation may be presented as in Algorithm 3.1, where an assembly step (line $as_{\mathcal{N}}$) always requires the frontal matrix of the parent to be in memory. In an out-of-core assembly, we assume that a contribution block can be partially on disk during assembly operations.

```

foreach node  $\mathcal{N}$  in the tree (postorder traversal) do
   $al_{\mathcal{N}}$ : Allocate memory for the frontal matrix associated with  $\mathcal{N}$  ;
  if  $\mathcal{N}$  is not a leaf then
     $as_{\mathcal{N}}$ : Assemble contribution blocks from children ;
     $f_{\mathcal{N}}$ : Perform a partial factorization of the frontal matrix of  $\mathcal{N}$ , writing factors to disk on the fly;
    keep the contribution block (in memory or on disk, possibly partially) for later use;

```

Algorithm 3.1: Multifrontal method with factors on disk.

3.4.3.1 Illustrative example and formal expression of the I/O volume

Before providing a formal expression of the I/O volume, we illustrate the memory and I/O behaviours on the small example given in Figure 3.7 (left): we consider a root node (**e**) with two children (**c**) and (**d**). The frontal matrix of (**e**) requires a storage $m_e = 5$ (let us assume, for example, that this means 5 GB). The contribution blocks of (**c**) and (**d**) require a storage $cb_c = 4$ and $cb_d = 2$, while the storage requirements for their frontal matrices are $m_c = 6$ and $m_d = 8$, respectively. (**c**) has itself two children (**a**) and (**b**) with characteristics $cb_a = cb_b = 3$ and $m_a = m_b = 4$. We assume that the core memory available is $M_0 = 8$.

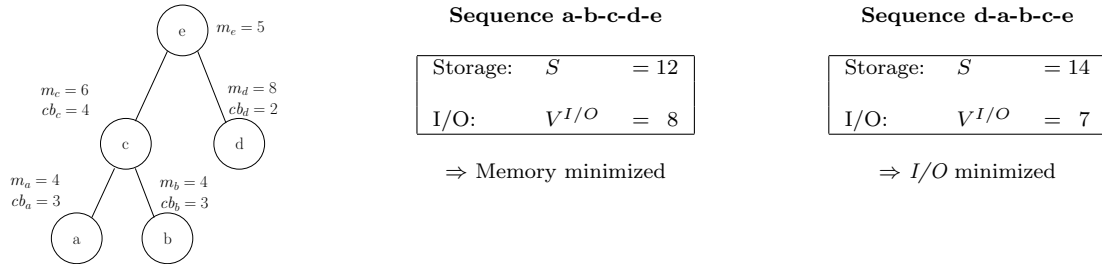


Figure 3.7: Influence of the postorder on the storage requirement and on the volume of I/O (with $M_0 = 8$).

To respect a postorder traversal, there are two possible ways to process this tree: (a-b-c-d-e) and (d-a-b-c-e). (Note that (**a**) and (**b**) are identical and can be swapped.) We now describe the memory behaviour and I/O operations in each case. We first consider the postorder (a-b-c-d-e). (**a**) is first allocated ($m_a = 4$) and factored (we write its factors of size $m_a - cb_a = 1$ to disk), and $cb_a = 3$ remains in memory. After (**b**) is processed, the memory contains $cb_a + cb_b = 6$. A peak of storage $S_c = 12$ is then reached when the frontal matrix of (**c**) is allocated (because $m_c = 6$). Since only 8 (GB) can be kept in core memory, this

forces us to write to disk a volume of data equal to 4 GB. Thanks to the postorder and the use of a stack, these 4 GB are the ones that will be reaccessed last; they correspond to the bottom of the stack. During the assembly process we first assemble contributions that are in memory, and then read 4 GB from disk to assemble them in turn in the frontal matrix of **(c)**. Note that (here but also more generally), in order to fit the memory requirements, the assembly of data residing on disk may have to be performed by panels (interleaving the read and assembly operations). After the factors of **(c)** of size $m_c - cb_c = 2$ are written to disk, its contribution block $cb_c = 4$ remains in memory. When the leaf node **(d)** is processed, the peak of storage reaches $cb_c + m_d = 12$. This leads to a new volume of I/O equal to 4 (and corresponding to cb_c). After **(d)** is factored, the storage requirement is equal to $cb_c + cb_d = 6$ among which only $cb_d = 2$ is in core (cb_c is already on disk). Finally, the frontal matrix of the parent (of size $m_e = 5$) is allocated, leading to a storage $cb_c + cb_d + m_e = 11$: after cb_d is assembled in core (into the frontal matrix of the parent), cb_c is read back from disk and assembled in turn. Overall the volume of data written to (and read from) disk² is $V_e^{I/O}(\text{a-b-c-d-e}) = 8$ and the peak of storage was $S_e(\text{a-b-c-d-e}) = 12$.

When the tree is processed in order (d-a-b-c-e), the storage requirement successively takes the values $m_d = 8$, $cb_d = 2$, $cb_d + m_a = 6$, $cb_d + cb_a = 5$, $cb_d + cb_a + m_b = 9$, $cb_d + cb_a + cb_b = 8$, $cb_d + cb_a + cb_b + m_c = 14$, $cb_d + cb_c = 6$, $cb_d + cb_c + m_e = 11$, with a peak $S_e(\text{d-a-b-c-e}) = 14$. Nodes **(d)** and **(a)** can be processed without inducing I/O , then 1 unit of I/O is done when allocating **(b)**, 5 units when allocating **(c)**, and finally 1 unit when the frontal matrix of the root node is allocated. We obtain $V_e^{I/O}(\text{d-a-b-c-e}) = 7$.

We observe that the postorder (a-b-c-d-e) minimizes the peak of storage and that (d-a-b-c-e) minimizes the volume of I/O . This shows that minimizing the peak of storage is different from minimizing the volume of I/O .

All the process described above is illustrated in Figure 3.8, which represents the evolution of the storage in time for the two postorders (a-b-c-d-e) and (d-a-b-c-e) (subfigures 3.8(a) and 3.8(b), respectively). The storage increases when memory is allocated for a new frontal matrix of size x ($al_N(x)$); it decreases when contribution blocks of size y are assembled into the frontal matrix of their parent ($as_N(y)$) and when factors of size z are written to disk ($f_N(z)$). When the storage is larger than the available memory M_0 , this means that part of the stack is on disk. The core window is shaded in the figure, so that the white area below the core window corresponds to the volume of data on disk. Finally write and read operations on the stack are noted w_x and r_y , where x and y are written and read sizes, respectively. We can see that each time the storage is about to exceed the upper bound of the core window, a write operation is necessary. The volume of data of each read operation depends on the size of the contribution blocks residing on disk that need to be assembled.

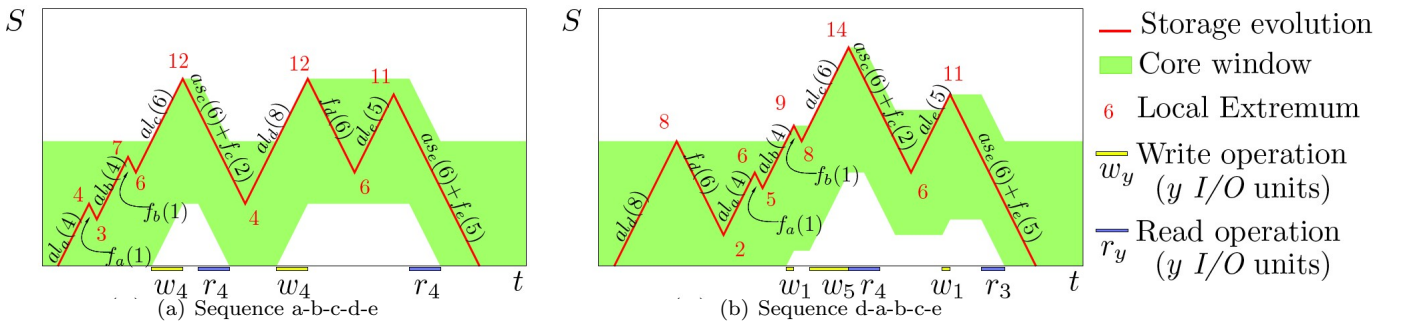


Figure 3.8: Evolution of the storage requirement S when processing the sample tree of Figure 3.7 with the two possible postorders, and subsequent I/O operations. Notations $al_N(x)$, $as_N(y)$ and $f_N(z)$ were introduced in Algorithm 3.1.

Since contribution blocks are stored using a stack mechanism, some contribution blocks (or parts of

²We do not count I/O for factors, that are independent from the postorder chosen: factors are systematically written to disk in all variants considered.

contribution blocks) may be kept in memory and consumed without being written to disk. Assuming that the contribution blocks are written only when needed (possibly only partially), that factors are written to disk as soon as they are computed, and that a frontal matrix always fits in core memory, we focus on the computation of the volume of I/O on this stack of contribution blocks.

We remind that the working storage requirement for the classical assembly, as obtained in Section 3.3.2 is (same equation as 3.1):

$$S^{term} = \max \left(\max_{j=1,n} (S_j^{term} + \sum_{k=1}^{j-1} cb_k), m + \sum_{k=1}^n cb_k \right) \quad (3.19)$$

In our out-of-core context, we now assume that we are given a core memory of size M_0 . If $S > M_0$, some I/O will be necessary. The data that must be written to disk are given by Property 3.1 (write bottom of the stack in priority), which was already used in an informal way in the example at the beginning of this section.

To simplify the discussion we first consider a set of subtrees and their parent, and suppose that $S_j \leq M_0$ for all children j . The volume of contribution blocks that will be written to disk corresponds to the difference between the memory requirement at the moment when the peak S is obtained and the size M_0 of the memory allowed (or available). Indeed, each time an I/O is done, an amount of temporary data located at the bottom of the stack is written to disk. Furthermore, data will only be reused (read from disk) when assembling the parent node. More formally, the expression of the volume of I/O for the terminal allocation scheme with classical assembly, using Formula (3.1) for the storage requirement, is:

$$V^{I/O,term} = \max \left(0, \max_{j=1,n} (S_j^{term} + \sum_{k=1}^{j-1} cb_k), m + \sum_{k=1}^n cb_k - M_0 \right) \quad (3.20)$$

As each contribution written is read once, $V^{I/O,term}$ will arbitrarily refer to the volume of data written.

We now suppose that there exists a child j such that $S_j > M_0$. We know that the subtree rooted at child j will have an intrinsic volume of I/O $V_j^{I/O,term}$ (recursive definition based on a bottom-up traversal of the tree). Furthermore, we know that the memory for the subtree rooted at child j cannot exceed the physical memory M_0 . Thus, we will consider that it uses a memory exactly equal to M_0 ($A_j^{term} \stackrel{def}{=} \min(S_j^{term}, M_0)$), and that it induces an intrinsic volume of I/O equal to $V_j^{I/O,term}$. With this definition of A_j as the *active memory*, *i.e.* the amount of core memory effectively used to process the subtree rooted at child j , we can now generalize Formula (3.20). We obtain:

$$V^{I/O,term} = \max \left(0, \max_{j=1,n} (A_j^{term} + \sum_{k=1}^{j-1} cb_k), m + \sum_{k=1}^n cb_k - M_0 \right) + \sum_{j=1}^n V_j^{I/O,term} \quad (3.21)$$

To compute the volume of I/O on the complete tree, we recursively apply Formula (3.21) at each level (knowing that $V^{I/O,term} = 0$ and $A = S = m$ for leaf nodes). The volume of I/O for the factorization is then given by the value of $V^{I/O,term}$ at the root.

3.4.3.2 Minimizing the I/O volume

It results from Formula (3.21) that minimizing the volume of I/O is equivalent to minimizing the expression $\max_{j=1,n} (A_j + \sum_{k=1}^{j-1} cb_k)$, since it is the only term sensitive to the order of the children.

Thanks to Theorem 3.1 (proved in [126]), we deduce that we should process the children nodes in decreasing order of $A_j - cb_j = \min(S_j, M_0) - cb_j$. (This implies that if all subtrees require a storage $S_j > M_0$ then **MinIO** will simply order them in increasing order of cb_j .) An optimal postorder traversal of the tree is then obtained by applying this sorting at each level of the tree, constructing Formulas (3.1) and (3.21) from bottom to top. We will name **MinIO** this algorithm.

Note that, in order to minimize the peak of storage (defined in Formula (3.1)), children had to be sorted (at each level of the tree) in decreasing order of $S_j - cb_j$ rather than $A_j - cb_j$. Therefore, on the example discussed before, the subtree rooted at **(c)** ($S_c - cb_c = 12 - 4 = 8$) had to be processed before the subtree rooted at **(d)** ($S_d - cb_d = 8 - 2 = 6$). The corresponding algorithm (that we name **MinMEM** and that leads to the postorder (a-b-c-d-e)) is different from **MinIO** (that leads to (d-a-b-c-e)): minimizing the storage requirement is different from minimizing the I/O volume; it may induce a volume of I/O larger than needed. Conversely, when the stack fits in core memory, M_0 is larger than S_j and $A_j = S_j$ for all j . In that case, **MinMEM** and **MinIO** lead to the same tree traversal and to the same peak of core memory.

3.4.4 In-place assembly of the last contribution block

As explained before, this variant assumes that the memory of the frontal matrix of the parent may overlap with (or include) that of the contribution block from the last child. The contribution block from the last child is then expanded (or assembled *in-place*) in the memory of the parent. Since the memory of a contribution block can be large, this scheme can have a strong impact on both storage and I/O requirements. In this context, the storage requirements needed to process a given subtree is given by Formula (3.2).

In an out-of-core context, the use of this *in-place* scheme induces a modification of the amount of data that has to be written to/read from disk. As previously for the memory requirement, the volume of I/O to process a given node with n children (Formula (3.21)) becomes:

$$V^{I/O,term,ip} = \max \left(0, \max_{j=1,n} (A_j^{term-ip} + \sum_{k=1}^{j-1} cb_k), m + \sum_{k=1}^{\boxed{n-1}} cb_k) - M_0 \right) + \sum_{j=1}^n V_j^{I/O,term,ip},$$

where this time, $A_j^{term-ip} \stackrel{def}{=} \min(S_j^{term-ip}, M_0)$ (see Formula (3.2)). Once again, the difference comes from the *in-place* assembly of the contribution block coming from the last child. Because $m + \sum_{k=1}^{n-1} cb_k = \max_{j=1,n} (m + \sum_{k=1}^{j-1} cb_k)$, this formula can be rewritten as:

$$V^{I/O,term,ip} = \max \left(0, \max_{j=1,n} (\max(A_j^{term-ip}, m) + \sum_{k=1}^{j-1} cb_k) - M_0 \right) + \sum_{j=1}^n V_j^{I/O,term,ip} \quad (3.22)$$

Thanks to Theorem 3.1, minimizing the above quantity can be done by sorting the children nodes in decreasing order of $\max(A_j^{term-ip}, m) - cb_j$, at each level of the tree.

3.4.5 In-place assembly of the largest contribution block

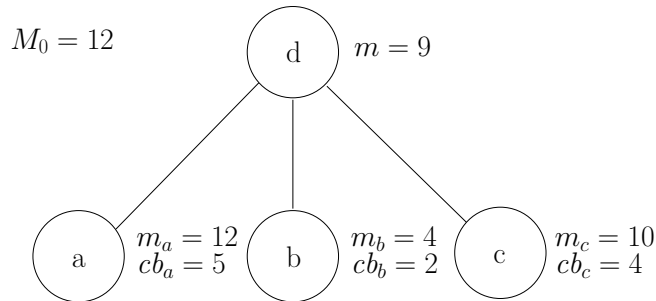


Figure 3.9: Example of a tree where **MinIO** + *last-in-place* is better than the *max-in-place* variant.

In an out-of-core context, it is not immediate nor easy to generalize **MinIO** to the in-place assembly of the largest contribution block (see Section 3.2). The problem comes from the fact that the largest contribution

block, if it does not correspond to the last child, may have to be written to disk to leave space for the subtrees that come after it in the postorder. Let us illustrate the difficulties one may encounter on the example provided in Figure 3.9. We first remark that the optimal order for the `MinIO` + *last-in-place* variant gives a sequence of children nodes (a-b-c), to which corresponds a volume of I/O equal to 5 (see Section 3.4.5). Let us now consider the *max-in-place* case. Assuming for the moment that the order is still (a-b-c), we process child (a) and child (b) without performing I/O. In order to allocate the memory for $m_c = 10$, at least 5 units of data have to be written to disk among cb_a and cb_b , for example one may write all of cb_b and 3 units of data from cb_a . We process (c) and have in memory $cb_c = 4$ together with two units of data of cb_a . Assembling the largest contribution cb_a in-place then requires reading back the 3 units of data from cb_a from disk, and writing 1 unit of data from cb_c to disk to make space for the frontal matrix of node (d), of size $m = 9$. This is far less natural and it requires overall more I/O than an *in-place* assembly of the contribution block of the last child (which is in memory). By trying all other possible orders (a-c-b), (b-a-c), (b-c-a), (c-a-b), (c-b-a), we can observe with this example that it is not possible to obtain a volume of I/O with a *max-in-place* assembly smaller than the one we obtained with a *last-in-place* assembly (equal to 5). Thus, the *max-in-place* strategy in an out-of-core context appears complicated, and non optimal at least in some cases. Therefore, we propose to only apply the *max-in-place* strategy on parts of the tree that can be processed in-core. This is done in the following way: we first apply `MinMEM` + *max-in-place* in a bottom-up process to the tree. As long as this leads to a storage smaller than M_0 , we keep this approach to reduce the intrinsic in-core memory requirements. Otherwise, we *switch* to `MinIO` + *last-in-place* to process the current family and any parent family. In the following we name `MinIO` + *max-in-place* the resulting heuristic.

3.4.6 Theoretical comparison of `MinMEM` and `MinIO`

Theorem 3.4. *The volume of I/O induced by `MinMEM` (or any memory-minimization algorithm) may be arbitrarily larger than the volume induced by `MinIO`.*

Proof. In the following, we provide a formal proof for the *classical* and *last-in-place* assembly schemes, but it also applies to the strategies defined in Section 3.4.5 for the *max-in-place* scheme (which is identical to *last-in-place* on families where I/O are needed). Let M_0 be the core memory available and $\alpha (> 2)$ an arbitrarily large real number. We aim at building an assembly tree (to which we may associate a matrix, see remark in Section 3.3.1), for which:

- $S(\text{MinIO}) > S(\text{MinMEM})$ and
- the I/O volume induced by `MinMEM` (or any memory minimization algorithm), $V^{I/O}(\text{MinMEM})$, is at least α times larger than the one induced by `MinIO`, $V^{I/O}(\text{MinIO})$ – i.e., $V^{I/O}(\text{MinMEM})/V^{I/O}(\text{MinIO}) \geq \alpha$.

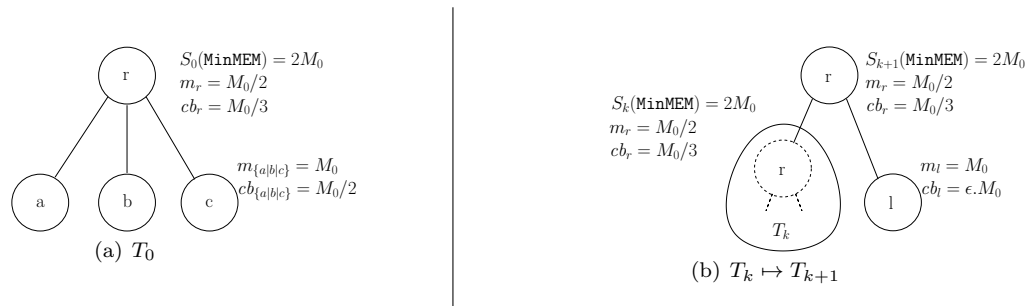


Figure 3.10: Recursive construction of an assembly tree illustrating Theorem 3.4.

We first consider the sample tree T_0 of Figure 3.10(a). It is composed of a root node (r) and three leaves (a), (b) and (c). The frontal matrices of (a), (b), (c) and (r) respectively require a storage $m_a = m_b = m_c = M_0$ and $m_r = M_0/2$. Their respective contribution blocks are of size $cb_a = cb_b = cb_c = M_0/2$ and $cb_r = M_0/3$. Both for the *classical* and *last-in-place* assembly schemes, it follows that the storage required

to process T_0 is $S_0(\text{MinMEM}) \stackrel{\text{def}}{=} S_r(\text{MinMEM}) = 2M_0$, leading to a volume of I/O $V_0^{I/O} \stackrel{\text{def}}{=} V_r^{I/O} = M_0$. We now define a set of properties \mathcal{P}_k , $k \geq 0$, as follows.

Property \mathcal{P}_k : Given a subtree T , T has the property \mathcal{P}_k if and only if: (i) T is of height $k + 1$; (ii) the peak of storage for T is $S(\text{MinMEM}) = 2M_0$; and (iii) the frontal matrix at the root (r) of T is of size $m_r = M_0/2$ with a contribution block of size $cb_r = M_0/3$.

By definition, T_0 has property \mathcal{P}_0 . Given a subtree T_k which verifies \mathcal{P}_k , we now build recursively another subtree T_{k+1} which verifies \mathcal{P}_{k+1} . To proceed we root T_k and a leaf node (l) to a new parent node (r), as illustrated in Figure 3.10(b). The frontal matrix of the root node has characteristics $m_r = M_0/2$ and $cb_r = M_0/3$, and the leaf node (l) is such that $m_l = S_l = M_0$ and $cb_l = \epsilon M_0$. The value of ϵ is not fixed yet but we suppose $\epsilon < 1/10$. The active memory usage for T_k and (l) are $A_k = \min(S_k, M_0) = M_0$ and $A_l = \min(S_l, M_0) = M_0$. Because all trees T_k (including T_0) verify the constraints defined at the beginning of Section 3.3.1, it is possible to associate a matrix to each of these trees. **MinMEM** processes such a family in the order (T_k - l - r) because $S_k - cb_k > S_l - cb_l$. This leads to a peak of storage equal to $S_{k+1}(\text{MinMEM}) = 2M_0$ (obtained when processing T_k). Thus T_{k+1} verifies \mathcal{P}_{k+1} . We note that **MinMEM** leads to a volume of I/O equal to $V_{k+1}^{I/O}(\text{MinMEM}) = M_0/3 + V_k^{I/O}(\text{MinMEM})$ (Formulas (3.21) and (3.22) for the *classical* and *last-in-place*, respectively).

Since $S_k(\text{MinIO})$ is greater than or equal to $S_k(\text{MinMEM})$, we can deduce that **MinIO** would process the family in the order (l - T_k - r) because $A_l - cb_l > A_k - cb_k$ (or $\max(A_l, m_r) - cb_l > \max(A_k, m_r) - cb_k$ in the *last-in-place* case). In that case, we obtain a peak of storage $S_{k+1}(\text{MinIO}) = \epsilon M_0 + S_k(\text{MinIO})$ and a volume of I/O $V_{k+1}^{I/O}(\text{MinIO}) = \epsilon M_0 + V_k^{I/O}(\text{MinIO})$.

Recursively, we may build a tree T_n by applying n times this recursive procedure. As $S_0(\text{MinIO}) = 2M_0$, we deduce that $S_n(\text{MinIO}) = (2 + n\epsilon)M_0$ which is strictly greater than $S_n(\text{MinMEM}) = 2M_0$. Furthermore, because $V_0^{I/O}(\text{MinMEM}) = V_0^{I/O}(\text{MinIO}) = M_0$, we conclude that $V_n^{I/O}(\text{MinMEM}) = nM_0/3 + M_0$ while $V_n^{I/O}(\text{MinIO}) = n\epsilon M_0 + M_0$. We thus have: $V_n^{I/O}(\text{MinMEM})/V_n^{I/O}(\text{MinIO}) = (1 + n/3)/(1 + n\epsilon)$. Fixing $n = \lceil 6\alpha \rceil$ and $\epsilon = 1/\lceil 6\alpha \rceil$ we finally get: $V_n^{I/O}(\text{MinMEM})/V_n^{I/O}(\text{MinIO}) \geq \alpha$.

We have shown that the I/O volume induced by **MinMEM**, $V^{I/O}(\text{MinMEM})$, is at least α times larger than the one induced by **MinIO**. To conclude we have to show that it would have been the case for any memory-minimization algorithm (and not only **MinMEM**). This is actually obvious since the postorder which minimizes the memory is unique: (l) has to be processed after T_k at any level of the tree. □

3.4.7 Flexible parent allocation

We recall that the *flexible* allocation scheme consists in allowing the parent allocation to be done right after an arbitrary child subtree (instead of after all child subtrees) The parent allocation can still be done *in-place* or not, similar to what has been studied for the *terminal* allocation (Equation (3.21) for the *terminal* non-in-place allocation, and Equation (3.22) for the *terminal in-place* allocation).

In the case of the *flexible* allocation, the main difference with Formula (3.21) is that a child j processed after the parent allocation may also generate I/O . If such a child cannot be processed in-core together with the frontal matrix of the parent, then part of that frontal matrix (or all of it) has to be written to disk to make room for the child subtree. This possible extra- I/O corresponds to underbrace (**a**) of Formula (3.23). After that, the factor block of the frontal matrix of child j is written to disk and its contribution block is ready to be assembled into the frontal matrix of the parent. However, we assume that we cannot easily determine which rows of the contribution block, if any, can be assembled into the part of the frontal matrix available in memory³. Therefore this latter frontal matrix is fully re-loaded into memory (reading back from disk the part previously written). This operation may again generate I/O : if the contribution block of child j and the frontal matrix of its parent cannot fit together in memory, a part of cb_j has to be written to disk, then read back (panel by panel) and finally assembled. This second possible extra- I/O is counted in underbrace (**b**) of Formula (3.23). All in all, and using the storage definition from Formula (3.11), the

³See [5], Section 4.6.2 for ideas of what could be done.

volume of I/O required to process the subtree rooted at the parent node is given by:

$$\begin{aligned}
V^{I/O,flex} = & \max \left(0, \max \left(\max_{j=1, \boxed{P}} \left(A_j^{flex} + \sum_{k=1}^{j-1} cb_k \right), m + \sum_{k=1}^{\boxed{P}} cb_k \right) - M_0 \right) \\
& + \underbrace{\sum_{j=p+1}^n \left(\max(0, m + A_j^{flex} - M_0) \right)}_{\text{(a)}} + \underbrace{\sum_{j=p+1}^n \left(\max(0, m + cb_j - M_0) \right)}_{\text{(b)}} \\
& + \sum_{j=1}^n V_j^{I/O,flex}
\end{aligned} \tag{3.23}$$

Again, $A_j^{flex} \stackrel{def}{=} \min(S_j^{flex}, M_0)$ and a recursion gives the I/O volume for the whole tree.

As done in other situations, the formula can be adapted to a *last-in-place* assembly scheme, meaning that child p is assembled *in-place* in the frontal matrix of the parent, leading to:

$$\begin{aligned}
V^{I/O,flex,ip} = & \max \left(0, \max \left(\max_{j=1,p} \left(A_j^{flex-ip} + \sum_{k=1}^{j-1} cb_k \right), m + \sum_{k=1}^{\boxed{P-1}} cb_k \right) - M_0 \right) \\
& + \sum_{j=p+1}^n \left(\max(0, m + A_j^{flex-ip} - M_0) \right) + \sum_{j=p+1}^n \left(\max(0, m + cb_j - M_0) \right) \\
& + \sum_{j=1}^p V_j^{I/O,flex,ip},
\end{aligned} \tag{3.24}$$

where $A_j^{flex-ip} \stackrel{def}{=} \min(S_j^{flex-ip}, M_0)$.

With the *terminal* allocation scheme, it was possible to minimize the volume of I/O by sorting the children in an appropriate order. In the *flexible* allocation scheme, one should moreover determine the appropriate *split point*, *i.e.*, the best value for p . The flexible I/O volume is minimized when both:

- i the children processed *before* the parent allocation are correctly separated from the ones processed *after*;
- ii each one of this set is processed in an appropriate order.

Exploring these $n.n!$ combinations is not always conceivable since some families may have a very large number n of children (more than one hundred for instance for matrix **GUPTA3**). However, we have shown earlier that the order in the first set is known: decreasing order of $A_j^{flex} - cb_j$ (resp. $\max(A_j^{flex-ip}, m) - cb_j$ in the *in-place* case). Moreover, the I/O volume on the children processed after the allocation is independent of their relative processing order. Said differently, these two remarks mean that condition **(ii)** is actually immediate when **(i)** is determined. Therefore we mainly have to determine to which set (before or after the parent allocation) each child belongs to. However, this still makes an exponential (2^n) number of possibilities to explore and motivates to further reduce the complexity.

Actually, the decision problem associated with this minimization problem is NP-complete. In other words, given an arbitrary volume of I/O V , there is no deterministic polynomial algorithm that can consistently decide whether there exists a partition of the children inducing a volume of I/O lower than or equal to V (except if $P = NP$). The proof of NP-completeness (reduction from 2-PARTITION) is available in [5].

To further reduce the complexity, remark that if a child is such that $m + S_j^{flex} \leq M_0$, ordering this child *after* the parent allocation does not introduce any additional I/O (**(a)** and **(b)** are both 0 in Equation (3.23)), whereas this may not be the case if it is processed before the parent allocation. Therefore, we conclude that we can place all children verifying $m + S_j^{flex} \leq M_0$ after the parent allocation. Furthermore, consider the case where $S_j^{flex} \geq M_0 - m + cb_j$ and $m + cb_j \leq M_0$. Processing this child *after* the parent allocation (see

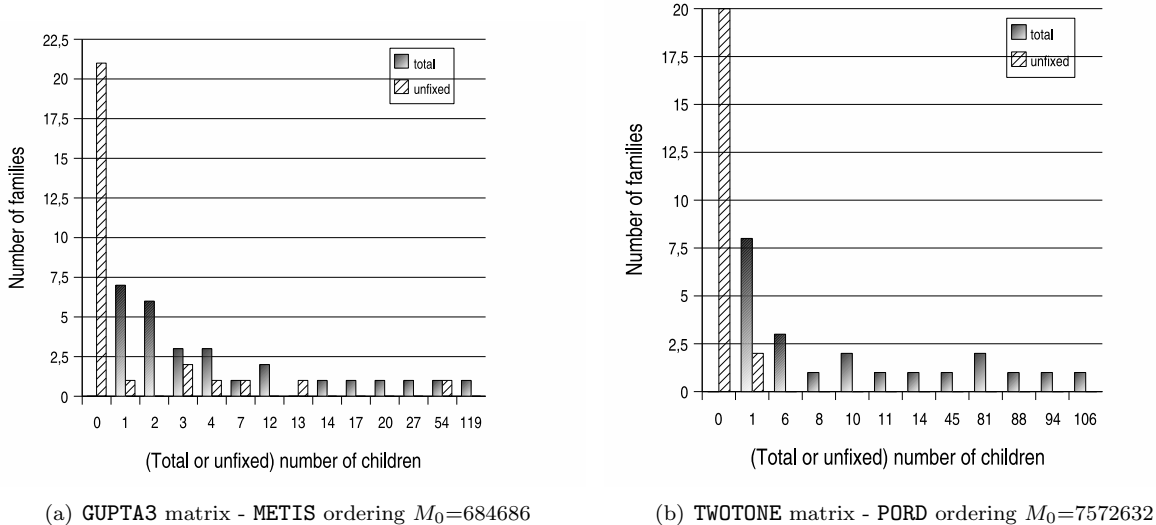


Figure 3.11: Distribution of the families in function of their total and unfixed number of children. After a straightforward analysis, most families have few (or no) unfixed children.

Formula (3.23)) leads to a volume of I/O either equal to m (if $S_j^{flex} \geq M_0$) – which is greater than cb_j , or to $S_j^{flex} - M_0 + m$ (if $S_j^{flex} \leq M_0$) – which is also greater than cb_j . On the other hand, treating that child *first* (this may not be optimal) will lead to a maximum additional volume of I/O equal to cb_j . Therefore, we can conclude that we should process it *before* the parent allocation. For the same type of reasons, children verifying $S_j^{flex} \geq 2(M_0 - m)$ and $m + cb_j > M_0$ should also be processed *before* the parent allocation.

We will say that a child is *fixed* if it verifies one of the above properties: a straightforward analysis - independent of the metrics of its siblings - determines if it should be processed before or after the allocation of the parent node. Even though the number of *fixed* children can be large in practice, some matrices may have a few families with a large number of *unfixed* children, as shown in Figure 3.11 for two sparse problems, in the *in-place* allocation case. For instance, among the 28 families inducing I/O for the GUPTA3 matrix ordered with METIS when a memory of $M_0 = 684686$ scalars is available, 21 families have no unfixed children (thus for them the optimum process is directly known), but one family keeps having 54 unfixed children. In such cases, heuristics are necessary. The one we designed consists in moving repeatedly *after* the parent allocation the child which is responsible for the peak of storage, until one move does not decrease the volume of I/O anymore. The whole process is called **Flex-MinIO**.

3.4.8 Experimental results

Minimizing the volume of I/O can be critical when processing large problems. The first thing that one should observe is that as soon as I/O 's are required, it is critical to give as much memory as possible to the factorization algorithm: decreasing M_0 by x GB (say) implies an increase of I/O volume equal to $k.x$, where k is an integer greater or equal to 1 (see also Section 3.4.1).

We provide here a few illustrative results and remarks with the *terminal* allocation, then with the *flexible* allocation.

3.4.8.1 Terminal allocation

First, we observed that using an *in-place* assembly scheme is in general very useful: on most test matrices that were used, it reduces the I/O volume by a factor of two. Second, significant gains are obtained when comparing the volumes of I/O obtained with **MinMEM** and **MinIO**. We report in Figure 3.12 the largest gains

observed for each matrix on the range of values of M_0 larger than the size of the largest frontal matrix. The largest gain is obtained for the case **SPARSINE-PORD**, where **MinIO** is better than **MinMEM** by a factor of 5.58. Generally, the largest profits from **MinIO** are obtained when matrices are preprocessed with orderings which tend to build irregular assembly trees: **AMF**, **PORD** and - to a lesser extent - **AMD** (see [107] for more illustrations on the impact of ordering on tree topologies). Intuitively, this is because on such trees, there is a higher probability to be sensitive to the order of children.

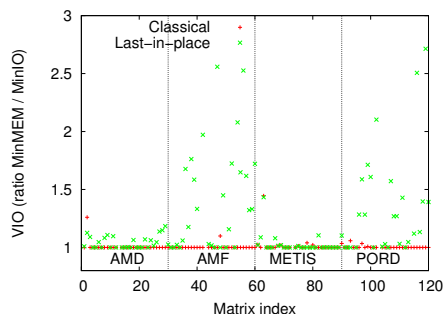


Figure 3.12: I/O volume obtained with **MinMEM** divided by the one obtained with **MinIO**. Thirty test matrices are ordered with four reordering heuristics separated by vertical bars. For each matrix-ordering, we report the largest gain obtained over all values of M_0 that exceed the size of the largest frontal matrix. The ratios for **GEO3D-25-25-100-AMF** and **SPARSINE-PORD** in the *last-in-place* scheme are equal to 5.12 and 5.58 and are not represented in the graph.

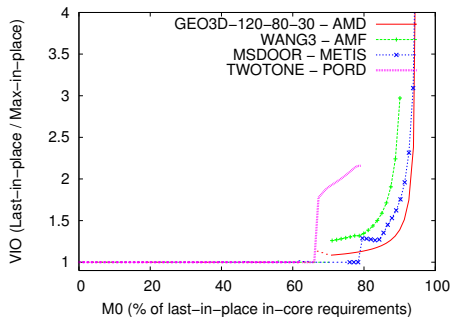


Figure 3.13: Impact of *max-in-place* assembly scheme on the volume of I/O (**MinIO** algorithms).

Finally, Figure 3.13 shows by how much the **MinIO** algorithm with a *max-in-place* assembly scheme improves the **MinIO** *last-in-place* one, on four selected matrices. When the available core memory decreases, the ratio is equal to 1 because in this case, the heuristic for the *max-in-place* assembly variant switches to the *last-in-place* scheme (as explained in Section 3.4.5) and the switch occurs very early. More experimental results and discussions are available in [13].

