On the Out-Of-Core Factorization of Large Sparse Matrices

Emmanuel Agullo

N° d'ordre : 496

N° attribué par la bibliothèque : 09ENSL496

THÈSE

en vue d'obtenir le grade de

Docteur de l'Université de Lyon - École Normale Supérieure de Lyon

spécialité : Informatique

Ecole Doctorale de Mathématiques et Informatique Fondamentale

présentée et soutenue publiquement le 28/11/2008

par Monsieur Emmanuel AGULLO

Titre :

Méthodes directes hors-mémoire (out-of-core) pour la résolution de systèmes linéaires creux de grande taille

---

Après avis de : *Monsieur Iain DUFF*

Monsieur Sivan TOLEDO

Devant la commission d'examen formée de :

Monsieur Iain DUFF, Membre/Rapporteur

Monsieur Jean-Yves L'EXCELLENT, Membre

Madame Xiaoye Sherry LI, Membre

Monsieur Yves ROBERT, Membre

Monsieur Jean ROMAN, Président

Monsieur Sivan TOLEDO, Membre/Rapporteur

Titre français de la thèse:

Méthodes directes hors-mémoire (out-of-core) pour la résolution de systèmes linéaires creux de grande taille

English title of the Ph.D. thesis:

On the Out-Of-Core Factorization of Large Sparse Matrices

Par / By: Emmanuel AGULLO

Directeur de thèse / Main supervisor: *Jean-Yves L'EXCELLENT*

Co-encadrant / Co-supervisor: *Abdou GUERMOUCHE*

Document rédigé en anglais / Document written in English

Remerciements

J'adresse tout naturellement mes premiers remerciements à Jean-Yves L'Excellent. Ces années sous ta direction ont été un immense plaisir. Merci d'avoir été autant à l'écoute. Merci pour ta générosité. Tout ce que tu m'as appris, sur les plans scientfiques et humains, j'espère que je saurai le transmettre à mon tour. Merci également à Abdou Guermouche pour avoir co-encadré cette thèse. Merci de m'avoir permis de rentrer petit à petit dans les lignes de code de `MUMPS`. Merci d'avoir eu la patience d'échanger de longues heures au téléphone, preuves et contre-exemples à l'appui. Merci également de m'avoir fait découvrir les villes de Toulouse et Bordeaux. Merci à Sherry Li et Esmond Ng de m'avoir reçu au LBNL pendant une demi-année. Merci à Sherry d'avoir pris le temps de me faire plonger dans `SuperLU`, d'avoir rendu le séjour si agréable par sa gentillesse et d'avoir accepté de faire partie du jury de thèse. Merci à Esmond pour son accueil. Merci également à Patrick Amestoy pour avoir suivi de si près le déroulement de cette thèse, pour son aide et pour ses conseils.

Je remercie Iain Duff et Sivan Toledo d'avoir accepté de rapporter cette thèse, d'avoir pris le soin de formuler des retours extrêmement constructifs et d'avoir été membres du jury. Merci à Jean Roman d'avoir présidé le jury. Merci à Yves Robert d'avoir également accepté de participer au jury en plus de tous les précieux conseils promulgués.

Je remercie Frédéric Desprez et Frédéric Vivien de m'avoir accueilli dans leur équipe puis soutenu tout au long de cette période. Merci également à tous les membres de l'équipe Graal. Merci à Yves et Anne pour les parties de pétanque, à Eddy pour sa bonne humeur. Merci à Loris et Lionel pour nos discussions NP-complètes. Merci à Jean-Sébastien pour son courage exemplaire, à Cédric pour nos discussions, à Raphaël pour ses défaites avec le sourire et à Jean-François pour sa sympathie. Merci à Mila, Alfredo, Bora, Philippe, et Aurélia. Merci à tous les membres du LIP de rendre le temps passé au laboratoire agréable. À Alain et Serge pour les tournois de foot. À Sylvain, Damien et Florent pour les pauses café. Merci aussi à Osni et Tony pour leurs grillades exceptionnelles.

Merci à ma famille pour m'avoir permis de me lancer dans cette aventure, pour leur soutien constant et pour le bonheur partagé. Merci en particulier à mes parents. Enfin merci à Amélie d'être là.

Contents

Introduction **1**

1 Introduction to the field **3**
1.1 Direct methods for the factorization of sparse matrices 3
1.1.1 Sparsity, fill-in and reordering 4
1.1.2 Elimination trees and pivoting 6
1.1.3 Left-looking, right-looking and multifrontal methods 9
1.1.4 Supernodes and assembly tree 12
1.2 Out-of-core sparse direct methods 14
1.2.1 Approaches based on virtual memory 15
1.2.2 Work of Liu 16
1.2.3 Work of Rothberg and Schreiber 16
1.2.4 Work of Toledo et al. 17
1.2.5 Work of Dobrian and Pothen 19
1.2.6 Summary 19
1.3 I/O mechanisms 20
1.4 Experimental environment 21
1.4.1 Software environment 21
1.4.2 Hardware environment 23
1.4.3 Test problems 24

I Models and algorithms for out-of-core serial sparse direct methods **25**

2 On the volume of I/O: Case study with the multifrontal method **27**
2.1 Out-of-core multifrontal methods 27
2.1.1 Assumptions related to out-of-core multifrontal methods 29
2.1.2 Variants of the multifrontal method 29
2.2 Evolution of the active storage, core memory and I/O volume 30
2.3 Notations 33
2.4 Formal expression of the I/O volume 33
2.5 Experiments on real problems 34
2.6 Models for an out-of-core assembly scheme 36

3 Reducing the *I/O* volume in the multifrontal method: Terminal allocation 41
3.1 Classical assembly scheme . . . 43
3.1.1 Illustration of the impact of different postorders . . . 43
3.1.2 Optimum postorder tree traversal . . . 44
3.2 In-place assembly of the last contribution block . . . 45
3.3 In-place assembly of the largest contribution block . . . 46
3.4 Theoretical comparison of `MinMEM` and `MinIO` . . . 47
3.5 Experimental results . . . 49
3.6 Conclusion . . . 52

4 Reducing the *I/O* volume in the multifrontal method: Flexible allocation scheme case 53
4.1 Restating results of Chapter 3 . . . 53
4.2 Flexible parent allocation . . . 54
4.3 Volume of *I/O* in a flexible multifrontal method . . . 55
4.4 Reducing the complexity (discrimination process) . . . 56
4.5 Minimizing the *I/O* volume in the flexible multifrontal method is NP-complete . . . 59
4.5.1 Intuition . . . 59
4.5.2 Preliminary result: the problem of a knapsack that can spill out is NP-complete . . . 61
4.5.3 Proof of the NP-completeness . . . 62
4.5.4 Another difficulty which also makes the problem NP-complete . . . 64
4.6 In-place assembly schemes . . . 67
4.6.1 In-place assembly before the parent allocation . . . 67
4.6.2 In-place assembly after the parent allocation . . . 68
4.7 A Heuristic based on the discrimination process . . . 69
4.8 Experimental results . . . 70
4.9 Conclusion . . . 71

5 Memory management schemes for multifrontal methods 75
5.1 In-core stack memory . . . 76
5.1.1 Recalling the classical and last-in-place assembly schemes . . . 76
5.1.2 In-place assembly of the largest contribution block . . . 77
5.1.3 Flexible allocation of the frontal matrices . . . 78
5.1.3.1 Classical and last-in-place assembly schemes . . . 78
5.1.3.2 Max-in-place assembly scheme . . . 79
5.2 Out-of-core stacks . . . 80
5.2.1 Cyclic memory management . . . 80
5.2.2 Using information from the analysis: terminal allocation scheme . . . 82
5.2.3 Using information from the analysis: flexible allocation scheme . . . 85
5.3 Conclusion . . . 86

6 A study of out-of-core supernodal sparse direct methods 89
6.1 Assumptions on models for out-of-core supernodal algorithms 90
6.2 Minimum core problem - W1/R0 scheme 91
6.3 Combining *left-looking* and *right-looking* methods 93
6.3.1 *Right-looking* approach between SuperPanels 95
6.3.2 *Left-looking* approach between SuperPanels 98
6.3.3 Comparison 102
6.4 *I/O* volume reduction partitioning algorithms 105
6.4.1 *I/O* volume induced with a *left-looking* approach between SuperPanels 105
6.4.2 Partitioning a chain 106
6.4.3 Left-looking/right-looking approach 107
6.4.4 Left-looking/left-looking approach 109
6.5 Preliminary prototype implementation 112
6.5.1 Reducing the scatter-gather overhead - HyperNodes 114
6.5.2 Symbolic factorization - out-of-core depth-first-search 115
6.5.3 Out-of-core numerical factorization - *left-looking/left-looking* method 115
6.5.4 Preliminary validation 115
6.6 Conclusion 117

II A parallel out-of-core multifrontal solver 119

7 Preliminary study 121
7.1 Parallel scheme for the multifrontal method 121
7.2 Instrumentation 123
7.3 Experimental results 123

8 A robust parallel code with factors on disk 127
8.1 Direct and buffered (at the system level) I/O mechanisms 128
8.2 Synchronous and asynchronous approaches (at the application level) . . 129
8.3 Sequential Performance 130
8.4 Parallel Performance 133
8.5 Discussion 135

9 Effects of a parallel out-of-core stack on memory 137
9.1 Models to manage the contribution blocks on disk in a parallel context . 137
9.2 Analysis of the memory needs of the different schemes 138
9.3 Analysing how the memory peaks are obtained 139
9.4 Decreasing the memory peaks 142
9.5 Conclusion 144

10 Improving the memory scalability 147
10.1 Proportional mapping and memory scalability 147
10.1.1 Practical study on real-life matrices 147
10.1.2 Quantification of the sub-optimality of proportional mapping . . . 149

10.2 A memory-aware mapping algorithm 150
10.3 First experiments 151
10.4 Discussion 153
10.5 Other memory reductions 154
10.6 Conclusion 155

Conclusion **157**

A Bibliography **161**

B On the shape of the I/O graphs: Formalization **169**

C Complements to Chapter 4 **177**
C.1 The decision problem associated with a knapsack that can spill out is NP-complete 177
C.2 Multiple allocation 178

Introduction

Solving sparse systems of linear equations is central in many simulation applications. Because of their robustness and performance, direct methods can be preferred to iterative methods, especially in an industrial context. However, they require a large amount of memory because the factorization of a sparse matrix A leads to factors with an amount of non-zeros much larger than in the original matrix. Even if the memory of modern computers keeps increasing, applications always require to solve larger and larger problems which still need more memory than available. Furthermore, some types of architectures have a limited amount of memory per processor or per core. The use of *out-of-core* solvers, where the disk is used to extend the physical memory, is then compulsory. In this dissertation, we study out-of-core direct methods from both a theoretical and a practical point of view. We focus on two classes of direct methods, so-called multifrontal and supernodal algorithms, and introduce the main concepts we rely on in Chapter 1.

Part I: Models and algorithms for out-of-core serial sparse direct methods

In Part I, we study those methods in a sequential out-of-core context. A critical parameter in out-of-core is the I/O volume, which corresponds to the amount of traffic between disk and memory. In the context of multifrontal methods, we propose models to compute and represent the I/O volume in Chapter 2. We then design algorithms to schedule the tasks of the multifrontal algorithm in a way that limits that I/O volume in chapters 3 and 4. In Chapter 3, we consider the classical multifrontal method and show that minimizing the I/O volume on the stack of temporary data is different from minimizing the overall size of the stack; we propose a new algorithm that minimizes the I/O volume. The results of Chapter 3 are used as a basis for Chapter 4, which addresses a more general variant of the multifrontal method that we call *flexible allocation scheme* [44]. In this variant task allocation can be anticipated and this significantly reduces the working storage and the I/O volume compared to classical approaches. We show that the minimization of the I/O volume is NP-complete with this allocation scheme and propose an efficient heuristic based on practical considerations. In Chapter 5, we propose memory management algorithms for our models that could be used as a basis to build a new serial out-of-core code.

The last chapter of Part I is dedicated to the study of out-of-core supernodal methods

and corresponds to the work done during a 6-month visit at the Lawrence Berkeley National Laboratory under the supervision of Xiaoye S. Li, in the group of Esmond G. Ng. We present an out-of-core prototype for the serial version of `SuperLU` and propose contributions to address the problem of *I/O* volume minimization in the context of supernodal methods.

Part II: A parallel out-of-core multifrontal solver

In Part II, we consider a particular method, the multifrontal algorithm, that we push as far as possible in a parallel context. Starting (in Chapter 7) from a preliminary study on the memory behaviour of an existing in-core solver, `MUMPS`, we design a robust out-of-core extension that allows to process significantly larger problems by storing factors on disk with a high efficiency (Chapter 8). We propose different *I/O* mechanisms and study their impact on performance. We also show the limits of our method in terms of scalability. To address that problem, we then study two complementary aspects. A first approach consists in studying the behaviour of our out-of-core code if temporary data were also processed on disk (Chapter 9). This study illustrates both the bottlenecks related to the current implementation and the intrinsic limits of the method. A second approach consists in rethinking the whole schedule of the factorization (Chapter 10) to limit the memory usage. We show that this latter approach allows us to process large problems with a very good memory scalability.

All along this dissertation, our approaches are validated on large matrices from different application fields. Note that we focus on the factorization step since the solution step is the object of a separate study [8]. Concerning the *multifrontal* method, we will assume that frontal matrices - which are dense matrices corresponding to the elementary tasks but may represent a large amount of memory - can fit in core memory and go as far as possible in this context. We show that we can process very large problems doing so, especially in parallel.