3.4.8.2 Flexible allocation

Considering now the case of *flexible* allocation, we illustrate the potential of this approach on four problems, for the *in-place* assembly scheme: Figure 3.14 shows the evolution of the volume of I/O depending on the available memory on the target machine. When a large amount of memory is available (right part of the graphs), the flexible allocation schemes (both **Flex-MinMEM** and **Flex-MinIO**) induce a small amount of I/O compared to the terminal allocation scheme (**Term-MinIO**). Indeed, with such an amount of memory, many children can be processed after the allocation of their parent without inducing any I/O (or inducing a small

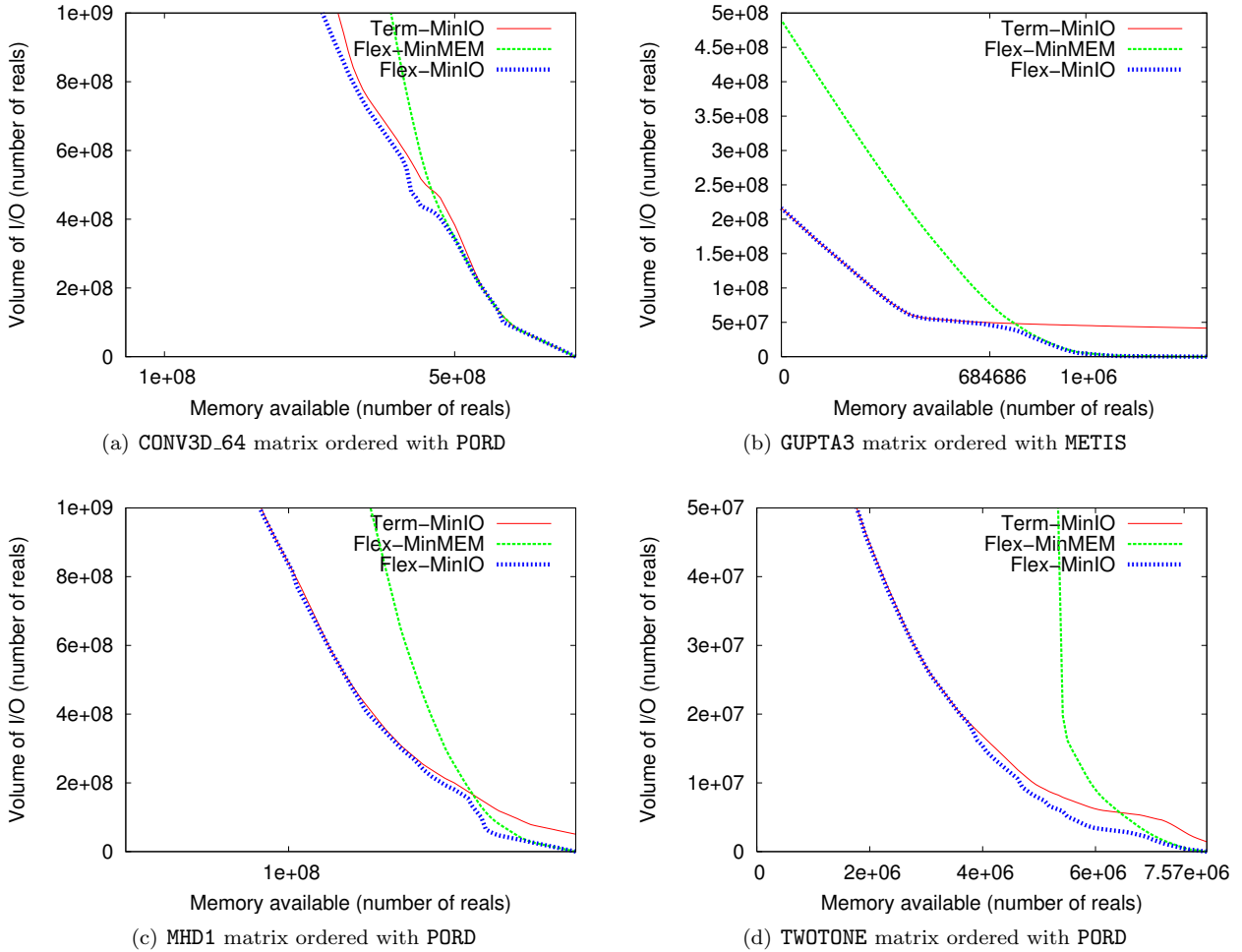


Figure 3.14: I/O volume on the stack of contribution blocks as a function of the core memory available for the three heuristics with four different matrices.

amount of I/O): the possible extra- I/O s corresponding to underbraces (a) and (b) of Formula (3.23) are actually equal (or almost equal) to zero for those children.

When the amount of available memory is small (left part of the graphs), the memory-minimizing algorithm (Flex-MinMEM) induces a very large amount of I/O compared to the I/O -minimization algorithms (both Flex-MinIO and Term-MinIO). Indeed, processing a child after the parent allocation may then induce a very large amount of I/O (M_0 is small in underbraces (a) and (b) of Formula (3.23)) but memory-minimization algorithms do not take into account the amount of available memory to choose the *split point*.

Finally, when the amount of available memory is intermediate, the heuristic we have proposed (Flex-MinIO) induces less I/O than both other approaches. Indeed, according to the memory, not only does the heuristic use a flexible allocation scheme on the families for which it is profitable, but it can also adapt the number of children to be processed after the parent allocation.

To conclude on flexible allocation schemes and I/O volumes, we refer the reader to [5], Chapter 4, for various discussions and extensions of this section.

3.5 Memory management algorithms

The different `MinMEM` and `MinIO` algorithms presented in the previous sections provide a particular postorder of the assembly tree. With *flexible* allocation schemes, they also compute the positions of the parent allocations. These algorithms can be applied during the analysis phase of a sparse direct solver. Then the numerical factorization phase relies on this information and should respect the predicted optimal metrics (memory usage, *I/O* volume). In this Section, we focus on the case where factors are written to disk as soon as they are computed, and consider the remaining working storage. We show that in this case, efficient memory management algorithms can be designed for the various assembly schemes considered. Because we consider that the factors are written to disk on the fly, we only have to store temporary frontal matrices and contribution blocks. We assume that those must be stored in a preallocated contiguous workarray W of maximum size M_0 , the available core memory. In this workarray, we manage one or two stacks depending on our needs, as illustrated in Figure 3.15. Another approach would consist in relying on dynamic allocation routines (such as `malloc` and `free`). Although those may still be efficient from a performance point of view, the use of such a preallocated workarray has several advantages over dynamic allocation, allowing for a tighter memory management as long as complicated garbage collection mechanisms can be avoided. In particular, several memory operations are possible with a workarray managed by the application that would be difficult or even impossible with standard dynamic allocation tools:

- In-place assemblies: with dynamic allocation, expanding the memory of the contribution block of a child node into the memory of the frontal matrix of the parent node would require to rely on a routine that extends the memory for the contribution block (such as `realloc`). This may imply an extra copy which cancels the advantages of in-place assemblies; with a preallocated workspace, we simply shift some integer pointers.
- Assuming that the frontal matrix uses a dense row-major or column-major storage and that the factors have been copied to disk, we can copy the contribution block of such a frontal matrix into a contiguous memory area that overlaps with the original location. With dynamic allocation, we would need to allocate the memory for the contribution block, perform the copies and then free the memory for the original frontal matrix. Even assuming that the contribution block is compacted in-place (inside the memory allocated for the frontal matrix), then it is not clear how to free the rest of the frontal matrix with dynamic allocation tools, whereas this can be done by shifting an integer pointer in our case.

Finally, the preallocated workarray allows for a very good locality, for example, when assembling the contributions from children into the frontal matrix of the parent, the entries of all the contribution blocks are contiguous in memory.

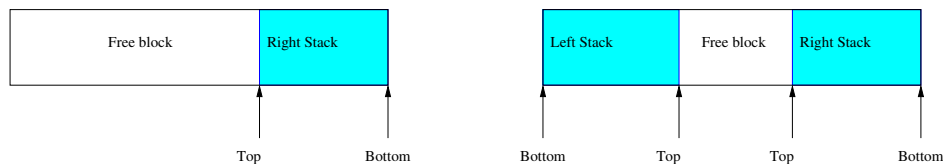


Figure 3.15: Subdivision of the main workarray, W , into one stack (left) or two stacks (right) of contribution blocks. The free block can be used to store the temporary frontal matrices.

3.5.1 In-core stack memory

In this section, we assume that the contribution blocks are processed in core. We first recall memory management algorithms that are used in existing multifrontal codes in Section 3.5.1.1. In Section 3.5.1.2, we then more specifically explain how to handle our new *max-in-place* assembly scheme (see the previous chapter). We generalize those algorithms to the multifrontal method with a *flexible* allocation in Section 3.5.1.3.

3.5.1.1 Recalling the classical and last-in-place assembly schemes

The *classical* and *last-in-place* approaches with a *terminal* allocation are already used in existing multifrontal codes. We recall them in this section in order to introduce notions that we will use in the other subsections. We have seen earlier (first part of Property 3.1) that since we have a postorder traversal, the access to the contribution blocks has the behaviour of a stack (in general, one uses the stack on the right of W). In other words, thanks to the postorder:

Property 3.3. *If the contribution blocks are stacked when they are produced, each time a frontal matrix is allocated, the contribution blocks from its children are available at the top of the stack.*

For example, at the moment of allocating the frontal matrix of node **(6)** in the tree of Figure 3.16, the stack contains, from bottom to top, cb_1 , cb_2 , cb_3 , cb_4 , cb_5 . The frontal matrix of **(6)** is allocated in the free block, then cb_5 and cb_4 (in that order) are assembled into it and removed from the stack. Once the assembly at the parent is finished, the frontal matrix is factorized, the factors are written to disk, and the contribution block (cb_6) is moved to the top of the stack.

The only difference between the *classical* and the *last-in-place* assembly schemes is that in the *last-in-place* case, the memory for the frontal matrix of the parent is allowed to overlap with the memory of the child available at the top of the stack. In the example, this means that if the free block on the left is not large enough for the frontal matrix of **(6)**, that frontal matrix is allowed to overlap with the memory of the contribution block of **(5)**, of size cb_5 , leading to significant memory gains. The contribution block of the child is expanded into the memory of the frontal matrix of the parent, and the contribution blocks from the other children are then assembled normally.

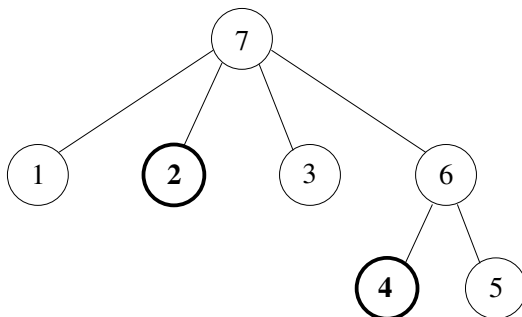


Figure 3.16: Example of a tree with 7 nodes. Nodes in bold correspond to the nodes with the largest contribution block among the siblings. (This property will be used in Section 3.5.1.2.)

3.5.1.2 In-place assembly of the largest contribution block

The new *max-in-place* assembly scheme that we have introduced consists in overlapping the memory of the parent with the memory of the largest child contribution block. For this to be possible, the largest contribution block must be available in a memory area next to the free block where the frontal matrix of the parent will be allocated. By using a special stack for the largest contribution blocks (the one on the left of W , see Figure 3.15), Property 3.3 also applies to the largest contribution blocks. Thus, when processing a parent node,

- the largest child contribution is available at the top of the left stack and can overlap with the frontal matrix of the parent; and
- the other contributions are available at the top of the right stack, just like in the *classical* case.

This is illustrated by the tree of Figure 3.16. When traversing that tree, we first stack cb_1 on the right of W , then stack cb_2 (identified as the largest among its siblings) on the left of W , then cb_3 on the right, cb_4 on the left, and cb_5 on the right. When node **(6)** is processed, the workarray W contains:

cb_2	cb_4	Free block	cb_5	cb_3	cb_1
--------	--------	------------	--------	--------	--------

The memory for the frontal matrix of **(6)** can overlap with cb_4 so that cb_4 is assembled *in-place*; cb_5 is then assembled normally. Note that the same type of situation will occur for the root node **(7)**: cb_2 (now available at the top of the left stack) will first be assembled *in-place*, the cb_6 , cb_3 and cb_1 (in that order) will be assembled from the right stack.

3.5.1.3 Flexible allocation of the frontal matrices

We now consider the *flexible* multifrontal method, as discussed in Section 3.4.7. Since the frontal matrix of a parent is allowed to be allocated before all children subtrees have been processed, several frontal matrices may be in core memory at the same time. Let us first consider the *classical* and *last-in-place* assembly scheme. On the example of Figure 3.17, we assume that the frontal matrix f_7 of node **(7)** is allocated after the treatment of node **(3)** and that the frontal matrix f_6 of node **(6)** is allocated after the treatment of node **(4)**.

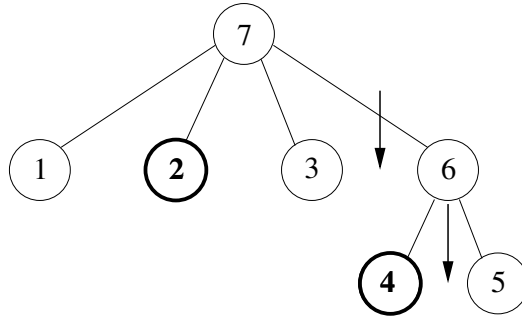


Figure 3.17: Reconsidering the example of Figure 3.16 with a *flexible* allocation. The arrows indicate the position at which the frontal matrices of the parents are allocated. Nodes in bold correspond to the nodes with the largest contribution block among the siblings processed before a parent allocation.

When processing node **(5)**, both f_7 and f_6 have been allocated in memory, although they cannot be factored yet. Similarly to the contribution blocks, we have the property that frontal matrices are accessed with a LIFO (Last In First Out) scheme: on our example, frontal matrices f_7 and f_6 are allocated in this order but f_6 is factored and released before f_7 . It is thus natural to store the frontal matrices in a stack too. Again, it is possible to manage both stacks in a single array and this approach allows for an overlapping in time of the stacks: i) one of the stack may be large when the other is small and vice-versa; ii) the frontal matrix may overlap with the last contribution block in the *last-in-place* case). We suppose that the right stack is used for the contribution blocks, and, this time, the left stack is used for the frontal matrices.

On our example, after node **(7)** has been allocated, the contributions of nodes **(1)**, **(2)** and **(3)** are assembled and released. Then, node **(4)** is factored and produces a contribution block so that at this time, the workarray contains:

f_7	Free block	cb_4
-------	------------	--------

 Frontal matrix of **(6)** is then allocated in the left stack. Remark that it is allowed to overlap with cb_4 in the *last-in-place* scheme. Assuming no overlap between f_6 and cb_4 , the workarray W contains:

f_7	f_6	Free block	cb_4
-------	-------	------------	--------

 cb_4 is assembled in f_6 and released. Next: node **(5)** is processed; its contribution is assembled into f_6 and released; **(6)** is factored; its contribution is assembled into f_7 and released; finally, **(7)** is factored.

Let us now consider the *max-in-place* assembly scheme. We need to store

- frontal matrices,
- normal contribution blocks,
- largest contribution blocks,

which would indicate the need for three stacks. However, the key idea here consists in observing that the stack for frontal matrices and for largest contribution blocks can use a single stack. For that, we must simply verify that (i) a largest contribution block produced before a front allocation is released after the front; (ii) a large contribution produced before a front allocation is released before. When a subtree has been processed, all the frontal matrices and contribution blocks related to other nodes than its root node have been released. Therefore, we only have to check that (i) and (ii) stand for the nodes that compose a family (we do not need to investigate the data related to the nodes inside the subtrees of the children). Let us consider a family. A number of p children are processed before the parent allocation. One of them, say j_0 ($j_0 \leq p$), provides the largest contribution block. This block is pushed on top of the left stack of the workarray W . When child p has been processed, this contribution block is still on the top of the left stack and can be extended *in-place* to constitute the frontal matrix. Contribution blocks from children j , $j \leq p, j \neq j_0$ are assembled from the right stack. Then, the children $j, j > p$ (and their subtrees) are processed in the available space and their contribution block are assembled into the frontal matrix on the fly. Next, the frontal matrix is factored, produces a contribution block that is either pushed on the left (if it is in turn the largest of its siblings) or on the right (otherwise). For instance, with the tree of Figure 3.17, the workarray W is as follows before the allocation of f_7 :

cb_2	Free block	cb_3	cb_1
--------	------------	--------	--------

Then f_7 overlaps with cb_2 which is on top of the left stack as required. After node (4) is processed, the left stack contains f_7 and cb_4 ; f_6 is allocated, overlapping with cb_4 ; f_5 is allocated and factored; cb_5 is stored in the right stack and assembled into f_6 , and so on. Overall, the left stack was used for the frontal matrices and cb_2 and cb_4 and the right stack was used for the other contribution blocks.

3.5.2 Out-of-core stacks

We now assume that contribution blocks may be written to disk when needed. When there is no more memory, Property 3.1 suggests that the bottom of the stack(s) should be written to disk first. Therefore, the question of how to reuse the corresponding workspace arises. We give a first natural answer to this question in Section 3.5.2.1, but it has some drawbacks and does not apply to all cases. Based on information that can be computed during the analysis phase, we then propose in Section 3.5.2.2 a new approach that greatly simplifies the memory management for all the considered assembly schemes.

3.5.2.1 Dynamic cyclic memory management

In the *classical* and *last-in-place* cases, only one stack is required. In order for new contribution blocks (stored at the top of the stack) to be able to reuse the space available at the bottom of the stack after write operations, a natural approach consists in using a cyclic array. From a conceptual point of view, the cyclic memory management is obtained by joining the end of the memory zone to its beginning, as illustrated in Figure 3.18. In this approach, the decision to free a part of the bottom of the stack is taken dynamically, when the memory is almost full. We illustrate this on the sample tree of Figure 3.7 processed in the postorder (d-a-b-c-e) with a *classical* assembly scheme. After processing nodes (d) and (a), one discovers that *I/O* has to be performed on the first contribution block produced (cb_d) only at the moment of allocating the frontal matrix of (b), of size $m_b = 4$ (see Figure 3.19(a)).

Note that a significant drawback of this approach is that a specific management has to be applied to the border, especially when a contribution block or a frontal matrix is split on both sides of the memory area (as occurs for frontal matrix m_b in Figure 3.19(a)).

Moreover, in the *max-in-place* case, such an extension is not as natural because of the existence of two stacks. That is why we propose in the next subsection another approach, which avoids a specific management of the borders for the *classical* and *last-in-place* cases, and allows to efficiently handle the *max-in-place* case.

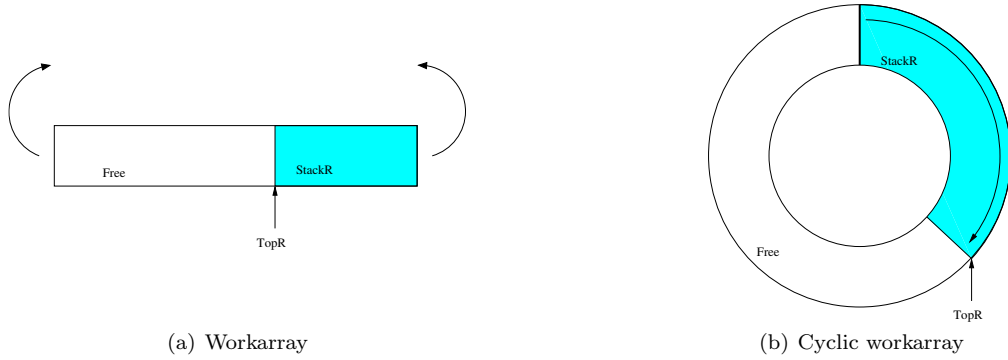


Figure 3.18: Folding a linear workarray (left) into a cyclic workarray (right).

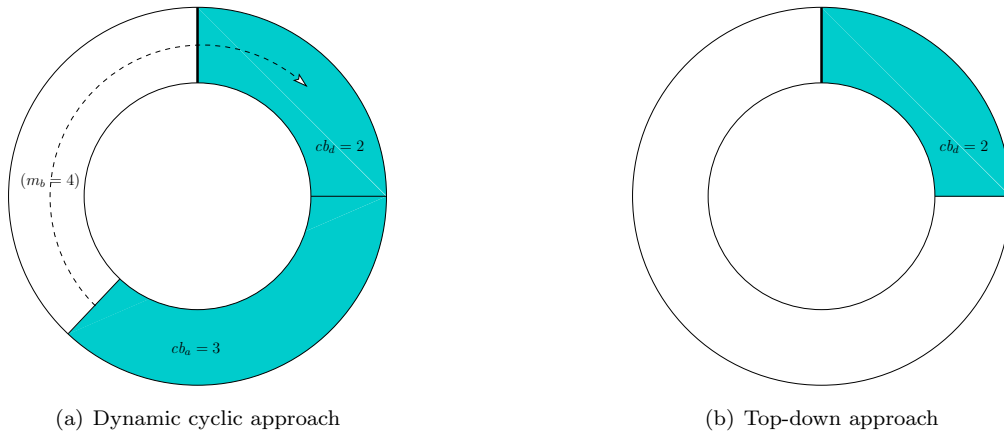


Figure 3.19: Memory state while processing the tree of Figure 3.7 in the postorder (d-a-b-c-e). The size of the workarray is $M_0 = 8$. With a dynamic approach (left), one discovers that I/O will be performed on cb_a only before dealing with node (b). With the approach of Section 3.5.2.2 (right), we know *a priori* that cb_a must be fully written to disk thanks to the analysis phase.

3.5.2.2 Using information from the analysis: static top-down formulation

In order to minimize the I/O volume in the previous approach, a contribution is only written to disk when the memory happens to be full: the decision of writing a contribution block (or a part of it) is taken dynamically. However, a better approach can be adopted. We explain it by listing some properties, each new property being the consequence of the previous one. For the moment, let us consider a *terminal* allocation.

Property 3.4. *While estimating the volume of I/O , the analysis phase can forecast whether a given contribution block will have to be written to disk or not.*

This property results from an instrumentation of the analysis phase that we describe in the following. When considering a parent node with n child subtrees, the volume of I/O $V_{family}^{I/O}$ performed on the children of that parent node is given by the first member (the recursive amount of I/O on the subtrees is not counted) of Formulas (3.21) and (3.22) respectively for the *classical* and *in-place* cases. For example,

$$V_{family}^{I/O} = \max \left(0, \max_{j=1, n} \left(A_j + \sum_{k=1}^{j-1} cb_k \right), m + \sum_{k=1}^n cb_k \right) - M_0 \quad (3.25)$$

for the *classical* assembly scheme. Given $V_{family}^{I/O}$ and knowing that we are going to write the contribution blocks produced first in priority, one can easily determine if the contribution block cb_j of the j^{th} child must be written to disk:

- if $\sum_{i=1}^j cb_i \leq V_{family}^{I/O}$, the volume of *I/O* for that family is not reached even when cb_j is included; therefore, cb_j must be entirely written to disk;
- if $\sum_{i=1}^{j-1} cb_i < V_{family}^{I/O} < \sum_{i=1}^j cb_i$, then cb_j should be partially written to disk and the volume written is $V_{family}^{I/O} - \sum_{i=1}^{j-1} cb_i$;
- otherwise, cb_j remains in-core.

In the tree of Figure 3.7 processed in the order (d-a-b-c-e), the volume of *I/O* for the family defined by the parent (e) and the children (d) and (c) is equal to 3. According to what is written above, this implies that $cb_d = 2$ must be entirely written to disk, and that 1 unit of *I/O* must be performed on cb_c .

Property 3.5. *Because the analysis phase can forecast whether a contribution block (or part of it) will be written to disk, one can also decide to write it (or part of it) as soon as possible, that is, as soon as the contribution is produced. This will induce the same overall I/O volume.*

Thanks to Property 3.5, we will assume in the following that:

Strategy 1. *We decide to write all the contribution blocks which have to be written as soon as possible.*

This is illustrated in Figure 3.19(b): as soon as the contribution block of node (d) (cb_d) is produced, we know that it has to be written to disk and we can decide to write it as soon as possible, *i.e.*, before processing node (a). Therefore, we can free the memory for the contribution block of (d) before allocating the frontal matrix of (a) (by using synchronous *I/O*s or, more efficiently, with pipelined asynchronous *I/O*s).

Property 3.6. *Each time a contribution block has to be written, it is alone in memory: all the previous contribution blocks are already on disk.*

In other words, it is no longer required to write the bottom of a stack, as it was suggested in Property 3.1. A slightly stronger property is the following:

Property 3.7. *If a subtree requires some I/O, then at the moment of processing the first leaf of that subtree, the workarray W is empty.*

This is again because we should write the oldest contribution blocks first and those have been written as soon as possible. A corollary from the two previous properties is the following:

Property 3.8. *When we stack a contribution block on a non-empty stack, we will never write it. Otherwise, we would have written the rest of the stack first. In particular, if a given subtree can be processed in-core with a memory $S \leq M_0$, then at the moment of starting this subtree, the contiguous free block of our workarray W is necessarily at least as large as S .*

It follows that by relying on Strategy 1 a cyclic memory management is not needed anymore: a simple stack is enough for the *classical* and *last-in-place* assembly schemes, and a double stack is enough for the *max-in-place* assembly scheme. In the latter case, a double stack is required only for processing in-core subtrees since our *max-in-place* + **MinIO** heuristic switches to *last-in-place* for subtrees involving *I/O* (as explained in Section 3.4.5).

We illustrate this strategy on the *max-in-place* + **MinIO** variant of Section 3.4.5 (although it applies to all **MinIO** approaches). We assume that the analysis phase has identified in-core subtrees (processed with **MinMEM** + *max-in-place*) and out-of-core subtrees (processed with **MinIO** + *last-in-place*). We also assume that the contribution blocks that must be written to disk have been identified. The numerical factorization is then illustrated by Algorithm 3.2. It is a top-down recursive formulation, more natural in our context, which starts with the application of `Algo00C_rec()` on the root of the tree. A workarray W of size M_0 is used.


```

% W:  workarray of size M0
% n:  number of child subtrees of tree T
for j = 1 to n do
  if the subtree Tj rooted at child j can be processed in-core in W then
    % We know that the free contiguous block in W is large enough thanks to
    % Property 3.8
    Apply the max-in-place approach (see Section 3.5.1.2);
  else
    % Some I/O are necessary on this subtree, therefore W is empty (Property 3.7)
    % We do a recursive call to Algo00C_rec(), using all the available workspace
    Algo00C_rec(subtree Tj) ;
  Write cbj to disk or stack it (decision based on Property 3.4 and Strategy 1);
Allocate frontal matrix of the parent node; it can overlap with cbn;
for j = n downto 1 do
  Assemble cbj in the frontal matrix of the root of T (reading from disk vj units of data, possibly by
  panels);
Factorize the frontal matrix; except for the root node, this produces a contribution block;

```

Algorithm 3.2: Algo00C_rec(tree T).

3.5.2.3 Application to the flexible allocation scheme

Assuming this time that we have a *last-in-place* (for example) allocation scheme before the parent allocation, $V_{family}^{I/O}$ (see also Property 3.4 for the *terminal* allocation) can be expressed as:

$$V_{family}^{I/O} = \max \left(0, \max_{j=1, \boxed{p}} (A_j^{flex} + \sum_{k=1}^{j-1} cb_k), + \sum_{k=1}^{\boxed{p-1}} cb_k) - M_0 \right) \quad (3.26)$$

Again, oldest contributions blocks are written first and if $\sum_{i=1}^j cb_i < V_{family}^{I/O}$, this means that cb_j should be written to disk at least partially. We decide to do so as soon as possible. Therefore, Property 3.8 (and the previous ones) still hold. We now state two new properties, the second one being a direct consequence of the first one.

Property 3.9. *We consider a family involving some I/O, using either a classical or last-in-place assembly scheme. Before processing the first subtree of this family, we know that the memory is empty (Property 3.7). When the parent of the family is allocated, the contribution blocks of the children already processed are either on disk or available in the right stack (classical or last-in-place assembly schemes). In particular cb_p is available on the top of that stack.*

Property 3.10. *Considering a family involving I/O, once the contribution blocks from the children $j \leq p$ have been assembled into the frontal matrix of the parent, that frontal matrix is alone in memory.*

We now consider a child j processed after the parent allocation. Thanks to Section 3.4.7, we know that such a subtree can be processed in-core alone ($S_j^{flex} < M_0$). If $S_j + m > M_0$, part of the frontal matrix ($S_j + m - M_0$) has to be written to disk in order to make room for the subtree T_j rooted at child j , which is then processed in-core. With our previous assumptions, if $m + cb_j > M_0$, we might then have to write part of the contribution block to disk, read back the frontal matrix m and assemble cb_j into it panel by panel.

This leads to the top-down recursive formulation provided in Algorithm 3.3. The analysis phase has identified in-core subtrees (where **MinMEM** + *flexible* + *max-in-place* can be applied, as explained in Section 3.5.1.3). On out-of-core subtrees, we decide to use the combination (**MinIO** + *flexible* + *last-in-place*), see Section 3.4.5.

```

% W: workarray of size  $M_0$ 
% n: number of child subtrees of tree  $T$ 
% p: position of the parent allocation
% This algorithm is only called on subtrees that do not fit in memory
for  $j = 1$  to  $p$  do
    if the subtree  $T_j$  rooted at child  $j$  can be processed in-core in  $W$  then
        % We know that the free contiguous block in  $W$  is large enough thanks to
        % Property 3.8
        Apply the max-in-place flexible in-core approach (see Section 3.5.1.3)
    else
        % Some I/O are necessary on this subtree, therefore  $W$  is empty (Property 3.7)
        % We do a recursive call to Algo00C_flex_rec(), using all the available
        % workspace
        Algo00C_flex_rec(subtree  $T_j$ );
        Write  $cb_j$  to disk or stack it (decision based on Property 3.4 and Assumption 1, but using
        Formula (3.26)) at the right of  $W$ ;
% Thanks to Property 3.9:
Allocate the frontal matrix of the root of  $T$ , of size  $m$  (say), at the left of the workspace (in  $W(1 : m)$ );
it can overlap with  $cb_p$  because we decided to use a last-in-place scheme on out-of-core families;
for  $j = p$  downto 1 do
    Assemble  $cb_j$  in the frontal matrix of the root of  $T$  (reading from disk the part of  $cb_j$  previously
    written, if any, possibly by panels);
% The frontal matrix of the parent is now alone in memory (Property 3.10)
for  $j = p + 1$  to  $n$  do
    % We know that  $S_j \leq M_0$  thanks to Section 3.4.7
    if the subtree  $T_j$  rooted at child  $j$  cannot be processed in-core with its parent in  $W$  then
        Write an amount  $m + S_j - M_0$  units of the parent frontal matrix;
    % A free contiguous block of size  $S_j$  is now available in memory
    Apply the max-in-place flexible in-core approach to  $T_j$  (Section 3.5.1.3);
    % On output  $cb_j$  is available in memory
    Assemble  $cb_j$  into the frontal matrix of the root of  $T$  (temporary writing part of  $cb_j$  to disk, and
    reading the part of the parent frontal matrix previously written, if any);
Factorize the frontal matrix; this step produces a contribution block (except for the root) that we
stack on the right of  $W$ ;
Algorithm 3.3: Algo00C_flex_rec(tree  $T$ ), using a max-in-place (resp. last-in-place) before the parent
allocation for the in-core (resp. out-of-core) parts.

```

Note that only one stack, on the right of W , is manipulated in Algorithm 3.3, although more stacks are used temporarily on subtrees that can be processed in-core.

Table 3.4 sums up how to organize the data in the in-core case depending on the variant of the multifrontal method considered.

Allocation scheme	Assembly scheme	Data	
		Left stack	Right stack
<i>terminal</i>	<i>classical</i>	\emptyset	all CB's
	<i>last-in-place</i>	\emptyset	all CB's
	<i>max-in-place</i>	largest CB's	other CB's
<i>flexible</i>	<i>classical</i>	fronts	all CB's
	<i>last-in-place</i>	fronts	all CB's
	<i>max-in-place</i>	fronts + largest CB's	other CB's

Table 3.4: Summary of the in-core management of data (other than the current frontal matrix). *Front* is used for *frontal matrix* and *CB* is used for *contribution block*.

3.5.3 Limits of the models

In the out-of-core approaches relying on information from the analysis, we have considered multifrontal solvers without delayed pivots between children and parent nodes. In that case, the forecasted metrics from the analysis are exactly respected during the numerical factorization and the tree traversals obtained are optimal. In particular, Algorithms 3.2 and 3.3 can be applied and implemented as presented.

Let us now allow dynamic pivoting that results in delayed pivot eliminations between some children nodes to their parents or ancestors [80]. The size of the associated contribution blocks increases to include the delayed pivot rows/columns, leading to an increase of the quantities cb and m . Because such numerical difficulties can hardly be predicted but often remain limited in practice with proper preprocessing, it seems reasonable to us to keep the tree traversal obtained with the original metrics from the analysis. In the case of an in-core stack, the memory management algorithms from Section 3.5.1 can still be applied – including the memory management for our new *max-in-place* scheme presented in Section 3.5.1.2 – as long as the memory size is large enough.

In the context of an out-of-core stack, the approaches from Section 3.5.2.2 do not apply directly because the storage for a subtree may be larger than forecasted when numerical difficulties occur. Imagine for example that a subtree which was scheduled to be processed in-core no longer fits in memory because of delayed eliminations within the subtree. Alternative strategies to deal with those numerical difficulties are required, which are outside the scope of this paper. Recovering from a situation where the strategy has been too optimistic may require a significant amount of extra, unpredicted *I/O* and/or memory copies. A safer approach could consist in relaxing the forecasted metrics with a predefined percentage and artificially limit the amount of delayed eliminations to remain within that percentage. Finally, storing delayed rows/columns into separate data structures with a separate out-of-core management when necessary might be another option.

3.6 Concluding remarks

This work and the associated memory-management algorithms can be useful for large problems in limited-memory environments. They can be applied to shared-memory multifrontal solvers relying on threaded BLAS libraries. In a parallel distributed context, they can help limiting the memory requirements and decreasing the *I/O* volume in the serial parts of the computations. In parallel environments, memory usage also depends a lot on scheduling strategies, as will be seen in Chapter 4. In particular, we hope that the memory-aware algorithms of Section 4.3 will allow generalizations of some of the sequential mechanisms describe in this chapter.

Chapter 4

Task Scheduling in Parallel Distributed-Memory Environments

The way we adapted the multifrontal method to parallel distributed-memory environments using message passing and distributed dynamic schedulers was described in Section 2.1. Mapping and scheduling algorithms have a strong impact on the behaviour of our method and this has been the object of a lot of research and development work [24, 31, 33, 103, 101, 34]. We remind that in our framework, the nodes of the assembly tree are mapped statically, but the processor in charge of a node determines dynamically, at runtime, the set of processors that will participate in the associated factorization task. In practice, at least 80% of the work is mapped at runtime, and the mapping decisions are based on estimates of the load of the other processors (the load estimations will be further detailed in Section 4.1). To limit the dynamic choices of the schedulers and improve locality of communications, [33] suggested to limit the choice of *slave* processors for a given task to a set of *candidates* determined statically. Before that, any processor participating in the execution could be chosen to work on the subtasks.

This chapter is organized in two parts. In the first one (Section 4.1), we present different approaches to get accurate load and memory information on entry to the distributed scheduling algorithms, that have all been implemented and experimented. In the second part (Section 4.2), we then summarize the work done in the past concerning scheduling and mapping algorithms, both on the static and dynamic aspects. Those works have aimed at improving performance, reduce memory usage, or do both at the same time. Particular architectures (clusters of SMP's) have also been considered.

4.1 Obtaining accurate load estimates

In this section, we describe, study and compare different load exchange mechanisms and justify the choice for the one we decided to use by default.

Our parallel multifrontal algorithm can be considered as a set of dependent tasks executing on a distributed system of N processors that only communicate by message passing. From time to time, any processor P (called *master*) needs to send work to other processors. In order for such scheduling decisions to use accurate information, each processor facing a scheduling decision relies on a view of the workload and memory usage of the other processors. The workload and memory of a processor P vary: (i) when P processes some work (less work waiting to be done, temporary memory freed at the end of a task) or (ii) when a new task appears on processor P (that can come either from a new dependency met or from another processor). Whereas errors on workload estimates typically result in a longer execution time (or makespan), an error in the memory estimation can be more dramatic and lead to a failure if the processor chosen does not have enough memory to receive and process the assigned task. The algorithms presented in this section aim at providing state information from the processors that will be used during distributed dynamic scheduling decisions. To fix the ideas, the simplified model algorithm is Algorithm 0.1. Looking at

Algorithm 0.1, note that all messages discussed in this section are of type *state information*, and they are processed in priority compared to the other messages. In practice, a specific channel (or MPI communicator) is used for those messages.

We consider two classes of algorithms.

- The first class (discussed in Section 4.1.1) consists in maintaining the view of the load information during the computation: when quantities vary significantly, processes exchange information and maintain an approximate view of the load of the others.
- The second class of approaches (Section 4.1.2) is more similar to the distributed snapshot problem of [51] and is demand-driven: a process requiring information (to proceed to a scheduling decision) asks for that information to the others. Although less messages are involved, there is a stronger need for synchronization. In this section, we discuss possible algorithms for those two classes of approaches, and compare their impact on the behaviour of a distributed application using dynamic scheduling strategies.

4.1.1 Maintaining a distributed view of the load

In this approach, each process broadcasts information when its state changes. Thus, when a process has to take a dynamic decision (we called this type of dynamic decisions a *slave selection*), it already has a view of the state of the others. A condition to avoid a too incoherent view is to make sure that all pending messages related to load information are received before taking a decision implying to assign work to other processors. This is the case in the context of Algorithm 0.1.

4.1.1.1 Naive mechanism

Early versions of MUMPS used the mechanism described by Algorithm 4.1; each process P_i is responsible of knowing its own load; for each significant variation of the load, the absolute value of the load is sent to the other processes, and this allows them to maintain a global view of the load of the system. A threshold mechanism ensures that the amount of messages to exchange load information remains reasonable.

- 1: **Initialization**
- 2: $last_load_sent = 0$
- 3: Initialize(my_load)
- 4: **When my_load has just been modified:**
- 5: **if** $|my_load - last_load_sent| > threshold$ **then**
- 6: **send** (in a message of type *Update*, asynchronously) my_load to the other processes
- 7: $last_load_sent = my_load$
- 8: **end if**
- 9: **At the reception of load l_j from P_j (message of type *Update*):**
- 10: $load(P_j) = l_j$

Algorithm 4.1: Naive mechanism to exchange load information.

The local load l_i should be updated on the local process regularly, at least when work is received from another process, when a new local task becomes ready (case of dependent tasks), and when a significant amount of work has just been processed.

Limitations

Some problems can arise with the mechanism described above for the dynamic scheduling parts of our system. Indeed, with this mechanism, if several successive slave selections occur, there is nothing to ensure that a slave selection has taken into account the previous ones. Thus, a slave selection can be done based on invalid

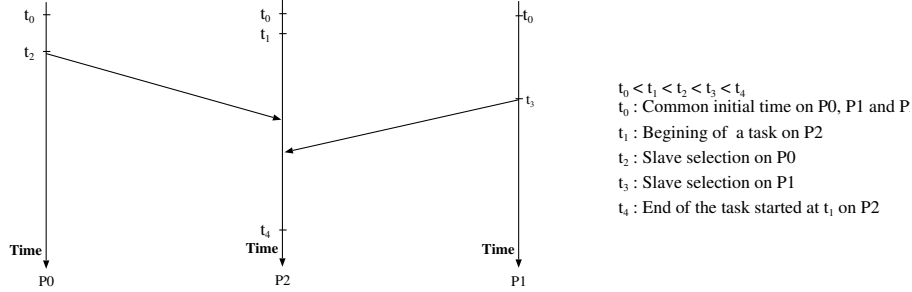


Figure 4.1: Example exhibiting the incorrectness of the naive mechanism.

information and this can lead to critical situations (in practice, large imbalance of the workload or critical increase of the memory).

Figure 4.1 gives an illustration of the problem. Suppose that P_2 has started a relatively costly task at time t_1 and is chosen as slave by P_0 at time t_2 . P_2 will not be able to receive the subtask from P_0 before the end of its own task. As a result, P_2 that does not know yet that it has been chosen by P_0 , cannot inform the others. P_1 , which is the second process that has to select slaves, will then select P_2 without taking into account the amount of work already sent by P_0 . This simple example exhibits the limitations of this approach.

4.1.1.2 Mechanism based on load increments

In this section we present another mechanism based on load increments to improve the correctness of the load information during the execution, and avoid situations like in Figure 4.1. Each time a process selects slaves, it sends (to all processes) a message of type *Master.To.All* containing the identity of the slaves and the amount of workload/memory assigned to each of them (it is a kind of reservation mechanism). At the reception of a message of this type, each process updates its local information on the processes concerned with the information contained in the message.

A formal description of the mechanism is given in Algorithm 4.2. For each variation of the workload on a process P_i , P_i broadcasts the increment representing the variation in a message of type *update*. Again, a threshold mechanism is applied to avoid too many messages: $\Delta load$ accumulates smaller $\delta load$ increments and is sent when larger than the threshold.

Note that when a (slave) process starts a task that was sent by another, it need not broadcast a message of type *Update* if the increment is positive: the master has already sent the information relative to its selected slaves (see line 7 in Algorithm 4.2).

4.1.1.3 Reducing the number of messages

To control the number of messages, the threshold should be chosen adequately. For example it is consistent to choose a threshold of the same order as the granularity of the tasks appearing in the slave selections. The number of messages will increase with the number of processes, since a message is broadcasted to all processes for each load variation in the system. However, some processes may never be master and never send work to others; this information may be known statically. Those processes do not need any knowledge of the workload/memory of the others. More generally, if at some point a process P_i knows that it will not proceed to any further slave selection in the future, it can inform the others. After P_i has performed its last slave selection, it can thus send a message of type *No.more.master* to the other processes (including to processes which are known not be master in the future). On reception of a message of type *No.more.master* from P_i by P_j , P_j stops sending load information to P_i . Note that the experiments presented later in this paper use this mechanism. Typically, we observed that the number of messages could be divided by 2 in the case of our test application, MUMPS.

```

1: Initialization:
2:  $my\_load = 0$ 
3:  $\Delta load = 0$ 
4:
5: When my load varies of  $\delta load$ :
6: if  $\delta load$  concerns a task where I am slave then
7:   if  $\delta load > 0$  return
8: end if
9:  $my\_load = my\_load + \delta load$ 
10:  $\Delta load = \Delta load + \delta load$ 
11: if  $\Delta load > threshold$  then
12:   send  $\Delta load$  (in a message of type Update, asynchronously) to the other processes
13:    $\Delta load = 0$ 
14: end if
15:
16: At the reception of load increment  $\Delta l_j$  from processor  $P_j$  (message of type Update):
17:  $load(P_j) = load(P_j) + \Delta l_j$ 
18:
19: At each slave selection on the master side:
20: for all  $P_j$  in the list of selected slaves do
21:   Include in a message of type Master_To_All the load  $\delta l_j$  assigned to  $P_j$ 
22: end for
23: send (asynchronously) the message Master_To_All to the other processes
24:
25: At the reception of a message of type Master_To_All:
26: for all  $(P_j, \delta l_j)$  in the message do
27:   if  $P_j \neq myself$  then
28:      $load(P_j) = load(P_j) + \delta l_j$ 
29:   else
30:      $my\_load = my\_load + \delta l_j$ 
31:   end if
32: end for

```

Algorithm 4.2: Mechanism based on load increments.

4.1.2 Exact algorithm

The second solution to this problem is close to the distributed snapshot approach [51, 111], coupled with a distributed leader election algorithm; the snapshot is demand-driven and initiated by the process that requires information from the others. This approach avoids the cost of maintaining the view during the computations, but loses some of the asynchronous properties of the application. Indeed, when a process requires information from the others, it has to wait for all others to be ready to send that information. Furthermore, since in our case the information is strongly linked to the dynamic scheduling decisions taken, two simultaneous snapshots should be “serialized” in order for the second one to take into account the slave selection resulting from the first one.

Each time a process has to take a dynamic decision (that uses and can modify the state of the others), it initiates a snapshot. After completion of the snapshot, it can take its dynamic decision, inform the others about its choice (message *master_to_slave* to the processes that have been selected as slaves) and finally restart the others. A more formal description of this scheme is given in Algorithm 4.3. Note that on reception of a message *master_to_slave*, a processor updates its state information (load) with the information contained in that message, so that the result of a first slave selection is taken into account if another snapshot is initiated from another process. Apart from that particular case, a processor is responsible for updating its own load information regularly.

- 1: Initiate a snapshot (see below)
- 2: Proceed to a dynamic slave selection
- 3: **for all** *islave* slave chosen **do**
- 4: Send a message of type *master_to_slave* to *islave* containing information to update its state (typically flops and memory corresponding to share of the work)
- 5: **end for**
- 6: Finalize the snapshot (see below)

Algorithm 4.3: Context in which the snapshot algorithm is applied.

The algorithm we use to build the snapshot of the system is similar to the one proposed by Chandy and Lamport [51]. In addition, since we are in a distributed asynchronous environment, several snapshots may be initiated simultaneously. They are in that case serialized to ensure that each process needing a snapshot takes into account the variation of the state (*i.e.*, workload, available memory, ...) of the processes chosen during the previous dynamic decision. For that, a distributed leader election [88, 161], for example based on process ranks, is performed. The process elected is the one that will complete its snapshot in priority. After termination of the snapshot of the leader, a new leader election is done within the set of processes having already initiated a snapshot. The algorithm is still based on message passing between the processes; a preliminary step consists in initializing the data structures that will be used during the execution to manage the snapshot mechanism:

- 1: **Initialization:**
- 2: *leader = undefined* % current leader
- 3: *nb_snp = 0* % number of concurrent snapshots except myself
- 4: *during_snp = false* % flag telling if I am trying to be the current leader
- 5: *snapshot = false* % flag telling if there is an active snapshot for which I am not leader
- 6: **for** *i = 1* **to** *nprocs* **do**
- 7: *request(P_i) = 0* % request identifier
- 8: *snp(P_i) = false* % array of flags telling if *P_i* has initiated a snapshot
- 9: *delayed_message(P_i) = false* % array of flags telling if I delayed the emission of a message to a processor
- 10: **end for**

The rest of the algorithm uses three types of messages: *start_snp*, *snp* and *end_snp*. When a process initiates a snapshot, it broadcasts a message of type *start_snp*. Then it waits for the information relative to the state of all the others. Note that if there are several snapshots initiated simultaneously, a “master”

(*i.e.*, process that initiates a snapshot) may have to broadcast a message of type *start_snp* several times with different request identifiers to be able to gather a correct view of the system, in the case where it was not the leader among the “master” processes.

```

1: Initiate a snapshot:
2: leader = myself
3: snp(myself) = true
4: during_snp = true
5: while snp(myself) == true do
6:   request(myself) = request(myself) + 1
7:   send asynchronously a message of type start_snp containing request(myself) to all others
8:   nb_msgs = 0
9:   while nb_msgs ≠ nprocs - 1 do
10:    receive and treat a message
11:    if during_snp == false then
12:      during_snp = true
13:      nb_msgs = 0
14:      break
15:    end if
16:  end while
17:  if nb_msgs == nprocs - 1 then
18:    snp(myself) = false
19:  end if
20: end while

```

After receiving the load information from all other processes, the process that initiated the snapshot can proceed to a scheduling decision (dynamic slave selection at line 2 of Algorithm 4.3), and update the load information resulting from that decision. After that (see the algorithm below), it informs the other processes that its snapshot is finished (message of type *end_snp*) and waits for other snapshots in the system to terminate.

```

1: Finalize the snapshot:
2: send asynchronously a message of type end_snp to all other processes
3: leader = undefined
4: if nb_snp ≠ 0 then
5:   snapshot = true
6:   for i=1 to nprocs do
7:     if snp(Pi) == true then
8:       leader = elect(Pi, leader)
9:     end if
10:  end for
11:  if delayed_message(leader) == true then
12:    send asynchronously my state and request(leader) to leader in a message of type snp
13:    delayed_message(leader) = false
14:  end if
15:  while nb_snp ≠ 0 do
16:    receive and treat a message
17:  end while
18: end if

```

When a process P_j receives a message of type *start_snp* from a process P_i (see the algorithm below), it can either ignore the message (if P_j is the current leader, see lines 8-11), either send a message of type *snp* that contains its state (lines 15 or 21), or delay the message to be sent in order to avoid a possible inconsistency in the snapshot. This last case can occur if P_j detects that P_i is not the leader (line 18) or because of asynchronism.

To give an example illustrating how asynchronism can be managed, consider a distributed system with three processes P_1 , P_2 , P_3 , where P_1 receives a message *start_snp* both from P_3 and P_2 , in that order. P_1 first answers to P_3 and then to P_2 which is the leader (we assume that the leader is the process with smallest rank). When P_2 completes its snapshot, suppose that P_3 receives *end_snp* from P_2 before P_1 . In addition, suppose that P_3 re-initiates a snapshot (sending a message of type *start_snp*) and that P_1 receives the *start_snp* message from P_3 before *end_snp* from P_2 arrives. Then P_1 will not answer to P_3 until it receives the message *end_snp* from P_2 . This ensures that the information sent from P_1 to P_3 will include the variation of the state information induced by the dynamic decision from P_2 . Such a situation may occur in case of heterogeneous links between the processes.

Note that the algorithm is recursive. After the first reception of a message of type *start_snp*, the process does not exit from the algorithm until all snapshots have terminated (lines 26-28 in the algorithm below). Note that the test at line 24 is there to avoid more than one level of recursion.

```

1: At the reception of a message start_snp from  $P_i$  with request number req:
2: leader = elect( $P_i$ , leader)
3: request( $P_i$ ) = req
4: if snp( $P_i$ ) == false then
5:   nb_snp = nb_snp + 1
6:   snp( $P_i$ ) = true
7: end if
8: if leader == myself then
9:   delayed_message( $P_i$ ) = true
10: return
11: end if
12: if snapshot == false then
13:   snapshot = true
14:   leader =  $P_i$ 
15:   send asynchronously my state and request( $P_i$ ) to  $P_i$  in a message of type snp
16: else
17:   if leader ≠  $P_i$  or delayed_message( $P_i$ ) == true then
18:     delayed_message( $P_i$ ) = true
19:     return
20:   else
21:     send asynchronously my state and request( $P_i$ ) to  $P_i$  in a message of type snp
22:   end if
23: end if
24: if nb_snp == 1 then {loop on receptions for the first start_snp message (if nb_snp is greater than 1, I
   am already waiting for the completion of all the snapshots)}
25:   during_snp = false
26:   while snapshot == true do
27:     receive and treat a message
28:   end while
29: end if

```

On the other hand, when a process receives a message of type *end_snp*, it checks if there is another active snapshot in the system (different from the sender of the message). If not, the receiving process exits and continues its execution. Otherwise, it sends its state information only to the process viewed as the leader (*leader*) of the remaining set of processes that have initiated a snapshot. It stays in snapshot mode (*snapshot* = *true*) until all ongoing snapshots have completed.

```

1: At the reception of a message of type end_snp from  $P_i$ :
2: leader = undefined
3: nb_snp = nb_snp - 1
4: snp( $P_i$ ) = false

```

```

5: if nb_snp == 0 then
6:   snapshot = false
7: else
8:   for i=1 to nprocs do
9:     if snp(Pi) == true then
10:      leader = elect(Pi, leader)
11:     end if
12:   end for
13:   if leader == myself then
14:     return
15:   end if
16:   if delayed_message(leader) == true then
17:     send asynchronously my state and request(leader) to leader in a message of type snp
18:     delayed_message(leader) = false
19:   end if
20: end if

```

Finally, when a “master” process receives a message of type *snp* from another one, it first checks that the request identifier contained in the message is equal to its own. In that case, it stores the state of the sender. Otherwise, the message is ignored since there is in that case no guarantee about the validity of the information received.

```

1: At the reception of a message of type snp from Pi with request id req:
2: if req == request(myself) then
3:   nb_msgs = nb_msgs + 1
4:   Extract the state/load information from the message and store the information for Pi
5: end if

```

4.1.3 Experiments

The mechanisms described in Sections 4.1.1.1, 4.1.1.2 and 4.1.2 have been implemented inside the MUMPS package. In fact, the mechanism from Section 4.1.1.1 used to be the one available in MUMPS, while the mechanism of Section 4.1.1.2 is the default one since MUMPS version 4.3. In order to study the impact of the proposed mechanisms, we experiment them on several problems (see Table 4.1) extracted from various sources including Tim Davis’s collection at University of Florida¹ or the PARASOL collection². The tests have been performed on an IBM SP system of IDRIS³ composed of several nodes of either 4 processors at 1.7 GHz or 32 processors at 1.3 GHz.

We have tested the algorithms presented in the previous sections (naive, based on increments and based on snapshot) on 32, 64 and 128 processors of the above platform. By default, we used the METIS package [120] to reorder the variables of the matrices. The results presented in the following sections have been obtained using two different strategies:

- a dynamic memory-based scheduling strategy, that will be later described in Section 4.2.5, and
- a dynamic workload-based scheduling strategy, similar to the one of Section 4.2.3, but with small improvements (irregular partitions of slave tasks, proportional mapping enforced on L_0 layer).

Using a memory-based strategy is motivated by the fact that a memory-based scheduling strategy is very sensitive to the correctness of the view. The workload-based dynamic scheduling strategy (also sensitive to the correctness of the view) will be used to also illustrate the cost and impact of each mechanism in terms of time.

¹<http://www.cise.ufl.edu/~davis/sparse/>

²<http://www.parallab.uib.no/parasol>

³Institut du Développement et des Ressources en Informatique Scientifique

Small test problems				
Matrix	Order	NZ	Type	Description
BMWCR_A1 (PARASOL)	148770	5396386	SYM	Automotive crankshaft model
GUPTA3 (Tim Davis)	16783	4670105	SYM	Linear programming matrix (A*A')
MSDOOR (PARASOL)	415863	10328399	SYM	Medium size door
SHIP_003 (PARASOL)	121728	4103881	SYM	Ship structure
PRE2 (Tim Davis)	659033	5959282	UNS	AT&T,harmonic balance method
TWOTONE (Tim Davis)	120750	1224224	UNS	AT&T,harmonic balance method.
ULTRASOUND3	185193	11390625	UNS	Propagation of 3D ultrasound waves generated by X. Cai (Simula Research Laboratory, Norway) using Diffpack.
XENON2 (Tim Davis)	157464	3866688	UNS	Complex zeolite,sodalite crystals.
Large test problems				
Matrix	Order	NZ	Type	Description
AUDIKW_1 (PARASOL)	943695	39297771	SYM	Automotive crankshaft model
CONV3D64	836550	12548250	UNS	provided by CEA-CESTA; generated using AQUILON (http://www.enscpb.fr/master/aquilon)
ULTRASOUND80	531441	330761161	UNS	Propagation of 3D ultrasound waves, provided by M. Sosonkina, larger than ULTRASOUND3

Table 4.1: Test problems.

For the memory-based strategy, we measure the memory peak observed on the most memory consuming process. The tests using memory-based scheduling have been made on 32 and 64 processors which are enough for our study. For the workload-based scheduling strategy, we measure the time to factorize the matrix on the largest test problems on 64 and 128 processors. Each set of results (test problem/number of processors) is performed on the same configuration of computational nodes. However, when going from one test problem to another, the configuration can change: because of the characteristics of the machine, 64 processors can either be 16 nodes of quadri-processors, either 2 nodes of 32 processors, or some intermediate configuration, including cases where some processors are not used in some nodes. Therefore, results presented in this section should not be used to get a precise idea of speed-ups between 64 and 128 processors. Finally, note that the number of dynamic decisions for the set of small test problems (see Table 4.1) is comprised between 8 and 92 on 32 processors, and between 8 and 152 on 64 processors. For the set of larger test problems, the number of dynamic decisions is between 119 and 169 on 64 processors and between 199 and 274 on 128 processors.

4.1.3.1 Memory-based scheduling strategy

In Table 4.2, we give the peak of active memory (maximum value over the processors) required to achieve the factorization. We compare the influence of the naive mechanism introduced in Section 4.1.1.1, of the mechanism based on increments introduced in Section 4.1.1.2, and of the algorithm presented in Section 4.1.2, on the dynamic memory-based scheduler (see Section 4.2.5).

On 32 processors (Table 4.2(a)), we observe that the peak of memory is generally larger for the naive mechanism than for the others. This is principally due to the limitation discussed in Section 4.1.1.1 for that mechanism: some dynamic scheduling decisions are taken by the schedulers with a view that does not include the variations of the memory occupation caused by the previous decisions. In addition, we observe that the algorithm based on distributed snapshots (Section 4.1.2) gives in most cases the best memory occupation and that the mechanism based on increments is not too far behind. For the GUPTA3 matrix, the algorithm based on snapshots provides the worst memory peak. In that case, we observed that there is a side effect of doing snapshots on the schedule of the application. The asynchronous and non-deterministic nature of the application explain such possible exceptions to the more important general tendency.

On 64 processors, we can observe a similar behaviour: the naive mechanism gives in most cases worse results than the other mechanisms. For the largest problems in this set (*e.g.*, matrix ULTRASOUND3), the algorithm based on snapshots gives the best results, followed by the mechanism based on increments and finally the naive mechanism.

The results of this section illustrate that when we are interested in a metric that has great variations (such as the memory), the algorithm based on snapshots is well-adapted, although costly. (We will discuss this in the next section.) We also see that in terms of quality of the information, the mechanism based on

	Increments based	Snapshot based	naive
BMWCRA_1	3.71	3.71	3.71
GUPTA3	3.88	4.35	3.88
MSDOOR	1.51	1.51	1.51
SHIP_003	5.52	5.52	5.52
PRE2	7.88	7.83	8.04
TWOTONE	1.94	1.89	1.99
ULTRASOUND3	7.17	6.02	10.69
XENON2	2.83	2.86	2.93

(a) 32 processors.

	Increments based	Snapshot based	naive
BMWCRA_1	2.30	2.30	3.55
GUPTA3	2.70	2.70	2.70
MSDOOR	1.01	0.84	0.84
SHIP_003	2.19	2.19	2.19
PRE2	7.66	7.87	7.72
TWOTONE	1.86	1.86	1.88
ULTRASOUND3	3.59	3.40	5.24
XENON2	2.45	2.41	3.61

(b) 64 processors.

Table 4.2: Peak of active memory (millions of real entries) on 32 and 64 processors as a function of the exchange mechanism.

increments is never far from the one based on snapshots.

4.1.3.2 Workload-based scheduling strategy

	Increments based	Snapshot based
AUDIkw_1	94.74	141.62
CONV3D64	381.27	688.39
ULTRASOUND80	48.69	85.68

(a) 64 processors.

	Increments based	Snapshot based
AUDIkw_1	53.51	87.70
CONV3D64	178.88	315.63
ULTRASOUND80	35.12	66.53

(b) 128 processors.

Table 4.3: Time for execution (seconds) on 64 and 128 processors as a function of the exchange mechanism applied.

We compare in Table 4.3 the factorization time from MUMPS with a workload-based scheduling strategy (see Section 4.2.3) when using the algorithm based on snapshots and the one based on increments. We can observe that the mechanism based on snapshots is less efficient than the one based on increments. This is principally due to the fact that the snapshot operation requires a strong synchronization that can be very costly in terms of time. In addition, when there are several dynamic decisions that are initiated simultaneously, those are serialized to ensure the correctness of the view of the system on each processor. Thus, this can increase the duration of the snapshots. Finally, the synchronization of the processors may have unneeded effects on the behaviour of the whole system. For example, if we consider the CONV3D64 matrix on 128 processors, the total time spent to perform all the snapshot operations is of 100 seconds. In addition, there were at most 5 snapshots initiated simultaneously. This illustrates the cost of the algorithm based on snapshots especially when the processors cannot compute and communicate simultaneously. (A long task involving no communication will delay all the other processes.) Furthermore, we remark that if we measure the time spent outside the snapshots for CONV3D64, we obtain $315.63 - 100.00 = 215$ seconds, which is larger than the 178.88 seconds obtained with the increments-based mechanism (see Table 4.3(b)). The reason is that after a snapshot, all processors restart their computation and data exchanges simultaneously.

The data exchanges can saturate the network. Another aspect could be the side-effect of the leader election on the global behaviour of the distributed system, where the sequence of dynamic decisions imposed by the criterion for the leader election (smallest processor rank in our case) has no reason to be good strategy compared to the natural one. Finding a better strategy is a scheduling issue and is out-of-scope in this study.

	Increments based	Snapshot based
AUDIkw_1	302715	11388
CONV3D64	386196	16471
ULTRASOUND80	208024	12400

(a) 64 processors.

	Increments based	Snapshot based
AUDIkw_1	1386165	39832
CONV3D64	1401373	57089
ULTRASOUND80	746731	50324

(b) 128 processors.

Table 4.4: Total number of messages related to the load exchange mechanisms on 64 and 128 processors.

Concerning the number of messages exchanged during the factorization, the results are given in Table 4.4. Note that the size of each message is larger for the snapshot-based algorithm since we can send all the metrics required (workload, available memory, . . .) in a single message. On the other hand, for the increments based mechanism, we send a message for each sufficient variation of a metric. We can observe that the algorithm based on snapshots uses less messages than the mechanism based on increments that tries to maintain a view of the system on each process. The communication cost of these messages had no impact on our factorization time measurement since we used a very “high bandwidth/low latency” network. For machines with high latency networks, the cost of the mechanism based on increments could become large and have a bad impact on performance. In addition, the scalability of such an approach may become a problem if we consider systems with a large number of computational nodes (more than 1000 processors for example).

To study the behaviour of the snapshot mechanism in a system where processors can compute and communicate at the same time, we slightly modified our solver by adding an extra thread that periodically checks for messages related to snapshots and/or load information. The algorithm executed by this second thread is given below:

```

1: while not end_of_execution do
2:   sleep(period)
3:   while there are messages to be received do
4:     receive a message
5:     if the received message is of type start_snp then
6:       block the other thread (if not already done)
7:     end if
8:     treat the received message
9:     if the received message is of type end_snp and there is no other ongoing snapshot then
10:      restart the other thread
11:    end if
12:  end while
13: end while

```

It is based on POSIX threads and only manages messages corresponding to state information, excluding the ones related to the application, which use another channel. Also, we fixed the sleep period experimentally to 50 microseconds. Furthermore, since our application is based on MPI [159], we have to ensure that there is only one thread at a time calling MPI functions using locks⁴. Finally, the interaction between the two threads can be either based on signals or locks. One way to block the other thread is to send a special signal to block it. Another way, which is the one used here, is to simply get the lock that protects the MPI calls

⁴Thread-safe implementations of MPI were not so common at the time this work was performed.

and to release it only at the end of the snapshot.

	Increments based	Snapshot based
AUDIKW_1	79.54	114.96
CONV3D64	367.28	432.71
ULTRASOUND80	49.56	69.60

(a) 64 processors.

	Increments based	Snapshot based
AUDIKW_1	41.00	59.19
CONV3D64	189.47	237.69
ULTRASOUND80	35.91	52.00

(b) 128 processors.

Table 4.5: Impact of the threaded load exchange mechanisms on the factorization time (seconds) on 64 and 128 processors.

We tested this threaded version of the application on 64 and 128 processors. The results are given in Table 4.5. Note that we also measured the execution time for the threaded increments mechanism with the intention to evaluate the cost of the thread management. We observe that using a thread has a benefic effect on the performance in most cases for the mechanism using increments (compare the left columns of Tables 4.3 and 4.5). We believe that this is because the additional thread treats the messages more often and thus avoids to saturate the internal communication buffers of the communication library (and from the application). Concerning the algorithm based on snapshots, the execution time is greatly reduced compared to the single-threaded version, thus illustrating the fact that processors spend less time performing the snapshot. For example if we consider the CONV3D64 problem on 128 processors, the total time spent to perform all the snapshot operations has decreased from 100 seconds to 14 seconds. However, we can observe that this threaded version of the snapshot algorithm is still less efficient than the one based on increments. This is principally due to the stronger synchronization points induced by the construction of a snapshot (even in the threaded version), as well as the possible contention when all processors restart their other communications (not related to state/snapshot information).

4.1.4 Concluding remarks

We have studied different mechanisms aiming at obtaining a view as coherent and exact as possible of the load/state information of a distributed asynchronous system under the message passing environment. We distinguished between two approaches to achieve this goal: maintaining an approximate view during the execution, and building a correct distributed snapshot.

We have shown that periodically broadcasting messages that update the view of the load/state of the other processes, with some threshold constraints and some optimization in the number of messages, could provide a good solution to the problem, but that this solution requires the exchange of a large number of messages. On the other hand, the demand-driven approach based on distributed snapshot algorithms provides more accurate information, but is also much more complex to implement in the context of our type of asynchronous applications: we had to implement a distributed leader election followed by a distributed snapshot; also, we had to use a dedicated thread (and mutexes to protect all MPI calls) in order to increase reactivity. In addition, this solution appears to be costly in terms of execution time and might not be well-adapted for high-performance distributed asynchronous applications. It can however represent a good solution in the case of applications where the main concern is not execution time but another metric to which the schedulers are very sensitive (*e.g.*, the memory usage). We also observed that this approach significantly reduces the number of messages exchanged between the processes in comparison to the first one; it might be well adapted for distributed systems where the links between the computational nodes have high latency/low bandwidth.

4.2 Hybrid static-dynamic mapping and scheduling strategies

As introduced at the beginning of this chapter, we now describe some of the work that has been carried out to define, map and schedule the tasks arising in our asynchronous parallel multifrontal framework. Several degrees of freedom allow for various possible strategies to take scheduling decisions. As discussed earlier, some of the decisions are static, and other decisions are dynamic, taken at runtime. The dynamic decisions use the workload and/or the memory information built thanks to the mechanisms described in Section 4.1.

The main degrees of freedom concerning **dynamic decisions** taken at runtime are:

task selection: the choice of the next local task to process, in case several tasks are available in the local pool of tasks.

slave selection: if the task is parallel (case of a type 2 node), the choice of the processes to help (*slave processes*). The order of the slave processes is also a degree of flexibility, together with the possibility of defining subtasks of equal or varying sizes. In some approaches (see below and Section 4.2.3), the slave processes must be chosen among a set of *candidate* processes chosen statically.

The dynamic decisions also depend on static mapping choices. The main objectives of the static mapping phase are to control the communication costs, and to balance the memory consumption and computation done by each processor. In our approach, those static choices concern the static **tree mapping**, which can be decomposed into:

- a partial mapping of the tasks in the tree, together with the decision to use several MPI processes or a single one for each node in the tree. More precisely (see Section 2.1.1), the mapping of a type 1 node is static whereas for type 2 nodes, only the mapping of the fully summed part (or *master* part) is static.
- for each type 2 node, a possible list of candidate processes that are allowed to work on that node, so that the dynamic slave selection algorithm is then restricted to selecting slaves among the candidates for that type 2 node, instead of all processors.

In the next subsections, we summarize previous work and explain what has been done with respect to the above degrees of freedom. When relevant, we also explain what type of load information is used.

4.2.1 History – PARASOL project (1996-1999)

The MUMPS project implements the parallel multifrontal approach discussed in this thesis. It started in 1996 with the PARASOL project which was an Esprit IV Long Term Research (LTR) European project. MUMPS was inspired by an experimental prototype of an unsymmetric multifrontal code for distributed-memory machines using PVM [85]. That experimental prototype was itself inspired by the code MUPS, developed by Amestoy during his PhD thesis [17] at CERFACS under the supervision of Duff. The first version of MUMPS that uses three levels of parallelism was MUMPS 2.0 in February 1998; it is the result of two years of intensive developments to introduce type 2 and type 3 parallelism in the multifrontal method, with the asynchronous approach to parallelism described in Chapter 2. At that time, no use of load estimates was made and the mapping of slave processes was static: assuming that a master process for a node in the tree had rank i in the MPI communicator, slave processes were $i + 1$, $i + 2$, ... (modulo the number of available processes). The number of *slaves* was chosen in such a way that the work per slave process is always smaller or equal to the work of the master, with the limit that the number of slaves should not exceed the number of available processes.

The use of load information appeared in version 2.1.3 of MUMPS (released to PARASOL partners in July 1998); this is the version that was used for the results presented in the reference article [27] and it is also the first version including a solver for symmetric problems. Once the number of slaves for a given type 2 node is known – at that time this depended mainly on the structure of the frontal matrix – the process in charge of the type 2 node (master) selects at runtime the processes with smallest estimated load. Concerning load exchange mechanisms, each MPI process is responsible of updating and communicating its load from time to time: this corresponds to the naive approach described in Section 4.1.1.1.

Several successive versions internal to the PARASOL project followed, and various improvements and new numerical functionalities were included. This led to the public version of MUMPS available at the end of the PARASOL project (version 4.0.4, September 1999), corresponding to the article published as [24], and also described in [27]. The main characteristics of the scheduling and mapping heuristics at the end of the PARASOL project are the following:

Static tree mapping. A partial mapping of the assembly tree to the processors is performed statically at the end of the analysis phase. A layer called L_0 is obtained by using an algorithm inspired from the one from Geist-Ng [90]; see Algorithm 4.4. In order to map the nodes of layer L_0 , an LPT (longest processing time first) list-scheduling algorithm is used; the subtrees are mapped in decreasing order of their workload, a subtree is packed in the bin (*i.e.*, is mapped on the processor) with the smallest workload. The workload of the processor is updated, and the process is repeated, so that the next subtree is mapped on the processor with smallest workload, until the load balance is accepted. The LPT algorithm achieves a theoretical time that is smaller than $\frac{4}{3} - \frac{1}{3p}$ times the optimal time, where p is the number of processors [98]. In early versions of MUMPS, this algorithm was indeed cheaper (and more naive): a processor was considered as a bin (binpacking) and a bin was filled with unsorted subtrees from L_0 until reaching but not exceeding the average expected cost per bin. Then the remaining subtrees were sorted and mapped to the processors using the LPT algorithm. In the algorithm, the computational cost is approximated by the number of floating-point operations.

$L_0 \leftarrow$ roots of the assembly tree

repeat

Find the node \mathcal{N} in L_0 with largest computational cost in subtree

$L_0 \leftarrow L_0 \cup \{\text{children of } \mathcal{N}\} \setminus \{\mathcal{N}\}$

Map L_0 subtrees onto the processors

Estimate load unbalance

until load unbalance < threshold

Algorithm 4.4: Principle of the Geist-Ng algorithm to define a layer L_0 .

Once layer L_0 is defined, we consider that the tree is roughly processed from bottom to top, layer by layer (see Figure 4.2). Layer L_0 is determined using Algorithm 4.4, illustrated in Figure 4.3. Then for $i > 0$, a node belongs to L_i if all its children belong to L_j , $j < i - 1$. First, nodes of layer L_0 (and associated subtrees) are mapped. This first step is designed to balance the work in the subtrees and to reduce communication since all nodes in a subtree are mapped onto the same processor. Normally, in order to get a good load balance, it is necessary to have many more nodes in layer L_0 than there are processors. Thus L_0 depends on the number of processors and a higher number of processors will lead to smaller subtrees.

The mapping of higher layers in the tree only takes into account memory balancing issues. At this stage, only the volume of factors is taken into account when balancing the memory used by the processors. For each processor, the memory load (total size of its factors) is first computed for the nodes at layer L_0 . For each layer L_i , $i > 0$, each unmapped node of L_i is mapped onto the processor with the smallest memory load and its memory load is revised.

The mapping is then used to explicitly distribute the permuted initial matrix onto the processors and to estimate the amount of work and memory required on each processor.

Finally, above layer L_0 , nodes larger than a certain threshold are defined statically to be of type 2 (parallel nodes), except the root node, which may be of type 3 (that is, it relies on a 2D block synchronous parallelization, see Section 2.1.1.2), if it is large enough. More precisely, a non-root node above L_0 is of type 2 if the size of its contribution block is larger than a given threshold. In an intermediate version, the threshold was based on the amount of work to be done by the slaves but nodes having a small fully-summed block and a huge contribution block were causing memory problems if they were not parallelized; hence this simpler criterion.

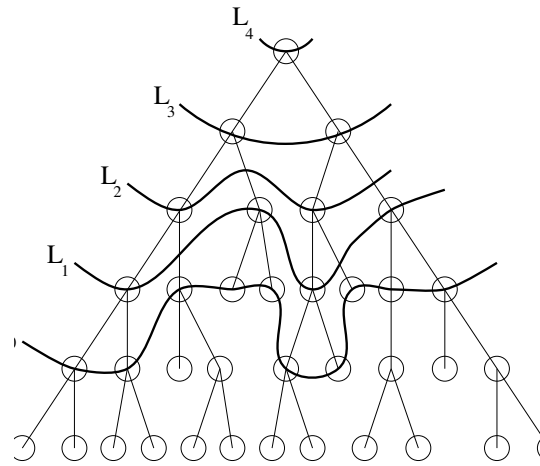


Figure 4.2: Decomposition of the assembly tree into levels.

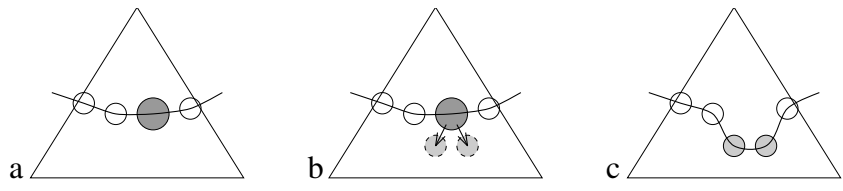


Figure 4.3: One step in the construction of the first level L_0 .

Only the masters of type 2 nodes are mapped, the rest of the node is split into several tasks without any pre-assigned mapping, so that at runtime, the master of a type 2 node will assign the slave tasks to any MPI process without any static constraint.

Dynamic task selection. The pool of tasks is managed as a stack, giving priority to the postorder. Although this limits the amount of parallelism and prevents the execution from following the layer-by-layer approach, experiments with a FIFO (first-in first-out) strategy showed huge limitations with respect to memory usage: FIFO would increase both parallelism and memory usage.

Dynamic slave selection. In this version, the number of slave processes is static and only depends on the structure of the front; however the choice of the identifiers for the slave processes is done depending on the estimate of the workload of the other processes: work is given to the less loaded processes. All slave processes roughly receive the same amount of work.

The main characteristics of this version are summarized in the left part of Figure 4.4. We refer the reader to the articles [27, 25, 24] for more information together with some performance results.

4.2.2 Improvements (1999-2001)

After the PARASOL project, MUMPS was ported to a CRAY T3E architecture in the context of a strong collaboration with NERSC/Lawrence Berkeley National Laboratory (France-Berkeley Fund project followed by sabbatical of Amestoy in Berkeley), in particular with X. S. Li, author of SuperLU_Dist [124], a fan-out (right-looking) [112] parallel sparse direct solver for unsymmetric matrices. Although the main principles of scheduling in MUMPS are the same as in the previous section, a lot of parameter tuning (task granularity, etc.) was performed in order to run on significantly larger numbers of processes (up to 512). A nice characteristic of the Cray T3E was the high speed of its network, in comparison to the processor speed. Still, it was observed that performance results depended a lot on MPI implementations and MPI parameters in both MUMPS and SuperLU and this motivated a deeper study of the so called *eager* versus *long* MPI protocols, which resulted in an implementation using asynchronous receives (MPI_IRecv) on top of the already existing asynchronous sends (MPI_Isend), see [29] for more information. Thanks to immediate MPI communication primitives, performance became much more independent from the underlying MPI implementation. Thus, the collaboration between the French and Berkeley teams was the opportunity to confront the behaviour of both solvers, improve them and perform a fair comparison between the approaches [28]. The resulting version roughly corresponds to MUMPS 4.1.6 (so called “TOMS” version), and it had a large success among users. It was also used for comparisons with a commercial package in [108]. Among improvements done at that time regarding dynamic scheduling decisions, the choice of the number of slave processors for a type 2 node is now dynamic and mixes several objectives: (i) try to give work only to processors less loaded than the master; (ii) allow more processes if this would result in tasks with a too large granularity, which would increase significantly memory usage or exceed the size of the preallocated communication buffers; (iii) allow for less processors if this would result in too much communication and too small granularities; in particular, avoid creating slave tasks involving significantly less work than the master task.

4.2.3 Static tree mapping: candidate processors

The text of this section is inspired by [33]. There are two problems with the previous approach:

1. First, memory is overestimated. In the previous implementation, the amount of memory needed for each processor is estimated during the analysis phase and is reserved as workspace for the factorization. Consequently, if every processor can possibly be taken as a slave of any type 2 node, then enough workspace has to be reserved, on each processor, for the potential corresponding computational task. This can lead to a dramatic overestimate of memory requirements because, during the factorization, typically not all processors are actually used as slaves.

2. Second, the choice of slaves is completely local and does not take into account locality of communication. The choice depends crucially on the instant when the master chooses the slaves and does not take into account scheduling decisions that are close in time and can have conflicting local objectives. More global information is needed to improve the quality of the scheduling decisions.

The concept of *candidate processors* originates in an algorithm presented in [142, 145] and has also been used in the concept of static task scheduling for Cholesky factorization [113]. With this concept it is possible to guide the dynamic task scheduling and to address the above issues. For each node that requires slaves to be chosen dynamically during the factorization, a limited set of processors is introduced from which the slaves can be selected. This allows to exclude all non candidates from the estimation of workspace during the analysis phase and leads to a tighter and more realistic estimation of the workspace needed. Furthermore, the list of candidates can be built using a more global view of the tasks graph.