Chapter 1

Introduction to the field

1.1 Direct methods for the factorization of sparse matrices

We are interested in solving the following linear system whose unknown is x:

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

A is a sparse matrix of order N ($A = (a_{ij})_{1 \leq i,j \leq N}$), and x and b are column vectors. A matrix is sparse when the number NZ of nonzero values is small compared to the number of entries ($NZ \ll N^2$).

Direct methods are based on the Gaussian elimination and proceed in two steps: the factorization and the solution steps. In the factorization step, matrix A is decomposed under the form

$$A = LU, \tag{1.2}$$

where L is a lower triangular matrix whose diagonal values are equal to 1 and U is an upper triangular matrix. The solution step then consists in solving in order two triangular systems: $Ly = b$ and $Ux = y$. One of the motivations of a two-step process is that the solution step is far less costly than the factorization step. Thus, when solving a succession of linear systems with the same matrix A but varying right-hand sides, only the solution step has to be repeated whereas the costly decomposition is performed once.

We admit that decomposition (1.2) exists if the matrix is invertible (non-singular) even if it means swapping some columns. Matrices L and U verify: $a_{ij} = (LU)_{ij}$, for each (i, j) in $\{1; ...; N\}^2$. Considering the respective triangular structures of L and U, it comes: $a_{ij} = \sum_{k=1}^{\min(i,j)} l_{ik} u_{kj}$. Because $l_{ii} \equiv 1$, we finally have:

$$\begin{cases} a_{ij} = \sum_{k=1}^{i-1} l_{ik} u_{kj} + u_{ij} & \text{if } i \leq j \\ a_{ij} = \sum_{k=1}^{j-1} l_{ik} u_{kj} + l_{ij} u_{jj} & \text{if } i > j \end{cases} \tag{1.3}$$

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

$$I_j \begin{cases} u_{kj} = a_{kj} - \sum_{i=1}^{k-1} l_{ki} u_{ij} & \text{for each } k \text{ in } \{1; ...; j\} \\ l_{kj} = \frac{1}{u_{jj}} (a_{kj} - \sum_{i=1}^{j-1} l_{ki} u_{ij}) & \text{for each } k \text{ in } \{j+1; ...; N\} \end{cases} \tag{1.4}$$

which allows their computation by iteratively applying I_j for $j = 1$ to N. We present in Figure 1.1 the data involved during an iteration. Computations are performed column by column. Each column j (terms u_{kj} or l_{kj} in Formula (1.4)) depends on the columns $i, i = 1, \ldots, j-1$ (term l_{ki} in Formula (1.4)).

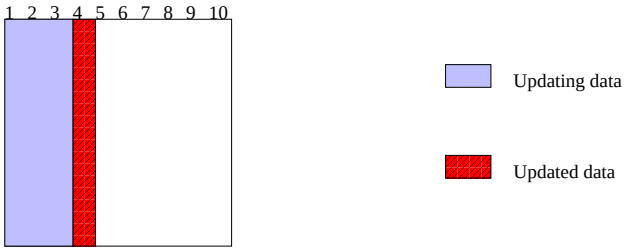


Figure 1.1: Iteration I_4.

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

1.1.1 Sparsity, fill-in and reordering

To reduce the storage requirement and the amount of computation, sparse direct methods aim at storing and computing only nonzero values. Usually the L and U factors are denser than the original matrix A (they have more nonzero values). This phenomenon is called *fill-in*. Indeed, Expression 1.4 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 $\{1; ...; \min\{i, j\}\}$ such that l_{ik} and u_{kj} are nonzeros. Figure 1.2 illustrates fill-in. Fill-in increases the number of nonzero values. These new nonzero values in turn induce more computation. Said differently, fill-in increases the memory requirement as well as the number of operations and is critical. Many studies have been done to reduce the amount and the effects of fill-in. They consist in permuting the rows/columns of the matrix and are called *ordering algorithms*.

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 $V = \{1; 2; ...; N\}$ 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.

The problem which consists in finding the optimum ordering of the variables to minimize the amount of computation being NP-complete [72], the use of heuristics is required. In practice these heuristics are performed on the graph associated with the matrix. Among all the possible heuristics, here are two main classes of ordering algorithms that apply to the symmetric case.

- Some heuristics are based on local criteria to decide which variable to eliminate first. They are called *bottom-up* approaches. For instance, Minimum Degree algorithm consists in eliminating the variable which has the lowest degree of connectivity in the graph associated with the matrix at each iteration. Approximate Minimum Degree [5] (`AMD`) and Multiple Minimum Degree [54] (`MMD`) libraries implement efficient variants of this algorithm. Another approach consists in selecting the variable which induces a minimum fill at each iteration. Approximate Minimum Fill [57] (`AMF`) and Multiple Minimum Fill [62] (`MMF`) libraries implement variants of this algorithm.
- Some heuristics are based on a more global approach (*top-down* approach). They consist in partitioning the graph associated with the matrix recursively. Nested dissections [38, 37] belong to this class of heuristics.

Some other heuristics are hybrid between these two schemes. The graph associated with the matrix is recursively partitioned down to a certain granularity and then local heuristics are performed on the obtained partitions. `METIS` [51] and `PORD` [69] libraries implement hybrid approaches.

Finally, some heuristics are specific to the unsymmetric case. Column Approximate Minimum Degree [24] (`COLAMD`) for instance is a variant of `AMD` for the unsymmetric case.

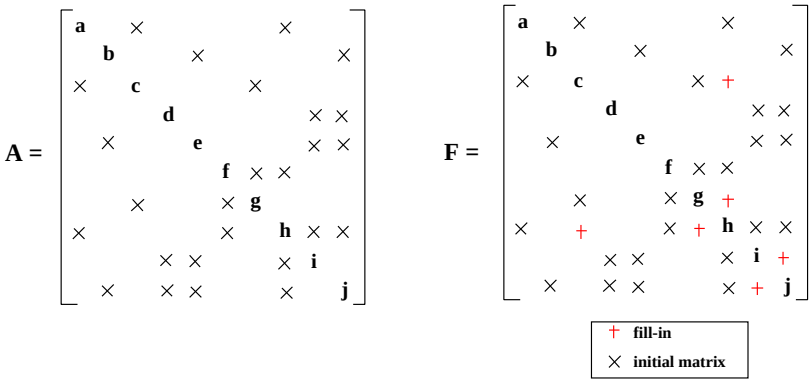


Figure 1.2: Fill-in. Matrix F contains l_{ij} and u_{ij} values after factorization.

1.1.2 Elimination trees and pivoting

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 [56, 39]. Formula (1.4) provides these dependencies that we express as a binary relation $\rightarrow$ on the set of columns $\{1; ...; N\}$ 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 $\{i+1; ...; N\}$ such that $l_{ki}u_{ij} \neq 0$.*

The transitive closure $\xrightarrow{+}$ of $\rightarrow$ expresses whether a column i must be computed before a column j: $i \xrightarrow{+} j$ *if and only if* column i must be computed before column j. This information can be synthesized with a transitive reduction $\xrightarrow{-}$ of $\rightarrow$ (or of $\xrightarrow{+}$): column i must be computed before column j *if and only if* there is a path in the directed graph associated with $\xrightarrow{-}$ from i to j. This statement would be true for any of the relations $\rightarrow$, $\xrightarrow{+}$ or $\xrightarrow{-}$, but $\xrightarrow{-}$ presents the advantage to be the most compact way to code this information [2].

The graph associated with $\xrightarrow{-}$ reflects some available freedom to reorder the variables without changing the sparse 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 [3]. We can thus introduce the notion of *descendant* and *ancestor* between columns as follows: i descendant of $j \Leftrightarrow j$ ancestor of $i \Leftrightarrow i \xrightarrow{+} j$. 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 [2].

Symmetric elimination tree

In the symmetric ($\mathbf{LL}^T$ or $\mathbf{LDL}^T$) case, the transitive reduction of the graph of explicit dependencies is a tree and is called *symmetric elimination tree* [68, 56]. As we will heavily rely on this property in this dissertation, 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 $\{i+1; ...; N\}$ such that $l_{ki}l_{ji} \neq 0$. Thus $i \rightarrow j$ 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 $k = j$. ■

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 1.4 states that $l_{kj} \neq 0$ (except in case of numerical cancellation). ■

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

Proof. We have to show that there is no cycle (neither directed nor undirected) in the transitive reduction of the considered graph. As we have already seen that there is no directed cycle (it is a *dag*) we suppose to the contrary that there is an undirected cycle whose column of smallest index is i. Then there exist two other columns j and k $(i < j < k)$ in this cycle such that $i \rightarrow j$ and $i \rightarrow k$. Lemma 1.2 implies $j \rightarrow k$. Subsequently $i \rightarrow k$ is reduced by the pair $(i \rightarrow j, j \rightarrow k)$ and cannot be part of the reduced graph. This is a contradiction. ■

Figure 1.3 illustrates the different stages of the construction of the symmetric elimination tree.

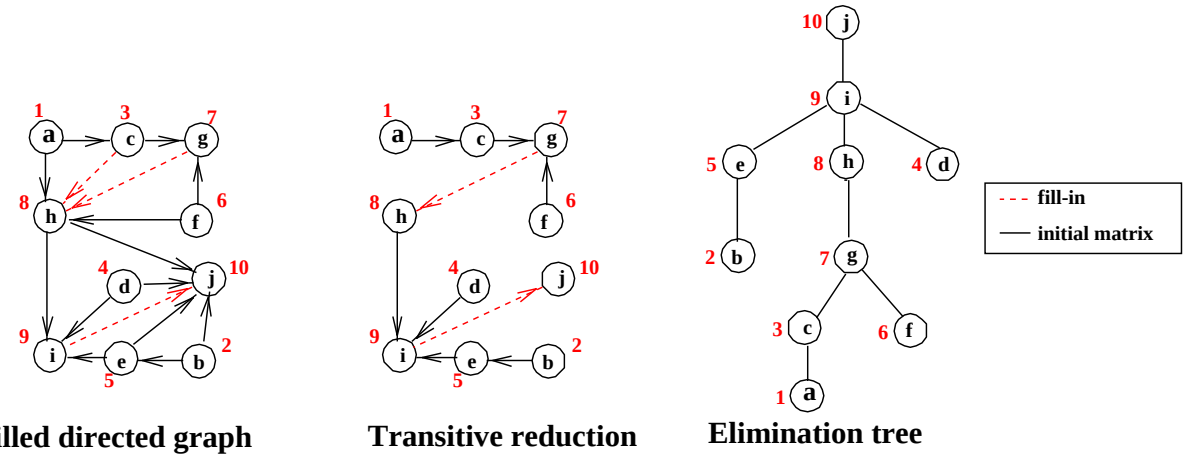


Figure 1.3: Construction of the symmetric elimination tree of the matrix presented in Figure 1.2.

Liu shows in [56] that the graph associated with $\rightarrow$ is exactly the graph associated with the triangular factor $(\mathcal{G}(L^T))$ and that the symmetric elimination tree is thus the transitive reduction of $\mathcal{G}(L^T)$. He furthermore explains how to compute the structure of L (*i.e.* $\mathcal{G}(L^T)$) from the the symmetric elimination tree (*i.e.* the transitive reduction of $\mathcal{G}(L^T)$) and from the structure of the original matrix (A). The motivation is that the structure of the elimination tree is more compact and thus most efficient to traverse than the structure of the factor itself. Therefore, the computation of $\mathcal{G}(L^T)$ is enhanced by the use of its transitive reduction which is maintained during the computation of $\mathcal{G}(L^T)$. In this sense, the symmetric elimination tree characterizes the structure of the triangular factor.

Some unsymmetric methods use a symmetric pattern to handle unsymmetric matrices. The structure of the initial matrix is symmetrized according to the structure of $A + A^T$:

each initial structural zero in the pattern of A that is nonzero in $A + A^T$ is filled with a numerical zero value. These methods can thus rely on the symmetric elimination tree too. For instance `MUMPS` [9, 10] is based on this approach.

Unsymmetric elimination dags

In the unsymmetric case, the transitive reduction of the graph of explicit dependencies does not correspond to the graph of a special matrix involved in the decomposition. However, Gilbert and Liu have generalized the notion of elimination tree to the unsymmetric case in [39]: for unsymmetric matrices, the nonzero structure of the lower and upper triangular factors can be characterized by the elimination *dags* that are the respective transitive reductions of $\mathcal{G}(L^T)$ and $\mathcal{G}(U)$. Indeed, they explain how to efficiently deduce the structure of the factors from these elimination *dags* and from the original matrix (similarly to the structure of L that could be deduced from the symmetric elimination tree for a symmetric matrix). Intuitively, the motivation for the use of these elimination structures is the need to handle two factored matrices (L and U) which are structurally different yet closely related to each other in the filled pattern. These elimination structures are used for instance in unsymmetric methods such as the distributed memory version of `SuperLU` [53].

Pivoting

In practice, some factorization methods do not respect the initial ordering. Floating point storage and arithmetic are not exact. Thus rounding errors may accumulate and prevent from computing an accurate solution. This phenomenon is called numerical instability. In particular, the division by a small diagonal value (u_{jj} in Formula 1.4) will lead to elements of large magnitude, inducing rounding errors when added to small numbers. This is why some methods allow to swap the diagonal variable (and the corresponding row) with another variable (and its corresponding row) of greater value. This swapping may modify the sparse structure of the factors and thus the graph of dependencies. A classical method of pivoting is the *partial pivoting* [42] which consists in swapping the diagonal variable with the variable of the column which has the largest magnitude. Another method called *threshold pivoting* only swaps the diagonal variable if its value divided by the largest value of the column is lower than a given threshold.

Column elimination tree

Some methods aim at anticipating such possible structural change. 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 potential partial pivoting [40]. The column elimination tree is the symmetric elimination tree of A^TA (provided that there is no cancellation in computing A^TA). Note that A^TA 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 [25, 40]. For instance, the serial version of `SuperLU` [25] is based on this approach.