4.2.3.1 Preamble: proportional mapping

In the proportional mapping [142], the assembly tree is processed from top to bottom, starting with the root node. For each child of the root node, the work associated with the factorization of all nodes in its subtree is first computed, and the available processors are distributed among the subtrees according to their weights. Each subtree thus gets its set of *preferential* processors. The same mapping is now repeated recursively: the processors that have been previously assigned to a node are again distributed among the children according to their weights (as given by the computational costs of their subtree). The recursive partitioning stops once a subtree has only one processor assigned to it. The proportional mapping both achieves locality of communication (similar to the subtree-to-subcube mapping, see [130]) and guides the partitioning from a *global* point of view, taking into account the weight of the subtrees even for irregular trees.

4.2.3.2 Main ideas of the mapping algorithm with candidates

In the retained approach, described in [33], a variant of the Geist-Ng algorithm is still applied, but an extra condition to accept a layer L_0 is that the memory must also be balanced.

Then, whereas only the masters of the nodes in the higher parts of the tree were mapped statically in the previous approach, each node now also receives a list of candidate processors defined statically. First, preferential processors are defined thanks to the proportional mapping algorithm. A relaxation is done to add flexibility: if at a node \mathcal{N} with two child subtrees, a strict proportional mapping gives 30% processors of the nodes allocated to \mathcal{N} on the left branch and 70% in the right branch, we use $30\% \times (1 + \rho)$ (resp. $70\% \times (1 + \rho)$) preferential processors in the left (resp. right) branch, where ρ is a relaxation parameter. Given the list of preferential processors, the tree is processed from bottom to top, layer by layer. At each layer, the types of the nodes of the layer are determined (type 1 or type 2) and each node receives its list of preferential processors as determined by the proportional mapping. If needed, node splitting (see Figure 2.11) or node amalgamation (see the end of Section 1.1.6) can occur. In case there are large nodes rooting small subtrees, there is an optional algorithm to redistribute candidates within the layer so that each node of the layer is assigned a number of candidates proportional to the cost of the node. In that case, a variant of a list scheduling algorithm is used, giving priority to preferential processors and only allowing non-preferential processors as candidates when all preferential processors have been chosen. However, best results were obtained when respecting the preferential processors, and only allowing extra processors when this would break granularity constraints. Finally, a top-down pass on the whole tree is made to exchange master and candidate processors of type 2 nodes: an exchange is done if this improves memory balance. At runtime, slaves are then chosen among the candidates, the least loaded first. The candidate processors less loaded than the master are chosen in priority, and the less loaded processors are ordered first in the list of chosen slaves. More information can be obtained in [33]; in particular, it is shown that the new scheduling algorithm behaves significantly better than the fully dynamic approach in terms of

- factorization time,
- accuracy of the memory estimates, and

- volume of communication.

4.2.4 Scheduling for clusters of SMP nodes

Starting from [33], Pralet et al. [31] consider the adaptation of the mapping and scheduling strategies in order to take into account architectures composed of clusters of SMP nodes. On such architectures, communication inside a node is faster than outside the nodes and this impacts the performance of our solver. For example, on an SP3 from CINES⁵, using 16 Power3+ processors within a single node is about 25% more efficient than using 8 processors from 1 node and 8 processors from the second node.

In order to take into account the architecture, both the dynamic slave selection and the static choice of candidate processors were modified.

Dynamic slave selection

As before, when a master processor of a type 2 node performs a slave selection, the candidate processors are sorted in increasing order of their load and the least loaded processors are chosen in priority. In order to take into account the SMP architecture, a penalty is now given to the load before the sort. Typically, the load of a processor is multiplied by a factor λ if it is outside the SMP node of the master processor (this simple algorithm was shown to behave better than a model of the network taking into account latency and bandwidth). As a result, less processors outside the SMP node are chosen and this leads to better performance. As a side effect, since the loads are larger, the average number of processors participating to each type 2 node is reduced, limiting communication but also limiting the amount of parallelism, which can be critical in some cases. In order to obtain further gains, the modified dynamic scheduling algorithm is thus combined to improvements of the candidate-based algorithm using relaxed proportional mapping, as explained below.

Static tree mapping

Although [31] describe the approach in a very general case, for simplicity, we assume here that each SMP node contains the same number of processors. First, the preferential processors are chosen cyclically in the list of processors not according to their MPI rank, but rather according to their position in the list of processors, where processors belonging to the same SMP node are numbered consecutively. Then, during the layer-by-layer bottom-up process, an *architecture criterion* is used: typically, if several SMP nodes are in the list of candidates, the master –because it communicates the most with the slaves– must be on the SMP node that appears most often in the list of candidates.

Large benefits from these approaches are obtained on IBM clusters of SMP's with 16 Power3+ or Power4 processors. In [31], the authors also experiment mixing MPI parallelism with parallelism based on a threaded BLAS library, and show that using 2 or 4 threads per MPI process is a good compromise. We discuss again such hybrid MPI/thread parallel issues in Section 6.6, in the context of multicore architectures.

4.2.5 Memory-based dynamic scheduling

For large problems, memory usage may be more critical than time of execution: if a problem is too large, memory allocation may fail and it is then necessary to reduce the memory footprint in order to get a chance of solving the problem. In [103], we discuss memory-based dynamic scheduling strategies for the parallel multifrontal method as implemented in MUMPS. These strategies are a combination of a memory-based slave selection strategy and a memory-aware task selection strategy.

⁵Centre Informatique National de l'enseignement supérieur, Montpellier, France.

Task selection (pool management)	stack (LIFO)	→	stack in general, with exceptions in case of memory problem
Number of slave processors	static	→	dynamic
Slave partitions	same amount of work to each slave	→	irregular partitions
Static mapping of masters	mainly memory-based	→	more complex
Static choice of candidates	no constraint (all MPI processes)	→	determined statically
Load information	flops of ready tasks	→	flops and memory information, various other information (<i>e.g.</i> , near-to-be-ready tasks)
Load mechanism	naive mechanism	→	increment-based (from Section 4.2.5 onwards)

Figure 4.4: Main evolutions of scheduling algorithm characteristics between Sections 4.2.1 (left) and Section 4.2.6 (right).

Dynamic slave selection: The slave processors are selected with the goal to obtain the best memory balance, and we use an irregular 1D-blocking by rows for both symmetric and unsymmetric matrices. In fact, the slave selection strategy attempts at choosing the minimum number of slaves that will balance the current memory usage without increasing the local memory peak.

Dynamic task selection: Concerning the task selection strategy, we have adapted the task selection to take into account memory constraints. Whereas the pool of tasks is always managed as a stack (last task inserted is the first extracted) in all previous approaches, we try to better take into account memory. In particular, a task in a sequential subtree should in some cases be processed even if a new task higher in the tree has appeared: this is because it may be better to finish the subtree and its local peak of memory before starting new activities at a different place in the tree. Furthermore, if the task at the top of the pool is large in comparison to the peak of memory already observed, we attempt to extract other tasks. More precisely, the task selection algorithm is given by Algorithm 4.5.

```

if the node at the top of the pool is part of a sequential subtree then
    return the node at the top of the pool;
else
    for  $\mathcal{N}$  in the pool of ready tasks, starting from the top do
        if  $\text{memory\_cost}(\mathcal{N}) + \text{current memory} \leq \text{memory peak observed since the beginning of the factorization}$  then
            return  $\mathcal{N}$ ;
        else
            if  $\mathcal{N}$  belongs to a sequential subtree then
                return  $\mathcal{N}$ ;
            end if
        end if
    end for
    return the node at the top of the pool;
end if

```

Algorithm 4.5: Memory-aware task selection algorithm.

Both the task selection strategy and the slave selection strategy require memory information to circulate between the processors, so that each processor now has a view of the memory load of other processors,

not just the workload (see Section 4.1). Whereas the workload allows approximations, inaccurate memory information can be dramatic in the sense that this may lead to a failure of the execution. In order to avoid such situations, the load information mechanism was modified to forecast the arrival of large new ready tasks: before a new task is ready (last child in the process of factorizing its frontal matrix), a message is sent so that the memory for the master part of the frontal matrix of the parent is included in the memory estimate of the ready tasks of the master processor.

Overall, this scheduling strategy allows a significant reduction of the memory usage, but at the cost of a performance penalty (see [103]).

4.2.6 Hybrid scheduling

[34] pushes the previous ideas further. By reconsidering both the static and dynamic aspects of scheduling, its objectives are to

- preserve the memory behaviour obtained in Section 4.2.5,
- further decrease the difference between memory estimates and actual memory consumption,
- accelerate the factorization compared to previous approaches.

In order to reach these objectives, both the static and dynamic aspects of scheduling are revisited.

Static tree mapping Concerning the static mapping and choice of candidates, a layer L_0 is defined as before, under which tree parallelism is used exclusively. However, the mapping of the nodes from L_0 is forced to respect the proportional mapping; this is because we observed that the strategy consisting in better balancing the load of the subtrees of the L_0 layer has a worse locality of processors and leads to a larger memory usage (more simultaneous contribution blocks corresponding to subtree roots because we are further from a stack behaviour). We call the zone of the tree under L_0 zone 4, because three other zones are defined above the L_0 layer, as show in Figure 4.5.

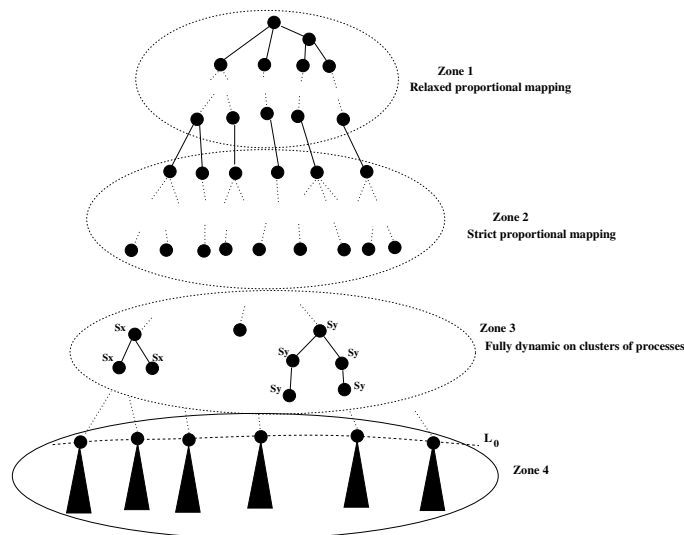


Figure 4.5: The four zones of the assembly tree. S_x and S_y are sets of preferential processors.

Zone 1 uses a relaxed proportional mapping in order to deal with the possible unbalances of zone 2, where a stricter proportional mapping is used. Moreover, we decided to implement a third zone – zone 3, in which each child inherits all candidate processors from its parent, which are defined by the

proportional mapping. The upper limit of zone 3 depends on a parameter $proc_{max}$ which corresponds to a number of processors. During the top-down approach of the proportional mapping, if the number of preferential processors of a node x is smaller or equal to $proc_{max}$, then x and all its descendants (above zone 4) belong to zone 3 and have the same set of candidate processors (see sets S_x and S_y in Figure 4.5). The motivation for zone 3 is that the fully dynamic code behaves well on small numbers of processors. Remark that, on architectures such as clusters of SMPs, $proc_{max}$ should be set to the number of processors inside the SMP node in order to naturally take into account memory locality.

Dynamic slave selection Given a set of candidate processors for a type 2 node, they are first sorted by increasing estimated workload. The slave selection algorithm done by the master of the type 2 node aims at balancing the workload of the selected processors (among candidate processors), subject to the memory constraints of each processor:

- estimated amount of available memory on the processor;
- maximum factor size: the maximum factor size is a relaxed value of an estimation of the factors done during the analysis. It is updated dynamically such that, when at a node a processor takes less than its share, it may take more for another node in the tree (and vice versa).
- maximum buffer size: the contribution block of a slave of a type 2 node should not exceed the size of the preallocated communication buffer⁶.

If one of those constraints is saturated, the given slave does not get more work and the extra work is shared between others. The maximum number of selected processors $nlim$ is first estimated to the minimum between the number of candidate processors and a number of processors such that the average work given to each slave is not too small compared to the work of the master processor. If the mapping of all the rows of the front does not succeed, the number $ntry$ of selected processors is increased, until reaching $nlim$ (Algorithm 3 in [34]). $nlim$ is then increased if needed, until reaching all candidate processors (Algorithm 2 in [34]).

Similar to the memory-based scheduling, it is critical to provide an accurate view of both load and memory information. The mechanisms for memory information have been slightly modified so that the frequency to send memory information increases when the available memory decreases.

Furthermore, the memory estimates have been modified. Thanks to all the above mechanisms to efficiently take memory constraints into account at runtime, the memory estimates computed during the analysis phase are now based on an optimistic scheduling scenario and not on a worst-case scenario anymore. Such an improvement of the reliability of memory estimates is of extreme importance in practical applications and was not possible in approaches not taking memory into account in dynamic scheduling decisions.

4.3 Memory scalability issues and memory-aware scheduling

Memory scalability is critical when increasing the number of processors. Ideally, when multiplying the number of processors by k , one would like the memory per processor to be divided by the same factor k . Memory is already taken into account in most strategies from Section 4.2. However, we step back a little in this section to show some intrinsic properties of the memory behaviour when processing in parallel a tree of tasks bottom-up.

Let us denote by S_{seq} the sequential peak of memory when the tree is reordered according to the techniques presented in Chapter 3. We denote by $S_{avg}(p)$ the average amount of storage per processor required to process that matrix with p processors, and by $S_{max}(p)$ the maximum amount (among all processors) of storage required to process that matrix with p processors.

The *memory efficiency* on p processors is defined as: $e_{max}(p) = \frac{S_{seq}}{p \times S_{max}(p)}$.

We can also define the *average memory efficiency* as $e_{avg}(p) = \frac{S_{seq}}{p \times S_{avg}(p)}$, which gives an idea of the overall loss of memory when increasing the number of processors.

⁶See Section 6.5 for a discussion on the memory associated to communication buffers.

A perfect memory scalability on p processors corresponds to $e_{\max}(p) = 1$. However, in the approach from Section 4.2.6, we typically observe that the memory efficiency is only between 0.1 and 0.3 on 128 processors, when the factors are stored on disk (see Chapter 5). With factors on disk, the storage only consists in temporary contribution blocks and current frontal matrices; memory efficiency is slightly better with factors in core memory. Although the scheduling algorithms from Section 4.2.6 rely on a *relaxed* proportional mapping, let us consider a strict proportional mapping [142] (see Section 4.2.3.1) to illustrate its memory aspects, compared to the postorder. We consider a perfect binary tree of depth k processed with $p = 2^k$ processors (see Figure 4.6(a)).

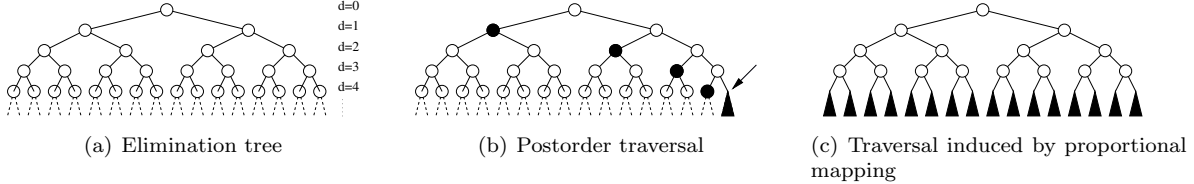


Figure 4.6: A perfect binary elimination tree (a) and possible traversals (b,c). Data in memory when the peak is reached during the tree traversal are black. There are p of them in (c) and only $\log(p)$ of them in (b). The arrow points to the node for which the peak is reached with a postorder traversal (b).

Using the notations of Chapter 3, we assume that the nodes that are at a depth lower than or equal to k have contribution blocks of identical sizes, cb , and frontal matrices of identical sizes, $m = 2 \times cb$. We furthermore assume that the subtrees at depth $d = k$ require a storage $S_k \geq 2 \times cb$. With a postorder traversal, the storage required to process the whole tree is equal to $S_{seq} = k \times cb + S_k$ (see Figure 4.6(b)).

If all the processors are mapped on each single node and if they synchronously follow a postorder traversal, the memory efficiency is equal to 1. (We assume for simplicity that contribution blocks and frontal matrices can be distributed with a balanced memory on the processors, which is not always the case in practice because of master tasks.) A possible *memory-aware mapping algorithm* is the following. . On the other hand, if we assume that a proportional mapping has been applied, each subtree at depth $d = k$ is processed on a different processor (which means that $p = 2^k$). The peak of memory of each processor is thus equal to S_k (see Figure 4.6(c)) and the memory efficiency is then equal to $e_{avg}(p) = \frac{S_{seq}}{p \times S_k} = \frac{k \times cb + S_k}{p \times S_k} \leq \frac{k/2+1}{p} = O(\log(p)/p)$.

In practice, the largest frontal matrices and contribution blocks are often near the top of the elimination tree. Therefore, the proportional mapping may induce a different memory efficiency. However, we see that in parallel, an algorithm in-between proportional mapping and postorder should be sought.

Let M_0 be the available memory per processor. Note that if $M_0 > S_{seq}/p$, then the tree can be processed with p processors (in the worst case, using a postorder traversal with all processors mapped on each single node. The tree is processed from top to bottom, similar to proportional mapping. All processors are assigned to the root node. At each step, each child inherits from a subset of p_i processors among the p processors that are mapped on their parent ($\sum_i p_i = p$) in the following way:

1. A step of proportional mapping is first attempted to map children nodes
2. If the memory peaks S_i for the children are such that $S_i/p_i < M_0$, then the mapping is accepted.
3. If some memory peaks are too large ($S_i/p_i > M_0$), subtrees are serialized [5], or arranged into groups leading to reasonable memory per subtree [6]. In such cases, subtrees processed last should take into account the contributions blocks produced by the groups processed earlier in the tree, which have to be kept in memory: to process subtree j , contribution blocks of the previously processed subtrees must be kept in memory, whose (average) size per processor is $(\sum_{i=1}^{j-1} cb_i)/p$. Therefore, the memory constraint M_0 used for the lower levels of the tree becomes $M_0 - (\sum_{i=1}^{j-1} cb_i)/p$.
4. In the above tests, a relaxation is done in order to ensure that $S_{\max}(p) \leq M_0$, not just $S_{avg}(p) \leq M_0$. For that, a relaxation parameter t can be used that estimates memory unbalance. All comparisons

with M_0 are replaced by comparisons on $M_0 \times t$, where $t > 1$. If $S_{max}/S_{avg} < t$, this means that we will consume less memory than M_0 .

Remark that serializing subtrees introduces constraints in the scheduling algorithms of our asynchronous multifrontal factorization. Preliminary results are encouraging and show that, starting from the strategies of Section 4.2.6, it is possible to significantly reduce the memory usage without significant loss of performance. More information and a complete description of the associated algorithms and ideas will be available in [150]. Two issues should be tackled in the long term:

- The approach described is static. It would be nice to reintroduce dynamic decisions, with memory constraints, but without falling in dangerous situations where memory constraints will be impossible to respect in the future (this is similar to deadlock-avoidance algorithms, see [152], with memory as the critical resource). Dynamic scheduling is necessary to cope with numerical pivoting and with limits to performance models on more and more complex computer platforms.
- From an application point-of-view, providing the allowed memory M_0 is sometimes possible. However, there is no point in using too much memory if this only increases performance by a very small percentage, or if it decreases it (swapping). From experiments with different values of M_0 , it seems that it would be possible to find M_0 values providing both a good performance (although not optimal) and a reasonable memory efficiency. Defining various standard strategies would be interesting, *e.g.*, avoid decreasing the optimal performance by more than 20%.

Chapter 5

A Parallel Out-of-Core Multifrontal Method

The objective of this chapter (see also [7, 12]) is to show how out-of-core storage may help decreasing the memory requirements of parallel sparse direct solvers and, consequently, allow the solution of larger problems with a given physical memory. We also show how low-level I/O mechanisms affect performance.

Introduction

As mentioned in the introduction and in Section 1.3.4, the memory usage of sparse direct solvers is often the bottleneck to solve large sparse systems of linear equations. In order to solve larger problems, out-of-core storage must sometimes be used when memory is insufficient. In this chapter, we describe some of the work we have done to design a robust out-of-core solver, in which computed factors are stored on disk and report experiment on significantly large problems. We observe that the core memory usage can be significantly reduced in serial and parallel executions, with a time performance of the factorization phase comparable to that of a parallel in-core solver. A careful study shows how the *I/O* mechanisms impact the performance. We describe a low-level *I/O* layer that avoids the perturbations introduced by system buffers and allows consistently good performance results. To go significantly further in the memory reduction, it is interesting to also store the intermediate working memory on disk. We describe algorithmic models to address this issue, and study their potential in terms of both memory requirements and *I/O* volume.

Several authors [1, 68, 95, 148, 149, 134, 164] have worked on sequential or shared-memory out-of-core solvers (see also the survey paper [163]), but out-of-core sparse direct solvers for distributed-memory machines are less common. Furthermore, authors have sometimes neglected the effect of system buffering, which often introduces a bias in the performance analysis. Although Dobrian [67] shows that multifrontal methods are generally well-suited for the out-of-core factorization, contributions by [148] and [149] for uniprocessor approaches pointed out that these methods may not fit well an out-of-core context because large dense temporary matrices can represent a bottleneck for memory. Therefore, they prefer left-looking approaches (or switching to left-looking approaches)¹. However, in a parallel context, increasing the number of processors can help keeping such large frontal matrices in-core. Furthermore, note that pivoting issues in out-of-core left-looking approaches are not always natural [95]; on the contrary, we are interested in approaches with the exact same pivoting strategies in the in-core and out-of-core contexts.

An out-of-core multifrontal approach based on virtual memory was experimented in the past by [58]. In this work, the authors use a low-level layer [57] enabling to influence the system paging mechanisms. A limited number of calls to this layer was added at the application level in order to provide information on the application. For example, it is possible to inform the layer that a zone of memory has a high priority because it will be used very soon (it will then be prefetched if on disk, or kept if already in core memory), or that in

¹See Section 1.2.4 for a brief description of left-looking and right-looking approaches

another zone the contents are obsolete because they have been used already (and the corresponding pages can then be discarded without being written to disk). In particular the current active frontal matrix and the top of the stack will be set a high priority; the data corresponding to a contribution block just consumed by the parent are obsolete and need not be written to disk. The authors showed results significantly better than when relying on the default LRU (Least Recently Used) policy, with very limited local modifications to the sparse direct solver. However, this type of approach is not portable because it is too closely related to the operating system. In the rest of this chapter, we only consider out-of-core approaches with *explicit* calls to *I/O* routines.

As explained in Section 1.3.3, the memory in multifrontal methods consists of two parts: one corresponds to terminal data, the *factors*; the other one to temporary data, the *active memory* (or *active storage*). We also refer the reader to Chapter 3 for possible out-of-core models in the case of serial executions. Because the factors produced will only be accessed at the solve stage, it makes sense to write them to disk first. We use this approach to design an extension of a parallel solver (MUMPS, for MUltifrontal Massively Parallel Solver, see [24]) where factors are stored on disk during the factorization process. This approach allows to treat much larger problems, and to reduce significantly the memory usage (by a factor 5 to 10 on 1 to 4 processors, and a factor around 2 on 16 to 128 processors). An important issue is performance, which has to be analyzed in details. In particular, it must be noted that default *I/O* mechanisms based on the use of system buffers have several disadvantages and are not suitable for all applications. For simple applications (with small *I/O* requirements), the cost of an *I/O* is similar to the cost of a memory copy, and the effective *I/O* is performed asynchronously (or is not performed at all!) by the system. For more *I/O*-intensive cases (factorization of a matrix with large factors), some problems can occur: excessive memory usage of the system buffers, or bad performance. To avoid these problems, we have decided to study direct *I/O* mechanisms (that bypass the system buffers), and to couple them with an asynchronous approach at the application level. This allows to obtain consistent performance results and, thanks to the knowledge we have of the application, to control *I/O* in a more tight way.

In order to go further in the memory reduction (and treat larger problems on a given number of processors), it is interesting to use disk storage not only for the factors, but also for part of the temporary working memory (or active storage). To analyze this approach, we propose a theoretical study based on an instrumentation of the solver with out-of-core factors and study the variations of the working memory according to different models of out-of-core memory management. We assess the minimum core memory requirements of the method, and study which type of tasks is responsible for the peak memory for the different models.

This chapter is composed of two main sections. In the first one, we present and analyse in detail the performance of the approach consisting in storing the computed factors to disk (Section 5.1). We then study the memory limits of different strategies to process the active memory out-of-core (Section 5.2). In Section 5.2.3, we analyze more accurately and qualitatively the type of tasks responsible for the peak of core memory usage in each of these strategies, and discuss their relation with critical parameters in the management of parallelism. Note that the solution step is also critical in this out-of-core context and should not be neglected, as large amounts of data will be read from disk. This is the object of a specific and separate study [23] that we will not detail with here.

5.1 A robust out-of-core code with factors on disks

We present in this section a robust out-of-core code based on MUMPS in which computed factors are stored on disk during the factorization step. This parallel out-of-core code is already used by several academic and industrial groups, and enables them to solve problems much larger than before. All the functionalities available in MUMPS may be used in this new out-of-core code (*LU* or *LDL^T* factorization, pivoting strategies, out-of-core solution step [22], ...). The motivation to store the factors on disk is that, in the multifrontal method, produced factors are not re-used before the solution step. In the approach we present, the factors are written as soon as they are computed (possibly via a buffer) and only the active memory remains in-core.

Because *I/O* buffering at the operating system level makes performance results difficult to reproduce and to interpret [7, 22], we first discuss system buffering issues and ways to avoid such buffering problems in our

performance study. We explain why the management of asynchronism for the *I/Os* should be transferred from the system level to the application level and present the mechanisms we have implemented to do so. A detailed performance analysis follows; it highlights several drawbacks from default *I/O* mechanisms that have most often been neglected in the past.

5.1.1 Direct and buffered (at the system level) *I/O* mechanisms

The efficiency of low-level *I/O* mechanisms directly affects the performance of the whole application. Several *I/O* tools are available: AIO (POSIX asynchronous *I/O* layer), MPI-IO [162] (*I/O* extension of the MPI standard) or FG [56] (high level asynchronous buffered *I/O* framework). However, C *I/O* library provides best portability while offering a reasonable abstraction level for our needs.

By default, when a write operation is requested, modern systems copy data into a system buffer (named *pagecache*) and effectively perform the disk access later, using an asynchronous mechanism. Thanks to that mechanism (hidden to the user), the apparent cost of the write operation is in many cases only equal to the cost of a memory copy. However, in the context of a high-performance out-of-core application, such a mechanism suffers four major drawbacks:

1. As the allocation policy for the system buffer (*pagecache*) is not under user control (its size may vary dynamically), the size of the remaining memory is neither controlled nor even known; this is problematic since out-of-core algorithms precisely rely on the size of the available memory. Subsequently, one may exploit only part of the available memory or, on the contrary, observe swapping and even run out of memory.
2. The system is well adapted to general purpose applications and not necessarily optimized for *I/O*-intensive applications: for example, it is better to avoid the intermediate copy to the *pagecache* when a huge stream of data must be written to disk.
3. The management of the *pagecache* is system-dependent (it usually follows an LRU policy). As a consequence, the performance of *I/O* operations vary (for instance, the *I/O* time can increase if the system needs to partially flush the *pagecache*). This is particularly problematic in the parallel context, where load balancing algorithms will not be able to take this irregular and unpredictable behaviour into account.
4. The last drawback is related to performance studies: when analysing the performance of an out-of-core code, one wants to be sure that *I/Os* are effectively performed (otherwise, and even if the code asks for *I/O*, one may be measuring the performance of an in-core execution). We insist on this point because this has sometimes not been done in other studies relating to sparse out-of-core solvers. Authors we are aware of who have taken this type of issues into account are Rothberg and Schreiber [148]: in order to get sensible and reproducible results, they dynamically add artificial delays in their code when the time for a read or write operation is observed to be smaller than the physical cost of a disk access.

The use of direct *I/O* mechanisms allows one to bypass the *pagecache*. The four previous drawbacks are then avoided: we are sure that *I/Os* are performed; no hidden additional memory is allocated (the *pagecache* is not used in this case); we explicitly decide when disk accesses are performed; and the *I/O* costs become stable (they only depend on the latency and the bandwidth of the disks). Direct *I/Os* are available on most modern computers and can be activated with a special flag when opening the file (*O_DIRECT* in our case). However data must be aligned in memory when using direct *I/O* mechanisms: the address and the size of the written buffer both have to be a multiple of the page size and/or of the cylinder size. In order to implement such a low-level mechanism, we had to rely on an intermediate aligned buffer, that we write to disk when it becomes full. The size of that buffer has been experimentally tuned to maximize bandwidth: we use a buffer of size 10 MB, leading to an approximate bandwidth of respectively 90 MB/s and 50 MB/s on the IBM and Linux platforms (described later). Furthermore, asynchronism must be managed at the application level to allow for overlapping between *I/Os* and computations.

5.1.2 Synchronous and asynchronous approaches (at the application level)

One purpose of this chapter is to highlight the drawbacks of the use of system buffers (or *pagecache*) and to show that efficiency may be achieved with direct *I/O*. To do so, the management of the asynchronous *I/O*s (allowing overlapping) has to be transferred from the system level to the application level. In order to analyze the behaviour of each layer of the code (computation layer, *I/O* layer at the application level, *I/O* layer at the system level) we designed two *I/O* mechanisms at the application level:

Synchronous *I/O* scheme. In this scheme, the factors are directly written to disk (or to the *pagecache*) with a synchronous scheme. We use standard C *I/O* routines: either *fread/fwrite* (to read from or write to a binary stream), *read/write* (to read from or write to a file descriptor), or *pread/pwrite* when available (to read from or write to a file descriptor at a given offset).

Asynchronous *I/O* scheme. In this scheme, we associate with each MPI process of the application an *I/O* thread in charge of all the *I/O* operations for that process. This allows to overlap *I/O* operations with computations². The *I/O* thread uses the standard POSIX thread library (pthreads). The computation thread produces (computes) factors that the *I/O* thread consumes (writes to disk) according to a producer-consumer paradigm. Each time a block of factors is produced, the computation thread posts an *I/O* request: it inserts the request into a *queue of pending requests* in a critical section. The *I/O* thread loops endlessly: at each iteration it waits for requests that it handles using a FIFO strategy. Symmetrically, the *I/O* thread informs the computation thread of its advancement with a second producer-consumer paradigm in which this time the *I/O* thread produces the finished requests (inserts them into the *queue of finished requests*). The computation thread consumes the finished requests by removing them from the queue when checking for their completion. This second mechanism is independent from the first one. Note that we limited our implementation to the case where only one *I/O* thread is attached to each computation thread. It could be interesting to use multiple *I/O* threads to improve overlapping on machines with several hard disks per processor, or with high performance parallel filesystems.

Together with the two *I/O* mechanisms described above, we designed a buffered *I/O* scheme. This approach relies on the fact that we want to free the memory occupied by the factors (at the application level) as soon as possible, *i.e.*, without waiting for the completion of the corresponding *I/O*. Thus, we introduced a buffer into which factors can be copied before they are written to disk. We implemented a double buffer mechanism in order to overlap *I/O* operations with computations: the buffer is divided into two parts in such a way that while an asynchronous *I/O* operation is occurring on one part, computed factors can be stored in the other part. In our experiments, the size of the buffer (half a buffer in fact) is set to the size of the largest estimated factor among the nodes of the tree. Note that the asynchronous scheme always requires a buffer in order to free the factors from main memory. Furthermore, the buffer is not necessary in the synchronous scheme and implies an extra copy. Therefore, we only present results with the buffered asynchronous scheme (that we name *asynchronous scheme* and abbreviate as *Asynch.*) and with the non-buffered synchronous one (that we name *synchronous scheme* and abbreviate as *Synch.*). When the *pagecache* is used together with the synchronous scheme (at the application level), asynchronism is managed at the system level; when direct *I/O* mechanisms are applied together with the asynchronous scheme, asynchronism only occurs at the application level.

5.1.3 Testing environment

For our study, we use test problems (see Table 5.1) from standard collections (PARASOL collection³, University of Florida sparse matrix collection⁴), or from MUMPS users. Publicly available matrices from our application

² Modern systems use the direct memory access (DMA) feature which allows an efficient overlapping of computation and *I/O*s even when only one processor is used.

³<http://www.parallab.uib.no/parasol>

⁴<http://www.cise.ufl.edu/research/sparse/matrices/>

partners are available on the griddtlse.org website (TLSE collection). We use two types of target platforms, one with local disks, one with remote disks. The platform with local disks is a cluster of Linux dual-processors at 2.6 GHz from PSMN/FLCHP⁵, with 4 GB of memory and one disk for each node of 2 processors. In order to have more memory per process and avoid concurrent disk accesses, only one processor is used on each node. The observed bandwidth is 50 MB / second per node, independently of the number of nodes, and the filesystem is ext3. The other machine is the IBM SP system from IDRIS⁶, which is composed of several nodes of either 4 processors at 1.7 GHz or 32 processors at 1.3 GHz. On this machine, we have used from 1 to 128 processors with the following memory constraints: we can access 1.3 GB per processor when asking for 65 processors or more, 3.5 GB per processor for 17-64 processors, 4 GB for 2-16 processors, and 16 GB on 1 processor. The *I/O* system used is the IBM GPFS [154] filesystem. With this filesystem we observed a maximal *I/O* bandwidth of 108 MBytes per second (using direct *I/O* to ensure that the *I/O*s are effectively performed, without intermediate copy). However, it is not possible to write files to local disks with the configuration of this platform. This results in performance degradations when several processors simultaneously write/read an amount of data to/from the filesystem: the bandwidth decreases by a factor of 3 on 8 processors and by a factor of 12 on 64 processors when each processor writes one block of 500 MBytes. This filesystem is thus not optimal for parallel performance issues. However we chose to also run on this platform because it has a large number of processors, and allows to run large problems in-core on which we can compare out-of-core and in-core performance. By default, we used the METIS package [120] to reorder the matrices

Matrix	Order	nnz	Type	$nnz(L U)$ ($\times 10^6$)	Flops ($\times 10^9$)	Description
AUDIKW_1	943695	39297771	SYM	1368.6	5682	Crankshaft model (PARASOL collection).
BRGM	3699643	155640019	SYM	4483.4	26520	Ground mechanics model from BRGM (TLSE collection).
CONESHL_mod	1262212	43007782	SYM	790.8	1640	Cone with shell and solid element from SAMTECH (TLSE collection).
CONESHL2	837967	22328697	SYM	239.1	211.2	Provided by SAMTECH (TLSE collection).
CONV3D64	836550	12548250	UNS	2693.9	23880	Provided by CEA-CESTA; generated using AQUILON (http://www.enscpb.fr/master/aquilon).
GUPTA3	16783	4670105	SYM	10.1	6.3	Linear programming matrix (AA'), Anshul Gupta (Univ. Florida collection).
SHIP_003	121728	4103881	SYM	61.8	80.8	Ship structure (PARASOL collection).
SPARSINE	50000	799494	SYM	207.2	1414	Structural optimization, CUTEr (Univ. Florida collection).
QIMONDA07	8613291	66900289	UNS	556.4	45.7	Circuit simulation problem provided by Reinhard Schultz, Qimonda AG (TLSE collection).
ULTRASOUND80	531441	330761161	UNS	981.4	3915	Propagation of 3D ultrasound waves, provided by M. Sosonkina.
XENON2	157464	3866688	UNS	97.5	103.1	Complex zeolite, sodalite crystals, D. Ronis (Univ. Florida collection).

Table 5.1: Test problems. Size of factors ($nnz(L|U)$) and number of floating-point operations (Flops) computed with METIS.

and limit the number of operations and fill-in arising during the numerical factorization. In the following, parallel executions rely on the dynamic scheduling strategy proposed in [34]. When reporting memory usage, we focus on real data (factors, temporary active memory), excluding storage for integers and symbolic data structures (which is comparatively negligible).

5.1.4 Sequential performance

Because the behaviour of our algorithms on a platform with remote disks might be difficult to interpret, we first validate our approaches on machines with local disks. For these experiments, we use the cluster of dual-processors from PSMN/FLCHP presented in Section 5.1.3. Because this machine has a smaller memory,

⁵Pôle Scientifique de Modélisation Numérique/Fédération Lyonnaise de Calcul Haute Performance

⁶Institut du Développement et des Ressources en Informatique Scientifique

the factorization of some of the largest test problems swapped or ran out of memory. We first present results concerning relatively small problems (SHIP_003 and XENON2) because they allow us to highlight the perturbations induced by the pagecache and because we have an in-core reference for those problems. We then discuss results on larger problems. Table 5.2 reports the results.

Matrix	Direct <i>I/O</i>	Direct <i>I/O</i>	P.C.	P.C.	IC
	Synch.	Asynch.	Synch.	Asynch.	
SHIP_003	43.6	36.4	37.7	35.0	33.2
XENON2	45.4	33.8	42.1	33.0	31.9
AUDIkw_1	2129.1	[2631.0]	2008.5	[3227.5]	(*)
CONESHL2	158.7	123.7	144.1	125.1	(*)
QIMONDA07	152.5	80.6	238.4	144.7	(*)

Table 5.2: Elapsed time (seconds) for the sequential factorization using direct *I/O* mechanisms or the pagecache (P.C.) for both the synchronous (Synch.) and asynchronous (Asynch.) approaches, and compared to the in-core case (IC) on a machine with local disks (PSMN/FLCHP).

(*) The factorization ran out of memory. [2631.0] Swapping occurred.

For problems small enough so that the in-core factorization succeeds (top of Table 5.2), we have measured average bandwidths around 300 MB/s when relying on the pagecache, whereas the disk bandwidth cannot exceed 60 MB/s (maximum physical bandwidth). This observation highlights the perturbations caused by the system pagecache; such perturbations make the performance analysis unclear. Moreover, the system can in these cases allocate enough memory for the pagecache so that it needs not perform the actual *I/O*s. When an *I/O* is requested, only a *memory copy* from the application to the pagecache is done. This is why the factorization is faster when using the pagecache: this apparent efficiency comes from the fact that the execution is mostly performed in-core. In other words, a performance analysis of an out-of-core code using the system pagecache (it is the case of most out-of-core solvers) makes sense only when performed on matrices which require a memory significantly larger than the available physical memory. This illustrates the fourth drawback from Section 5.1.1.