In this dissertation, we will not discuss further methods based on elimination dags 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;*
- *the column elimination tree for unsymmetric direct methods with unsymmetric structure.*

1.1.3 Left-looking, right-looking and multifrontal methods

There are two main types of operations occurring during the factorization algorithm. Using the notations of [29], 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 (1.4)):
- FACTO(A_j): $A_j(j+1:n) \leftarrow A_j(j+1:n)/a_{jj}$;
- UPDATE(A_i,A_j): $A_j(i+1:n) \leftarrow A_j(i+1:n) - a_{ij}.A_i(i+1:n)$;

where A_k is the column k 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, FACTO(A_j) has to be performed after FACTO(A_i) if j parent of i (*i.e.* if $i \bar{\rightarrow} j$). And there is an effective 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 UPDATE(A_i,A_j) operation has to be performed after FACTO(A_i) and before FACTO(A_j). We will note UPDATE($*$,A_j) an update *of* column j and UPDATE(A_i,$*$) an update *from* column i.

In spite of these constraints of dependency, the structure of the elimination tree provides some flexibility and freedom to schedule the computations. Indeed, any topological traversal [56] of the elimination tree respects the dependencies between FACTO operations. We will see the interest of exploiting this freedom in chapters 3 and 4. 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 UPDATE($*$,A_j) are performed just

before FACTO(A_j). On the opposite, *right-looking* algorithms perform the UPDATE operations as soon as possible: all the UPDATE(A_i,$*$) operations are performed just after FACTO(A_i). Algorithms 1.1 and 1.2 respectively illustrate *left-looking* and *right-looking* factorizations. Note that Algorithm 1.1 exactly corresponds to applying iteration I_j from Formula 1.4 for $j = 1$ to N.

```
for j = 1 to N do
    foreach i such that i→j (j explicitly depends on i) do
        UPDATE(A_i,A_j) ;
    FACTO(A_j) ;
```

Algorithm 1.1: General *left-looking* factorization algorithm.

```
for i = 1 to N do
    FACTO(A_i) ;
    foreach j such that i→j (j explicitly depends on i) do
        UPDATE(A_i,A_j) ;
```

Algorithm 1.2: General *right-looking* factorization algorithm.

We will see how to further exploit that freedom in our context in Chapter 6, related to the study of out-of-core supernodal methods.

The *multifrontal* method [31, 34] 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_i^k,A_i) and consists in adding the temporary column T_i^k associated with child k of i from A_i. Otherwise, the destination column is a temporary column T_k^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 1.3 describes the whole algorithm:

```
for i = 1 to N do
    foreach j such that i→j (j explicitly depends on i) do
        INIT(T_j^i) ;
    foreach k such that k⇢i (k child of i) do
        ASSEMBLE(T_k^i, A_i) ;
        foreach j such that j > i and k→j (j explicitly depends on k) do
            ASSEMBLE(T_j^k, T_j^i) ;
    FACTO(A_i) ;
    foreach j such that i→j (j explicitly depends on i) do
        UPDATE(A_i, T_j^i) ;
```

Algorithm 1.3: 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 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 handed to the parent. The following tasks are performed at each node i of the tree:

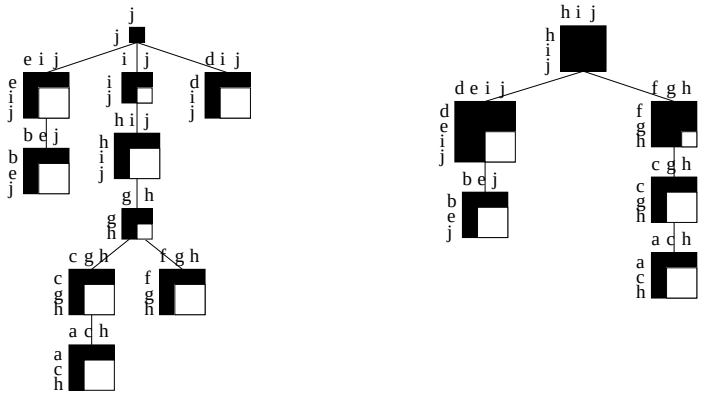
(MF-1) allocation of the frontal matrix in memory; gather entries of column i of matrix A into the first column of the front;

(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 1.4(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 [23, 14].

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 1.5. In all three methods, the nodes are processed one after the other, following a topological ordering of the elimination tree. 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



(a) Elimination tree (b) Assembly tree

Figure 1.4: Frontal matrices associated with the elimination tree (left) or to the assembly tree (right) of the matrix presented in Figure 1.2. The black part of the frontal matrices correspond to their factor block and the white part to their contribution block (that has to be assembled into their respective parent).

updated. In the *multifrontal* method, only the children nodes are accessed (to assemble the contributions blocks).

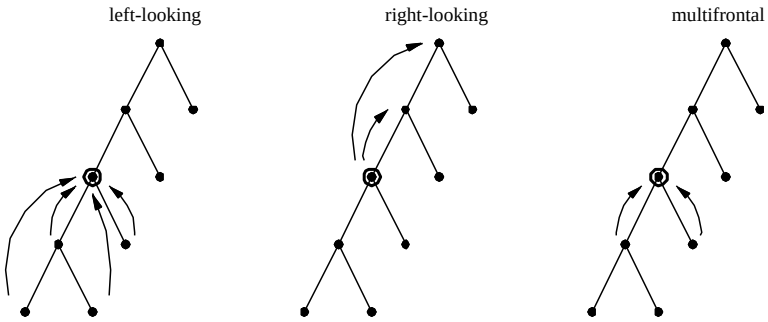


Figure 1.5: Data access pattern for the *left-looking*, *right-looking* and *multifrontal* methods.

1.1.4 Supernodes and assembly tree

The idea of a supernode[17, 32] is to group together columns of the on-going updated matrix with the same nonzero structure so that they can be treated as a dense matrix for storage and computation. A supernode is a range of contiguous columns of L with the same nonzero structure below their diagonal. When the vertices of the elimination tree are supernodes and not anymore single columns, the term *assembly tree* [17] is used. Coming back to the matrix presented in Figure 1.2, we can pick out three supernodes:

(d,e), (f,g) and (h,i,j). The assembly tree corresponding to this matrix is then given in Figure 1.6.

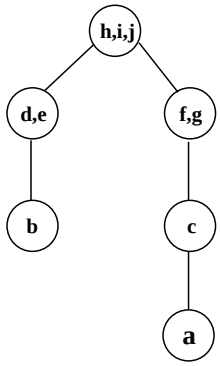


Figure 1.6: Assembly tree corresponding to the matrix presented in Figure 1.2.

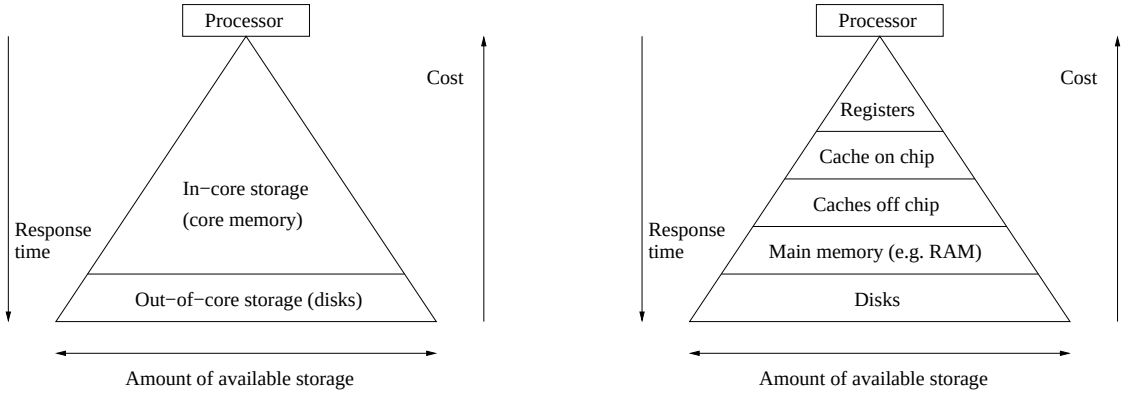
In the case of the multifrontal method, the use of supernodes impact the structure of the frontal matrices: their factor block is no longer constituted of a single column (and row) but contains as many columns (and rows) as there are nodes in the supernodes. Figure 1.4(b) illustrates the impact of the use of supernodes on the pattern of the frontal matrices.

In practice, supernodes are used in the implementation of sparse direct solvers, whatever the method (*left-looking*, *right-looking* or *multifrontal*). Here is a non exhaustive list of such solvers: `SuperLU` [25], `PaSTiX` [48], `UMFPACK` [23], `TAUCS` [1], `Oblio` [28], `PARDISO` [65, 66], `PSPASES` [47], `HSL` library [49], `SPOOLES` [15], `WSMP` [46] and `MUMPS` [9, 10].

Some of these solvers implement a parallel distributed version of the factorization algorithm with different approaches related to the type of scheduling used. For instance, `PaSTiX` [48] and `SuperLU_DIST` [53] use a fully static scheduling policy. On the contrary, `MUMPS` is based on a partial static mapping of the tasks that dynamic decisions complement in order to balance the load and improve the reactivity.

Because most modern sparse direct methods are based on supernodes, we should consider any actual method as supernodal. However, we have seen that the multifrontal method does not exactly perform its computation on the supernodes (the group of columns of the on-going updated matrix) but on frontal matrices that also involve the use of temporary data (the contribution blocks). In this dissertation, we will restrictively call *supernodal methods* methods that directly perform their computation between supernodes, *de facto* excluding *multifrontal* methods. Part I of the dissertation discusses all these methods in our context. Multifrontal methods will be studied in chapters 2, 3, 4 and 5. Supernodal methods will be separately studied in Chapter 6. In Part II, we will explore further one of these methods - the multifrontal method - in order to process very large problems in the parallel case.

1. www.tau.ac.il/~stoledo/taucs/



(a) Out-of-core architecture (b) Full storage hierarchy

Figure 1.7: Storage hierarchy.

1.2 Out-of-core sparse direct methods

Even on modern supercomputers, some applications - among which sparse direct solvers - may require an amount of memory larger than the one available on the target platform. If no specific treatment is performed, the application cannot succeed and is said to run *out-of-memory*. However modern systems can overcome this limit by using other units of storage like disks to extend the main memory and thus allowing the application to end up successfully. Such a process is said to be *out-of-core* and this design is motivated by economic reasons, disks being much cheaper than core memory. Figure 1.7(a) illustrates the out-of-core architecture. In a way, disks are just part of a set of units of storage that form a large and global *virtual memory*. From the application interface point of view, this storage is contiguous and any part of it can be addressed uniformly.

But this simple view masks a hierarchy of storage, as illustrated in Figure 1.7(b). The management of the data transfers between the different layers of storage is shared between three entities: the application, the hardware and the operating system. In a classical architecture, the application decides of data transfers between virtual memory and registers (possibly through the delegation of that work to compilers). The hardware is in charge of caching data in fast units of memory using strategies based on temporal and spatial locality paradigms. Indeed recently accessed data are likely to be re-accessed in a near future, as well as data that are close (in the virtual address space) to these recently accessed data. Finally, the operating system (possibly relying on hardware mechanisms) controls data transfers between main memory and disks according to the same paradigms.

Considering that main memory is a cache for virtual memory, one may note that the operating system and the hardware control all the layers of cache. To fully benefit from these potential cache effects, the application has in turn to be cache-friendly or,

said differently, has to respect the locality paradigms as much as possible. Because sparse direct methods have already been highly optimized to fit caches (see the methods referred in Section 1.1.4) and because - as said above - the out-of-core problem is just a particular problem of cache management, we may wonder whether it makes sense to specifically study out-of-core direct methods. We provide here three motivations:

- The amount of main memory (usually several gigabytes) is much larger than the amount of cache memory (several megabytes maximum). Such a ratio implies that the data structures that have to fit in the caches are not the same (and the subsequent mechanisms have to add one to another).
- Contrary to the data traffic between the different layers of cache memory and between cache memory and main memory, it is possible to explicitly control the data moves between disks and main memory.
- The amount of available virtual memory on supercomputers is usually not much larger than the available physical memory. One of the reasons is that the administrators may prefer to prevent users from running applications that may swap because the slow down induced may cause poor exploitation of the platform. Explicit out-of-core techniques allow use of all the available space on disks.

In the following we present a state-of-the-art of out-of-core sparse direct methods. We can distinguish between two main classes of methods: the ones based on a virtual memory mechanism - succinctly presented in Section 1.2.1 - and the ones that explicitly perform the *I/O's* at the application level - that we present in sections 1.2.2 to 1.2.5.

1.2.1 Approaches based on virtual memory

Paging (or swapping) consists in delegating the management of the out-of-core treatment to the system: it handles an amount of memory greater than the physical memory available and is composed of memory pages either in physical memory or on disk. Some authors have developed their own paging mechanism in Fortran [60]. When only relying on the system, paging mechanisms do not usually exhibit high performance [21, 61] because they have no particular knowledge of the memory access pattern of the application. However, through *paging monitoring* [20] the application can adapt the paging activity to the particularities of its memory access scheme at the level of the operating system kernel. The application can then define the priority of a page to be kept in memory and specify which pages are obsolete so that they can be freed. This improvement can reach a performance 50 % higher than the LRU (Least Recently Used) policy used by the system. However, this approach is too closely related to the operating system and not adapted when designing portable codes.

Another possible approach would consist in mapping parts of memory to files using C primitives such as `mmap`. Again, it is difficult to obtain portable code and attain good performance (even if some mechanisms like the `madvise` system call can help).