However, when direct *I/O* mechanisms are used with the asynchronous out-of-core scheme for these relatively small problems, the factorization remains efficient (at most 10% slower than the in-core one). The slight overhead compared to the asynchronous out-of-core version relying on the pagecache results from the cost of the last *I/O*. After the last factor (at the root of the tree) is computed, the *I/O* buffer is written to disk and the factorization step waits for this last *I/O* without any computation to overlap it. When using direct *I/O*, this last *I/O* is performed synchronously and represents an explicit overhead for the elapsed time of the factorization. On the contrary, when the pagecache is used, only a memory copy is performed: the system may perform the effective *I/O* later, after the end of the factorization. For some larger matrices (CONESHL2 or QIMONDA07), the results show a very good behaviour of the asynchronous approach based on direct *I/O*, even when the last *I/O* is included. In the case of the AUDIKW_1 matrix, the asynchronous approaches swapped because of the memory overhead due to the *I/O* buffer. Note that even in this case, the approach using direct *I/O* has a better behaviour. More generally, when comparing the two asynchronous

Direct <i>I/O</i>	P.C.
Asynch.	Asynch.
1674	[2115]

Table 5.3: Elapsed time (seconds) for the factorization of matrix AUDIKW_1 when the ordering strategy PORD is used. Platform is PSMN/FLCHP. [2115] Swapping occurred.

approaches to each other on reasonably large matrices, we notice a higher overhead of the pagecache-based one, because it consumes extra memory hidden to the application. To further illustrate this phenomenon,

we use the PORD [156] ordering (see Table 5.3), which reduces the memory requirements in comparison to METIS for matrix AUDIKW_1. In this case the memory required for the asynchronous approach is of 3783 MB. We observe that the asynchronous scheme allows a factorization in 1674 seconds when based on direct *I/O*, without apparent swapping. However, when using the pagecache, the factorization requires 2115 seconds: the allocation of the pagecache makes the application swap and produces an overhead of 441 seconds. This illustrates the first drawback (introduced in Section 5.1.1). Let us now discuss the case of the matrix of our collection that induces the most *I/O*-intensive factorization, QIMONDA07. For this matrix, assuming a bandwidth of 50 MB/s, the time for writing factors (85 seconds) is greater than the time for the in-core factorization (estimated to about 60 seconds). We observe that the system (columns “P.C.” of Table 5.2) does not achieve a good performance (even with the buffered asynchronous scheme at the application level that avoids too many system calls). Its general policy is not designed for such an *I/O*-intensive purpose. On the other hand, the use of direct *I/O* mechanisms with an asynchronous scheme is very efficient. *I/O*s are well overlapped by computation: the factorization only takes 80.6 seconds during which 60 seconds (estimated) of computation and 78.8 seconds (measured) of disk accesses are performed (with a measured average bandwidth of 53.8 MB/s). This illustrates the second drawback of the use of the pagecache: we have no guarantee of its robustness in an *I/O*-intensive context, where *I/O* should be performed as soon as possible rather than buffered for a while and then flushed. (Note that the synchronous approach with direct *I/O* mechanisms is not competitive because computation time and *I/O* time cumulate without possible overlap.) To confirm these results on another platform, Table 5.4 reports the performance obtained on the IBM machine, where remote disks are used. Again we see that even with remote disks, the use of direct

Matrix	Direct <i>I/O</i>	Direct <i>I/O</i>	P.C.	P.C.	IC
	Synch.	Asynch.	Synch.	Asynch.	
AUDIkw_1	2243.9	2127.0	2245.2	2111.1	2149.4
CONESHL_MOD	983.7	951.4	960.2	948.6	922.9
CONV3D64	8538.4	8351.0	[[8557.2]]	[[8478.0]]	(*)
ULTRASOUND80	1398.5	1360.5	1367.3	1376.3	1340.1
BRGM	9444.0	9214.8	[[10732.6]]	[[9305.1]]	(*)
QIMONDA07	147.3	94.1	133.3	91.6	90.7

Table 5.4: Elapsed time (seconds) on the IBM machine for the factorization (sequential case) using direct *I/O*s or the pagecache (P.C.) for both the synchronous (Synch.) and asynchronous (Asynch.) approaches, and compared to the in-core case (IC), for several matrices.

(*) The factorization ran out of memory.

[[8857.2]] Side effects (swapping, ...) of the pagecache management policy.

I/O coupled with an asynchronous approach is usually at least as efficient as any of the approaches coupled with the use of the pagecache and that relying only on the pagecache (P.C., Synch.) leads to additional costs. Furthermore, note that this table provides a representative set of results among several runs, each matrix corresponding to one submission at the batch-scheduler level. Indeed, performance results vary a lot from execution to execution. For instance, we were sometimes able to observe up to 500 seconds gain on the very large matrix CONV3D64 thanks to the use of direct *I/O*s (with an asynchronous scheme) compared to the use of the pagecache. Finally, note that for matrix AUDIKW_1 the performance is sometimes better with the out-of-core approach than with the in-core approach (2149.4 seconds in-core versus 2111.1 seconds for the system-based asynchronous approach and 2127.0 seconds for the direct *I/O* approach). This comes from machine-dependent (in-core) cache effects resulting from freeing the factors from main memory and always using the same memory area for active frontal matrices: a better locality is obtained in the out-of-core factorization code.

5.1.5 Parallel performance

Table 5.5 gives the results obtained in the parallel case on our cluster of dual-processors. We can draw conclusions similar to the sequential case. For large matrices (see results for `CONESHL_MOD` and `ULTRASOUND80`), the use of the asynchronous approach relying on direct *I/O* has a good behaviour: we achieve high performance without using the pagecache and avoid its possible drawbacks. In the *I/O*-dominant case (`QIMONDA07` matrix), the pagecache again has serious difficulties to ensure efficiency (second drawback).

We observe that the execution sometimes swaps (`CONESHL_MOD` on 1 processor or `ULTRASOUND80` on 4 processors) because of the additional space used for the *I/O* buffer at the application level. This leads to a slowdown so that the benefits of asynchronism are lost. In this asynchronous case, when comparing the system and the direct *I/O* approaches, it appears that the additional memory used by the operating system (the pagecache) leads to a larger execution time, probably coming from a larger number of page faults (extra memory for the pagecache and first drawback).

Provided that enough data are involved, the out-of-core approaches appear to have a good scalability, as illustrated, for example, by the results on matrix `CONESHL_MOD`. The use of local disks allows to keep a good efficiency for parallel out-of-core executions.

Matrix	#P	Direct <i>I/O</i>		P.C.		IC
		Synch.	Asynch	Synch	Asynch	
<code>CONESHL_MOD</code>	1	4955.7	[5106.5]	4944.9	[5644.1]	(*)
	2	2706.6	2524.0	2675.5	2678.8	(*)
	4	1310.7	1291.2	1367.1	1284.9	(*)
	8	738.8	719.6	725.6	724.7	712.3
<code>ULTRASOUND80</code>	4	373.2	[399.6]	349.5	[529.1]	(*)
	8	310.7	260.1	275.6	256.7	(*)
<code>QIMONDA07</code>	1	152.5	80.6	238.4	144.7	(*)
	2	79.3	43.4	88.5	57.1	
	4	43.5	23.1	42.2	31.1	[750.2]
	8	35.0	21.1	34.0	24.0	14.6

Table 5.5: Elapsed time (seconds) for the factorization on 1, 2, 4, and 8 processors using direct *I/O* mechanisms or the pagecache (P.C.), for both the synchronous (Synch.) and asynchronous (Asynch.) approaches, and compared to the in-core case (IC) on a machine with local disks (PSMN/FLCHP).

(*) The factorization ran out of memory. [750.2] Swapping occurred.

We now present results on a larger number of processors, using the IBM machine at IDRIS. Note that the *I/O* overhead is more critical in the parallel case as the delay from one processor has repercussions on other processors waiting for it (third drawback). We show in Table 5.6 (for matrix `ULTRASOUND80`) that we can achieve high performance using direct *I/O*s with an asynchronous scheme.

When the number of processors becomes large (64 or 128) the average volume of *I/O* per processor is very small for this test problem (15.3 MB on 64 processors, 7.7 MB on 128) and the average time spent in *I/O* mode is very low (less than 2.4 seconds) even in the synchronous scheme. Therefore, the synchronous approach with direct *I/O*, which does not allow overlapping of computations and *I/O*s is not penalized much. Concerning the comparison of the asynchronous approach with direct *I/O* to the system approach, performance are similar. However, when we have a critical situation, the use of the system pagecache may penalize the factorization time, as observed on 128 processors in the synchronous case. In Table 5.7, we report the results obtained on one large symmetric matrix. We observe here that it is interesting to exploit asynchronism at the application level, both for the direct *I/O* approach and for the system (pagecache) approach.

<i>I/O</i> mode	Scheme	1	2	4	8	16	32	64	128
Direct <i>I/O</i>	Synch.	1398.5	1247.5	567.1	350.9	121.2	76.9	44.6	36.5
Direct <i>I/O</i>	Asynch.	1360.5	(*)	557.4	341.2	118.1	74.8	45.0	33.0
P.C.	Synch.	1367.3	1219.5	571.8	348.8	118.5	69.6	44.8	90.0
P.C.	Asynch.	1376.3	(*)	550.3	339.2	109.4	73.8	45.2	30.0
	IC	1340.1	(*)	(*)	336.8	111.0	64.1	40.3	29.0

Table 5.6: Elapsed time (seconds) for the factorization of the **ULTRASOUND80** matrix using direct *I/O* mechanisms or the pagecache (P.C.), for both the synchronous (Synch.) and asynchronous (Asynch.) approaches, and compared to the in-core case (IC) for various numbers of processors of the IBM machine.

(*) The factorization ran out of memory.

<i>I/O</i> mode	Scheme	1	2	4	8	16	32	64	128
Direct <i>I/O</i>	Synch.	983.7	595.3	361.3	158.2	69.8	41.6	26.9	21.5
Direct <i>I/O</i>	Asynch.	951.4	549.5	340.5	156.9	65.7	41.5	24.7	16.3
P.C.	Synch.	960.2	565.6	358.8	159.0	68.2	41.8	28.1	18.9
P.C.	Asynch.	948.6	549.6	336.6	153.7	65.8	40.4	26.8	16.1
	IC	922.9	(*)	341.4	162.7	64.3	39.8	20.7	14.7

Table 5.7: Elapsed time (seconds) for the factorization of the **CONESHL_MOD** matrix using direct *I/O* mechanisms or the pagecache (P.C.) for both the synchronous (Synch.) and asynchronous (Asynch.) approaches, and compared to the in-core case (IC), for various numbers of processors of the IBM machine.

(*) The factorization ran out of memory.

5.1.6 Discussion

Overlapping of *I/O*s and computations allows to achieve high performance both when asynchronism is ensured at the system level (pagecache) and when it is managed at the application level (and uses the direct *I/O* approach). However, we have shown that in critical cases (either when a high ratio *I/O*/computation is involved - as for matrix **QIMONDA07** - or when a huge amount of *I/O* is required - as for matrix **CONV3D64**) the asynchronous scheme using direct *I/O* is more robust than the schemes using the pagecache. Similar difficulties of the system approach for read operations have also been shown in [22]. Furthermore, notice that even when the system approach has a good behaviour, we have observed that it often achieves better performance when used with a buffered asynchronous scheme at the application level: calling *I/O* routines (system calls) too frequently decreases performance.

To conclude this section, let us mention the memory gains that can be obtained when storing the factors to disk. For a small number of processors, the memory requirements of the application decrease significantly (more than 90% on some problems in the sequential case, as shown in column “1 processor” of Table 5.8).

When the number of processors increases (16 or more), an out-of-core execution usually allows to save between 40% and 50% of memory, as reported in Table 5.8. Note that in some cases, the amount of memory saved can be much larger, as illustrated by the **QIMONDA07** matrix. It is also interesting to note that even if the memory scalability of our out-of-core solver is not good when going from 1 to 16 processors, it somehow stabilizes after 16 processors: the memory ratio $\frac{OOC}{IC}$ is then almost constant and the memory usage scales reasonably. A possible explanation is that, after some point, increasing the number of processors does not increase too much the overall memory usage because most of the memory is consumed in the large fronts near the top of the tree, and those fronts are distributed over the processors. In the rest of this chapter, the code described in this section will be referred to as **Factors-on-disk**.

Matrix	1 processor		16 processors		32 processors		64 processors		128 processors	
	OOC	IC	OOC	IC	OOC	IC	OOC	IC	OOC	IC
AUDI_KW_1	2299	12188	909	1402	589	742	272	353	179	212
CONESHL_MOD	1512	7228	343	780	167	313	103	176	61	96
CONV3D64	6967	(17176)	1047	1849	540	930	265	471	148	251
QIMONDA07	29	4454	6	283	5	143	4	72	(*)	(*)
ULTRASOUND80	1743	8888	339	662	178	323	92	176	52	92

Table 5.8: Average space effectively used for scalars (in MBytes) per processor, for sequential and parallel executions on various numbers of processors, in the out-of-core (OOC) and in-core (IC) cases, for some large matrices. The IBM machine was used.

(*) The analysis ran out of memory. (17176) Estimated value from the analysis phase (the numerical factorization ran out of memory).

5.1.7 Panel version

One conclusion of the above results is that an asynchronous (Asynch.) approach is worth using. The drawback of this approach is that the granularity of I/O is large, leading to possibly very large (sometimes prohibitively large) buffers in order to overlap computation and I/O . Therefore, a so called “panel version” was developed, which decreases the granularity of I/O and the associated size of I/O buffers. Thanks to this mechanism, the out-of-core buffers are limited to the storage of a few panels: each panel contains columns of the L factors or rows of the U factors. This led to a lot of specification and code development, necessary to make this work effective in practical large-scale applications and usable by others. The main points concern:

- new design of the I/O layer to manage panels: specification of the storage for panels, variable size of the panels, special cases (for example, a panel should not finish in the middle of a 2×2 pivot);
- extension of the solve algorithm to work with panels: computations on each loaded panel, instead of each frontal matrix;
- management of numerical pivoting issues: inside a frontal matrix, we used to pivot in the LAPACK style, that is, we pivot rows that are in the previous panels. If previous panels are on disk, this is unfortunately not possible. Therefore, we store all necessary pivoting information and not just swapped lists of indices for each front, as already mentioned in Section 2.2.3.
- experimentation, testing and validation of the interaction and synchronizations between the I/O thread and the factorization kernels.

The advantages of working with panels instead of frontal matrices are as follows: (i) the buffer size has been strongly reduced when asynchronous I/O are managed at the application level (see Table 5.9); (ii) I/O 's can be overlapped with computations during the factorization of a frontal matrix whereas they used to be overlapped only between the factorizations of different frontal matrices; and (iii) the L and U factors can be written to independent files: this allows a better data access during the solution step and strongly improves the efficiency of that step, which is even more sensitive to I/O 's than the factorization phase.

5.2 Description and analysis of models to further reduce the memory requirements

Either to further reduce the memory requirements on large numbers of processors, or to process even larger problems on machines with few processors, one also needs to store the intermediate active memory to disk. In this section we propose a theoretical study to evaluate the interest of storing the contribution blocks out-of-core. Our motivation is that the problem of managing the active memory out-of-core in a parallel

Matrix	#procs	Out-of-core elementary data	
		<i>Factor block</i>	<i>Panel</i>
AUDIKW_1	1	1067.1	12.8
AUDIKW_1	32	155.5	12.8
CONESHL_MOD	1	1292.8	13.8
CONESHL_MOD	32	125.1	10.6
CONV3D64	1	3341.5	40.2
CONV3D64	32	757.6	40.2
ULTRASOUND80	1	1486.6	20.4
ULTRASOUND80	32	208.3	20.4

Table 5.9: Size of the I/O buffers (MB) with an asynchronous factorization.

asynchronous context is novel and needs to be studied before any real-life implementation. In addition, the dynamic and asynchronous schemes used in the parallel multifrontal method (at least as implemented in the MUMPS solver) make the behaviour difficult to forecast. It is thus natural to evaluate the gains that can be expected from such a parallel out-of-core method. To reach this objective, we present several models (first introduced in [7]), to perform the assembly of the contribution blocks in an out-of-core context. We use these models to better understand the memory limits of the approach and to identify the bottlenecks to treat arbitrarily large problems. Note that treating problems as large as possible (topic of this section) is a different issue from achieving good performance (as discussed in the previous section).

5.2.1 Models to manage the contribution blocks on disk

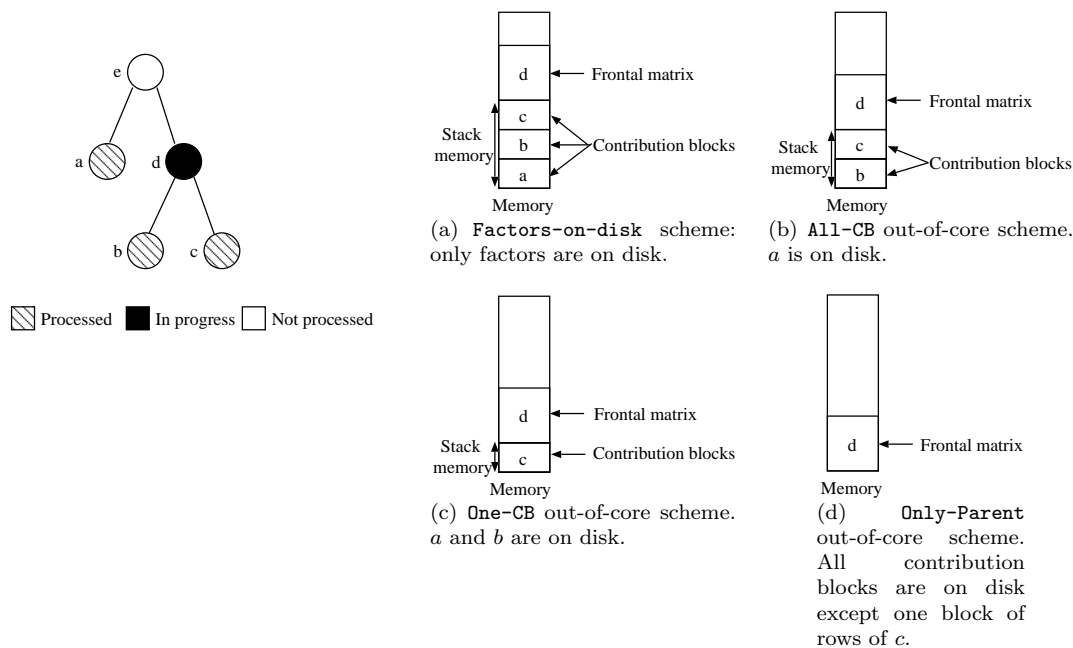


Figure 5.1: Out-of-core assembly schemes for the contribution blocks. Left: the frontal matrix of node d is being assembled. Right: data that must be present in core memory when assembling the contribution block of c into the frontal matrix of d .

We are interested in schemes where contribution blocks will be written at most once (after they are produced) and read at most once (before they are assembled into the frontal matrix of their parent). We will assume that frontal matrices can hold in-core (but they can be scattered over several processors); note that

by doing so, we maintain the write-once/read-once property. Figure 5.1 illustrates the different schemes we have modeled for the assembly of a frontal matrix:

- **All-CB scheme.** In this scheme, all the contribution blocks of the children must be available in core memory *before* the frontal matrix of the parent is assembled. The assembly step (consisting of extend-add operations) is identical to the in-core case, the only difference is that contribution blocks may have been stored to disk earlier.
- **One-CB out-of-core scheme.** In this scheme, during the assembly of an active frontal matrix, the contribution blocks of the children may be loaded one by one in core memory (while the other ones remain on disk).
- **Only-Parent out-of-core scheme.** In this scheme, we authorize all the contribution blocks from children to stay on disk: they may be loaded in memory row by row (or block of rows by block of rows) without being fully copied from disk to memory.

Prefetching all the required data before the assembly step of a parent node (as in the All-CB scheme) allows to perform computations (extend-add operations) at a high rate. On the other hand, for the Only-Parent and One-CB schemes, the assembly operations will be interrupted by *I/O* and there will usually not be enough operations to overlap the next *I/O*. This will result in some overhead on the execution time. Remark that, because we consider parallel executions, frontal matrices can be scattered over several processors. In that case, there are several contribution blocks for a given node, one for each slave processor. Such contribution blocks may be written to disk and read back when they need to be assembled or sent.

5.2.2 Analysis of the memory needs of the different schemes

In order to study the memory requirements corresponding to each out-of-core assembly scheme, we have instrumented our parallel solver (the one from Section 5.1, **Factors-on-disk**) with a software layer that simulates *I/Os* on the contribution blocks. The idea is to assume that a contribution block is written to disk as soon as it is computed. Then we assume that it is read back when needed (for the assembly of the parent node) depending on the assembly scheme used. Data are at most written once and read once and a counter holds the size of the memory used for each scheme: (i) the counter is increased when a new task is allocated or when a contribution block is “read” from disk; (ii) the counter is decreased when a factor block or a contribution block is “written” to disk, or when a contribution block is freed (because it has been assembled into the frontal matrix of the parent).

In parallel, when a contribution block is produced, the mapping of the parent node may not be known (dynamic scheduling). Therefore, the contribution block stays on the sender side until the master of the parent node has decided of the mapping of its slave tasks. In our model, we assume that this contribution block is written to disk on the sender side (thus decreasing the counter), until the information on where to send it is known. At the reception of such a contribution, if the task (master or slave part of a frontal matrix) depending on the contribution has already been allocated on the receiver, the considered processor consumes it on the fly.

This count is done during the parallel numerical factorization step of a real execution: indeed, the memory requirements measured thanks to this mechanism exactly correspond to those we would obtain if contribution blocks were effectively written to disk. Clearly our main goal is to study the potential of a parallel out-of-core multifrontal method that stores temporary active memory to disk in terms of reduction of the core memory requirements. To reach this objective, a full implementation of the *I/O* mechanisms for each assembly scheme (together with the associated memory management for each scheme) is not necessary.

We report in Figure 5.2 a comparison of the memory peaks obtained when using our different assembly schemes for two large test problems (a symmetric one and an unsymmetric one). These two problems are representative of the behaviour we observed on the other matrices from Table 5.1. The top curve (**Factors-on-disk**), used as a reference, corresponds to the actual memory requirements of the code from Section 5.1, where contributions blocks are in-core; the others were obtained with the instrumentation of

that code described above. We observe that the strategies for managing the contribution blocks out-of-core provide a reduction of the working memory requirement that scales similarly to the **Factors-on-disk** version. We also notice that the peak of core memory for the **All-CB** assembly scheme is often close to the one where only factors are stored on disk. On the other hand, we observe that the **One-CB** scheme significantly decreases the memory requirements, and that the **Only-Parent** scheme further reduces the memory needed for the factorization. The relative gain observed with the **Only-Parent** scheme is large enough to conclude that it is worthwhile applying this scheme, in spite of the possible overhead on efficiency (and complexity) due to the need to be able to interleave *I/O* operations with assembly operations on small blocks of rows. It represents the minimal memory requirement that can be reached with our model, in which active frontal matrices are kept in-core.

Finally, notice that the memory requirement measured for each scheme corresponds to specific tasks (subtrees, master tasks, slave tasks) that have been allocated to the processor responsible of the peak of memory. In the next section, we analyze the content of the memory when the peak is reached in order to understand the critical features of the parallel multifrontal method that can affect it.

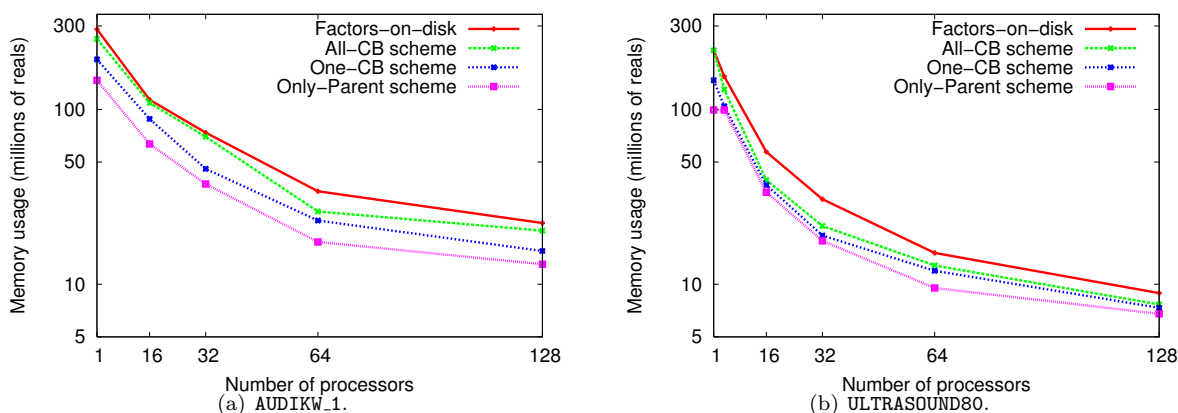


Figure 5.2: Memory behaviour (memory requirement per processor) for the different assembly schemes on various numbers of processors for the (symmetric) AUDIKW_1 and (unsymmetric) ULTRASOUND80 matrices. A logarithmic scale is used for the y-axis.

5.2.3 Analysing how the memory peaks are reached

We now analyze in more detail which type of tasks is involved in the peak of core memory for each strategy. Table 5.10 shows the state of the memory on the processor on which the peak memory is reached, in the case of an execution on 64 processors for the AUDIKW_1 and CONV3D64 problems. Note that, based on load balancing criteria, the dynamic schedulers may allocate several tasks simultaneously on a given processor. With the AUDIKW_1 matrix, we notice that for the **Only-Parent** and **One-CB** schemes as well as for the **Factors-on-disk** case, the peak memory is reached when a subtree is processed (more precisely when the root of that subtree is assembled). In the **Only-Parent** case, the processor also has a slave task activated. For the **All-CB** scheme, the peak is reached because the schedulers have simultaneously allocated too many slave tasks (3, corresponding to 3 different nodes) to one processor, reaching together 42.97% of the memory; at that moment, the memory also contains a master task but its size is less important (5.93%).

Similarly to matrices AUDIKW_1 and CONV3D64, we have performed this study for most problems in Table 5.1, on various numbers of processors. Rather than presenting all the results, we report here the main phenomena observed for two examples and we summarize in the following the typical behaviour observed for symmetric and unsymmetric problems. (i) For symmetric problems, between 8 and 128 processors, the peak is reached when a sequential subtree is being processed (see Figure 2.1), most often when the root of

Scheme	Memory percentage of the active tasks			Memory percentage of the contribution blocks	
	master tasks	slave tasks	sequential subtrees		
AUDIKW_1	Factors-on-disk	0%	0%	*27.11%	72.89%
	All-CB	5.93%	*42.97%	0%	51.10%
	One-CB	0%	0%	*75.10%	24.90%
	Only-Parent	0%	48.32%	*51.63%	0.04%
CONV3D64	Factors-on-disk	0%	*40.19%	0%	59.81%
	All-CB	0%	*65.71%	0%	34.29%
	One-CB	38.89%	*46.27%	0%	14.84%
	Only-Parent	47.82%	*52.06%	0%	0.12%

Table 5.10: Memory state of the processor that reaches the global memory peak when the peak is reached, for each out-of-core scheme and for the **Factors-on-disk** code, for the (symmetric) **AUDIKW_1** matrix and the (unsymmetric) **CONV3D64** matrix on 64 processors. Symbol * in a column refers to the last task activated before obtaining the peak, which is thus *responsible* for it.

that subtree is assembled; this occurs for all out-of-core schemes. Sometimes a slave task may still be held in memory when the peak arises (and it can then represent between 25 % and 75 % of the memory of the active tasks on the processor). (ii) For unsymmetric problems, on many processors (from 16 to 128), the peak is generally obtained because of a large master task (which requires more memory in the unsymmetric case than in the symmetric case, see Figure 2.7). This is increasingly true when going from the **Factors-on-disk** scheme to the **Only-Parent** scheme. These effects are sometimes hidden when many tasks are simultaneously active. For example, on 64 processors with the **All-CB** scheme, for the **CONV3D64** problem, the peak is obtained when a processor has four slave tasks in memory. With fewer processors (less than 8), the assembly of the root of a subtree is more often responsible for the peak.

Thanks to parallelism, memory needs of a particular task can be parcelled out over many processors. However, in order to be efficient, some tasks remain sequential and become the memory bottleneck when the other ones are parallelized. On the range of processors used, the limiting factor observed is the granularity of master tasks (which are processed in sequential on a given processor) for unsymmetric problems and the one of the subtrees in the symmetric case. In both cases, there is still some potential to decrease the memory requirements by doing static modifications to the tree of tasks, possibly at the cost of a performance degradation [10].

5.2.4 Summary

This study allows to analyze the memory behaviour of several models for an out-of-core storage of the active memory in a parallel asynchronous multifrontal method. The relative gains observed with the **Only-Parent** strategy make it the most relevant one, in spite of the fact that the implementation of the assembly process will be the most complex. We have also identified some key parameters - granularity of subtrees and of master tasks - which impact the minimum memory requirements of the method. On these aspects, specific tuning of the granularity of such tasks can be done to further reduce the memory requirements.

Note that in a limited memory environment, contribution blocks need not systematically be written to disk. For example, with the **One-CB** scheme, not all sets of parent and child contributions have to follow the **One-CB** scheme: at least one child contribution must be in memory during the assembly process but there may be more if memory allows it. Similarly, in the **Only-Parent** scheme, some frontal matrices will still be assembled with a **One-CB** or even **All-CB** scheme. To summarize, the **Only-Parent** scheme allows to go further in the memory reduction, but is not less efficient than the other schemes if memory is large enough.

5.3 Conclusion

In this chapter, we have presented a parallel out-of-core direct solver that stores computed factors on disk. It allows to handle problems significantly larger than an in-core solver. We have highlighted several drawbacks

of the *I/O* mechanisms generally used (which in general implicitly rely on system buffers): memory overhead that can result in excessive swapping activity, extra cost due to useless intermediate memory copies, dependency on the system policy and non reproducibility of the results. We have then proposed a robust and efficient *I/O* layer, which uses direct *I/O*s together with an asynchronous approach at the application level. This avoids the drawbacks of the system buffers and allows one to achieve good (and reproducible) performance. On a limited number of processors, storing factors on disk clearly allows to solve much larger problems. With more processors (16 to 128), because the active memory does not scale as well as the factors, the core memory usage is only reduced by a factor of two, on average.

In order to go further in the memory reduction with out-of-core techniques, especially on large numbers of processors, an out-of-core storage of the contribution blocks has also been studied. We have proposed several models for the assembly process of the multifrontal method and analyzed their impact in terms of minimum core memory for parallel executions. To do that, we have instrumented our solver (that stores factors to disk), performed parallel executions and measured the memory requirements for each model. This analysis showed that the most complex assembly scheme was worth implementing. We have also identified some key parameters related to the management of parallelism (granularity of subtrees and of master tasks) that can impact the core memory usage.

As stated in [148, 149] one difficulty of the sequential multifrontal approach in an out-of-core context comes from large frontal matrices that can be a bottleneck for memory: allowing the out-of-core storage of the contribution blocks sometimes only decreases the memory requirements by a factor of about 2. Fortunately, in this context, we have shown that parallelism can further decrease these memory requirements significantly (even when only the factors are stored to disk) as large frontal matrices are split over the processors. When this is not enough, an out-of-core approach should also be used for frontal matrices.

Finally, remark that the memory-aware mapping algorithms presented in Section 4.3 should allow significant reduction of the active memory usage on large numbers of processors (experiments in this section are based on the scheduling algorithms from Section 4.2.6): given the increasing parallelism in current architectures, it makes sense to develop and analyze further those algorithms before going into the complexity of managing an out-of-core stack memory.

Chapter 6

Solving Increasingly Large Problems

In this chapter, we describe a series of techniques and advances to solve larger problems, decrease memory requirements, and make better use of modern multicore machines. Simulation codes still require the solution of very large problems; for example in earth science applications, one is still limited when trying to solve very large cases [135, 160]. The previous chapters already do a step in that direction: out-of-core storage of the factors allows solving much larger problems than before, and efficient scheduling in parallel distributed environments is critical. As said before, using out-of-core temporary storage could also be useful to solve larger problems but it made sense to first improve temporary storage scalability (see Section 4.3). Given the evolution trends of computer architectures with increasing global memory size (although memory per core decreases) and increasing numbers of cores, out-of-core storage for the contribution blocks is still not the most critical priority compared to the efficient exploitation of such resources.

This chapter is organized as follows. In Section 6.1, we describe a way to parallelize the symbolic analysis phase, using parallel graph partitioning tools. We discuss the use of 64-bit integers in Section 6.2 and then consider the problem of performing the forward elimination during the factorization stage (Section 6.3), accelerating the solve phase (especially in an out-of-core environment) and avoiding the need to store L factors. Section 6.4 focuses on further improvements to the solve phase, both from a memory and complexity point of view. In Section 6.5, we show how to reduce the memory associated with communication buffers, at the cost of a small performance penalty. Section 6.6 and the included subsections focus on current work to better exploit multicores thanks to multithreading techniques (this is the object of an on-going PhD thesis). We conclude the chapter with some other work directions aiming at solving larger problems efficiently.

6.1 Parallel analysis

Large matrices may be too large to be stored in the memory of a single processor or of a single node. In our algebraic approaches where the analysis phase of the sparse direct solver relies on the entire graph of the matrix, it becomes critical to parallelize this analysis phase. The main reason is memory: using large numbers of processors and out-of-core approaches (see chapter 5), the memory per processor for the factorization is significantly reduced, and the memory for the analysis phase if the graph of the matrix is centralized on a single node can become the bottleneck. Furthermore, since there is a significant degree of parallelism during the factorization, the time for the analysis is far from being negligible on large numbers of processors.

Therefore, it became critical to parallelize the analysis phase. The core of the analysis phase consists of two main operations (sometimes tightly coupled):

- Elimination tree computation: this step provides a pivotal order that minimizes the fill-in generated at factorization time and identifies independent computational tasks that can be executed in parallel.
- Symbolic factorization: simulates the actual factorization in order to estimate the memory that has to be allocated for the factorization Phase.

The parallelization of this operation can be achieved by providing an interfacing mechanism to third party parallel ordering tools like PT-SCOTCH [52] or ParMETIS [121]. Parallel ordering tools like those mentioned above return an index permutation that describes the pivotal order and a separators tree which results from the application of an ordering algorithm based on nested dissection. Based on the result of the ordering step, the parallel symbolic factorization is conducted as in Figure 6.1 [18]. First, a number of subtrees, in the separators tree, is selected that is equal to the number of working processors; each of these subtrees is assigned to a processor that performs the symbolic factorization of the unknowns contained in it (Figure 6.1(*left*)). Once every processor has finished with its subtree, the symbolic elimination of the unknowns associated with the top part of the tree is performed sequentially on a designated processor (Figure 6.1(*right*)). The method used to perform the symbolic factorization locally on each processor is based on the usage of quotient graphs in order to limit the memory consumption both for the subtrees and the top of the tree.

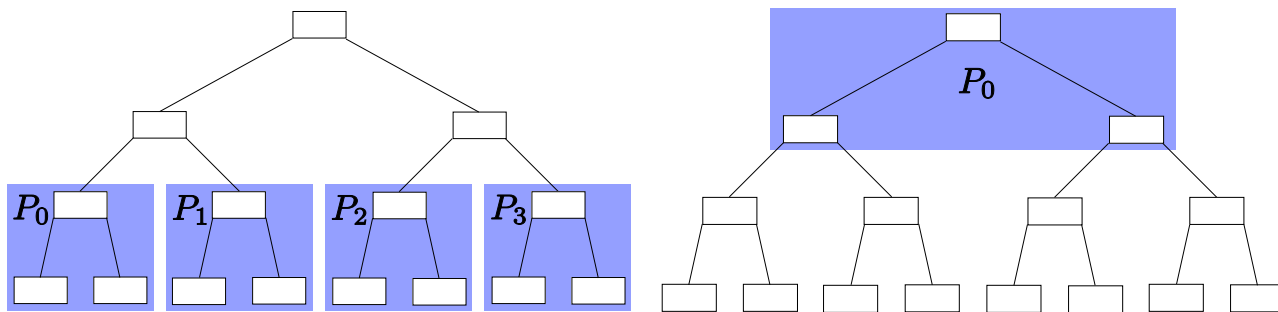


Figure 6.1: Parallel symbolic factorization.

Num. procs	Factors Size	Max. Front Size	Flops	Time	Memory per proc.
2	1.06	1.16	1.05	2.99	0.52
4	1.08	1.14	1.12	1.75	0.28
8	1.10	1.29	1.21	1.09	0.15
16	1.09	1.10	1.16	0.67	0.09
32	1.12	1.20	1.30	0.46	0.07
64	1.12	1.24	1.29	0.32	0.06
128	1.11	1.16	1.25	0.25	0.06

Table 6.1: Experimental results with PT-SCOTCH on matrix BRGM. Each entry in the table is the ratio of a metric when using PT-SCOTCH versus serial SCOTCH.

Table 6.1 reports experimental results measured using PT-SCOTCH to compute the pivotal order on the BRGM matrix from the GRID-TLSE collection; the numbers in the table are normalized with respect to the sequential case. The following conclusions can be drawn from these results:

1. quality of the ordering: as shown by the columns reporting the factors size, front size and number of floating-point operations, the quality of the ordering does not degrade with the degree of parallelism and it is comparable to what is obtained with a sequential ordering tool.
2. performance: the parallelization of the analysis phase provides significant reduction of the cost of this phase, when more than 8 processors are used.
3. memory consumption: the memory requirement per processor in the analysis phase can be considerably reduced thanks to parallelization. Because memory consumption is strictly dependent on the number of nonzeros-per-row in the matrix, higher benefits can be expected for denser problems.