More recently Scott and Reid have proposed an out-of-core Cholesky code based on the multifrontal method in [59]. They rely on a virtual memory management system [60] running in the user space which consists in a virtual array that can be addressed by

the application. This array is divided into pages that are all of the same size. These pages can be either physically in memory or on disk. If room for a new page is required in the buffer, by default the page that was least recently accessed is written to disk (if necessary) and is overwritten by the new page. They adapt to the out-of-core case a variant of the multifrontal method that aims at reducing the storage requirement for the active memory and that was first introduced in [44]. We will discuss this multifrontal variant, called *flexible multifrontal method*, in Chapter 4.

1.2.2 Work of Liu

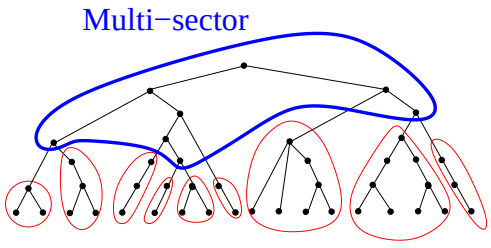
Liu proposed in [55] an algorithm to determine the tree traversal that minimizes the working storage requirement for the multifrontal method when the factors are processed out-of-core. He also suggests that the same algorithm should cut down the amount of *I/O* traffic when the working storage requirement has to be processed out-of-core. We will discuss this statement in Chapter 3.

1.2.3 Work of Rothberg and Schreiber

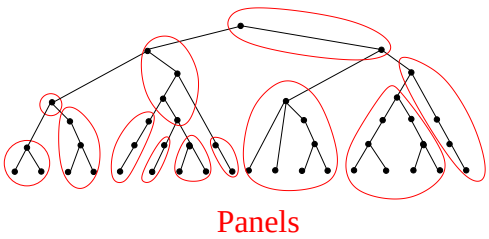
In [61] Rothberg and Schreiber propose several out-of-core direct methods for the Cholesky factorization. As the multifrontal method fits well the in-core case, they first propose an out-of-core multifrontal factorization, named `MF`. In the multifrontal method, the factors are not re-used during the factorization step. Therefore, they write them to disk as they are computed while the contribution blocks and the current frontal matrix remain in memory. However it may happen that those data cannot be held in memory either. The authors then identify the largest subtrees, called *domains*, in the elimination tree that can fit in memory (see Figure 1.8(a)). During the postorder traversal of the elimination tree, these subtrees are processed with the contribution blocks and the current frontal matrix held in core. Once the factorization of a domain is complete, the contribution block of its root node is written to disk. Once all the domains have been factored, the remaining part of the elimination tree, called *multisector* is processed in turn. When a supernode is processed, its frontal matrix may or may not fit in core. To deal with this possibility, it is divided into panels, *i.e.*, a set of adjacent matrix columns such that each panel might fit in half of the memory. These out-of-core frontal matrices are themselves factored with a *left-looking* approach. A panel is processed in three steps: (i) the panels associated to the contribution blocks of the children are read from disk and added to the current panel, (ii) each previous panel of the same frontal matrix is read back from disk and updates the current panel and (iii) the current panel is factored and written to disk. This method is robust as it never runs out-of-memory: at any time, at most two panels are together in memory.

However, when the frontal matrices do not fit in memory, they notice that this method induces a large volume of *I/O* which limits the efficiency of the factorization. They propose a hybrid approach between the *multifrontal* and *left-looking* methods to limit this

volume. The domains are still processed with the *multifrontal* method whereas the multisector is processed with a *left-looking* factorization. The size of the panels is adapted to fit the memory requirements of the *left-looking* method for which the elementary data to process is not anymore a frontal matrix but a supernode (smaller). A panel of a supernode of the multisector is processed as follows. It is initialized according to the nonzero entries of the original matrix. The previous panels are read one by one (not regarding whether they belong to a supernode of a domain or of the multisector) and update the current panel. The current panel is then factored in core and written to disk. Since the size of the panel is adapted to fit the memory requirements of the *left-looking* factorization, this method is called *pruned panel, left-looking* ($PPLL$ for short). Note that the multifrontal method applied on the domains is modified not to compute the updates from domain nodes to multisector nodes (which will be directly performed between supernodes with a *left-looking* method). They consider a variant of this hybrid method, called *pruned panel, left-looking with updates* ($PPLL_u$ for short), where the frontal matrix of the root of each domain is written to disk. This time, a regular multifrontal method is applied on the domains. When a panel in the multisector is factored, the updates from the domains are performed thanks to the use of the contribution blocks of the roots of the domains that are read back from disk. An experimental validation allowed them to show that this latter approach usually induces less I/O on very large problems and has a higher efficiency than the other ones.



(a) `MF`, $PPLL$ and $PPLL_u$



(b) `LL-RL`

Figure 1.8: Partitioning of the elimination tree for some out-of-core direct approaches.

1.2.4 Work of Toledo et al.

With various coauthors, Toledo has intensively contributed to the field of the out-of-core numerical linear algebra and out-of-core sparse direct solvers over the past ten years. Gilbert and Toledo designed a solver [41] for unsymmetric matrices (**LU** factorization). They also rely on a partition of the elimination tree into panels. Conceptually, in their case, a panel is a group of consecutive (in the postorder) columns (see Figure 1.8(b)) that can fit together in memory and that are written to disk once processed. They factorize the panels one after the other. To be processed, a panel is loaded into memory. The columns from previous panels that have to update columns of this panel are read back from disk

and the updates are performed. Because only columns that are on a single root-to-leaf path in the elimination tree may update each other, they notice that it is possible not to store all of a panel simultaneously. This property allows us to build wider panels and thus to induce less *I/O*. As we will discuss further in Chapter 6, this technique requires the use of a hybrid *left-looking/right-looking* method (*left-looking* between panels and *right-looking* within a panel). For the same purpose - getting wider panels and inducing less *I/O* - they offer the possibility to store the columns in the panel using a compressed format: since only nonzero values are stored, the panels are larger and the *I/O* volume decreases. Because they statically compute the partitioning of the elimination tree into panels before the numerical factorization step, they need to forecast the size of the panels. However partial pivoting (that may occur during the numerical factorization) modifies the forecast sparse structure of the factors and thus of the panels. This is why the authors compute an upper bound (based on [40]) on the number of nonzero values in the columns of the sparse factors instead. This upper bound takes into account any possible future change of the structure due to partial pivoting. To handle partial pivoting, part of the symbolic factorization is performed dynamically and interleaved with numerical factorization. In state-of-the-art *left-looking* in-core codes like `SuperLU`, this work is usually performed with a depth first search traversal of the pruned graph of L [25]. Because this might be costly and/or difficult to implement in an out-of-core process, Gilbert and Toledo instead use a priority queue (implemented thanks to a binary heap) to find columns of L that must update the current panel and row lists to find columns in the panel that must be updated.

The main drawback of Gilbert and Toledo's approach is that they do not use supernodes but individual column-column updates which is a serious bottleneck to achieve high-performance. Moreover, compressing the panels in a sparse format requires performing scatter-gather operations. This extra symbolic work counter-balances the advantage of decreasing the *I/O* volume obtained with compressed panels. To overcome these two drawbacks, Toledo and Uchitel proposed a variant of this code in [71]. Their main improvement is the use of supernode-panel updates (that are a variant of [25] using `BLAS-3` operations). On the other hand, they no longer use compressed panels and require that a panel fully fits in-core. This leads to narrower panels which are a bottleneck on the volume of *I/O*. Although they could reach a high efficiency on matrices of intermediate size, their experimental results showed a limited efficiency on very large problems.

The choice of giving up a *left-looking/right-looking* factorization comes from experimental results presented in [63] which showed the limited improvement on efficiency brought by this hybrid approach. The method presented in [63] is itself another (previous to [71]) extension of [41] proposed by Rotkin and Toledo for the Cholesky case. Because no dynamic pivoting occurs in a Cholesky ($\mathbf{LL}^T$) factorization, the structures can be fully forecast before the numerical factorization step. This allows us to decrease the overhead due to scatter-gather operations. Subsequently `BLAS-3` operations (supernode-supernode updates in their case) can be efficiently used with compressed panels.

This latter work was extended to the symmetric indefinite ($\mathbf{LDL}^T$) factorization by Meshar, Irony and Toledo in [58]. Their main improvement is the management of nu-

merical pivoting. The authors handle the subsequent dynamic evolution of the structure of the sparse factors with a dynamic partitioning of the elimination tree into panels that occurs during the numerical factorization step. Their pivot-search strategy is a less exhaustive variant of [16, 30] that can lead to more delayed pivots and thus increase the risk of running out of memory. However, they balance this handicap thanks to a more tolerant pivot-admissibility criterion. Indeed, they use a stability threshold (equal to 0.001) lower than the one of [16, 30] that induces less delays in the columns of the current supernode (but may induce numerical instability). The main features of all these works are summarized in Table 1.1. We will discuss further out-of-core supernodal approaches in Chapter 6.

Type of factorization	**LU**		**LL^T**	**LDL^T**
Coauthors, year [ref.]	Gilbert, 99 [41]	Uchitel, 07 [71]	Rotkin, 04 [63]	Meshar, Irony, 06 [58]
Compressed panels	√ (optional)	∅	√	√ (?)
`BLAS` level	1	3 (supernode-panel)	3 (supernode-supernode)	2-3 (`DSYTRF` variant)
`LL-RL`	√	∅	√ (optional)	∅
Pivoting	√	√	∅	√ (1-by-1 and 2-by-2)
Dynamic symbolic facto.	√ (priority queues)	√ (priority queues)	∅	√
Dynamic partitioning	∅	∅	∅	√

Table 1.1: Main features of Toledo et al. work on out-of-core direct methods.

1.2.5 Work of Dobrian and Pothen

In [27] Dobrian compares the efficiency (in terms of *I/O* volume and internal memory traffic) of the *left-looking*, *right-looking* and *multifrontal* methods (in the sequential case too). Thanks to analytical results (for model problems) and experimental ones from simulations (for irregular problems), he shows that the multifrontal method is a good choice even when the available core size is small. However, this method is not well suited for matrices whose peak of active memory is larger than the volume of factors. He concludes that to achieve good performance on a large range of problems, the solver should provide several algorithmic options including left-looking, right-looking and multifrontal as well as hybrid approaches.

1.2.6 Summary

Although both [61] and [63] insist on the problem of large frontal matrices arising in multifrontal methods, note that those large dense frontal matrices can be processed out-of-core (as done in [1]) and that in a parallel context, this may be less critical since a frontal matrix can be distributed over several processors. For this reason and also because multifrontal methods with factors and contribution blocks on disks allow to treat large problems (though not arbitrarily large), we aim at studying further the potential of the multifrontal method in chapters 3 and 4. Even though supernodal methods have been intensively studied, several points remain to be improved. First, the supernodal method that minimizes the *I/O* volume is an open problem (although [63] provides contributions

in that direction). Next, in the **LU** case, to our best knowledge, the codes that use efficient (cache-friendly) kernels of computations had difficulties to keep a high efficiency when processing very large problems: this illustrates the fact that the out-of-core problem is not the same as the cache problem but - on the contrary - brings one more degree of complexity. Finally, algorithms that aim at partitioning the matrix into panels have not been intensively studied and the choice of bottom-up approaches - that have been used so far - may not be suited. Chapter 6 discusses these issues.

1.3 I/O mechanisms

I/O mechanisms are essential for out-of-core applications as their own performance directly impacts the application's performance. We give below an overview of some existing *I/O* tools.

C library. Several C standard *I/O* routines are known to be efficient. The *read/write* functions allow one to read from or write to a file descriptor. To perform non sequential (random) file accesses, the *lseek* function can be used. It allows one to set the file offset for the file descriptor given as an argument. However, if the file descriptor is a shared memory object (as can occur in a multi-threaded environment) the result of *read* is unspecified. In particular, the section in between *lseek* and *read* calls has to be in mutual exclusion. Moreover, because of a lack of synchronization, *I/O* cannot be optimized at the kernel level (for instance it is not possible to use *RAID* [22] acceleration). The same type of problem occurs with the *write* routine. The *pread/pwrite* functions are equivalent to *read/write*, except that they read from a given position in the file without changing the file pointer. The management of the offset is directly handled by the kernel which takes care of the synchronizations and may optimize actual disk accesses. Finally the *fread/fwrite* functions allow to read from or write to a binary stream. All these functions are buffered at the system level which is not convenient in an out-of-core environment where duplication of information is usually not wished. However, it can still be useful in environments on which the other functions are not available.

Building a complete efficient asynchronous *I/O* mechanism based on those system calls is not immediate because a robust communication scheme between an *I/O* thread that manages disk accesses and the computational thread has to be built.

The new Fortran 2003 library includes an asynchronous *I/O* API as a standard. But this is too recent to be portable.

AIO. AIO is a POSIX asynchronous *I/O* mechanism. It should be optimized at the kernel level of the target platform and then permits high performance. However, its availability and portability are not so good.

MPI-IO. The Message Passing Interface MPI has an efficient *I/O* extension MPI-IO [70] that handles *I/O* in a parallel environment. However, this extension aims at managing parallel *I/O* applications which is not our target in this dissertation: disks

are there to extend the memory of each process and we are not planning to share out-of-core files between several processes.

FG. The FG [19] framework for high-performance computing applications aims at making *I/O* designing more efficient. It allows the developer to use an efficient asynchronous buffered *I/O* mechanism at a high level. However we will not use it at the moment because it manages concurrent *I/O* threads whereas, in our case, *I/O* threads do not interfere with each other.

Different file access modes. By default, system caches (pagecache) may be used by the operating system to speed-up *I/O* requests. The management of the pagecache is system-dependent and not under user control (it is usually managed with a `LRU` policy and its size may vary dynamically). Depending on whether the data is copied to the pagecache or written to disk (for example when the pagecache is flushed), the performance of *I/O* operations may vary. In order to enforce a synchronization between the pagecache and the disk, the *O_SYNC* flag can be specified when opening a file. However, the user still has no control on the size of the pagecache in this context: depending on the pagecache management policy, the behaviour of the user-space applications may be perturbed by the virtual memory mechanisms.
One way to avoid the caching at the system level consists in using direct *I/O*. This feature exists in various operating systems. In our experimental environment, it can be activated by specifying the *O_DIRECT* flag when opening the file. Note that data must be aligned in memory when using direct *I/O* mechanisms: the address and the size of the buffer must be a multiple of the page size and/or of the cylinder size. The use of direct *I/O* operations ensures that a requested *I/O* operation is effectively performed and that no caching is done by the operating system.

In the code used in the experiments in Part II, we decided to implement two *I/O* mechanisms: one based on the standard C *I/O* library and one based on AIO. However, since AIO was not portable on several of our target computers, we will only discuss results obtained with the standard C *I/O* library.

1.4 Experimental environment

1.4.1 Software environment

In order to experiment and validate the algorithms proposed in this dissertation, we have used two sparse direct solvers:

`MUMPS`. The `MUMPS` library allows for the solution of large sparse linear systems with symmetric positive definite matrices, general symmetric matrices and general unsymmetric matrices with symmetrized structure. It handles real and complex arithmetics in both single and double precisions. `MUMPS` implements a multifrontal method and relies on a symmetric elimination tree. It offers a serial and a parallel version. The serial version will be instrumented to model an out-of-core behaviour

of the multifrontal methods. We present the related results in chapters 2, 3 and 4. The parallel version is based on a distributed memory asynchronous algorithm. The purpose of Part II is to study and extend this code in order to process matrices as large as possible in an out-of-core context. A strength of this software is the large panel of provided features that are compatible with each other. On the other hand, maintaining these functionalities in an out-of-core context represents a major difficulty. For instance, one may wonder how to handle 2x2 pivoting in an asynchronous distributed memory out-of-core environment. MUMPS is available from the http://graal.ens-lyon.fr/MUMPS/ and http://mumps.enseeiht.fr/ webpages.

SuperLU. SuperLU is a general purpose library for the direct solution of large, sparse, unsymmetric systems of linear equations. It supports both real and complex data types, in single or double precision. The SuperLU package comes in three different flavours: a sequential version, a version for shared memory parallel machines and a version for distributed memory environments. In this dissertation Section 6, we will focus on the sequential version to illustrate our study of supernodal methods in an out-of-core context. This version implements a supernodal *left-looking* factorization and relies on a column elimination tree. It handles partial pivoting which consists in using as a pivot the largest absolute value of the column being factored. This requires to swap the row of the selected pivot with the one that matches the diagonal. Therefore, partial pivoting prevents to forecast the structure of the factors before performing the actual numerical step, which turns out to be a major difficulty in an out-of-core context as we will see in Section 6. SuperLU is available from the http://crd.lbl.gov/~xiaoye/SuperLU webpage.

As discussed in Section 1.1.1, ordering algorithms have a strong impact on the efficiency of the factorization. As one may presage, it is also the case in an out-of-core context. This is why we aim at testing our algorithms with different ordering libraries. We will see for instance in Chapter 3 that the impact on the I/O volume of our algorithms varies depending on whether the ordering algorithm provides a well balanced elimination tree. An ordering may be directly interfaced with the solver. MUMPS is interfaced with Approximate Minimum Degree (AMD), Approximate Minimum Fill (AMF), PORD, METIS and SCOTCH. SuperLU allows the use of natural ordering, a Multiple Minimum Degree (MMD) applied to the structure of A^TA, Multiple Minimum Degree (MMD) applied to the structure of $A^T + A$ and Column Approximate Minimum Degree (COLAMD). Note that COLAMD is particularly designed for unsymmetric matrices with unsymmetric structure when partial pivoting is needed. It does not require the explicit formation of A^TA, usually gives comparable orderings to MMD on A^TA, and is faster. The main reason why MUMPS and SuperLU do not interface the same orderings is that MUMPS processes matrices with symmetric structure whereas SuperLU allows one to handle an unsymmetric structure. Another possibility for testing an ordering library consists in performing the ordering phase in an environment independent from the direct solver, to record the result into a file and to reuse this file as an entry to a direct solver. Both MUMPS and SuperLU allow this feature. This latter method is furthermore required to perform reproducible experi-

ments and get comparable results between different experiments. Indeed, some ordering libraries like `METIS` use random algorithms and thus do not necessarily provide identical results from one execution to another.

1.4.2 Hardware environment

During the experiments, we have used different types of hardware platforms. The use of different architectures allows for a better validation. Particular attention has been paid to using different configurations of the *I/O* system: disks local to the processor, local to the node or remote disks. Here are the three main computers used.

PSMN/FLCHP. This platform is a cluster of Linux dual-processors at 2.6 GHz from PSMN/FLCHP [2] 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. With this configuration, the disks are thus local to the processors (one disk per processor). The observed bandwidth is 50 MB / second per node, independently of the number of nodes, and the filesystem is *ext3*.

IDRIS. This machine is an IBM SP system from IDRIS [3] 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` [67] 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, allows to run large problems in-core on which we can compare out-of-core and in-core performance, and gives a feedback on a widespread parallel filesystem.

Borderline. This machine is an IBM System x3755 using AMD Opteron 2218 dual-core processors at 2.6 GHz. It is composed of 10 nodes. Each node has 4 processors (8 cores), 32 GB of memory, and a local disk. The machine uses a Myri-10G network.

2. Pôle Scientifique de Modélisation Numérique/Fédération Lyonnaise de Calcul Haute Performance
3. Institut du Développement et des Ressources en Informatique Scientifique

1.4.3 Test problems

We now briefly present the test problems we will use in the dissertation. The main test problems are gathered in Table 1.2. Some other test problems that are punctually used are not presented in this table. However, all these test problems come from standard collections (PARASOL collection [4], University of Florida sparse matrix collection [5]), or from MUMPS users. Publicly available matrices from MUMPS users are available on the gridtlse.org website (TLSE collection).

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).
MHD1	485597	24233141	UNS	1222.8	8500	Unsymmetric magneto-hydrodynamic 3D problem, provided by Pierre Ramet.
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).
TWOTONE	120750	1224224	UNS	27.3	33.5	AT&T,harmonic balance method, two-tone. More off-diag nz than onetone (Univ. Florida collection).
QIMONDA07	8613291	66900289	UNS	556.4	45.7	Circuit simulation problem provided by Reinhart 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 1.2: Main test problems. Size of factors ($nnz(L|U)$) and number of floating-point operations (Flops) computed with METIS.

4. http://www.parallab.uib.no/parasol
5. http://www.cise.ufl.edu/research/sparse/matrices/

Part I

Models and algorithms for out-of-core serial sparse direct methods

Chapter 2

On the volume of I/O: Case study with the multifrontal method

In this chapter, we introduce the fundamental notions we will deal with in the whole dissertation. We study them in the context of the out-of-core multifrontal method. The particularity of this method, that makes it specially interesting to study, is that data are accessed with a stack mechanism, as we explain in Section 2.1. The volume of *I/O* is the amount of traffic between disks and core memory. Reducing the volume of *I/O* is critical in out-of-core applications because disk accesses are very slow compared to core memory accesses. The volume of *I/O* is thus a possible bottleneck on efficiency. However, some *I/O* are necessary when the amount of storage required to process a sparse matrix is larger than the amount of core memory available on the platform used. The volume of *I/O* is thus closely related to the amount of core memory and to the evolution of the storage requirement. In Section 2.2, we explain how to deduce the (minimum) volume of *I/O* from the evolution in time of the storage requirement. We then show how this volume evolves depending on the available memory. We isolate a key property that characterizes this evolution and which is also worth studying to get familiar with the notion of *I/O* volume. Some notations are introduced in Section 2.3. In Section 2.4, we explain how to compute the (minimum) *I/O* volume using a formulation based on the elimination tree of the matrix. This result is used to instrument a direct solver based on the multifrontal method, `MUMPS`. In Section 2.5, we use this instrumented solver to illustrate the impact of the available memory on the *I/O* volume with experiments on real-life matrices. We finally discuss several possible models and their respective limits in Section 2.6.

2.1 Out-of-core multifrontal methods

In the multifrontal method, the frontal matrices are processed one after the other following a postorder of the elimination tree. A postorder tree traversal allows for a `LIFO` (Last In First Out) access to the contribution blocks. We illustrate this property on the elimination tree given in Figure 2.1. Nodes 1, 2, 3, 4, 5 are factored. Next, the frontal

matrix of node 6 is allocated. The contribution blocks of nodes 5 and 4 (in that order) are assembled into that frontal matrix which can in turn be factored. The contributions blocks of nodes 6, 3, 2 and 1 are assembled into the frontal matrix of node 7 which is finally factored. Thanks to their `LIFO` access, the contribution blocks can be managed

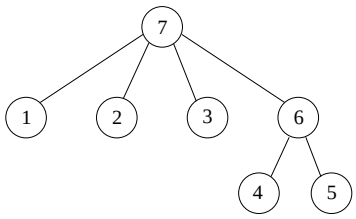


Figure 2.1: Postorder numbering of an elimination tree.

with a stack mechanism: when a frontal matrix is allocated, the contribution blocks of its children are *popped* (and assembled in the frontal matrix); the frontal matrix is then factored and its own contribution block is *pushed* into the stack.

Figure 2.2 represents the frontal matrix of a node in the tree. The *fully summed block* is factored and will not be re-used before the solution step. The non-fully summed block cannot be factored yet but will be updated and used later at the parent node, after it has been replaced by a *Schur complement* or *contribution block*. Because factors are not re-accessed during the factorization step, they can be written to disk as soon as possible in order to free some storage. In this context, the core memory only contains contribution blocks waiting to be assembled and the current frontal matrix.

Together, the current frontal matrix and the stack of contribution blocks constitute what is called the *active memory* in the literature. However, because this active memory may represent an amount of data larger than the available core memory, it may have to be processed out-of-core too. Part of this *active memory* may thus have to be stored on disks. This is why we prefer to call it *active storage*, the *active memory* being in our context the part of the active storage that is kept in main memory.

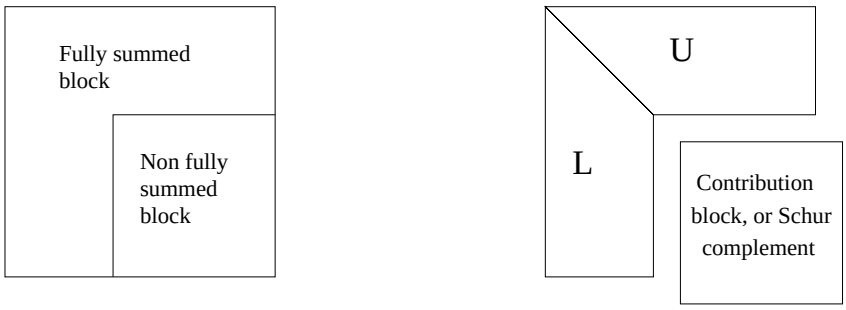


Figure 2.2: Frontal matrix before (left) and after (right) the partial factorization of step (MF-3) in the structurally unsymmetric case. In an out-of-core context, the L and U factor blocks are written to disk whereas the contribution block may be stacked in core memory.

In a limited memory environment, one drawback of multifrontal methods comes from large dense matrices that give a lower bound on the minimum core memory requirements. However, those dense matrices may fit in memory, or they can be treated with an out-of-core process. Apart from these dense matrices, the out-of-core multifrontal method follows a write-once/read-once scheme, which is an interesting property when studying the volume of *I/O*. In this dissertation, we assume that frontal matrices fit in core memory.

2.1.1 Assumptions related to out-of-core multifrontal methods

Assumption 2.1. *We assume that:*

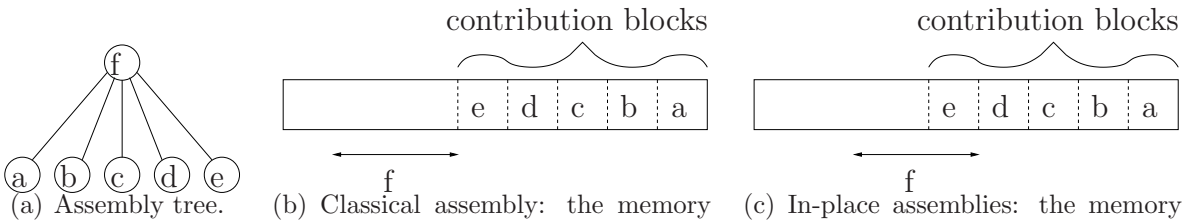
- *Factors are written to disk as soon as computed;*
- *contribution blocks are processed out-of-core when the core memory is full, possibly partially;*
- *the active frontal matrix is held in-core;*
- *the original matrix has been ordered by some fill-reducing ordering.*

This implies that the volume of *I/O* for the factors is constant and that there is no *I/O* for the frontal matrices. Therefore, in this chapter, the volume of *I/O* will only refer to the volume of *I/O* performed on the stack of contribution blocks.