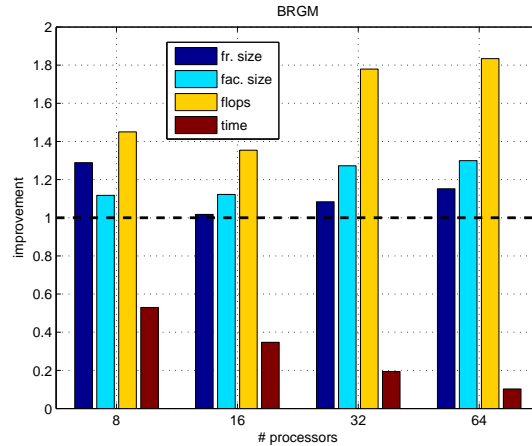


Figure 6.2: Comparison between PT-SCOTCH and ParMETIS: (maximum front size, factor size, flops, time) of ParMetis divided by PT-Scotch.

Figure 6.2 shows the performance of PT-SCOTCH with respect to ParMETIS: although PT-SCOTCH is slower than ParMETIS, it provides a considerably better quality ordering. Moreover, because the quality of the ordering provided by PT-SCOTCH does not vary much with the degree of parallelism (see Table 6.1) it is easy to conclude that ParMETIS provides worse quality orderings when more processors are added to the computation.

Remark that the symbolic analysis corresponding to the top of the tree is currently not parallelized. Although this has not appeared as a bottleneck yet thanks to the use of quotient graphs, it may become necessary to also parallelize the top of the tree (as was done in [100]) in the future when working directly on the uncompressed graphs of huge matrices.

6.2 64-bit addressing

Because of the various efforts aiming at processing larger problems (parallel analysis, out-of-core storage, better scheduling), and thanks to the increasing memory available on today’s computers, it became critical to rely on 64-bit addressing to address that memory. Still, since our implementation is mainly done in Fortran, standard integers of typical size 4 are used to address large arrays. In practice, a large array is allocated on each processor to hold the factors, contribution blocks, and active frontal matrix (see Section 1.3.3), as this allows for various optimizations and reductions of the memory usage (typically, locality in the management of the stack of contribution blocks, in-place assemblies). With signed 32-bit integers, even on 64-bit machines, the size of this array is limited to 2.147 billion ($2^{31} - 1$) entries. Each entry is either a float, a double, a complex or a double complex scalar. In the case of doubles for example, the maximum memory per workarray is 16 GB, which is definitely too small for many problems and justifies the use of 64-bit integers. Remark also that even with dynamic allocation, 64-bit integers are also necessary when the order of a dense frontal matrix is bigger than 46340, which is something frequent on large problems.

However, 32-bit integers had to be kept at the interface level (backward compatibility), for MPI and BLAS calls, and for all what concerns matrix indices. So the work consisted in separating the integers into two classes, the ones that should stay standard and the ones that should become 64-bit integers, including all types of integers that could possibly overflow on large problems. For example, many statistics returned to the user must rely on 64-bit internal computations, some integers in the minimum degree routines are 64-bit integers, and Fortran/C interfacing of 64-bit integers had to be done in a portable way. Doing this work on more than 200 000 lines of code has been time consuming, but was necessary for experimenting and validating other research features on real-life large-scale problems. It is also useful to applications by being

available since release 4.9 of MUMPS (July 2009).

Whereas this approach should be fine for sparse matrices of order up to 2 billion (32-bit integers are still used to store matrix indices, between 1 and the order of the matrix), 32-bit vs. 64-bit integers also become an issue in third-party codes. For example, since release R2006b of Matlab, integers for sparse matrices are all 64-bit on 64-bit systems (indices and pointers). While this non-backward compatibility can be arranged at the interface level between Matlab and our solver, ordering packages also start requiring 64-bit integers internally, even though the matrix dimension does not exceed 2 billions. In such cases, all integers should simply be 64-bit integers and this can be arranged with compiler flags and macros.

6.3 Forward elimination during factorization

The parallel forward and backward solutions were described in Algorithms 2.4 and 2.5, Section 2.4. The forward elimination accesses the factors of the tree from bottom to top (from leaves to roots), similar to the factorization. When factorizing the final root node, factors associated with the leaves are at high levels of the memory hierarchy, possibly on disk, so that it can be particularly inefficient to access them again right at the beginning of the forward elimination phase: all L and U factors during the factorization, then all the L factors for forward elimination, then all U factors again for the backward substitution.

When the right-hand side vector b only has a few columns that can be processed in a single block, the idea of forward elimination during factorization consists in performing all the operations from the forward elimination during the factorization, at the moment when the L factors are available in close memory. In fact, it can be viewed as appending a column on the right of matrix A , to which all update operations are applied. Looking back at Algorithm 1.2, line 3 then becomes:

$$A(i+1:n, i+1:n+1) = A(i+1:n, i+1:n+1) - A(i+1:n, i) \times A(i, i+1:n+1),$$

so that only the backward substitution algorithm remains to be done on the intermediate solution $A(1:n, n+1)$.

In the sparse case with a multifrontal approach, the data corresponding to right-hand sides appear directly in the frontal matrices (rather than using a separate data structure). In the unsymmetric case, those data appear as extra columns. In the symmetric case, where only the lower triangular part of the fronts is significant, it is more natural to use extra rows. Since all our approaches and memory management rely on square fronts, extra rows (resp. columns) are currently also allocated in the unsymmetric (resp. symmetric) case, although their content is not significant and computations are avoided for them. Both type 1 and type 2 factorization kernels generalize naturally, with some exceptions: the pivot stability test does not take into account the right-hand sides when looking for the maximum element.

On exit from the factorization, the intermediate solution y of the $Ly = b$ forward substitution phase is simply part of the frontal matrices and represents an extra column (considering the unsymmetric case). It is thus naturally distributed over the processors. The forward elimination can then be skipped during the solve phase. Let us consider the backward substitution; one can write:

$$\begin{pmatrix} U & y \\ & 0 \end{pmatrix} \times \begin{pmatrix} x \\ -1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad (6.1)$$

In other words, starting from the value of solution -1 for variables $n+1$ (and $n+2, \dots, n+nb_{rhs}$ in case of multiple right-hand sides), the standard backward solve algorithm can then be applied. Those -1 values are not stored in practice, inducing only minor modifications to the backward substitution algorithm.

Special case of a 2D block cyclic root in the assembly tree

In the case of a 2D block cyclic root node to be processed by ScaLAPACK, some difficulties arise. For example in ScaLAPACK, in order to perform solves, the block of right-hand sides is not allowed to start in the middle of a block of the 2D grid of processors. Thus, it is not possible to let the block of right-hand sides be assembled naturally, as a simple extension (extra rows or columns) of the front of the root node.

It has therefore to be built in a separate 2D block cyclic data structure. Furthermore, the distribution of the right-hand sides must be compatible with the one of the matrix in order to use ScaLAPACK routines to build a solution. More precisely, the right-hand side should be a block of columns, 2D block cyclic, with the same 2D block-cyclic distribution as the root matrix. Whereas in the unsymmetric case, the existing communication schemes can be generalized to send extra columns from the children of the root to the correct processors of the 2D cyclic grid of processors, this is not the case in the symmetric case, where the necessity to transpose blocks of rows (in the children of the root) into blocks of columns (in the root) implies different destination processors than the natural ones for rows of children and a new communication scheme. The associated developments were much heavier than expected (specifications, development, validation) because it was more useful to have a longer term investment avoiding

- a pre-assembly of the block of right-hand sides that would be stored by rows in the grid of processors, which would then have needed to be explicitly transposed before the solve, with an extra cost both in terms of temporary memory and communication.
- an increase in the number of assembly messages that would have been at the cost of amplifying the problems of latency that were sometimes observed when building the 2D block cyclic frontal matrix of a root with many children; for that, we take advantage of the existing messages used to assemble frontal matrix data from the children to the root, even though the data to send for right-hand sides are intrinsically of another nature and distributed differently.

The drawback is a more complex code more difficult to maintain, but with an interest for performance. Remark that this mechanism is also applied for symmetric indefinite matrices, where frontal matrices of our parallel multifrontal approach are symmetric, but where we use ScaLAPACK LU after an expansion of the root because of the absence of parallel LDL^T kernel on dense matrices.

Finally, on the root node, in order to optimize the data access to the factors, it was decided to build the complete solution on the root node (forward and backward); this is because all factors are available in memory at the moment of the factorization of the 2D block cyclic root. Thus, factors of the root do not need to be loaded back during the solution phase. Furthermore, it is then possible to completely free the factors associated with the root (see also the paragraph “Options to discard factors” below).

Schur complement and partial condensation

As explained in Section 2.5, it is sometimes useful to compute a reduced or condensed right-hand side on a set of variables, for instance corresponding to the interface of a domain. In case of forward elimination during factorization, it is then natural to build this reduced right-hand side during the factorization. In case of a 2D block cyclic Schur, the reduced right-hand side will be naturally distributed (see previous paragraph) but can be centralized to be returned to the user application. In case of a type 1 front at the root (*i.e.*, processed by a single processor), both the Schur and the reduced right-hand side must be copied or sent from the frontal matrix of the root to the user data, in order to keep a simple API to the solver.

Option to discard factors

In the case of unsymmetric matrices where an LU factorization is computed, it is not necessary to keep the L factors during the factorization. Indeed, since the intermediate right-hand side y has already been computed during factorization, only U will be used for the backward substitution. This gains storage in the in-core case, and I/O in the out-of-core case.

In case of a Schur complement, all factors can even be discarded if the application only requires the solution of a problem on an interface, excluding the solution on the internal problem.

Finally, in case of a 2D block cyclic root, all factors corresponding to the root have been used during factorization and have been freed.

Performance

Thanks to the forward elimination during factorization, we observed that the cost of the solution phase can be divided by a factor 2 when the factors are in-core, without noticeable additional cost of the factorization. When factors of large matrices are out-of-core, one observes a better stability of the solve time thanks to this functionality. Table 6.2 illustrates the gains obtained on a few matrices on one core of a laptop with an Intel Core 2 Duo P9700 at 2.8 GHz with 8GBytes of memory, for in-core and out-of-core executions. In the out-of-core case, we observe that I/O time varies a lot from one run to another, especially with system I/O where we have no control on the buffering mechanisms from the operating system. However, the gains are significant in the out-of-core case. We would expect the gains to be larger in the out-of-core case (in both serial and parallel executions), on very large matrices or with direct I/O , where system buffers will not be able to interfere with the out-of-core mechanisms. This deserves more experimentation. Interesting performance gains, although not quantified precisely, have also been reported by users who activated this feature.

Matrix de test	Measured phase (time in seconds)	Condensation during factorization	
		OFF	ON
CONESHL2 (factor size: 2.2 GB)	in-core factorization	142.0	142.4
	in-core solve	1.4	0.7
CONESHL (factor size: 6.4 GB)	out-of-core factorization (run 1)	818.1	822.4
	out-of-core factorization (run 2)	821.6	826.0
	out-of-core solve (run 1)	78.7 (CPU: 13.6)	52.9 (CPU: 6.2)
	out-of-core solve (run 2)	142.2 (CPU: 12.7)	47.7 (CPU: 5.3)
GRID 11pt (factor size: 23.3 GB)	out-of-core solve (run 1)	877.9 (CPU: 82.4)	556.5 (CPU: 47.1)
	out-of-core solve (run 2)	711.4 (CPU: 82.5)	497.0 (CPU: 46.7)

Table 6.2: Effects of the condensation functionality on the factorization and solve step of symmetric problems on one core of an Intel Core 2 Duo P9700 at 2.8 GHz with 8 GB of memory. System buffers are used in the out-of-core runs.

6.4 Memory and performance improvements of the solve algorithms (forward and backward substitutions)

Before reading this section, the reader should be familiar with the algorithms from Section 2.4, and with the modifications described in Section 2.5, on which it depends. We first describe the work done aiming at improving memory usage and locality, then describe the practical impact before a short discussion.

6.4.1 Reduction of workspace and locality issues

In Algorithms 2.4 and 2.5, Wb and $Wsol$ are workspaces of size n used in the forward and backward substitutions, respectively¹, whereas the workspace $WRHS$ introduced in Section 2.5 is a workspace of average size $\frac{n}{nprocs}$ which scales with the number of processors and contains on each processor data corresponding to the fully summed part of the fronts owned by that processor. When dealing with several right-hand sides, those are processed by blocks of arbitrary size b (see Algorithm 2.6), such that all those sizes are multiplied by b columns.

In case the right-hand side is sparse, or in case only a subset of entries of the solution is required (*e.g.*, computation of entries of the inverse, see [30]), those workarrays are much larger than needed: most of the rows are zero and are not even accessed. This is also the case in parallel executions, where Wb and $Wsol$ do not scale with the number of MPI processes and prevent increasing the blocksize b . Unfortunately, having

¹See also the modifications of these algorithms described in Section 2.5.

a large-enough value of the block size b is critical for obtaining good BLAS 3 performance in an in-core environment, and is even more critical in an out-of-core environment where the cost of accessing the factors on disk often dominates the solve time.

In order to get rid of Wb and $Wsol$, one should only allocate the useful workspace. In other words, variables that are not touched during the algorithm should not appear in Wb or $Wsol$, similar to what was done for WRHS to access variables corresponding to the pivot block of each front. Since Wb and $Wsol$ are accessed for variables between 1 and n , such an approach requires indirections. The idea we have retained is to fully suppress Wb and $Wsol$ and to extend WRHS, in order to include all variables possibly touched by a processor into that workarray. In the general unsymmetric case, because the row and column index lists of the pivot block may be unsymmetric in the presence of numerical difficulties (off-diagonal pivoting and delayed pivots), the indirections for the rows differ from the indirections for the columns. We illustrate this on a simple example on the tree of Figure 6.3, mapped on two processors P0 and P1. Remark that the discussion is for type 1 nodes but it would be the same for type 2 nodes since only the master processes access Wb and $Wsol$ in the solve algorithms.

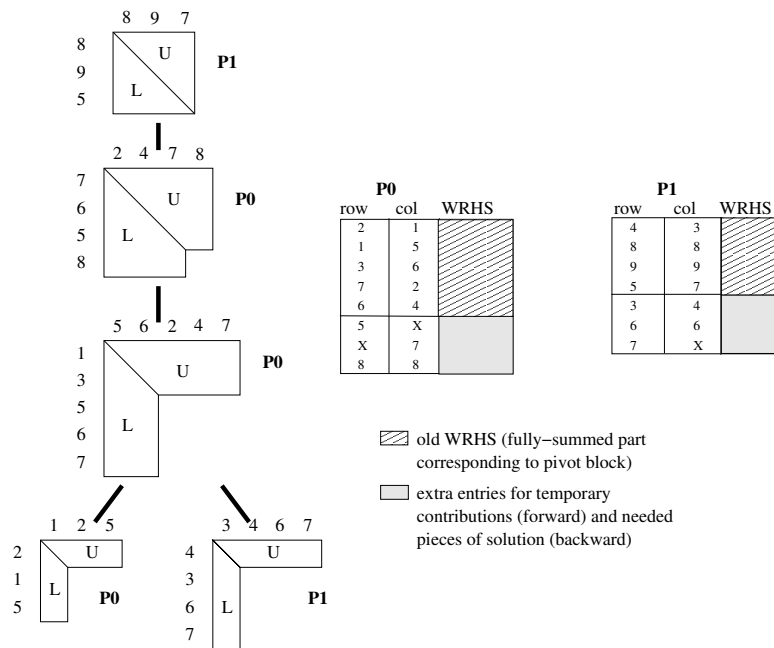


Figure 6.3: Example tree of factors mapped on 2 processors. The mapping of the masters is indicated for each node (P0, P1). Because of delayed pivots, the structure of the pivot block has some unsymmetry in it; the structure of the nodes of the original tree if no pivoting had occurred was 125; 3467; 567; 78; 89 (left-to-right, bottom to top, underlined variables correspond to fully-summed rows/columns and would have been eliminated at each node in the absence of numerical difficulties). WRHS is of size 8 on P0 and of size 7 on P1. The columns "row" and "col" correspond to the indirections to access WRHS: for instance on P0, POSinWRHS_row(7)=POSinWRHS_col(2)=4 means that row variable 7 and column variable 2 appear at position 4 of WRHS. An 'X' indicates that the corresponding entry is not used.

Before discussing the modifications of the aforementioned algorithms, we explain how to build the indirection arrays, which we call POSinWRHS_row and POSinWRHS_col. Except when solving the transposed system $A^T x = b$, POSinWRHS_row is used with row indices of the fronts during the forward elimination, and corresponds to positions in the right-hand side vector, whereas POSinWRHS_col is used with column indices of the fronts during the backward substitution and corresponds to positions in the solution. The procedure to build those indirections is given in Algorithm 6.1. In the first part of that algorithm, the indirections

corresponding to the fully summed block are computed. In the second part, the indirections corresponding to other variables are computed. At the end, m contains the size of the workarray `WRHS`. Notice the separate treatment of rows and columns in the second part of the algorithm, working on variables in the off-diagonal blocks of the factors: sometimes, m is incremented because of a row variable, sometimes because of a column variable, and sometimes because of both. It is in fact possible to have one loop for the rows and another one for the columns and keep the maximum of the two values of m obtained at the end to dimension the workarray `WRHS`. This way, `WRHS` will be slightly smaller (7 instead of 8 on $P0$ in the example of Figure 6.3, by suppressing the two unused positions denoted by 'X'). Compared to Section 2.4, `WRHS` does not scale perfectly with the number of processors because of the extra entries added. However, we will see that the memory scalability of `WRHS` is still very good in practice and that a huge memory gain comes from the fact that Wb (during forward elimination) and $Wsol$ (during backward elimination) have disappeared. Remark that at line 3, the order in which the nodes are visited can be arbitrary. However, visiting the nodes in the same order as they will be visited during the solve algorithm (which is close to a postorder traversal) leads to a better memory locality during the solve algorithm. This will be illustrated with experiments later in this section.

We now give in Algorithms 6.2 and 6.3 the modified algorithms for the forward and backward substitutions. At line 3 of Algorithm 6.2, the right-hand side must be distributed to the processors. Assuming that the right-hand side is initially on processor with rank 0, each processor loops on the row variables of its pivot block, and asks for rows of the original right-hand side to processor 0, which responds with the requested rows. On reception, `POSinWRHS_row` is used to store the result in `WRHS`. In the serial case, this step is mainly a permutation, which was not needed when Wb was used; in parallel, extra indirections are needed. On the other hand, we have a much better locality afterwards, during the forward elimination itself.

In Algorithm 6.3 (backward substitution), accesses to $Wsol$ have been replaced by accesses to `WRHS`. At line 13, rows of `WRHS` corresponding to variables in pivot blocks (upper part of the array `WRHS`) represent the solution. In both the forward and backward substitution algorithms, we have kept the indirection `POSinWRHS(\mathcal{N})` introduced in Section 2.5. However, we have the equality `POSinWRHS(\mathcal{N})=POSinWRHS_row(i)=POSinWRHS_col(j)`, where i (resp. j) is the first row index (resp. column index) of node \mathcal{N} . Therefore, `POSinWRHS` is suppressed in practice and should be interpreted as accesses to `POSinWRHS_row/col`.

Since rows of $Wtmp1$ (in the forward elimination) and x_1, y_1 (in the backward substitution) correspond to variables which are contiguous in `WRHS`, $Wtmp1, x_1$ and y_1 could normally also disappear, gaining some memory copies, and one could work directly in the array `WRHS`. An implementation problem related to the diagonal solve in the LDL^T case combined with nodes of type 2 makes this apparently simple modification more complicated than it looks, but it is on the “TODO” list!

In case of reduced right-hand sides (see Section 2.5), `WRHS` has to be kept between the forward elimination and the backward elimination for all columns of the right-hand side, whereas it is otherwise allocated for only one block of right-hand side. In that case, the cost of suppressing the temporary arrays Wb and $Wsol$ (for each block of right-hand sides) is the non perfect scalability of `WRHS` (whose number of columns is the total number of columns of the right-hand side in that case), inducing a memory usage slightly larger than before on return from the forward elimination algorithm. For reasonable tree mappings, this cost is negligible.

6.4.2 Performance

We now report on some experiments that were done while studying the performance of the solve phase on large matrices with multiple dense right-hand sides. The number of right-hand sides and the block size are the same and are equal to 128. Two matrices are used to illustrate the results:

- MATRIX2D: the discretization of a 5-point finite-difference operator on a 1000x1000 grid
- AUDI: a 3D model of a car, with 943695 equations, available from the TLSE and University of Florida collections (Parasol set of test problems).

Figure 6.4 shows the gains obtained thanks to the algorithmic improvements presented in the previous subsection in a sequential environment for matrix MATRIX2D. The terms used in the figure are explained below and will also be used in other results:

```

m ← 0; initially POSinWHRS_row=POSinWRHS_col=0
{Loop on variables in pivot block}
for all node  $\mathcal{N}$  mapped on Myid do
  {npiv $\mathcal{N}$  is the number of eliminated pivots }
  {nfront $\mathcal{N}$  is the order of the front }
  {row_list $\mathcal{N}$  is the list of row indices }
  {col_list $\mathcal{N}$  is the list of column indices }
  for k=1 to npiv do
    i ← row_list $\mathcal{N}$ (k); j ← col_list $\mathcal{N}$ (k)
    m ← m+1
    POSinWRHS_row(i)=m
    POSinWRHS_col(j)=m
  end for
end for
{Loop on variables in off-diagonal blocks}
for all node  $\mathcal{N}$  mapped on Myid do
  for k=npiv $\mathcal{N}$ +1 to nfront $\mathcal{N}$  do
    i ← row_list $\mathcal{N}$ (k); j ← col_list $\mathcal{N}$ (k)
    if POSinWRHS_row(i) = 0 or POSinWRHS_col(j) = 0 then
      m ← m+1
      if POSinWRHS_row(i) = 0 then
        POSinWRHS_row(i) ← m
      end if
      if POSinWRHS_col(j) = 0 then
        POSinWRHS_col(j) ← m
      end if
    end if
  end for
end for

```

Algorithm 6.1: Initialization of the indirections arrays for the modified solve algorithm.

```

1: Main Algorithm (forward elimination):
2: Initialize a pool with the leaf nodes mapped on Myid
3: Store into WRHS rows of the right-hand side corresponding to variables in the pivot block of nodes
   mapped on Myid. Set other rows of WRHS to 0.
4: while Termination not detected do
5:   if message is available then
6:     Process the message
7:   else if pool is not empty then
8:     Extract a node  $\mathcal{N}$  from the pool
9:     Fwd_Process_node( $\mathcal{N}$ )
10:  end if
11: end while
12:
13: Fwd_Process_node( $\mathcal{N}$ )
14: { $L_{11}$  and  $L_{21}$  are the  $L$  factors of  $\mathcal{N}$  }
15: { $P_{parent}$  be the process owning the master of the parent of  $\mathcal{N}$  }
16:  $Wtmp1 \leftarrow$  Rows of WRHS corresponding to the pivot block of  $\mathcal{N}$ , starting at position  $POSinWRHS(\mathcal{N})$ 
17:  $Wtmp1 \leftarrow L_{11}^{-1} \times Wtmp1$ 
18: Copy rows of  $Wtmp1$  back into  $WRHS(POSinWRHS(\mathcal{N}))$ 
19: Gather in  $Wtmp2$  rows of WRHS corresponding to row indices of  $L_{21}$  (use  $POSinWRHS\_row$ )
20: Reset the corresponding rows of WRHS to zero
21: if  $\mathcal{N}$  is of Type 1 then
22:    $Wtmp2 = Wtmp2 - L_{21} \times Wtmp1$ 
23:   Send the resulting contribution ( $Wtmp2$ ) to  $P_{parent}$ 
24: else if  $\mathcal{N}$  is of Type 2 then
25:   for all slave  $Islave$  of  $\mathcal{N}$  do
26:     Send  $Wtmp1$  together with the rows of  $Wtmp2$  corresponding to rows of  $L_{21}$  owned by  $Islave$  to
       the process in charge of  $Islave$ 
27:   end for
28: end if
29:
30: On reception of  $Wtmp1 +$  rows of  $Wtmp2$  by a slave
31: Multiply rows of  $L_{21}$  owned by the slave by  $Wtmp1$  and subtract the result from the received rows of
    $Wtmp2$ 
32: Send the resulting contribution to  $P_{parent}$ 
33:
34: On reception of a contribution corresponding to  $\mathcal{N}$  by  $P_{parent}$ 
35: Assemble the contribution into WRHS (Scatter using  $POSinWRHS\_row$ )
36: if all contributions for node  $\mathcal{N}$  have been received by  $P_{parent}$  then
37:   Insert parent of  $\mathcal{N}$  into the pool of ready nodes
38: end if

```

Algorithm 6.2: Forward elimination algorithm with reduced memory.

- 1: **Main Algorithm (backward substitution):**
- 2: On input: WRHS is the workspace obtained on output from Algorithm 2.4
- 3: POSinWRHS_col is used to access rows of WRHS
- 4: Initialize the pool with the roots mapped on *Myid*
- 5: **while** Termination not detected **do**
- 6: **if** message is available **then**
- 7: Process the message
- 8: **else if** pool is not empty **then**
- 9: Extract a node \mathcal{N} from the pool
- 10: **Bwd_Process_node**(\mathcal{N})
- 11: **end if**
- 12: **end while**
- 13: Gather solution from distributed WRHS arrays to the host (or keep it distributed)
- 14: Return solution to the user
- 15:
- 16: **Bwd_Process_node**(\mathcal{N})
- 17: $x_2 \leftarrow$ known entries of solution corresponding to columns of U_{12} (gather from WRHS, using POSinWRHS_col)
- 18: **if** \mathcal{N} is of type 1 **then**
- 19: $y_1 \leftarrow$ entries of WRHS corresponding to variables in the pivot block (copy from position POSinWRHS(\mathcal{N}))
- 20: Solve $U_{11}x_1 = y_1 - U_{12}x_2$ for x_1
- 21: Save x_1 in WRHS (copy at position POSinWRHS(\mathcal{N}))
- 22: Send partial solution x_1, x_2 to masters of children nodes
- 23: **else if** \mathcal{N} is of type 2 **then**
- 24: Send (distribute) entries of x_2 to the slaves, according to their structure
- 25: **end if**
- 26:
- 27: **On reception of x_1, x_2 , sent by the master of node \mathcal{N}**
- 28: Update my view of the solution (scatter into WRHS, using POSinWRHS_col)
- 29: Insert children of \mathcal{N} mapped on *Myid* into the local pool
- 30:
- 31: **On reception of parts of x_2 by a slave of \mathcal{N}**
- 32: Multiply the part of U_{12} mapped on *Myid* by the piece of x_2 just received
- 33: Send the negative of the result back to the master process of \mathcal{N}
- 34:
- 35: **On reception of a portion of $-U_{12}x_2$ from a slave by a master for node \mathcal{N}**
- 36: Add it into WRHS, starting at position POSinWRHS(\mathcal{N})
- 37: **if** this is the last update (all slaves sent their part) **then**
- 38: $y_1 \leftarrow$ entries of Wb corresponding to variables in the pivot block of U_{11} (row list, gather)
- 39: Solve $U_{11}x_1 = y_1$ for x_1
- 40: Save x_1 in WRHS (Copy at position POSinWRHS(\mathcal{N}))
- 41: Send partial solution x_1, x_2 to masters of children nodes
- 42: **end if**

Algorithm 6.3: Backward substitution algorithm with reduced memory.

4.10.0 corresponds to the original algorithm, where data on each MPI process (1 in the sequential case) are stored in a workspace of 1 million rows and 128 columns, using a column-major storage. Thus, the workspace increases linearly with the number of processors and is critical².

trunk is the approach described by Algorithms 6.2 and 6.3, with the indirections arrays `POSinWRHS_row` and `POSinWRHS_col`. Despite the new indirection to access data in `WRHS`, we observe a gain of almost 2 on the sequential performance of `MATRIX2D` thanks to the fact that access to data in `WRHS` is now done with a significantly better locality of reference.

trunk+postorder means that on top of this, we force data corresponding to variables in the pivot block in `WRHS` to be organized following a postorder, which is the order in which nodes in the tree will be accessed during the triangular substitutions. This is done by forcing a postorder at line 3 of Algorithm 6.1.

trunk+postorder+byrows means that the column-major storage for `WRHS` is replaced by a row-major storage: this way, `Wtmp1` and x_1, y_1 in the above algorithms correspond to a block of `WRHS` that is fully contiguous in memory.

trunk+postorder+byrows+best amalg. finally corresponds to the best execution time when the amount of amalgamation (see end of Section 1.1.6) has been tuned for the solve phase.

In Figure 6.5, where only the performance of the initial and final algorithms are shown, we observe that there are also significant gains in the parallel case for a 3D problem (matrix `AUDI`).

In Table 6.3, we show the memory usage of the solve phase with respect to the initial algorithm. In an out-of-core context, the memory reduction is very significant as soon as more than one processor is used. This is because the workarray `WRHS`, which accounts for a large part of the memory, now scales reasonably well with the number of processors (its non perfect scalability is due to some non-fully summed variables: bottom part of `WRHS` in the example of Figure 6.3).

# procs	4.10.0	New	WRHS
1	1773.1	1751.9	966.3
10	1145.7	291.7	108.5
20	1124.6	226.1	63.5

Table 6.3: Comparison of initial (4.10.0) and new solve algorithm in terms of memory (MBytes) with 1, 10, and 20 processes, on test case `AUDI`, when factors are out-of-core. In the in-core case, the memory is dominated by the factors.

Finally, we give in Table 6.4 the contents of the memory for in-core executions. We observe that `WRHS` scales as before, but remains small compared to the huge workspace required for the factorization for the `AUDI` matrix. This is not the case for `MATRIX2D`, where the memory scaling of `WRHS` is very important to limit the total memory usage.

6.4.3 Discussion

The cost of the solve phase is critical in some applications. During the concluding discussion of the last MUMPS Users'days in April 2010, the performance and memory usage of the solve phase appeared to be a bottleneck to several users, especially in applications which spend most of their time in the solve and not in the factorization (thousands of right-hand sides, simultaneous or successive). The preliminary work described above is a first step towards optimizing the behaviour of the solve step: starting from a problem of memory scalability with respect to numbers of processors, the memory reduction gave a better potential to exploit memory locality. This better memory locality resulted in significantly improved performance. In parallel, we have seen that the scaling of the solve phase is reasonably good. Still, the type of parallelization and task mapping inherited from the factorization might not be optimal for the solve. More generally, parameters

²It is even more critical with sparse right-hand sides

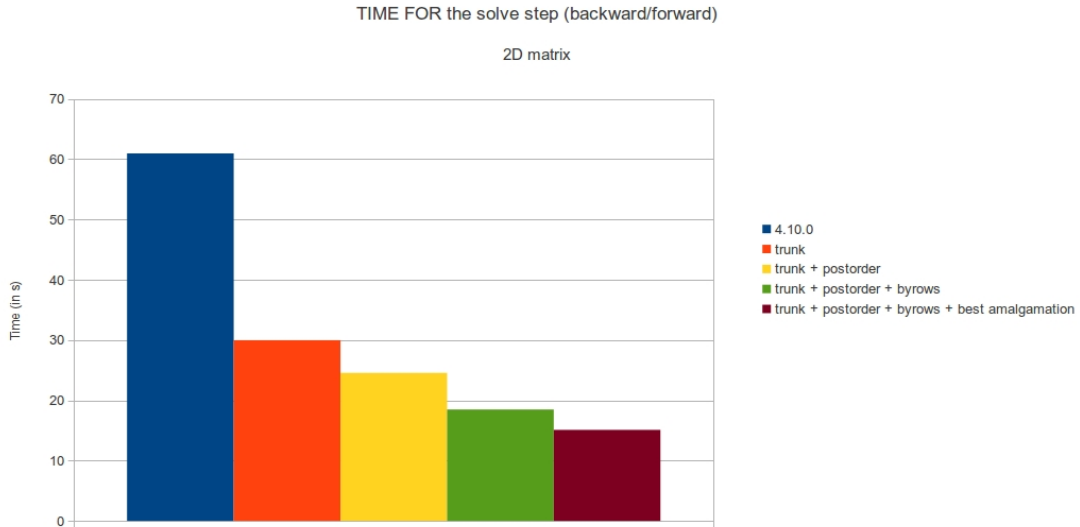


Figure 6.4: Performance improvements with respect to initial algorithm (4.10.0) of the serial performance on the test case MATRIX2D.

Number of processors	Total memory	Workarrays for facto		Workarray for solve WRHS	Other data (tree, ...)
		real	integer		
AUDI					
1	26999	25583.12	80.68	966.34	368.86
10	3508	3320.46	9.72	108.51	69.30
20	1801	1679.07	5.14	63.52	53.28
MATRIX2D					
1	2517	1316.24	72.01	1024.00	104.76
10	320	156.75	8.31	104.37	50.58
20	187	81.38	4.21	53.16	48.25

Table 6.4: Peak memory usage (MBytes) of the new solve algorithm when factors are kept in core memory, for matrices AUDI and MATRIX2D.

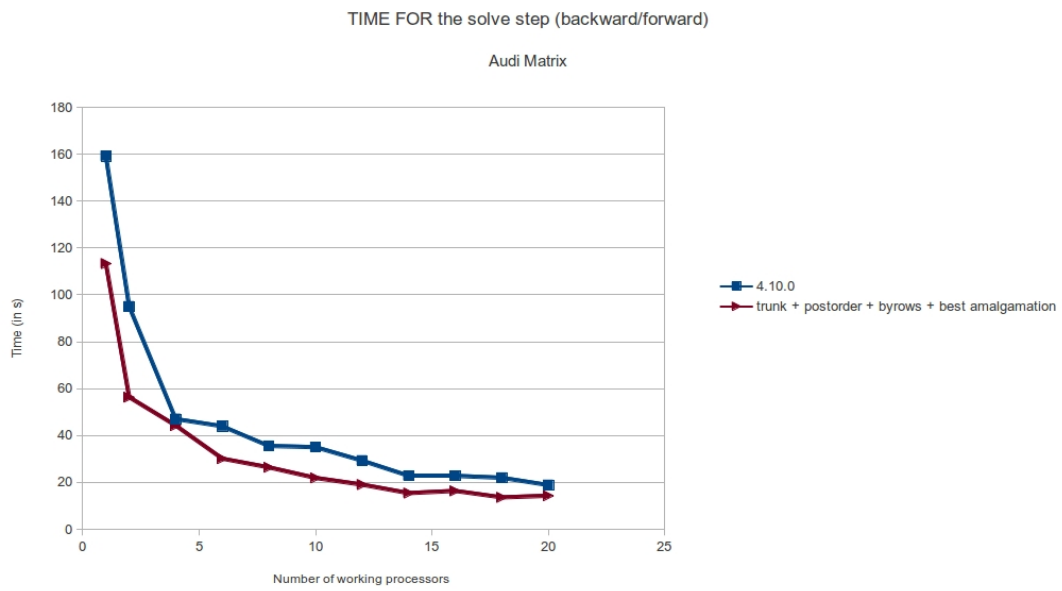


Figure 6.5: Performance improvements with respect to initial algorithm (4.10.0) of the serial and parallel performance on the test case AUDI.

from the factorization may not be the best for the solve. For instance, we observed significant gains in the solve by performing more amalgamation and so limiting the number of nodes in the tree compared to what is done for the factorization. When dealing with machines with hundreds of thousands of cores, the factorization phase becomes a huge challenge but the solve algorithms of sparse direct solvers could be even more challenging: the solve phase seems more communication- and memory-bound, with smaller attainable GFlops/s rates. Still, it deserves a lot of attention and could benefit from specific mapping and scheduling algorithms, together with a multithreaded approach to parallelism inside each MPI process in order to better exploit multicore systems.

Finally, the memory reduction described in this section was a critical first step in order to process much bigger problems in the context of sparse right-hand sides and for the computation of entries of the inverse of a matrix, see [16] and [30] for more information on those aspects.

6.5 Communication buffers

Because we cannot fully rely on internal MPI buffers whose size is difficult to control, and because of memory management issues, we have seen in Chapter 2 that we must manage our own communication buffers. The send buffer is cyclic and each time a message must be sent, the corresponding data are built in that cyclic buffer, and sent asynchronously with an associated MPI request. When the MPI request is free, this means that the corresponding zone in the buffer can be freed.

It appeared that the size of communication buffers necessary during the factorization became critical compared to the total memory usage. This is even more true in an out-of-core context, where the working memory is significantly reduced thanks to the possibility of using disk storage for the factor matrix. The reason for these large buffers arises from the fact that this parallel multifrontal method requires large messages, compared, for example to a supernodal approach like SuperLU_{dist} (while the overall communication volume is comparable, see [28]). In order to be able to send several messages asynchronously without waiting for the actual reception, the send buffer needs to be significantly larger than the largest message estimated during the analysis phase. In practice we use to have a send buffer around twice larger than the largest estimated message.

The messages whose size might be critical in our approach are the following:

- messages holding entire contribution blocks, sent from a process in charge of a type 1 child in the tree to a process in charge of the corresponding type 2 parent.
- messages corresponding to pieces of contributions, when either the child or the parent frontal matrix is of type 2, that is, is processed by more than one process. This type of messages covers two cases, depending on the rows of the considered block:
 - rows corresponding to numerical problems, whose elimination must be delayed (see Figure 1.14 and associated comments), sent from the master of a child to the master of a parent.
 - other rows, sent by a slave of a child to either a slave or the master of the parent.

Furthermore, this type of messages covers both the case where the parent is of type 2 (1D pipelined factorization) and the case of a 2D block cyclic distribution of the frontal matrix processed with ScaLAPACK [53].

The work consisted in authorizing the largest messages to be split into smaller ones, as described below. Although one would hope that rows corresponding to numerical problems are generally of limited size, we have seen cases where they also had to be splitted. The communication schemes for the three types of messages above have thus been modified. However, messages related to 1D pipelined factorizations have not been modified: they are already controlled by the blocking factor of the pipelined factorization and their (much smaller size) is now used as a lower bound of the new buffer sizes.