2.1.2 Variants of the multifrontal method

In this dissertation we consider several variants of the multifrontal method that we briefly introduce here and which correspond to different existing and/or possible implementations of the method. A first set of variants is related to the assembly scheme of the first contribution block that is assembled into the frontal matrix of the parent. 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 as illustrated in Figure 2.3(b). This scheme is implemented for instance in the `MA41` solver [14, 50].
- The *last-in-place* assembly scheme: the memory of the frontal matrix at the parent node is allowed to overlap with the contribution block of the last child processed (which is the first one assembled) as illustrated in Figure 2.3(c). To do so, the first contribution block assembled is expanded in place to form the frontal matrix. We save space by not summing the memory of the contribution block of that child with the memory of the frontal matrix of the parent. This scheme is available for example in a code like `MA27` [30].
- The *max-in-place* assembly scheme: we overlap the memory for the frontal matrix of the parent with the memory of the child having the largest contribution block (even if that child is not processed last). This is a new variant of *in-place* assembly; we will describe it more accurately in Section 3.3.



(a) Assembly tree.

(b) Classical assembly: the memory for the frontal matrix f of the parent does not overlap with the memory of the stack of contribution blocks.

(c) In-place assemblies: the memory for the frontal matrix f of the parent is allowed to overlap with the memory of the first contribution block assembled.

Figure 2.3: 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.

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 two such cases in this dissertation:

- The terminal allocation scheme: the memory for the frontal matrix is allocated after all the children have 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.

In this chapter, we focus on the multifrontal method with a *classical* assembly scheme and a terminal allocation of the frontal matrix. In Chapter 3 we will focus on the terminal allocation case in the context of which we examine the three possible assembly schemes quoted above. Chapter 4 will discuss those three assembly schemes in the context of the flexible allocation variant. Chapter 5 presents several memory management algorithms that fit the different possible combinations of assembly and allocation schemes (*max-in-place* assembly scheme with terminal allocation, *last-in-place* with flexible allocation, and so on).

2.2 Impact of the available core memory and of the evolution of the active storage on the I/O volume

Because the contribution blocks are produced once and accessed once, they will be written and read to/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 2.1:

Objective 2.1. *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 2.2. *Can we characterize the (optimum) volume of* I/O *as a function of the available core memory* M_0 *?*

Actually, Objective 2.1 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 2.1 states this result in other words:

Property 2.1. *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 dissertation (in the context of the multifrontal method) that the *I/O's* on the stack of contribution blocks are performed with respect to Property 2.1.

In particular, we can deduce the following result that aims at answering to Objective 2.2:

Property 2.2. *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 being technical, we report it in Appendix B and prefer to illustrate this behaviour here on simple examples.

In Figure 2.4(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 2.4(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

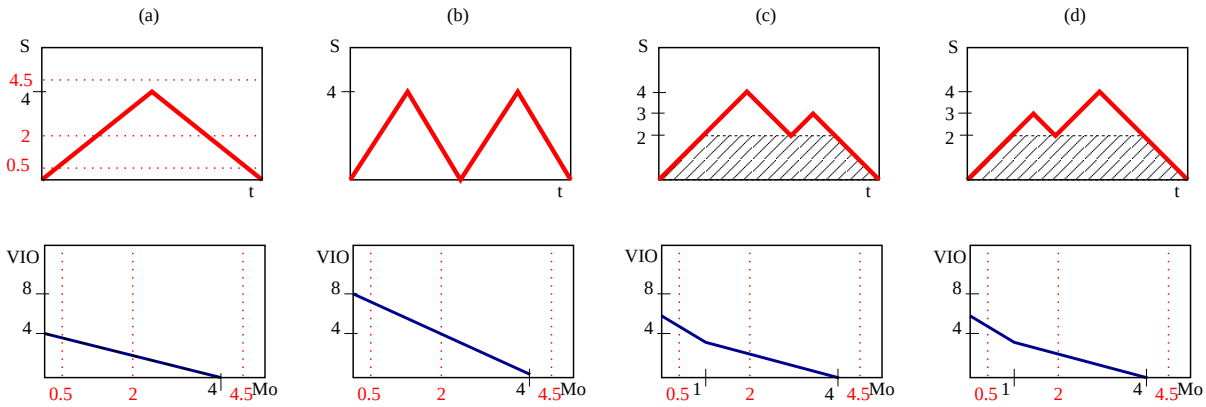


Figure 2.4: Evolution of the storage requirement of a stack (top) and I/O volume as a function of the available memory (bottom) on four examples (a, b, c and d).

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 2.4(c). In that case, we start doing I/O again when the physical memory available M_0 becomes smaller than the 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 only be processed out-of-core 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 2.4(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 $y(M_0) = \textit{peak storage} - M_0$, 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 following section 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 2.5.

2.3 Notations

Before discussing further the volume of I/O, we introduce some general notation. In a limited memory environment, we define M_0 as the amount of core memory. As described above, the multifrontal method is based on a tree in which a parent node is allocated in memory after all its child subtrees have been processed. When considering a generic parent node and its n children numbered $j = 1, \ldots, n$, we note:

- cb / cb_j, the storage for the contribution block of the parent node / of child j (note that $cb = 0$ for the root of the tree);
- m / m_j, the storage of the frontal matrix associated with the parent node / to its j^{th} child (note that $m \geq cb_j$, $m_j \geq cb_j$, and that $m_j - cb_j$ is the size of the factors produced by child j);
- S / S_j, the storage required to process the subtree rooted at the parent / at child j (note that if $S_j \leq M_0$, no I/O is necessary to process the subtree rooted at j);
- 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 node j given an available memory of size M_0.

2.4 Formal expression of the I/O volume

We now compute the volume of I/O on the stack of contribution blocks under Assumption 2.1. Recall we rely on a terminal allocation multifrontal method with a *classical* assembly scheme.

When processing a child j, the contribution blocks of all previously processed children have to be stored. Their memory size is added to 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 theses values, that is:

$$S = \max\left(\max_{j=1,n}(S_j + \sum_{k=1}^{j-1} cb_k), m + \sum_{k=1}^{n} cb_k\right) \tag{2.1}$$

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 [55]), allows us to determine the storage requirement for the complete tree.

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 and the amount of data that must be written to disk is given by Property 2.1. As discussed in Section 2.2, the bottom of the stack should be written first.

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 storage requirement at the moment when the peak S is obtained and the amount M_0 of memory available to process it. 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 $V^{I/O}$, using Formula (2.1) for the storage requirement, is:

$$V^{I/O} = \max\left(0, \max(\max_{j=1,n}(S_j + \sum_{k=1}^{j-1} cb_k), m + \sum_{k=1}^{n} cb_k) - M_0\right) \tag{2.2}$$

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

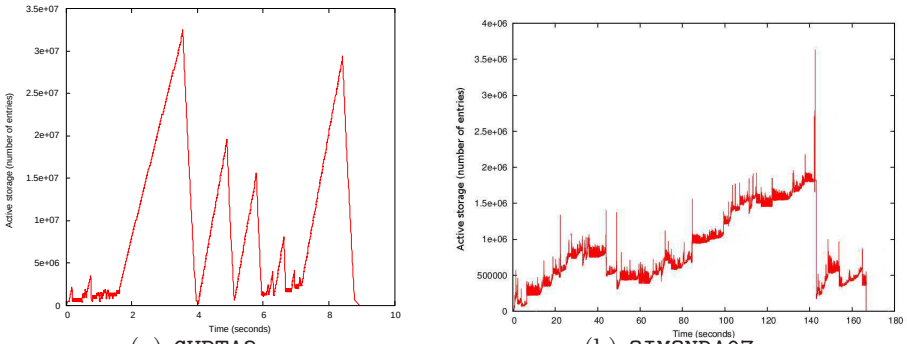
We now suppose that there exists a child j such that $S_j > M_0$. We know that the subtree rooted at child j will have an intrinsic volume of I/O $V_j^{I/O}$ (recursive definition based on a bottom-up traversal of the tree). Furthermore, we know that the memory for the subtree rooted at child j cannot exceed the physical memory M_0. Thus, we will consider that it uses a memory exactly equal to M_0 ($A_j \stackrel{def}{=} \min(S_j, M_0)$), and that it induces an intrinsic volume of I/O equal to $V_j^{I/O}$. 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 (2.2). We obtain:

$$V^{I/O} = \max\left(0, \max(\max_{j=1,n}(A_j + \sum_{k=1}^{j-1} cb_k), m + \sum_{k=1}^{n} cb_k) - M_0\right) + \sum_{j=1}^{n} V_j^{I/O} \tag{2.3}$$

To compute the volume of I/O on the complete tree, we recursively apply Formula (2.3) at each level (knowing that $V^{I/O} = 0$ for leaf nodes). The volume of I/O for the factorization is then given by the value of $V^{I/O}$ at the root.

2.5 Experiments on real problems

Figure 2.5 presents the active storage requirement as a function of time during the numerical factorization. This experiment corresponds to a sequential execution of the multifrontal solver `MUMPS` (see Section 1.4.1) in which we monitored the evolution of the



(a) GUPTA3. (b) QIMONDA07.

Figure 2.5: Evolution of the active storage during the numerical factorization of two matrices: GUPTA3 and QIMONDA07. The storage for the factors is 1.01×10^7 (real entries) for GUPTA3, and 5.58×10^8 for QIMONDA07.

active storage. We use two matrices to illustrate two very different behaviours: GUPTA3 and QIMONDA07 (see Table 1.2). For GUPTA3, we observe several large peaks of storage, that have similar orders of magnitudes. Between each peak, the storage almost reaches 0; this means that there is no hope to keep common data in core memory between those peaks. For the QIMONDA07 matrix, there is really only one (relatively small) peak of active storage. As done earlier for simpler cases (see Figure 2.4), it is interesting to relate such shapes of storage evolution to the *I/O* volumes. Because there are several large peaks of storage in the first case, we expect the steepness of the function providing the *I/O* volume to grow significantly when the available memory decreases. On the opposite, we expect a situation similar to the one of Figure 2.4(a) for the second case, with an *I/O* volume curve of steepness -1 for a wide range of core memory values. We report in Figure 2.6 the *I/O* volume computed thanks to Formula (2.3) (MUMPS is used to generate the tree data structure associated with each matrix) as a function of the available memory for several matrices extracted from Table 1.2. The graphs corresponding to GUPTA3 and QIMONDA07 confirm the above expectations regarding the steepness of the curves obtained.

For all problems the results also confirm that the geometrical shape obtained is a piecewise affine function (as discussed in Section 2.2). When the available memory is large, the *I/O* curve matches the line of equation $y(M_0) = \textit{peak storage} - M_0$; when the available memory becomes smaller, the volume of *I/O* increases faster and the steepness of the curve increases.

Another interesting phenomenon is that, for most matrices, the volume of *I/O* for the contribution blocks remains reasonable in comparison to the volume of *I/O* for the factors. Two extreme cases are, again, QIMONDA07 and GUPTA3. As seen earlier, the evolution of the active storage for the QIMONDA07 matrix (arising from circuit simulation) implies that the *I/O* volume never exceeds the straight line of equation $y(M_0) = \textit{peakstorage} - M_0$ (at least in the memory range where one frontal matrix fits in memory). Furthermore, it represents less than 1% of the volume of factors. Thus treating the stack out-of-core would be cheap. However, the peak storage (29 MB) is so small compared to the volume

of factors (7.2 GB) that treating the stack out-of-core is not necessary: once factors are on disk, the stack can be kept in-core. On the contrary, with the GUPTA3 matrix (arising from a linear programming problem), the peak storage is larger than the size of the factors, and the volume of *I/O* is even larger because there are many peaks (increased steepness). Another remark we can make on the GUPTA3 matrix is that the frontal matrices are small; subsequently with an out-of-core management of the contribution blocks, we can process it with a very little amount of core memory (compared to the memory required to process it with an in-core stack) ... at the cost of performing a huge amount of *I/O* (up to 11.2 times the volume of factors).

As a conclusion, for most matrices, the volume of *I/O* on the contribution blocks remains reasonable (but not negligible), even when the amount of available memory is small. However, some extreme cases may either be processed with an in-core stack even with a small amount of memory (QIMONDA07 matrix) or, on the contrary, induce a huge amount of *I/O* (compared to other metrics of the matrix) when only a small amount of memory is available (GUPTA3 matrix). To limit the impact of *I/O's* on efficiency, they have to overlap with computations. We explain in the following section that it is not always possible and propose several models to handle the assembly step of the multifrontal method in an out-of-core context.

2.6 Models for an out-of-core assembly scheme

Processing the contribution blocks out-of-core not only means that they have to be written to disk but also that they have to be read back and assembled into their parent. In other words, an out-of-core assembly scheme is required. Figure 2.7 illustrates the different schemes we have modeled for the out-of-core 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.

These schemes can be viewed as an extension of a scheme where only factors are processed on disk:

- `Factors-on-disk` **scheme.** In this reference scheme, only factors are on disk and the whole active storage remains in core memory. Note the implementation of such a scheme is described (in the parallel context) in Chapter 8.

In the `All-CB` scheme, all the data required can be prefetched before performing the assembly step of the parent node. This should allow to perform computations (extend-add operations) at a high rate as in the in-core case. On the contrary, for the `Only-Parent` and `One-CB` schemes, the assembly operations have to be interrupted by *I/O*. Moreover, the volume of extend-add operations is proportional to the number of data assembled. If those data are read from disk, we thus have to expect that subsequent *I/O's* cannot fully overlap with computation. The disk throughput then becomes a bottleneck inducing an overhead on the execution time.

On the other hand, the choice of the use of an assembly scheme determines the minimum amount of core memory required to process a matrix. Said differently, memory schemes have different memory requirements: each scheme has its own domain of applicability, corresponding to the memory range for which the factorization succeeds when the scheme is applied. Such a domain is defined by the lowest value for which the scheme succeeds (because then any larger value will lead to a successful factorization too). We have reported these domains in Figure 2.6: The `Only-Parent` scheme can be applied if the largest frontal matrix fits in memory. The whole represented part of the graph in Figure 2.6 matches this criterion. The `One-CB` domain of applicability is included in the `Only-Parent` one. Its leftmost limit is represented by a vertical plain line in Figure 2.6. The `All-CB` domain of applicability is included in the `One-CB` one. Its leftmost limit is represented by a vertical dashed line in Figure 2.6.