The approach applied to send messages possibly in several pieces is described further in Algorithm 6.4. The size of the send buffer has been estimated during the analysis based on the size of the messages that are

not split and on a *reasonable* size avoiding when splitting messages in too many pieces. Remark that the size of the reception buffer is also reduced.

- 1: **Initial state:** we assume that k rows among n are already sent; initially $k = 0$; r is the size of the reception buffer; the size of the send buffer is larger than r but part of it may be occupied by messages being sent and not freed yet.
- 2: Try to free requests in send buffer
- 3: Compute the largest contiguous size s available in send buffer
- 4: Estimate the number of rows that can be sent depending on:
 - the size of the necessary symbolic information in the message and of the header,
 - the number of rows already sent (for example a quadratic equation needs to be solved in the symmetric case, as each block of rows has a trapezoid shape),
 - what remains to be sent.
- 5: Estimate the size of the corresponding message (MPI_PACK_SIZE) and modify the above estimate, if necessary
- 6: **if** message size is big enough or message contains the last rows to be sent **then**
- 7: Build the message in the buffer and send it
- 8: **end if**
- 9: Return a status which can be, depending on cases:
 - ALLSENT: execution can continue, all rows were sent
 - TRY-AGAIN: nothing was sent, a new attempt should be made; in this case, the process should try to receive messages in order to avoid the deadlock situation where all processes try to send messages without anybody doing receptions.

Algorithm 6.4: Sending contribution blocks in several pieces.

Array 6.5 gives the size of the communication buffers for the original and new buffers. Whereas column *Original* corresponding to normal unsplit messages requires significant memory for communication buffers, a significant memory gain is obtained thanks to these modifications. The cost is a more complex code and a possible higher cost due to stronger synchronizations, since the receiver must receive pieces of a message before the next piece can be sent. However, no large degradation of performance was observed with this functionality, so that smaller communication buffers are now used systematically, both in the in-core and out-of-core approaches.

Matrix	Communication schemes	
	<i>Original</i>	<i>Modified</i>
AUDIKW_1	264	4.2
CONESHL_MOD	66	3.7
CONV3D64	286	16.1
ULTRASOUND80	75	8.2

Table 6.5: Average size per processor of the communication buffers (in Mbytes) on 32 processors. The overall core memory requirements for CONV3D64 are 1264 MBytes for the in-core approach, and 800 MBytes when factors are stored out-of-core. Memory for communication buffers was thus far from negligible in the original version.

6.6 Facing the multicore evolutions

In the recent years, with the advent of multicore machines, interest has shifted towards multithreading existing software in order to maximize utilization of all the available cores. Rather than writing a new code for multicore machines, we aim at adapting the parallel multifrontal solver `MUMPS` to take better advantage of multicore architectures. Although `MUMPS` was inspired by a shared-memory code [17], the parallelism is mainly based on message-passing (MPI). We believe that the MPI layer should be kept in order to naturally address NUMA architectures, memory locality, and distributed-memory, but that mixing message-passing with threads is necessary to reach a better scalability. Specialized solvers aiming at addressing multicore machines have been the object of a lot of work, see [43, 47, 60, 117, 109, 115, 122, 153], for example.

Here, we describe some preliminary work to optimize the performance of the factorization phase of our approach on multicore systems, with the objective to mix the shared-memory programming model with the distributed-memory programming model in order to scale to huge numbers of cores. We rely on the OpenMP [4] standard for multithreading, avoiding the direct use of threads.

6.6.1 Multithreaded BLAS libraries and OpenMP: preliminary experiments based on the fork-join model

In this section, we report on the work done to determine costly parts of the factorization, and use multithreading techniques in those parts. We will illustrate some behaviours using a few test matrices, the characteristics of which are summarized in Table 6.6.

Matrix	Order	Nonzeros	Symmetry	Origin
AMAT30	134335	975205	unsymmetric	French-Israeli Multicomputing project
BOXCAV	544932	3661960	symmetric	ANR Solstice project
DIMES	55640	13929296	unsymmetric complex	Multipole solver [97]
ULTRASOUND80	531441	33076161	unsymmetric	M. Sosonkina
HALTERE	1288825	10476775	symmetric complex	ANR Solstice project
THREAD	29736	4444880	symmetric	PARASOL collection

Table 6.6: Test matrices used in Section 6.6.1. Although complex, matrix `HALTERE` is treated as if it was real (imaginary part is ignored).

We used the TAU profiling tool³ [157], and observed that most of the time spent during the factorization of `MUMPS` was due to the following portions of the code, that can be scaled thanks to multithreading, either using OpenMP directives or an external multithreaded library:

BLAS operations. `MUMPS` uses Level 1, 2 and 3 BLAS operations and, especially for large 3D problems, the factorization time is largely dominated by the time spent in BLAS calls. Using multithreaded BLAS libraries such as `GOTO`⁴, `ACML`⁵ from AMD, `ATLAS`⁶ or `MKL`⁷ from Intel, helps improving the performance in multithreaded environments. The `GOTO` library was configured using the flag `USE_OPENMP=1`, in order to allow for compatibility with OpenMP. Although we observed that `GOTO` is the fastest among the BLAS libraries tested, it could not be used since even with `USE_OPENMP=1` it still seemed to conflict with the other OpenMP regions. It seems that `GOTO` creates threads and keeps some threads active after the main thread returns to the calling application – perhaps this is why the performance of OpenMP regions outside BLAS deteriorates a lot. `ACML` perturbed the OpenMP regions only a little while `MKL` was found to be the most compatible with all the OpenMP regions. Therefore we use `MKL` in the following experiments.

³Available from www.cs.uoregon.edu/research/tau.

⁴www.cs.utexas.edu/users/flame/goto

⁵www.amd.com/acml

⁶math-atlas.sourceforge.net

⁷software.intel.com/en-us/intel-mkl/

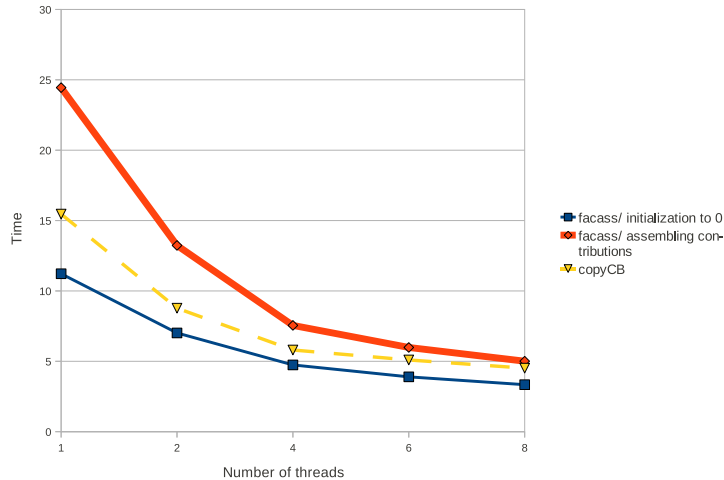


Figure 6.6: Assembly time as a function of the number of threads (testcase *AMAT30*), AMD Opteron 2218, 2.6 GHz. The figure shows separately initialization to 0 and assembly of contributions blocks (both are part of the assembly routine *facass*). Time for copying contribution blocks (*copyCB*) is also given for comparison.

Assembly operations. Such operations correspond to the assembly of contribution blocks (or Schur complements) from children into the frontal matrices of the parent nodes. In this phase the initializations to zero of the frontal matrices were also costly and could be multithreaded. The assembly of children contributions to parent nodes were also parallelized using OpenMP. Figure 6.6 shows the scaling of the assembly operations for the unsymmetric testcase *AMAT30*. We observed that multithreading is useful only for frontal matrices larger than 300 and should be avoided for smaller matrices. In the symmetric case, because of the difficulty of parallelizing efficiently small triangular loops, the parallelization is usually less efficient.

Stack operations. They consist in copying contribution blocks (*copyCB* operations) from the frontal matrix of a node to a separate zone, making them contiguous in memory. The factors are also compressed into contiguous space. Those copies are amenable to multithreading, as could already be seen in Figure 6.6 (*copyCB* operations). Figure 6.7 shows that the scaling strongly depends on the contribution block sizes. In this figure, each point gives the average time per call for a range of front sizes. For example, the value on the y-axis corresponding to 1500 on the x-axis represents the average time spent in the routine when the call is done with a contribution block in the range [1400,1600]. Our experience is that the potential for scaling of the stacking and assembly operations are similar: large blocks are needed to obtain reasonable speed-ups in these memory-bound operations. Unfortunately, the number of calls to those operations with small blocks is often much larger than the number of calls with large blocks, so that overall, on the range of matrices tested, the bad scalability with small blocks can still be a bottleneck when increasing the number of threads.

Pivot search operations. In some symmetric indefinite testcases, the pivot search operations were found to be costly. In the unsymmetric cases the pivot search is on rows whose elements are contiguous in memory, while in symmetric cases pivot search is mainly done within columns with a stride equal to the front size (this is because we use a row-major storage). The non-contiguous memory accesses in symmetric cases are the likely reason for this increased cost. Still, the pivot search operations were multithreaded using OpenMP reduction statements the results of which are shown in Table 6.7. In the testcase *THREAD* there was a reasonable gain in the two main pivot search loops, we call them RMAX and TMAX, however for most testcases there is a speed-down in these regions with the OpenMP directives. This is mostly caused within the regions of smaller loop sizes (< 300) or granularities which speed down

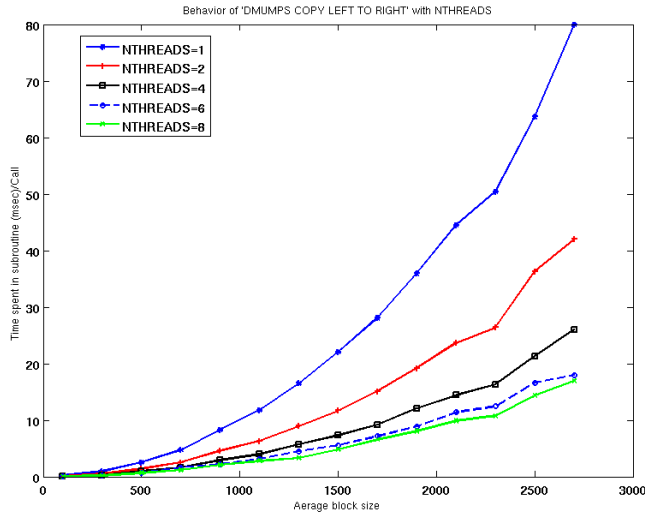


Figure 6.7: Behaviour of copyCB operations as a function of the number of threads. Each point on the x-axis refers to an interval of width 200. Testcase DIMES, Opteron.

distinctively as the number of threads is increased. We can stabilize this effect by using IF statements as a part of the OpenMP directive, which stops multithreading according to the IF statement. An example is shown in Table 6.8 where there is a disastrous slowdown without regulating the OpenMP region according to block sizes, and this effect is stabilized with addition of the IF statements.

	1 thread	2 threads	4 threads
Loop RMAX	3.35	2.33	2.07
Loop TMAX	1.65	1.11	0.84

Table 6.7: Performance (time in seconds) of the two main loops of the symmetric pivot search operations, testcase *THREAD*, Opteron.

We refer the reader to the technical report [54] for more experiments, case studies (symmetric positive definite, symmetric general, unsymmetric, Opteron-based and Nehalem-based processors), performance of the multithreaded solution phase, discussions about thread affinity, minimum granularity for parallelization with OpenMP.

We finish this section by showing the interest of mixing MPI and thread parallelism on the unsymmetric *ULTRASOUND80* matrix, on a 96-core machine from INRIA Bordeaux Sud-Ouest, with up to 4 threads per MPI process. We observe that it is interesting, from a performance point of view, to use 4 threads per MPI process. Furthermore, the OpenMP directives help a little, although the work consisting in inserting OpenMP directives has only been done in a few places (mainly type 1 nodes) and should also be done for assembly and stacking operations in type 2 nodes. Finally, let us remark that, for a given number of cores, it is better to use more threads and less MPI processes when memory usage is targeted. For example, with the same matrix *ULTRASOUND80*, the memory consumption is 8.97 GB, 11.0 GB, 11.3 GB, 13.3 GB when using, respectively, 1, 2, 4 and 8 MPI processes. Therefore, it is better in terms of memory usage to use 1

	1 thread	2 threads
Loop RMAX(without IF statements)	0.789	7.965
Loop RMAX(with IF statements)	0.835	0.789
Loop TMAX(without IF statements)	0.033	0.035
Loop TMAX(with IF statements)	0.037	.032

Table 6.8: Stabilizing the facildlt OpenMP operations using IF statements, testcase *BOXCAV*, Opteron. Times in seconds.

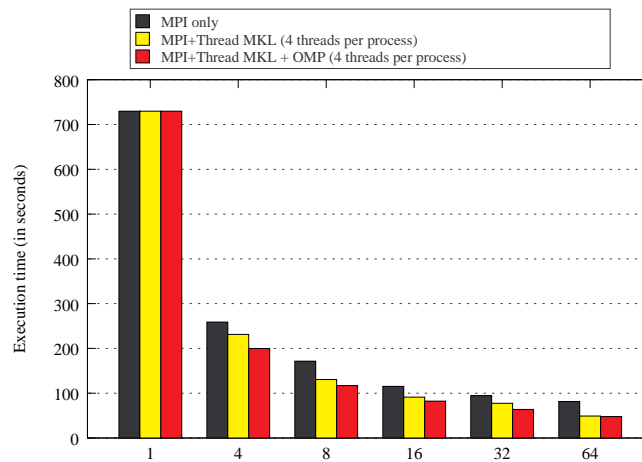


Figure 6.8: Time for factorization of matrix *ULTRASOUND80*, as a function of the number of cores used. In black, only MPI is used (for example 16 MPI processes for 16 cores). In the second and third experiments, 4 threads are used per MPI process (for example, 4 MPI processes with 4 threads each in the case of 16 cores) and a threaded BLAS library (MKL, 4 threads) is used. The difference between the second (yellow) and third (red) rectangles is that in the third rectangle, OpenMP directives are used to parallelize non-BLAS operations.

MPI process with 8 threads on 8 cores, than 8 MPI processes with 1 thread each. This behaviour is due to the non-perfect memory scalability with MPI (see Section 4.3).

6.6.2 OpenMP: optimization of a symmetric factorization kernel

In this section, we discuss the LDL^T factorization of a sparse matrix, and focus on a dense factorization kernel whose performance has been improved.

6.6.2.1 Performance bottleneck observed

While experimenting the fork-join model of parallelism discussed above, we observed that the symmetric indefinite case showed significantly poorer performance than the unsymmetric case. This motivated a deeper study of that behaviour. We remind that the symmetric factorization of a frontal matrix is using a blocked right-looking LDL^T algorithm (see Section 2.1.4) to update the fully summed part of the front, using Level 2 BLAS inside the pivot block and Level 3 BLAS outside the pivot block. The Schur complement is updated later.

Pivot factorization and pivot block update for LDL^T :

$A(nfr, nfr)$ is the array containing the symmetric frontal matrix

k is the current pivot

eb is the end (*i.e.*, the last column) of the current pivot block

1. Copy pivot column in pivot row (DCOPY)

$$A(k, k+1 : nfr) \leftarrow A(k+1 : nfr, k)$$

2. Update lower triangular block (DSYR)

$$A(k+1 : eb, k+1 : eb) \leftarrow A(k+1 : eb, k+1 : eb) - \frac{A(k+1:eb,k) \times A(k,k+1:eb)}{A(k,k)}$$

3. Scale column (DSCAL)

$$A(k+1 : nfr, k) \leftarrow \frac{A(k+1:nfr,k)}{A(k,k)}$$

4. Update rectangular block (DGER)

$$A(eb+1 : nfr, k+1 : eb) \leftarrow A(eb+1 : nfr, k) \times A(k, k+1 : eb)$$

Algorithm 6.5: Factorization of a pivot and update of the corresponding pivot block (in LDL^T decomposition).

We illustrate the poor performance of the symmetric indefinite case on the HALTERE matrix with METIS [120] on the SGI Altix machine *jade* from CINES (Montpellier, France). On this testcase, we observed that the central kernel performing the symmetric factorization of the current pivot block, does not scale correctly when increasing the number of threads. Each time a new pivot is chosen (see Section 1.3.2.2), it is used to update the column and update the pivot block. The unscaled column is first copied in the row of the frontal matrix in order to use BLAS 3 operations in the updates outside the pivot block (fully summed variables part and Schur complement). After each selected pivot, we therefore apply Algorithm 6.5, illustrated by Figure 6.9, which aims at updating the remaining columns of the current pivot block. Typically, the pivot block has 48 columns for large-enough fronts; the number of columns updated using DSYR and DGER is between 47 (for the first pivot of the pivot block) and 0 (for the last pivot of the pivot block).

Table 6.9 reports the accumulated times spent in the factorizations of the pivot blocks for the whole factorization, *i.e.*, for all frontal matrices, of matrix HALTERE, using double precision, real arithmetic. Clearly, the global results are disappointing in parallel, with bad speed-ups and even speed-downs. This can be explained by the fact that Algorithm 6.5 exhibits a poor memory locality of references: each element of the pivot column is accessed 3 times. Furthermore, noticing that we rely on a row-major storage, steps 1 and 2 of the algorithm force to load one cache line for each element of the column, when only one element is used inside

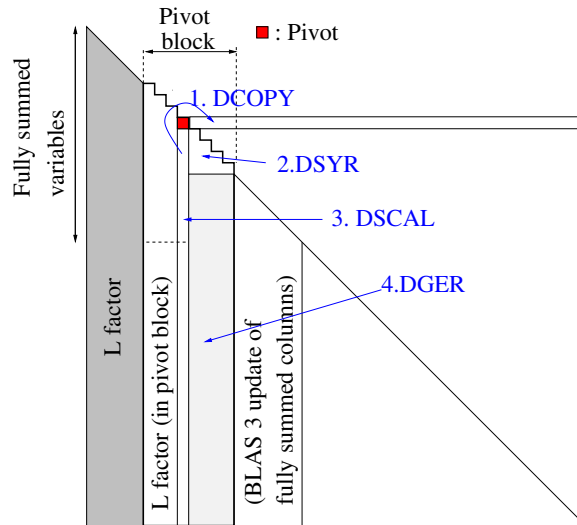


Figure 6.9: Illustration of Algorithm 6.5, performing the factorization of a pivot and the update of the pivot block in the symmetric indefinite factorization.

Number of threads	1	2	4	8
Time in Algorithm 6.5	16.7	16.7	19.9	29.0
Total factorization time	115	75	64	66

Table 6.9: Total time spent in Algorithm 6.5 after all pivots have been factored, and total time for factorization, as a function of the number of threads. Matrix *Haltere*, SGI Altix machine from CINES. Times in seconds.

that cache line. Whereas this was not too critical for serial executions, the memory bandwidth is exhausted in a multithreaded environment, resulting in no gains and even speed-downs when using multithreaded BLAS libraries.

6.6.2.2 Improvements of the factorization algorithm for the pivot block

In order to improve locality and avoid the limitations to parallelism due to memory accesses, we modified Algorithm 6.5 so that once an element in the pivot column is accessed and loaded into the cache, it is used as much as possible: for the copy in the upper triangular part of the front, the scaling, and the update of the rest of the row. This allows the elements of the pivot column to be loaded in cache once (temporal locality) and also improves the spatial locality between the element of the column and the corresponding row thanks to the row-major storage. Algorithm 6.6 gives the modified algorithm. First, the triangular block is updated (no parallelism is used because it is generally too small). Then, all the rows corresponding to the rectangular block can be processed in parallel. Remark that the new algorithm now only relies on OpenMP, without call to Level 1 or 2 BLAS. The results are given in Table 6.10 (to be compared with the results of the original algorithm in Table 6.9). We observe on this test case that it is very useful to avoid parallelization when the granularity is too small to benefit from parallelism. For example, on 8 threads, the overall time spent in Algorithm 6.6 decreases from 39.6 to 34.9 seconds when avoiding the parallelization in cases with less than 300 rows in the second loop, that is, when $nfr - eb < 300$. This clearly indicates that small blocks do not benefit from parallelism and were leading to significant speed-downs. Overall, the pivot factorization

Improved pivot factorization and pivot block update for LDL^T :

$A(nfr, nfr)$ is the array containing the symmetric frontal matrix

k is the current pivot

eb is the end (*i.e.*, the last column) of the current pivot block

```

% First, process symmetric part
for row = k + 1 to eb do
  1. Copy unscaled element of row to upper part
   $A(k, row) \leftarrow A(row, k)$ 

  2. Divide first element of row by pivot
   $A(row, k) \leftarrow \frac{A(row, k)}{A(k, k)}$ 

  3. Update remaining part of row
   $A(row, k + 1 : row) \leftarrow A(row, k + 1 : row) - A(row, k) \times A(k, k + 1 : row)$ 
end for
% Second, process rectangular part
for row = eb + 1 to nfr (in parallel) do
  1. Copy unscaled element of row to upper part
   $A(k, row) \leftarrow A(row, k)$ 

  2. Divide first element of row by pivot
   $A(row, k) \leftarrow \frac{A(row, k)}{A(k, k)}$ 

  3. Update remaining part of row
   $A(row, k + 1 : eb) \leftarrow A(row, k + 1 : eb) - A(row, k) \times A(k, k + 1 : eb)$ 
end for

```

Algorithm 6.6: Improved factorization of a pivot and update of the corresponding pivot block (in LDL^T decomposition).

and pivot block update now scales reasonably and the time spent in the factorization has decreased from

Number of threads	1	2	4	8
Without OMP IF				
Time in Algorithm 6.6	15.8	10.1	8.5	7.6
Total factorization time	110.7	65.4	47.5	39.6
With OMP IF				
Time in Algorithm 6.6	15.7	9.8	6.0	4.7
Total factorization time	110.4	64.9	44.1	34.9

Table 6.10: Total time spent in Algorithm 6.6 after all pivots have been factored, and total time for factorization, as a function of the number of threads. Matrix `Haltere`, SGI Altix machine from CINES. Times in seconds. The results “With OMP IF” corresponds to runs with an additional OpenMP “IF” directive, avoiding the parallelization of the second loop of Algorithm 6.6 when the number of rows in the loop is smaller than 300.

# threads	algorithm	Pivot search	Pivot block update
1	old	2.77	15.6
	new	0.91	15.3
2	old	1.95	9.8
	new	0.72	9.7
4	old	1.56	6.0
	new	0.64	6.0
8	old	1.58	4.7
	new	0.64	4.8

Table 6.11: Costs (in seconds) of the pivot block update and of the pivot search with and without the optimization consisting in checking for the stability of the pivot in the next column during the factorization of the pivot and panel update.

66 seconds (Table 6.9) to 34.9 seconds (Table 6.10) using 8 threads. Although the absolute speed-up is not that great compared to the sequential time (110 seconds), this shows that a careful look at just one of the factorization kernel allows for very significant gains. In fact, the speed-ups are much better for large frontal matrices but we remind that in the multifrontal tree, there are far more small frontal matrices than large ones. For small matrices near the bottom of the tree, it would then make sense to exploit tree parallelism (see Section 6.6.3).

6.6.2.3 Optimization of pivot search algorithm

We now describe a tiny optimization. Remark that, during the update of the pivot block with this new algorithm, the elements of the column next to the pivot column have all been loaded in cache once. It is therefore possible to check for the stability of the pivot candidate from the next column almost for free. Except for the last column of the pivot block, this can be done by computing the maximum of the elements in the column right to the pivot column, immediately after they have been computed and while in cache. In case the next pivot is stable (see Section 1.3.2.2) compared to the max in that column, the pivot search can be skipped. Some results on the same (`HALTERE`) matrix are given in Table 6.11. We observe that the overall cost of the pivot search is significantly reduced, and that the cost of the pivot block factorization remains unchanged with the additional computation of the maximum element in the next column (since only elements already in the cache are accessed anyway).

6.6.2.4 Discussion

Other factorization kernels should also be studied in order to push further the combination of BLAS and OpenMP directives. For example, the BLAS 3 update of the Schur complement in the LDL^T factorization

kernels currently works with many independent blocks processed one after each other and would benefit from parallelism with a larger granularity⁸. On the same `Haltere` matrix, 7 seconds out of 34 seconds on 8 threads and the same 7 seconds out of 110 seconds on 1 thread are also spent in assembly operations. There is thus also clearly scope for improvement in the assembly operations with symmetric matrices, even though the triangular loops involved are more difficult to parallelize.

6.6.3 OpenMP: exploiting the parallelism resulting from the assembly tree

The fork-join model to execute threaded regions wastes some time in activating sleeping threads in each threaded region or BLAS call; this was particularly visible on small frontal matrices, which usually appear near the bottom of the assembly tree. For example it was observed that for matrices smaller than a few hundreds, multithreading stack or assembly operations was leading to a performance degradation rather than improvement. The NUMA and cache penalties are indeed particularly critical on small frontal matrices, that usually appear near the bottom of the assembly tree. In Section 6.6.2, we also saw that multithreading the second loop of Algorithm 6.6 leads to a performance degradation on small matrices. To improve the parallelization of small frontal matrices, the multithreaded approach would thus benefit from exploiting the tree parallelism (similar what is done in the distributed-memory case), as shown in the preliminary results from Figure 6.10. In this figure, the unsymmetric version of our solver is used and a layer called `L0_OMP` is defined that such that:

- Under `L0_OMP`, tree parallelism and serial BLAS are used.
- Above `L0_OMP`, threaded BLAS libraries are used.

The definition of `L0_OMP` is based on serial/threaded benchmarks of MUMPS routines: we still use Algorithm 4.4 but the stopping criterion is new: we accept a layer `L0_OMP` if the estimated time below `L0_OMP` plus the estimated time above `L0_OMP` is

1. smaller than for all previous layers encountered;
2. smaller than the next 100 layers if algorithm is pursued further 100 times.

The second condition is there to avoid local minima due to possible imbalances after application of a greedy LPT (Largest Processing Time first) algorithm that aims at mapping subtrees onto the cores.

We observe in the figure that the speed-ups obtained are generally very good with the `L0_OMP` algorithm, compared to previous approaches. On large 3D problems (*e.g.*, AUDI test case), the gains are smaller than on problems leading to smaller fronts in the tree. An extreme case is the one of circuit-simulation matrices. On such matrices (see the results for G3-CIRCUIT and QIMONDA07), there is a very small amount of fill-in, leading to relatively small frontal matrices everywhere in the tree. In such cases, relying solely on node parallelism is clearly insufficient, and tree parallelism is compulsory to get performance.

Remark that, contrary to common intuition, we seem to still obtain reasonably good speed-ups by relying on tree parallelism first, perform a synchronization at the `L0_OMP` level, and work with all threads afterwards. Of course, the limits of this approach when increasing the number of threads has to be assessed. Short-term work-in-progress includes the adaptation of our approach to NUMA architectures, the study of memory management policies such as interleaving versus local, the suppression of idle time due to the `L0` synchronization, or the use of a penalty (around 10% to 50% on AMD processors) under `L0_OMP` due to the fact that all threads share a limited memory bandwidth. Multithreading the solve phase is also critical and should not be neglected.

⁸Ideally, we would like to rely on a threaded BLAS kernel performing the operation $C \leftarrow C - ADA^T$, where A and C are symmetric and D is a diagonal matrix with 1×1 and 2×2 pivots. More generally, such a BLAS kernel would be very useful for LDL^T factorizations.

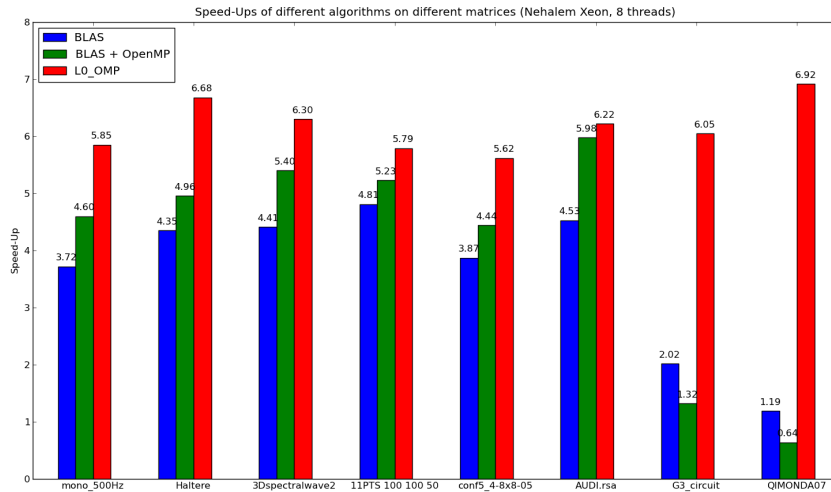


Figure 6.10: Comparison of the speed-ups obtained for the factorization of various matrices on 8 threads of a Nehalem-based computer when using (i) threaded BLAS, (ii) threaded BLAS and OpenMP directives outside BLAS calls, (iii) tree parallelism under LQ_OMP.

6.7 Sparse direct solvers and grid computing

Let us consider a different aspect not discussed earlier in this thesis and related to the transparent access to distant computing resources. We describe in this section two aspects where grid computing technologies and resources are interesting for sparse direct solvers.

- There exists a large range of algorithms and options in sparse direct solvers, and several sparse direct solvers are available, each implementing different variants of algorithms that may suit better a class of matrices or another. Although direct solvers are often considered as a black box when compared to iterative solvers, this range of algorithms, parameters and preprocessing options makes the adequate combination difficult to find for users. Finding the correct algorithms or parameters on a sparse matrix is however critical to solve larger problems, or decrease the time to the solution, or get a better accuracy. In many solvers, there exists the notion of “automatic” default choice for an option, which means that a solver will choose automatically the option based on some characteristics of the input matrix. However, those choices are limited and understanding which solver and which algorithms are best suited for a given class of matrix is not an easy issue. The GRID TLSE project[64] answers this problem by providing an expertise site for sparse linear algebra: typically, a user can upload and share matrices in private or public groups, and experiment and combine different functionalities from a variety solvers on their class of matrices. The expertise is based on scenarios, an example of which is the following: get all possible symmetric permutations from solvers capable of providing them, use them on all solvers and return statistics and synthetic graphs with the results. Metrics like time of execution, numerical accuracy or memory usage can be used to compare the results and, for experts, get some insight on the results. In the GRID TLSE site, each expertise request leads to a workflow, where each step of the workflow consists of a number of independent elementary requests. The middleware DIET [49] is used to execute and schedule those requests on a grid of computers.
- Usually, the solution of linear systems is the part of a simulation code that consumes most memory, while also being the bottleneck from the execution time point of view. In case a user does not have enough resources locally, it makes sense to use more powerful distant resources for the solution of the

linear system. We implemented a prototype with the `MUMPS` solver, using the `DIET` middleware again. To solve $Ax = b$, the matrix A and the right-hand side b can be sent over the network, and the solution x computed on the server is sent back to the client. Only this functionality was made available in the prototype, for unsymmetric matrices using double precision arithmetic. Matrix A remains small compared to the size of the factors and compared to the number of floating-point operations, contrary to dense matrices, where the size of the factors is the same as the size of the matrix. Therefore, the cost of transferring data could be reasonable in comparison to the size of the factors and the amounts of computations, at least the ratio is better than for dense matrices. Furthermore, when in practice many solves with the same matrix are requested, the matrix A can remain on the server(s). Within `DIET`, this can be done by setting the persistence mode to `PERSISTENT`, avoiding its transfer for each solve. Starting from the above prototype, experimentations showing the interest of using data persistence are given in [39].

It would make sense from an application point of view to go from this proof of concept to an implementation in production mode, using grid or cloud computing resources. Remark that this client-server model to use distant resources has already been used a lot in the context of linear algebra. For example, in the `Star-P` platform [55], the backslash operator from Matlab can be overloaded to work on distributed sparse matrices stored on distant HPC resources. Similar work was the object of the `OURAGAN` project [48] around `Scilab` or `Scilab//`.

In the case of simulation codes written in C or Fortran rather than Matlab or Scilab, operator overloading or simple interfaces like [87] are not what the application requires. Instead, the objective would consist in making all functionalities of a sparse direct solver available through the standard API of the solver. This requires redeveloping a library with the exact same API as the solver, allowing the application code to be exactly the same whether the linear solver is executed locally or on a more powerful distant server. Let us take the example of the `MUMPS` solver and the `DIET` middleware. At link time, there would be two possibilities on the client side:

1. The application code is linked with the `MUMPS` library: everything is executed locally.
2. The application code is linked with two libraries:
 - an intermediate library which transforms `MUMPS` calls into calls to the grid middleware, and
 - the grid middleware itself, that is the `DIET` library.

In this second case, everything should ideally be transparent from the application's code point of view: calls to the `MUMPS` solver are simply replaced by remote procedure calls. Data transfers must be minimized and data persistence on the servers is essential in order to transfer data only when necessary; such developments could naturally benefit from today's cloud infrastructures.

6.8 Conclusion and other work directions

We have described in this chapter recent work aiming at solving large problems. To conclude, we list in this section some other work directions and developments to solve increasingly large problems accurately and efficiently. A more *high-level* view will be given in the general conclusion of Chapter 7.

Multithreading within each MPI process. With the evolutions of computer architectures, the exploitation of multithreading inside each MPI process is critical. Three main issues are currently considered:

- The memory allocation policy has a very strong impact in many cases and we typically observed huge gains on multithreaded dense factorization kernels by allocating the memory on which threaded factorizations are applied with an interleave policy. Understanding how this affects a sparse multifrontal approach in which assemblies, copy operations, etc. are also parallelized is under investigation.

- Multithreaded dense factorization kernels that we currently use can be improved from the performance point of view, as was experimented for example in Section 6.6.2. Block sizes must be tuned, etc. As of today, the dense factorization kernels we use are specific (pivoting, partial factorization only, specific numerical aspects like detection of null pivots, out-of-core management, etc.) and cannot be immediately replaced by existing multithreaded decompositions from the dense linear algebra community.
- When the number of MPI processes increases (due to an increase in the number of nodes among which distributed-memory parallelism must be used), the amount of work in type 2 nodes increases, thus multithreading the corresponding kernels is critical.

MPI communications. When increasing the number of MPI processes, one must understand the limits of the existing communications schemes. For example, avoiding synchronizations and barriers is critical when working with thousands of nodes. We have recently been tackling two other examples related to communications within type 2 nodes:

- In the general case, the number of messages sent from type 2 children to type 2 parents is p^2 , if p is the number of workers (or slaves) in both the child and the parent. This number can be reduced to $O(p)$ by forcing the order of the variables in the child and in the parent to be compatible, and by initializing and managing the counters of messages-to-be-received in an appropriate manner. Although the volume communicated is similar, this decreases the latency associated with those messages by reducing drastically the number of messages when there are many processors involved in each node.
- The performance of the asynchronous broadcast algorithm in type 2 factorizations has been observed to be limited by the send bandwidth of the master node, in cases where many processors are assigned to a given frontal matrix. This typically occurs when serializing branches of the assembly tree for memory constraints, see Section 4.3. In the pipelined factorization implemented, the master processor must send a factored block to all its slaves. Given Algorithm 0.1, it is not possible to use the recent `MPI_IBCAST` primitives because each processor receives data with a general purpose asynchronous receive routine `MPI_IRECV` in Algorithm 0.1 and might be involved in several type 2 nodes. We have recently implemented a broadcast algorithm based on immediate sends and immediate receives, using a pipelined broadcast tree, and are expecting to significantly increase the performance of the factorization in type 2 nodes.

Related to MPI communications, the following points must also be considered:

- The splitting mechanism described in Figure 2.11 is evolving: by forcing a similar mapping between a parent and its child in a split chain, one can significantly limit the communication volume and improve performance on chains of splitted nodes.
- One must assess the limits of relying on ScaLAPACK kernels [53] for the root node and follow closely the evolution of libraries developed by the dense linear algebra community. As said before, our kernels are specific (numerical pivoting, various thresholds for delaying pivots, for detection of null pivots or for static pivoting, ...). It must be noted that a kernel like LDL^T that is heavily used in sparse linear algebra is not widely available in distributed-memory dense linear algebra libraries yet (for example it is not available in ScaLAPACK).
- Barriers and synchronizations involving all processors should be avoided. This looks fine thanks to our asynchronous approach, but we observed that on very large matrices with limited amount of fill (typically large reducible problems), a significant amount of time could be spent in reduction operations gathering statistics over all processors. The data structures involved in such global communications should be grouped into arrays in order to limit the associated costs.

Cost (memory, time) of phases other than the factorization. Although the cost of the factorization phase is usually predominant, the cost of the analysis and solve phases is sometimes critical too (see Section 6.1 for the parallelization of the analysis phase and Section 6.4 for some optimizations to the

solve phase). Concerning the analysis phase, the two following directions could help improving the analysis phase further in some applications:

- One could exploit a compressed graph on input instead of using the entire graph of the matrix. Indeed, many physical problems have several degrees of freedom at each node of the physical mesh (*e.g.*, temperature, pressure, velocities). Assuming that each node of a finite-element mesh has 6 variables, the number of edges in the graph associated with the matrix is 36 times larger than the number of edges in the mesh. Hence, in such cases, a sparse direct solver would gain a lot of memory by performing the analysis phase directly on the compressed graph corresponding to the mesh, pushing the limits of parallel analysis much further.
- In relation with the parallelization of the analysis phase, because simulation applications using direct solvers may already work on a distributed physical domain with balanced partitions, one could directly exploit a distributed mesh/graph on input and use the associated partitions in the analysis phase instead of calling parallel graph partitioning packages. Depending on the properties of the distribution, it could be possible to perform a local analysis on the internal variables of each domain and build local elimination trees, before a global analysis that will build the top of the tree based on the resulting information on the interfaces between domains.

Concerning the solve phase, we give a few remarks on this important topic in the general conclusion.