This figure shows that the `Only-Parent` scheme has a significantly wider range of applicability than the other ones. It usually makes it possible to factorize matrices with an amount of memory around 25 % smaller than the `One-CB` scheme and 50 % smaller than the `All-CB` scheme. We also notice that the gap to the `All-CB` scheme can be huge on some matrices that induce a large stack (`GUPTA3` matrix, for example).

Note that when several schemes can be applied for a given amount of available memory, they induce the same amount of *I/O*. This is due to the fact that the contribution blocks are managed with a stack mechanism. Let us take the example of the `One-CB` and `All-CB` schemes to illustrate this property. For both schemes, write operations are the same: when a new frontal matrix is allocated, we may need to write some contribution blocks to disk; those contribution blocks are the ones at the bottom of the stack and the volume written does not depend on the assembly scheme. When assembling a frontal matrix, the required contribution blocks are at the top of the stack. Either they are in memory, and in this case they can be assembled directly, or they are on disk, in which case no other contribution block from other part of the tree is in memory. In the latter case, they will be read one by one (`One-CB`) or all at once (`All-CB`), but the volume read is the same. Consequently, a single curve can represent the volume of *I/O* of all schemes; only the domain of applicability of each scheme changes, as presented in Figure 2.6.

To summarize, the study of the applicability of these different schemes shows that it is important to implement all the mechanisms: indeed, on one hand, to significantly limit the memory requirements, it is important to have the possibility to perform a `Only-Parent` assembly step; on the other hand, it is also important to assemble frontal matrices with a `One-CB` or even a `All-CB` scheme when memory allows for it so that read

operations can be prefetched. This study is also a basis for Chapter 9 in which we will discuss the possible extension of these models to the parallel case.

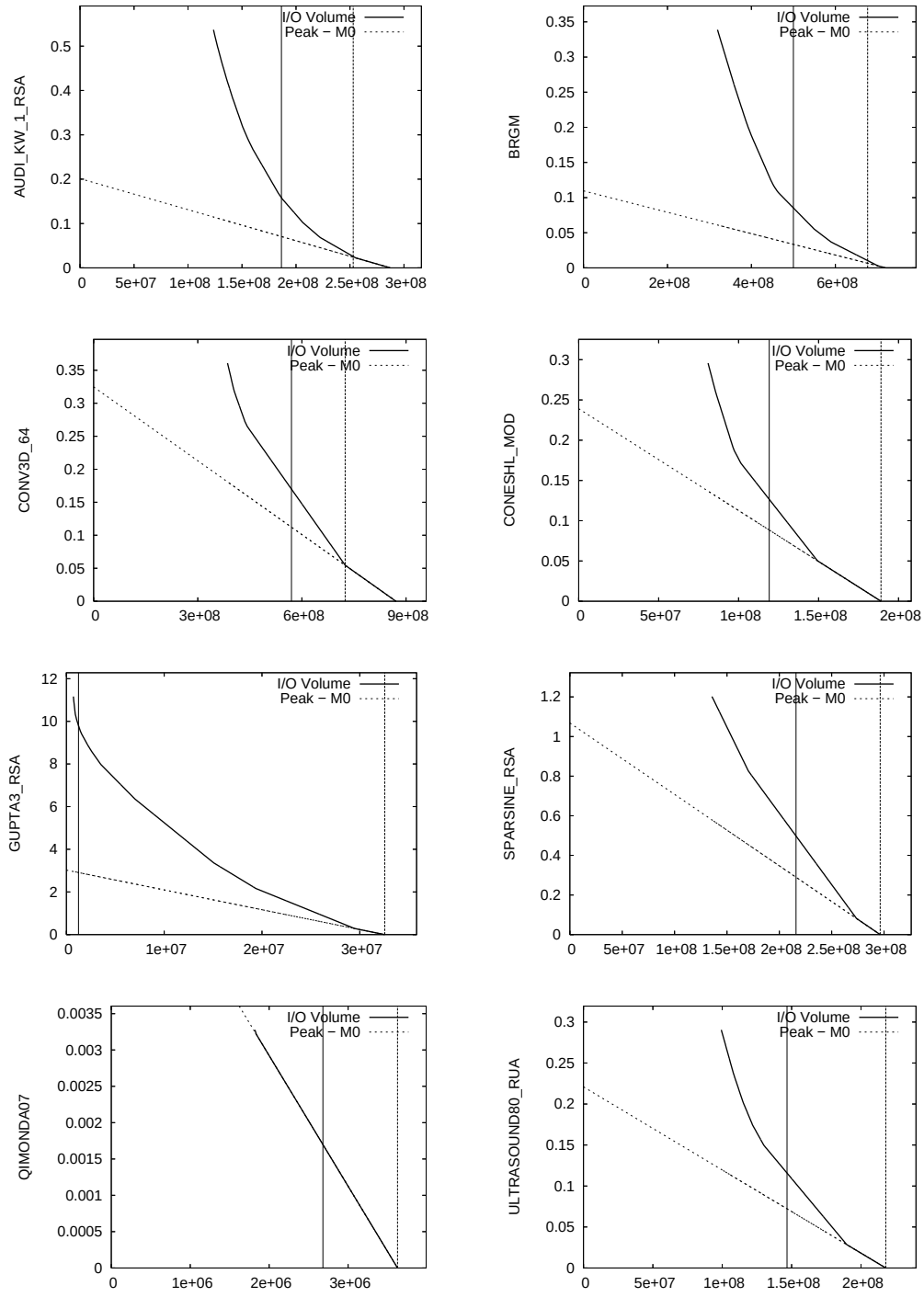
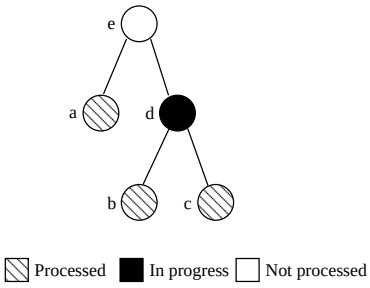
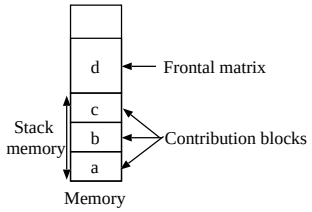
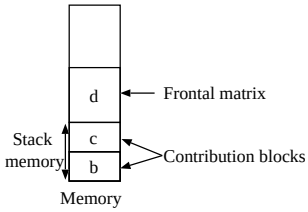


Figure 2.6: **Volume of *I/O*** (**y-axis**) related to the active storage divided by the volume of factors **depending on the available memory** M_0 (**x-axis**, expressed in number of real entries), for several problems. A lower bound for each curve is given by the dashed line of equation $y(M_0) = (\textit{peak storage} - M_0)/\textit{volume of factors}$. The vertical plain (*resp.* dashed) line represents the minimum amount of memory necessary for processing the matrix with a `One-CB` (*resp.* `All-CB`) scheme. The minimum memory for the `Only-Parent` scheme corresponds to the leftmost part of the curves.

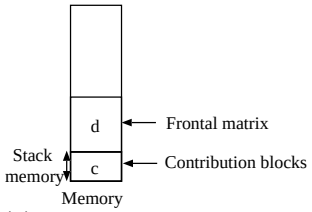




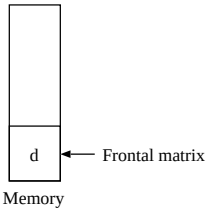
(a) `Factors-on-disk` scheme: only factors are on disk.



(b) `All-CB` out-of-core scheme. a is also on disk.



(c) `One-CB` out-of-core scheme. a and b are also on disk.



(d) `Only-Parent` out-of-core scheme. All contribution blocks are also on disk except one block of rows of c.

Figure 2.7: Out-of-core assembly schemes for the contribution blocks. Left: frontal matrix of 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.

Chapter 3

Reducing the I/O volume in the multifrontal method: Terminal allocation

The efficiency of the factorization of a sparse matrix strongly depends on the order of elimination of its variables. Sparse matrices are thus usually pre-processed to order the variables. These ordering algorithms (some of which have been introduced in Section 1.1.1) usually aim at decreasing the fill-in and thus reduce the storage requirement (as well as the amount of computation). Obviously, these pre-processsing techniques are even more important in an out-of-core context. However, once the variables have been ordered, there is still some freedom to reorder them without impacting the structure of the sparse factors. It is the case for instance, as introduced in Section 1.1.2, for any reordering that respects the dependencies of the elimination tree. More formally, an *equivalent* reordering [56] P of the (ordered) matrix A is a permutation matrix such that the filled graph of A has the same structure as that of PAP^T. It is known that equivalent orderings require the same amount of arithmetic for the sparse Cholesky decomposition of their permuted matrices (see for example [33]). Therefore, equivalent orderings result in the same storage for the factors and the same computational costs. They thus represent an excellent tool to improve some other metrics without decreasing the quality of the original ordering.

A natural class of equivalent orderings is the one of *topological* orderings [56]. A topological ordering on a rooted tree is one that numbers the children nodes before their parent nodes. Said differently, it is an ordering that respects the dependencies of the elimination tree. They offer some freedom in the tree traversal that can be exploited to decrease the amount of active storage required to process a matrix with the multifrontal method. Among the topological orderings, the *postorder* traversals are closely associated with the multifrontal method since they allow us to manage the contribution blocks with a stack mechanism. In [55], Liu proposed a postorder traversal that minimizes the active storage. In the multifrontal method with factors on disk (see Algorithm 3.1) this technique makes it possible to process larger problems. Liu suggested at the end of [55] that

foreach *node* ν **in** *the tree (postorder traversal)* **do**
  $al_{\mathcal{N}}(x)$: Allocate memory (of size x) for the frontal matrix associated with ν ;
  if ν *is not a leaf* **then**
    $as_{\mathcal{N}}(y)$: Assemble contribution blocks from children (of total size y);
  $f_{\mathcal{N}}(z)$: Perform a partial factorization of the frontal matrix of ν writing factors (of size z) to disk on the fly;

Algorithm 3.1: Multifrontal method with factors on disk.

minimizing the active storage is also well adapted when the stack of contribution blocks is processed out-of-core (as in the previous chapter). This conjecture raises the following question: *is the minimization of the active storage equivalent to the minimization of the volume of* I/O *?* Here are thus the assumptions and objective of this chapter. We rely on Assumption 2.1 and Property 2.1. We have seen in Chapter 2 that the use of this property results in an optimum volume of *I/O* for a given postorder of the elimination tree. In this chapter, we aim at reaching Objective 3.1:

Objective 3.1. *Given an amount of available core memory M_0, our purpose is to find the postorder that minimizes the* I/O *volume on the contribution blocks.*

We consider this problem in the context of the multifrontal method with terminal allocation and study the variants of the assembly scheme that have been introduced in Section 2.1.2: the *classical*, *last-in-place* and *max-in-place* assembly schemes.

We show that minimizing the storage requirements is different from minimizing the volume of *I/O*. For each variant of the allocation scheme, we present an algorithm that minimizes the storage requirement, called `MinMEM` (since it allows us to process a given problem with less memory); then, we describe a new algorithm called `MinIO` that, depending on the physical memory available, aims at finding the postorder that minimizes the *I/O* volume. We show that `MinIO` algorithms can generate a volume of *I/O* arbitrarily lower than `MinMEM` algorithms on contrived examples. We then show on real-life problems that these `MinIO` algorithms may significantly reduce the volume of *I/O* compared to the `MinMEM` approaches that focus on the storage requirements (such as [55]).

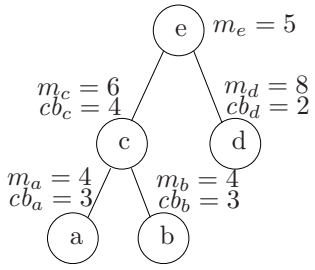
The chapter is organized as follows. In Sections 3.1 and 3.2, we explain how to find the postorder tree traversal that minimizes the volume of *I/O* induced by the *classical* and *last-in-place* schemes, respectively. In Section 3.3, we discuss the new variant of the *in-place* algorithm: the *max-in-place* scheme. We then show in Section 3.4 that the volume of *I/O* induced by `MinMEM` may be arbitrarily larger than the volume induced by `MinIO`. Section 3.5 illustrates the difference between `MinMEM` and `MinIO` on matrices arising from real-life problems, and shows the interest of the new *in-place* variant we propose.

3.1 Classical assembly scheme

In this section, we consider the multifrontal method with a *classical* assembly scheme. We first show on an illustrative example that minimizing the I/O volume is different from minimizing the storage requirement. This remark motivates our new algorithm that minimizes the I/O volume.

3.1.1 Illustration of the impact of different postorders

Let us consider the elimination tree of Figure 3.1(left). Its root node **(e)** has two children **(c)** and **(d)**. The frontal matrix of **(e)** requires a storage $m_e = 5$. 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 $M_0 = 8$.



Sequence a-b-c-d-e

Storage:	S	$= 12$
I/O:	$V^{I/O}$	$= 8$

$\Rightarrow$ Memory minimized

Sequence d-a-b-c-e

Storage:	S	$= 14$
I/O:	$V^{I/O}$	$= 7$

$\Rightarrow$ I/O minimized

Figure 3.1: 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 (GigaBytes, say) can be kept in core memory, this forces us to write to disk a volume of data equal to 4. Thanks to the postorder and the use of a stack, these 4 units of data are the ones that will be re-accessed 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 units of data 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) disk[1] is $V_e^{I/O}$(a-b-c-d-e)= 8 and the peak of storage was S_e(a-b-c-d-e)= 12.