Unsymmetric structure of frontal matrices. In our multifrontal approach we work on the structure of the symmetrized problem $|A| + |A|^T$ when the original matrix A has an unsymmetric structure. This implies introducing explicit zeros in A , and working on square fronts (with unsymmetric values). However, [37] showed that it is possible to use unsymmetric fronts and reduce the computational cost of the multifrontal factorization by working on unsymmetric frontal matrices. Allowing unsymmetric storage for frontal matrices in our distributed-memory approach would allow to handle in a much better way matrices with unsymmetric structures, where unsymmetric orderings like [36, 35] –see also [143]– could then be exploited.

Elemental format. Elemental format avoids the assembly of the matrix in finite element applications and can be used in a natural way in multifrontal methods. Instead of precomputing all elements before calling the solver, a call-back to build the elemental matrices only when needed for assembly in a frontal matrix would avoid the large storage required for the initial matrix in elemental format. It would also allow to parallelize completely the construction and the assembly of the global problem inside the solver.

Software engineering. One can imagine many other points towards building direct solvers with a wide range of functionalities capable of solving very large problems on very large computing platforms, but one thing to be kept in mind is that each of these points requires developments that must combine nicely with all or most of the existing ones. For that, we anticipate that software engineering aspects will be a key issue to be able to pursue the type of work and research we have been doing in the last 15 years.

Chapter 7

Conclusion

The research and the work described in this document has been the object of various research projects, industrial contracts and collaborations. This work has led to an original tool, **MUMPS** (see [2]), which aims at solving sparse systems of linear equations on parallel distributed platforms, with a wide range of functionalities developed over the years. Clearly, not all the inside of **MUMPS** could be presented in this document, but it was an opportunity to describe some of it. Today, **MUMPS** is both a research tool to test new ideas, perform experiments or simulations and gather statistics in the field of sparse linear algebra, and a competitive software package used worldwide, in academia and industry, which requires software engineering, validation, support and maintenance. Although it is primarily a research prototype in which new research is constantly injected, a number of industrial users have invested on the use of this tool in their applications and have become dependent on it and on its evolutions. Distributing our work widely under the form of a software library is also essential for us for validation, feedback, external debugging and reports on performance and numerical behaviour: this way, our research is widely validated and we can define and adapt research directions according to the feedback received by **MUMPS** users and according to the evolution of applications requirements and the evolution of computer platforms.

Computer architectures often evolve too quickly for algorithm designers and for developers of large scientific libraries. In this context, software choices are critical, together with the long term durability of the chosen programming models and solutions. For example, although **MUMPS** started in 1996 from a shared memory approach [17], it was at that time decided (PARASOL project [136, 137]) to replace everything related to the shared memory paradigm with explicit message-passing targeting distributed-memory computers. With the advent of multicore processors, the shared-memory paradigm becomes very critical again, as shown in Section 6.6. We currently rely on MPI for message-passing and OpenMP for multithreading, as this matches our existing software, and because it is not clear to us what the next programming standard for exascale computers will be. With OpenMP, we observed that taking into account NUMA architectures and memory placement is very critical. Nowadays, there are many evolutions of computer platforms that one should take into account and try to anticipate:

- Memory per core will decrease. Therefore, we must continue to pay a strong attention to memory usage and to memory scalability.
- Numbers of cores per computing node will increase. Therefore, it is critical to exploit them efficiently, thanks to multithreading. If the number of cores grows too much, we expect that an hybrid approach mixing threads and message-passing with an asynchronous approach will become necessary inside multicore processors: the message-passing layer will ensure data locality and avoid costly synchronizations.
- The relative cost of accessing data (memory, communication) will increase compared to the cost of performing floating-point operations. In this context, it is not clear how memory-consuming strategies like pivot search and numerical pivoting will behave and if they will remain affordable.

- The absolute number of computing nodes will increase. Therefore, performance and memory scalability on huge numbers of nodes, using a distributed-memory environment is essential. Asynchronous approaches in a distributed-memory paradigm are also essential to scale to large numbers of nodes and the communication patterns need to be carefully studied (and possibly revisited).
- Accelerators might become more and more spread and be more or less integrated with the main CPU. Although multifrontal approaches exhibit large blocks, allowing significant gains by using GPUs, it is still difficult to perform numerical pivoting, for example, on such architectures.

All-in-all, parallelism is more and more difficult to handle, and heterogeneity (for example, at the network level, or due to the use of accelerators) is another challenge. Much work is currently invested on runtime systems, where only the tasks and the dependencies between them need be provided to automatically schedule the work on heterogeneous platforms composed of multiple cores, multiple GPUs and multiple nodes. Again, long-lasting solutions or a standardization of such runtimes systems would be necessary before investing heavily on such approaches. It would be interesting to understand the limits of such approaches compared to a “by-hand” approach (and vice-versa).

Thanks to new algorithms and computer evolutions, the maximum size of the problems that can be solved with direct solvers are orders of magnitude larger than one or two decades ago and it keeps growing¹, although current software implementing sparse direct solvers is far from exploiting all resources of existing petascale computers. There is thus much scope for improvements on many challenging issues such as scheduling for memory, scheduling for performance, mapping irregular data-structures, memory scalability, out-of-core storage, locality of reference, efficient management of communications, etc. It is not clear when fault-tolerance will become an issue, but out-of-core storage of the factors as factorization goes along is a natural approach to incremental checkpointing.

Furthermore, in order to treat huge problems and exploit efficiently the memory available on large supercomputers, all data structures should scale with the number of processors; one can probably no more afford symbolic data structures of the order of the matrix or of the order of the number of nodes in the tree: those should scale with the number of processors. All preprocessing algorithms (scalings, symbolic factorization, maximum weighted matching algorithms, pre-selection of some 2×2 pivots, ...) should also scale in both memory usage and performance, working on fully distributed matrices or graphs.

Many solver developers have focused more on the performance of the factorization phase than on the performance of the solve phase. However, the solve phase deserves a lot of attention too for applications where most of the time is spent in that phase: sometimes, thousands of right-hand sides are solved for, for only one matrix factorization. Specific approaches can be useful in the case of multiple right-hand sides, or when right-hand sides are sparse, or when only a subset of the solution is needed. When the performance of the solve is getting critical, as this appeared to be the case for several applications at the last MUMPS users’group meeting², it might be worth guiding the preprocessing phase and the mapping of the factors by the performance of the solve phase.

Although much work remains to be done on the scalability and implementation of direct solvers on high performance computers, their complexity may still be unaffordable even on high-end computers when the problem size becomes too large: remember that a 3D grid of size $N \times N \times N$ may have $O(N^3)$ nonzeros in the original matrix, $O(N^4)$ entries in the factors, and require $O(N^6)$ floating-point operations. Because of this large complexity, direct solvers are sometimes used as a building-box in other approaches, typically in hybrid direct-iterative solvers: incomplete factorizations relying on direct solver technologies, domain decompositions methods using direct solvers within each domain, or Block-Cimmino approaches. Another approach we are currently studying is the notion of fast direct solvers. Several groups have also started to work on this subject, including in the context of multifrontal solvers. The idea is to rely on low-rank representations of certain blocks of the dense matrices arising because of fill-in during the factorization (the off-diagonal blocks are often low-rank), in order to compress data and reduce the amount of computations. Those low-rank representations can be of the same quality as the floating-point representations (*i.e.*, machine precision), or

¹For example, a problem with 87 million equations could be solved a few years ago with the Pastix [113] solver, whereas not so many people could imagine solving problems with more than 100000 equations with a direct solver 15 years ago.

²http://graal.ens-lyon.fr/MUMPS/ud_2010.html

they can be approximations of lower quality. In the latter case, this leads to preconditioners with a clear numerical criterion to control accuracy. The MUMPS framework should allow for efficient implementations of a low-rank solver in both distributed-memory and multithreading contexts.

With the above work directions, we aim at pursuing one of our main focus for the future: solving increasingly large problems arising in simulation applications efficiently and accurately. By efficiently, we mean that we must optimize resource usage (processor cores, memory accesses, locality) of computer platforms while aiming at obtaining the solution as quickly as possible. By accurately, we mean that we do not want to reduce the numerical stability, one of the main strengths of direct solvers, for performance or increased parallelism; in the context of increasing problem sizes and increasing numbers of operations, round-off errors may also become an issue to look at. We also aim at continuing work on numerical aspects and functionalities and transferring our research in a tool like MUMPS, which at the moment is both necessary to our future research and useful to many academic and industrial groups.

Bibliography

- [1] The BCSLIB Mathematical/Statistical Library. <http://www.boeing.com/phantom/bcslib/>.
- [2] MUltifrontal Massively Parallel Solver (MUMPS) users' guide. Available from <http://graal.ens-lyon.fr/MUMPS/> or <http://mumps.enseeiht.fr/>.
- [3] MUMPS: a MUltifrontal Massively Parallel sparse direct Solver. <http://graal.ens-lyon.fr/MUMPS>, <http://mumps.enseeiht.fr>.
- [4] The OpenMP application program interface. <http://www.openmp.org>.
- [5] E. Agullo. *On the Out-of-core Factorization of Large Sparse Matrices*. PhD thesis, École Normale Supérieure de Lyon, Nov. 2008.
- [6] E. Agullo, P. Amestoy, A. Buttari, A. Guermouche, J.-Y. L'Excellent, and F.-H. Rouet. Robust memory-aware mappings for parallel multifrontal factorizations, Feb. 2012. SIAM conference on Parallel Processing for Scientific Computing (PP12), Savannah, GA, USA.
- [7] E. Agullo, A. Guermouche, and J.-Y. L'Excellent. A preliminary out-of-core extension of a parallel multifrontal solver. In *EuroPar'06 Parallel Processing*, Lecture Notes in Computer Science, pages 1053–1063, 2006.
- [8] E. Agullo, A. Guermouche, and J.-Y. L'Excellent. On reducing the I/O volume in a sparse out-of-core solver. In *HiPC'07 14th International Conference On High Performance Computing*, number 4873 in Lecture Notes in Computer Science, pages 47–58, Goa, India, December 17-20 2007.
- [9] E. Agullo, A. Guermouche, and J.-Y. L'Excellent. Reducing the I/O volume in an out-of-core sparse multifrontal solver. Research Report 6207, INRIA, 05 2007. Also appeared as LIP report RR2007-22.
- [10] E. Agullo, A. Guermouche, and J.-Y. L'Excellent. Towards a parallel out-of-core multifrontal solver: Preliminary study. Research report 6120, INRIA, Feb. 2007. Also appeared as LIP report RR2007-06.
- [11] E. Agullo, A. Guermouche, and J.-Y. L'Excellent. On the I/O volume in out-of-core multifrontal methods with a flexible allocation scheme. In J. M. L. M. Palma, P. R. Amestoy, M. J. Daydé, M. Mattoso, and J. C. Lopes, editors, *VECPAR'08 International Meeting on High Performance Computing for Computational Science*, volume 5336 of *LNCS*, pages 328–335. Springer-Verlag Berlin Heidelberg, jun 2008.
- [12] E. Agullo, A. Guermouche, and J.-Y. L'Excellent. A parallel out-of-core multifrontal method: Storage of factors on disk and analysis of models for an out-of-core active memory. *Parallel Computing, Special Issue on Parallel Matrix Algorithms*, 34(6-8):296–317, 2008.
- [13] E. Agullo, A. Guermouche, and J.-Y. L'Excellent. Reducing the I/O volume in sparse out-of-core multifrontal methods. *SIAM Journal on Scientific Computing*, 31(6):4774–4794, 2010.
- [14] A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1:131–137, 1972.

- [15] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA., 1983.
- [16] P. Amestoy, I. Duff, J.-Y. L'Excellent, F.-H. Rouet, and B. Uçar. Parallel computation of entries of A-1, 2011. SIAM Workshop on Combinatorial Scientific Computing (CSC11), Darmstadt, Germany, 19/05/2011-21/05/2011.
- [17] P. R. Amestoy. *Factorization of large sparse matrices based on a multifrontal approach in a multi-processor environment*. PhD thesis, Institut National Polytechnique de Toulouse, 1991. Available as CERFACS report TH/PA/91/2.
- [18] P. R. Amestoy, A. Buttari, and J.-Y. L'Excellent. Towards a parallel analysis phase for a multifrontal sparse solver, June 2008. Presentation at the 5th International workshop on Parallel Matrix Algorithms and Applications (PMAA'08).
- [19] P. R. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17:886–905, 1996.
- [20] P. R. Amestoy and I. S. Duff. Vectorization of a multiprocessor multifrontal code. *International Journal of Supercomputer Applications*, 3:41–59, 1989.
- [21] P. R. Amestoy and I. S. Duff. Memory management issues in sparse multifrontal methods on multiprocessors. *Int. J. of Supercomputer Applics.*, 7:64–82, 1993.
- [22] P. R. Amestoy, I. S. Duff, A. Guermouche, and Tz. Slavova. Analysis of the out-of-core solution phase of a parallel multifrontal approach. Research report RT/APO/07/3, ENSEEIHT, Apr. 2007. Also appeared as CERFACS and INRIA technical report.
- [23] P. R. Amestoy, I. S. Duff, A. Guermouche, and Tz. Slavova. Analysis of the solution phase of a parallel multifrontal approach. *Parallel Computing*, 36:3–15, 2010.
- [24] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.
- [25] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent. Multifrontal solvers within the PARASOL environment. In B. Kågström, J. Dongarra, E. Elmroth, and J. Waśniewski, editors, *Applied Parallel Computing, PARA'98*, Lecture Notes in Computer Science, No. 1541, pages 7–11, Berlin, 1998. Springer-Verlag.
- [26] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent. MUMPS MULTifrontal Massively Parallel Solver Version 2.0. Technical Report RT/APO/98/3, ENSEEIHT-IRIT, Toulouse, France, 1998. Also CERFACS Report TR/PA/98/02.
- [27] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Comput. Methods Appl. Mech. Eng.*, 184:501–520, 2000.
- [28] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and X. S. Li. Analysis and comparison of two general sparse solvers for distributed memory computers. *ACM Transactions on Mathematical Software*, 27(4):388–421, 2001.
- [29] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and X. S. Li. Impact of the implementation of MPI point-to-point communications on the performance of two general sparse solvers. *Parallel Computing*, 29(7):833–847, 2003.
- [30] 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. *SIAM Journal on Scientific Computing*, 34(4):A1975–A1999, 2012.

- [31] P. R. Amestoy, I. S. Duff, S. Pralet, and C. Vömel. Adapting a parallel sparse direct solver to architectures with clusters of SMPs. *Parallel Computing*, 29(11-12):1645–1668, 2003.
- [32] P. R. Amestoy, I. S. Duff, D. Ruiz, and B. Uçar. A parallel matrix scaling algorithm. In J. M. L. M. Palma, P. R. Amestoy, M. J. Daydé, M. Mattoso, and J. C. Lopes, editors, *High Performance Computing for Computational Science, VECPAR'08*, number 5336 in Lecture Notes in Computer Science, pages 309–321. Springer-Verlag, 2008.
- [33] P. R. Amestoy, I. S. Duff, and C. Vömel. Task scheduling in an asynchronous distributed memory multifrontal solver. *SIAM Journal on Matrix Analysis and Applications*, 26(2):544–565, 2005.
- [34] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel Computing*, 32(2):136–156, 2006.
- [35] P. R. Amestoy, X. S. Li, and E. G. Ng. Diagonal Markowitz scheme with local symmetrization. *SIAM Journal on Matrix Analysis and Applications*, 29(1):228–244, 2007.
- [36] P. R. Amestoy, X. S. Li, and S. Pralet. Unsymmetric ordering using a constrained markowitz scheme. *SIAM Journal on Matrix Analysis and Applications*, 29(1):302–327, 2007.
- [37] P. R. Amestoy and C. Puglisi. An unsymmetrized multifrontal LU factorization. *SIAM Journal on Matrix Analysis and Applications*, 24:553–569, 2002.
- [38] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM Press, Philadelphia, PA, third edition, 1995.
- [39] G. Antoniu, E. Caron, F. Desprez, A. Fèvre, and M. Jan. Towards a transparent data access model for the GridRPC paradigm. In S. A. et al. (Eds), editor, *HiPC'2007. 14th International Conference on High Performance Computing.*, number 4873 in LNCS, pages 269–284, Goa. India, Dec. 2007. Springer Verlag Berlin Heidelberg.
- [40] M. Arioli, J. Demmel, and I. S. Duff. Solving sparse linear systems with sparse backward error. *SIAM Journal on Matrix Analysis and Applications*, 10:165–190, 1989.
- [41] C. Ashcraft and R. G. Grimes. The influence of relaxed supernode partitions on the multifrontal method. *ACM Transactions on Mathematical Software*, 15:291–309, 1989.
- [42] C. Ashcraft and J. W. H. Liu. Robust ordering of sparse matrices using multisection. *SIAM Journal on Matrix Analysis and Applications*, 19:816–832, 1998.
- [43] H. Avron, G. Shklarski, and S. Toledo. Parallel unsymmetric-pattern multifrontal sparse LU with column reordering. *ACM Transactions on Mathematical Software*, 34(2):8:1–8:31, 2008.
- [44] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petit, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM Press, 1997.
- [45] J. R. Bunch and L. Kaufman. Some stable methods for calculating inertia and solving symmetric linear systems. *Mathematics of Computation*, 31:162–179, 1977.
- [46] J. R. Bunch and B. N. Parlett. Direct methods for solving symmetric indefinite systems of linear systems. *SIAM J. Numer. Anal.*, 8:639–655, 1971.
- [47] A. Buttari. Fine granularity QR factorization for multicore based systems. Technical Report RT-APO-11-6, ENSEEIHT, Toulouse, France, 2011.

- [48] E. Caron, S. Chaumette, S. Contassot-Vivier, F. Desprez, E. Fleury, C. Gomez, M. Goursat, E. Jeannot, D. Lazure, F. Lombard, J.-M. Nicod, L. Philippe, M. Quinson, P. Ramet, J. Roman, F. Rubi, S. Steer, F. Suter, and G. Utard. Scilab to Scilab//, the OURAGAN project. *Parallel Computing*, 11(27):1497–1519, Oct. 2001.
- [49] E. Caron and F. Desprez. DIET: A scalable toolbox to build network enabled servers on the grid. *International Journal of High Performance Computing Applications*, 20(3):335–352, 2006.
- [50] S. Chakrabarti, J. Demmel, and K. A. Yelick. Modeling the benefits of mixed data and task parallelism. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 74–83, 1995.
- [51] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.
- [52] C. Chevalier and F. Pellegrini. PT-Scotch: A tool for efficient parallel graph ordering. In *Proceedings of PMAA2006, Rennes, France*, oct 2006.
- [53] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers - design issues and performance. Technical Report LAPACK Working Note 95, CS-95-283, University of Tennessee, 1995.
- [54] I. Chowdhury and J.-Y. L’Excellent. Some experiments and issues to exploit multicore parallelism in a distributed-memory parallel sparse direct solver. Research Report RR-4711, INRIA and LIP-ENS Lyon, Oct. 2010.
- [55] R. Choy, A. Edelman, J. R. Gilbert, V. Shah, and D. Cheng. Star-p: High productivity parallel computing. In *In 8th Annual Workshop on High-Performance Embedded Computing (HPEC 04)*, 2004.
- [56] T. H. Cormen, E. R. Davidson, and S. Chatterjee. Asynchronous buffered computation design and engineering framework generator (ABCDEFG). In *19th International Parallel and Distributed Processing Symposium (IPDPS’05)*, 2005.
- [57] O. Cozette. *Contributions systèmes pour le traitement de grandes masses de données sur grappes*. PhD thesis, Université de Picardie Jules Vernes, 2003.
- [58] O. Cozette, A. Guermouche, and G. Utard. Adaptive paging for a multifrontal solver. In *Proceedings of the 18th annual international conference on Supercomputing*, pages 267–276. ACM Press, 2004.
- [59] T. A. Davis. Algorithm 832: UMFPACK V4.3 — an unsymmetric-pattern multifrontal method with a column pre-ordering strategy. *ACM Transactions on Mathematical Software*, 30(2):196–199, 2004.
- [60] T. A. Davis. Algorithm 915: SuiteSparseQR: Multifrontal multithreaded sparse QR factorization. *ACM Transactions on Mathematical Software*, 38(1):8:1–8:22, 2011.
- [61] T. A. Davis and I. S. Duff. An unsymmetric-pattern multifrontal method for sparse LU factorization. *SIAM Journal on Matrix Analysis and Applications*, 18:140–158, 1997.
- [62] M. J. Daydé and I. S. Duff. Use of level 3 BLAS in LU factorization in a multiprocessing environment on three vector multiprocessors, the ALLIANT FX/80, the CRAY-2, and the IBM 3090/VF. *Int. J. of Supercomputer Applics.*, 5:92–110, 1991.
- [63] M. J. Daydé and I. S. Duff. A block implementation of level 3 BLAS for RISC processors. Technical Report RT/APO/96/1, ENSEEIHT-IRIT, 1996.

- [64] M. J. Daydé, L. Giraud, M. Hernandez, J.-Y. L'Excellent, C. Puglisi, and M. Pantel. An Overview of the GRID-TLSE Project. In *Poster Session of 6th International Meeting VECPAR'04, Valencia, Espagne*, Lecture Notes in Computer Science, pages 851–856. Springer, June 2004.
- [65] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20(3):720–755, 1999.
- [66] F. Desprez and F. Suter. Impact of mixed-parallelism on parallel implementations of Strassen and Winograd matrix multiplication algorithms. *Concurrency and Computation: Practice and Experience*, 16(8):771–797, 2004.
- [67] F. Dobrian. *External memory algorithms for factoring sparse matrices*. PhD thesis, Old Dominion University, Computer Science Department, 2001.
- [68] F. Dobrian and A. Pothen. Oblio: a sparse direct solver library for serial and parallel computations. Technical report, Old Dominion University, 2000.
- [69] F. Dobrian and A. Pothen. The design of I/O-efficient sparse direct solvers. In *Proceedings of Super-Computing*, 2001.
- [70] J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK Users Guide*. SIAM, Philadelphia, 1979.
- [71] J. Dongarra, R. Hempel, A. J. G. Hey, and D. W. Walker. MPI : A message passing interface standard. *Int. Journal of Supercomputer Applications*, 8:(3/4), 1995.
- [72] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. Algorithm 679. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16:1–17, 1990.
- [73] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. SIAM, Philadelphia, 1991.
- [74] I. S. Duff. Parallel implementation of multifrontal schemes. *Parallel computing*, 3:193–204, 1986.
- [75] I. S. Duff. Multiprocessing a sparse matrix code on the Alliant FX/8. *J. Comput. Appl. Math.*, 27:229–239, 1989.
- [76] I. S. Duff and J. Koster. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM Journal on Matrix Analysis and Applications*, 20(4):889–901, 1999.
- [77] I. S. Duff and J. Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM Journal on Matrix Analysis and Applications*, 22(4):973–996, 2001.
- [78] I. S. Duff and S. Pralet. Strategies for scaling and pivoting for sparse symmetric indefinite problems. *SIAM Journal on Matrix Analysis and Applications*, 27(2):313–340, 2005.
- [79] I. S. Duff and J. K. Reid. MA27—a set of Fortran subroutines for solving sparse symmetric sets of linear equations. Technical Report R.10533, AERE, Harwell, England, 1982.
- [80] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Transactions on Mathematical Software*, 9:302–325, 1983.
- [81] I. S. Duff and J. K. Reid. The multifrontal solution of unsymmetric sets of linear systems. *SIAM Journal on Scientific and Statistical Computing*, 5:633–641, 1984.
- [82] I. S. Duff and J. K. Reid. Exploiting zeros on the diagonal in the direct solution of indefinite sparse symmetric linear systems. *ACM Transactions on Mathematical Software*, 22(2):227–257, 1996.

- [83] S. C. Eisenstat and J. W. H. Liu. The theory of elimination trees for sparse unsymmetric matrices. *SIAM Journal on Matrix Analysis and Applications*, 26:686–705, 2005.
- [84] S. C. Eisenstat and J. W. H. Liu. A tree based dataflow model for the unsymmetric multifrontal method. *Electronic Transaction on Numerical Analysis*, 21:1–19, 2005.
- [85] V. Espirat. Développement d’une approche multifrontale pour machines a mémoire distribuée et réseau hétérogène de stations de travail. Technical Report Rapport de stage 3ieme Année, ENSEEIHT-IRIT, 1996.
- [86] C. Farhat, J. Mandel, and F.-X. Roux. Optimal convergence properties of the FETI domain decomposition method. *Comput. Methods Appl. Mech. Engrg.*, 115:365–385, 1994.
- [87] A. Fèvre, J.-Y. L’Excellent, and S. Pralet. Scilab and MATLAB interfaces to MUMPS. Technical Report RR-5816, INRIA, Jan. 2006. Also appeared as ENSEEIHT-IRIT report TR/TLSE/06/01 and LIP report RR2006-06.
- [88] H. Garcia-Molina. Election in distributed computing system. *IEEE Transactions on Computers*, pages 47–59, 1982.
- [89] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM : Parallel Virtual Machine – A User’s Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.
- [90] A. Geist and E. G. Ng. Task scheduling for parallel sparse Cholesky factorization. *Int J. Parallel Programming*, 18:291–314, 1989.
- [91] A. George and J. W. H. Liu. A quotient graph model for symmetric factorization. In I. S. Duff and G. W. Stewart, editors, *Sparse Matrix Proceedings*, pages 154–175. SIAM, 1978.
- [92] A. George and J. W. H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Englewood Cliffs, NJ., 1981.
- [93] J. R. Gilbert and J. W. H. Liu. Elimination structures for unsymmetric sparse LU factors. *SIAM Journal on Matrix Analysis and Applications*, 14:334–352, 1993.
- [94] J. R. Gilbert and E. G. Ng. Predicting structure in nonsymmetric sparse matrix factorizations. In J. G. A. George and J. Liu, editors, *Graph Theory and Sparse Matrix Computations*, pages 107–140. Springer-Verlag NY, 1993.
- [95] J. R. Gilbert and S. Toledo. High-performance out-of-core sparse LU factorization. In *Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing*, 1999. (10 pages on CDROM).
- [96] G. H. Golub and C. F. Van Loan. *Matrix Computations. 3rd ed.* The Johns Hopkins University Press, Baltimore and London, 1996.
- [97] D. Gope, I. Chowdhury, and V. Jandhyala. DiMES: Multilevel fast direct solver based on multipole expansions for parasitic extraction of massively coupled 3d microelectronic structures. In *42nd Design Automation Conference*, pages 159–162, June 2005.
- [98] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.
- [99] S. Graillat. Accurate floating-point product and exponentiation. *IEEE Trans. Comput.*, 58(7):994–1000, July 2009.
- [100] L. Grigori, J. W. Demmel, and X. S. Li. Parallel symbolic factorization for sparse LU with static pivoting. *SIAM J. Sci. Comput.*, 29(3):1289–1314, 2007.

- [101] A. Guermouche. Impact de l'ordonnancement sur l'occupation mémoire d'un solveur multifrontal parallèle. In *15ième Rencontres Francophones en Parallélisme, La Colle sur Loup, France*, pages 37–45, 2003.
- [102] A. Guermouche. *Étude et optimisation du comportement mémoire dans les méthodes parallèles de factorisation de matrices creuses*. PhD thesis, École Normale Supérieure de Lyon, July 2004.
- [103] A. Guermouche and J.-Y. L'Excellent. Memory-based scheduling for a parallel multifrontal solver. In *18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, page 71a (10 pages), 2004.
- [104] A. Guermouche and J.-Y. L'Excellent. Optimal memory minimization algorithms for the multifrontal method. Research report RR5179, INRIA, 2004. Also LIP report RR2004-26.
- [105] A. Guermouche and J.-Y. L'Excellent. Flexible task allocation for the memory minimization of the multifrontal approach, June 2005. Second International Workshop on Combinatorial Scientific Computing (CSC05), CERFACS, Toulouse, France.
- [106] A. Guermouche and J.-Y. L'Excellent. Constructing memory-minimizing schedules for multifrontal methods. *ACM Transactions on Mathematical Software*, 32(1):17–32, 2006.
- [107] A. Guermouche, J.-Y. L'Excellent, and G. Utard. Impact of reordering on the memory of a multifrontal solver. *Parallel Computing*, 29(9):1191–1218, 2003.
- [108] A. Gupta. Recent advances in direct methods for solving unsymmetric sparse systems of linear equations. *ACM Transactions on Mathematical Software*, 28(3):301–324, 2002.
- [109] A. Gupta. A shared- and distributed-memory parallel general sparse direct solver. *Applicable Algebra in Engineering, Communication and Computing*, 18(3):263–277, 2007.
- [110] A. Gupta and Y. Muliadi. An experimental comparison of some direct sparse packages. Technical Report TR RC-21862, IBM research division, T.J. Watson Research Center, Yorktown Heights, 2000. <http://www.cs.umn.edu/~agupta/wsmmp.html>.
- [111] L. He and Y. Sun. On distributed snapshot algorithms. In *Advances in Parallel and Distributed Computing Conference (APDC '97)*, 1997. 291–297.
- [112] M. T. Heath, E. G. Ng, and B. W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33:420–460, 1991.
- [113] P. Hénon, P. Ramet, and J. Roman. PaStiX: A high-performance parallel direct solver for sparse symmetric definite systems. *Parallel Computing*, 28(2):301–321, Jan. 2002.
- [114] N. J. Higham. A survey of condition number estimation for triangular matrices. *SIAM Review*, 29:575–596, 1987.
- [115] J. Hogg, J. K. Reid, and J. A. Scott. Design of a multicore sparse Cholesky factorization using DAGs. Technical Report RAL-TR-2009-027, Rutherford Appleton Laboratory, 2009.
- [116] HSL. HSL 2007: A collection of Fortran codes for large scale scientific computation, 2007.
- [117] D. Irony, G. Shklarski, and S. Toledo. Parallel and fully recursive multifrontal sparse Cholesky. *Future Generation Computer Systems*, 20(3):425–440, 2004.
- [118] M. Jacquelin, L. Marchal, Y. Robert, and B. Uçar. On optimal tree traversals for sparse matrix factorization. In *IPDPS'2011, the 25th IEEE International Parallel and Distributed Processing Symposium*. IEEE Computer Society Press, 2011, to appear.

- [119] C.-P. Jeannerod, N. Louvet, and J.-M. Muller. Further analysis of kahan’s algorithm for the accurate computation of 2 x 2 determinants. *Mathematics of Computation*, 2012. To appear.
- [120] G. Karypis and V. Kumar. METIS – *A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices – Version 4.0*. University of Minnesota, Sept. 1998.
- [121] G. Karypis and K. Schloegel. *ParMetis: Parallel Graph Partitioning and Sparse Matrix Ordering Library Version 4.0*. University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, Minneapolis, MN 55455, U.S.A., Aug. 2003. Users’ manual.
- [122] X. S. Li. Evaluation of SuperLU on multicore architectures. *Journal of Physics: Conference Series*, 125:012079, 2008.
- [123] X. S. Li and J. W. Demmel. Making sparse Gaussian elimination scalable by static pivoting. In *Proceedings of Supercomputing*, Orlando, Florida, November 1998.
- [124] X. S. Li and J. W. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Transactions on Mathematical Software*, 29(2), 2003.
- [125] J. W. H. Liu. Modification of the minimum degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software*, 11(2):141–153, 1985.
- [126] J. W. H. Liu. On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Transactions on Mathematical Software*, 12:127–148, 1986.
- [127] J. W. H. Liu. An application of generalized tree pebbling to sparse matrix factorization. *SIAM J. Algebraic Discrete Methods*, 8:375–395, 1987.
- [128] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11:134–172, 1990.
- [129] J. W. H. Liu. The multifrontal method for sparse matrix solution: Theory and Practice. *SIAM Review*, 34:82–109, 1992.
- [130] J. W. H. Liu, A. George, and E. Ng. Communication results for parallel sparse cholesky factorization on a hypercube. *Parallel Computing*, 10:287–298, 1989.
- [131] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, Jan. 1996.
- [132] L. Neal and G. Poole. A geometric analysis of Gaussian elimination. II. *Linear Algebra and its Applications*, 173:239 – 264, 1992.
- [133] E. G. Ng and P. Raghavan. Performance of greedy heuristics for sparse Cholesky factorization. *SIAM Journal on Matrix Analysis and Applications*, 20:902–914, 1999.
- [134] S. T. O. Meshar, D. Irony. An out-of-core sparse symmetric-indefinite factorization method. *ACM Transactions on Mathematical Software*, 32(3):445–471, 2006.
- [135] S. Operto, J. Virieux, P. R. Amestoy, J.-Y. L’Excellent, L. Giraud, and H. Ben Hadj Ali. 3D finite-difference frequency-domain modeling of visco-acoustic wave propagation using a massively parallel direct solver: A feasibility study. *Geophysics*, 72(5):195–211, 2007.
- [136] PARASOL. Deliverable 2.1a: MUMPS Version 2.0. A MULTifrontal Massively Parallel Solver. Technical report, January 10 1998.
- [137] PARASOL. Deliverable 2.1c: MUMPS Version 3.1. A MULTifrontal Massively Parallel Solver. Technical report, CERFACS and ENSEEIHT, February, 19 1999.

- [138] S. V. Parter. The use of linear graphs in Gauss elimination. *SIAM Review*, 3:119–130, 1961.
- [139] F. Pellegrini. SCOTCH 5.0 User’s guide. Technical Report, LaBRI, Université Bordeaux I, Aug. 2007.
- [140] F. Pellegrini, J. Roman, and P. R. Amestoy. Hybridizing nested dissection and halo approximate minimum degree for efficient sparse matrix ordering. In *Proceedings of Irregular’99, San Juan*, Lecture Notes in Computer Science 1586, pages 986–995, 1999.
- [141] F. Pellegrini, J. Roman, and P. R. Amestoy. Hybridizing nested dissection and halo approximate minimum degree for efficient sparse matrix ordering. *Concurrency: Practice and Experience*, 12:69–84, 2000. Preliminary version published in *Proceedings of Irregular’99*, LNCS 1586, 986–995.
- [142] A. Pothen and C. Sun. A mapping algorithm for parallel sparse Cholesky factorization. *SIAM Journal on Scientific Computing*, 14(5):1253–1257, 1993.
- [143] S. Pralet. *Constrained orderings and scheduling for parallel sparse linear algebra*. PhD thesis, Institut National Polytechnique de Toulouse, Sept 2004. Available as CERFACS technical report, TH/PA/04/105.
- [144] E. S. Quintana-Ortí and R. A. Van De Geijn. Updating an lu factorization with pivoting. *ACM Trans. Math. Softw.*, 35(2):11:1–11:16, 2008.
- [145] P. Raghavan. *Distributed sparse matrix factorization: QR and Cholesky decompositions*. Ph.D. thesis, Department of Computer Science, Pennsylvania State University, 1991.
- [146] J. Rigal and J. Gaches. On the compatibility of a given solution with the data of a linear system. *J. Assoc. Comput. Mach.*, 14:526–543, 1967.
- [147] E. Rothberg and S. C. Eisenstat. Node selection strategies for bottom-up sparse matrix ordering. *SIAM Journal on Matrix Analysis and Applications*, 19(3):682–695, 1998.
- [148] E. Rothberg and R. Schreiber. Efficient methods for out-of-core sparse Cholesky factorization. *SIAM Journal on Scientific Computing*, 21(1):129–144, 1999.
- [149] V. Rotkin and S. Toledo. The design and implementation of a new out-of-core sparse Cholesky factorization method. *ACM Transactions on Mathematical Software*, 30(1):19–46, 2004.
- [150] F.-H. Rouet. *Memory and performance issues in parallel multifrontal factorizations and triangular solutions with sparse right-hand sides*. PhD thesis, Institut National Polytechnique de Toulouse, Oct. 2012. To appear.
- [151] D. Ruiz. A scaling algorithm to equilibrate both rows and columns norms in matrices. Technical Report RT/APO/01/4, ENSEEIHT-IRIT, 2001. Also appeared as RAL report RAL-TR-2001-034.
- [152] C. Sánchez, H. B. Sipma, Z. Manna, and C. D. Gill. Efficient distributed deadlock avoidance with liveness guarantees. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software, October 22-25, 2006, South Korea*, pages 12–20.
- [153] O. Schenk and K. Gärtner. On fast factorization pivoting methods for sparse symmetric indefinite systems. Technical Report CS-2004-004, CS Department, University of Basel, August 2004.
- [154] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proc. of the First Conference on File and Storage Technologies*, 2002.
- [155] R. Schreiber. A new implementation of sparse Gaussian elimination. *ACM Transactions on Mathematical Software*, 8:256–276, 1982.
- [156] J. Schulze. Towards a tighter coupling of bottom-up and top-down sparse matrix ordering methods. *BIT*, 41(4):800–841, 2001.

- [157] S. S. Shende and A. D. Malony. The TAU parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, 2006.
- [158] Tz. Slavova. *Parallel triangular solution in the out-of-core multifrontal approach for solving large sparse linear systems*. Ph.D. dissertation, Institut National Polytechnique de Toulouse, Apr. 2009. Available as CERFACS Report TH/PA/09/59.
- [159] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*. The MIT Press, Cambridge, Massachusetts, 1996.
- [160] F. Sourbier, S. Operto, J. Virieux, P. R. Amestoy, and J.-Y. L'Excellent. FWT2D: a massively parallel program for frequency-domain full-waveform tomography of wide-aperture seismic data – part 2: numerical examples and scalability analysis. *Computer and Geosciences*, 35(3):496–514, 2009.
- [161] S. D. Stoller. Leader election in asynchronous distributed systems. *IEEE Transactions on Computers*, 49(3):283–284, Mar. 2000.
- [162] R. Takhur, W. Gropp, and E. Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*, pages 23–32. ACM Press, 1999.
- [163] S. Toledo. A survey of out-of-core algorithms in numerical linear algebra. In J. M. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 161–179. American Mathematical Society Press, Providence, RI, 1999.
- [164] S. Toledo and A. Uchitel. A supernodal out-of-core sparse Gaussian elimination method. In *Proceedings of PPAM 2007*, 2007.
- [165] J. H. Wilkinson. *Rounding Errors in Algebraic Processes*. Prentice-Hall, Englewood Cliffs, New Jersey, 1963.
- [166] M. Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 2:77–79, 1981.