When the tree is processed in order (d-a-b-c-e) (see Figure 3.2(b)), 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(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 $V_e^{I/O}$(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.

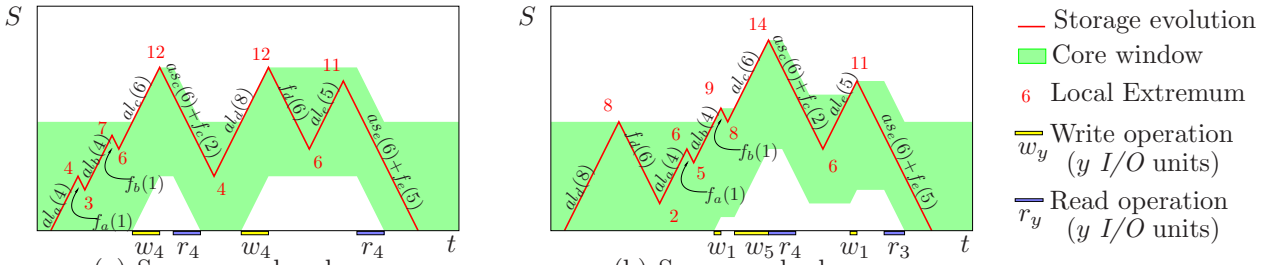
All the process described above is illustrated in Figure 3.2, which represents the evolution of the storage in time for the two postorders (a-b-c-d-e) and (d-a-b-c-e) (figures 3.2(a) and 3.2(b), respectively). The storage increases when memory is allocated for a new frontal matrix of size x ($al_{\mathcal{N}}(x)$); it decreases when contribution blocks of size y are assembled into the frontal matrix of their parent ($as_{\mathcal{N}}(y)$) and when factors of size z are written to disk ($f_{\mathcal{N}}(z)$). 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 read from disk depends on the size of the contribution blocks residing on disk that need to be assembled.

3.1.2 Optimum postorder tree traversal

We have just seen that minimizing the I/O volume is different from minimizing the storage requirement. We now exhibit an algorithm that computes a postorder traversal minimizing the I/O volume.

We have presented in the previous chapter how to compute the volume of I/O. It results from Formula (2.3) that minimizing the volume of I/O is equivalent to minimizing

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



(a) Sequence a-b-c-d-e (b) Sequence d-a-b-c-e

Figure 3.2: Evolution of the storage requirement S when processing the sample tree of Figure 3.1 with the two possible postorders, and subsequent I/O operations. Notations $al_{\mathcal{N}}(x)$, $as_{\mathcal{N}}(y)$ and $f_{\mathcal{N}}(z)$ were introduced in Algorithm 3.1.

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.

Theorem 3.1. (Liu, 86) *Given a set of values $(x_i, y_i)_{i=1,\ldots,n}$, the minimal value of $\max_{i=1,\ldots,n}(x_i + \sum_{j=1}^{i-1} y_j)$ is obtained by sorting the sequence (x_i, y_i) in decreasing order of $x_i - y_i$, that is, $x_1 - y_1 \geq x_2 - y_2 \geq \ldots \geq x_n - y_n$.*

Thanks to Theorem 3.1 (proved in [55]), 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 (2.1) and (2.3) from bottom to top. We will name `MinIO` this algorithm.

Note that, in order to minimize the peak of storage, we can also apply Theorem 3.1 but, in this case, to the term $S_j + \sum_{k=1}^{j-1} cb_k$ of Formula (2.1). This leads to ordering the children in decreasing order of $S_j - cb_j$ rather than $A_j - cb_j$ [45, 55]. Therefore, on the example from Section 3.1.1, 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 thus different from minimizing the *I/O* volume; it may induce a volume of *I/O* larger than needed.

3.2 In-place assembly of the last contribution block

In this variant (used in `MA27` [30] and its successors, for example) of the *classical* multifrontal algorithm, the memory of the frontal matrix of the parent is allowed to overlap with (or to 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 new context, the storage requirements needed to process a given node (Formula (2.1)) becomes:

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

The main difference with Formula (2.1) comes from the *in-place* assembly of the last child (see the boxed superscript in the sum in Formula (3.1)). In the rest of the paper we will use the term *last-in-place* to denote the memory management scheme where an *in-place* assembly scheme is used for the contribution block coming from the last child. Liu has shown[55] that Formula (3.1) could be minimized by ordering children in decreasing order of $\max(S_j, m) - cb_j$.

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 (2.3)) becomes:

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

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:

$$V^{I/O} = \max \left(0, \max_{j=1,n}(\max(A_j, m) + \sum_{k=1}^{j-1} cb_k) - M_0 \right) + \sum_{j=1}^{n} V_j^{I/O} \tag{3.2}$$

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

3.3 In-place assembly of the largest contribution block

In order to further reduce the storage requirement (in comparison to Equation (3.1)), one possibility is to overlap the memory of the parent with the **largest** child contribution block. Compared to Equation (2.1) corresponding to the *classical* scheme, cb_{max} must be subtracted from the term $m + \sum_j cb_j$. Since cb_{max} is a constant that does not depend on the order of children, minimizing the storage (`MinMEM`) is done by using the same tree traversal as for the classical scheme (decreasing order of $S_j - cb_j$). We call this new

scheme *max-in-place* as it constitutes a natural extension to the *in-place* assembly scheme from the previous section. We will see how the memory management can be adapted in Chapter 5.

In an out-of-core context, it is not immediate or easy to generalize MinIO to this *max-in-place* variant. Indeed, it may happen that the largest contribution block, if it does not correspond to the last child, had to be written to disk. In such a case it is better to have an *in-place* assembly of the contribution block of the last child (which is in memory) rather than of the largest contribution. 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 Theoretical comparison of MinMEM and MinIO

Theorem 3.2. *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 3.3 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 the beginning of Section 3.1) for which:

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

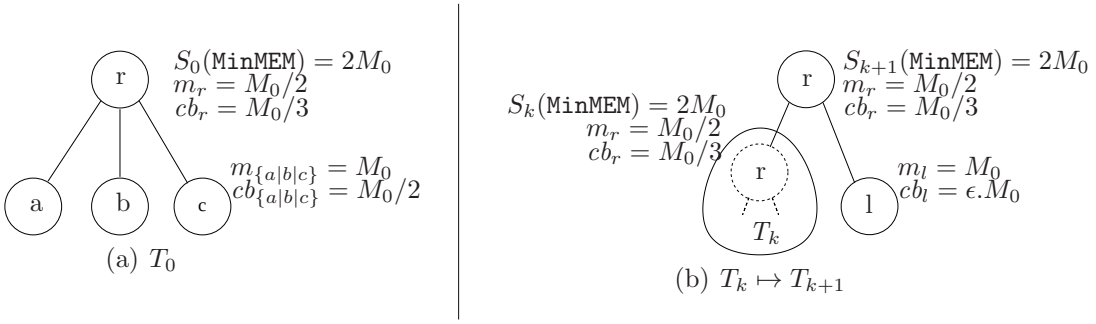


Figure 3.3: Recursive construction of an assembly tree illustrating Theorem 3.2.

We first consider the sample tree T_0 of Figure 3.3(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 $S_0(\texttt{MinMEM}) \stackrel{def}{=} S_r(\texttt{MinMEM}) = 2M_0$, leading to a volume of *I/O* $V_0^{I/O} \stackrel{def}{=} V_r^{I/O} = M_0$. 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 $S(\texttt{MinMEM}) = 2M_0$; 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 3.3(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 3.1, 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 $S_{k+1}(\texttt{MinMEM}) = 2M_0$ (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 $V_{k+1}^{I/O}(\texttt{MinMEM}) = M_0/3 + V_k^{I/O}(\texttt{MinMEM})$ (Formulas (2.3) and (3.2) for the *classical* and *last-in-place*, respectively).

Since $S_k(\texttt{MinIO})$ is greater than or equal to $S_k(\texttt{MinMEM})$, 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 $S_{k+1}(\texttt{MinIO}) = \epsilon M_0 + S_k(\texttt{MinIO})$ and a volume of *I/O* $V_{k+1}^{I/O}(\texttt{MinIO}) = \epsilon M_0 + V_k^{I/O}(\texttt{MinIO})$.

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

We have shown that the *I/O* volume induced by MinMEM, $V^{I/O}$ (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 that minimizes the memory is unique: (l) has to be processed after T_k at any level of the tree.

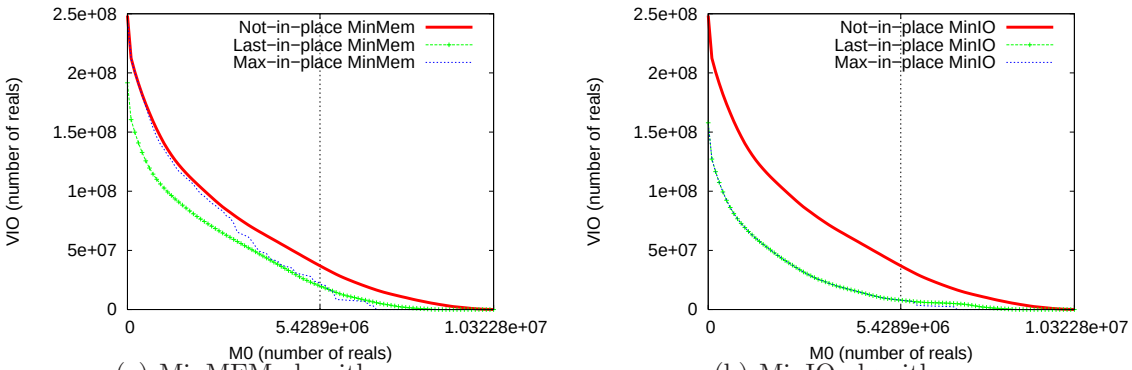
■

3.5 Experimental results

In this section we test the behaviour of the strategies presented in Sections 3.1, 3.2, and 3.3 on 30 matrices, numbered from 1 to 30: AUDIKW_1, BCSSTK, BMWCRA_1, BRGM, CONESHL_MOD, CONV3D_64, GEO3D-20-20-20, GEO3D-50-50-50, GEO3D-80-80-80, GEO3D-20-50-80, GEO3D-25-25-100, GEO3D-120-80-30, GEO3D-200-200-200, GUPTA1, GUPTA2, GUPTA3, MHD1, MSDOOR, NASA1824, NASA2910, NASA4704, SAYLR1, SHIP_003, SPARSINE, THERMAL, TWOTONE, ULTRASOUND3, ULTRASOUND80, WANG3 and XENON2. These matrices are from the Parasol[2], University of Florida[3] or TLSE[4] collections. Matrices GEO3D*, BRGM and CONV3D_64 come from Geosciences Azur, BRGM, and CEA-CESTA (code AQUILON), respectively.

We used several ordering heuristics – AMD [6], AMF [57], METIS [52] and PORD [69], that result in different task dependency graphs (or assembly trees) for a given matrix and impact the computational complexity. The volumes of *I/O* were computed by instrumenting the analysis phase of the MUMPS solver [13]. The matrices have a size from very small up to very large (a few million equations) and can lead to huge factors (and storage requirements). For example, the factors of matrix CONV3D_64 with AMD ordering represent 53 *GB* of data.

As previously mentioned, the *I/O* volume depends on the amount of core memory available. Figure 3.4 illustrates this general behaviour on a sample matrix, TWOTONE ordered with PORD, for the 3 assembly schemes presented above, for both MinMEM and MinIO. For all assembly schemes and algorithms used, we first notice that exploiting all



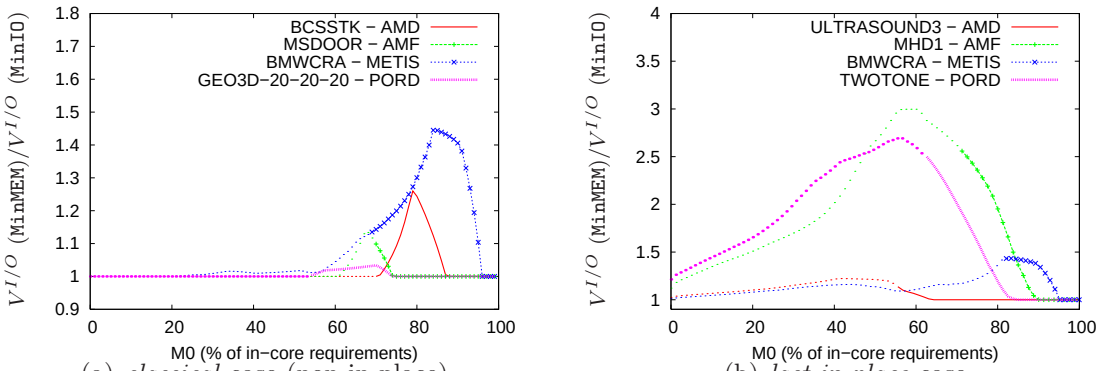
(a) MinMEM algorithms (b) MinIO algorithms

Figure 3.4: I/O volume on matrix TWOTONE with PORD ordering as a function of the core memory available, for the 3 assembly schemes presented above, for both MinMEM and MinIO algorithm. The vertical bar represents the size of the largest frontal matrix.

the available memory is essential to limit the *I/O* volume. Before discussing the results we remind the reader that the *I/O* volumes presented are valid under the hypothesis that

2. http://www.parallab.uib.no/parasol
3. http://www.cise.ufl.edu/research/sparse/matrices/
4. http://www.gridtlse.org

the largest frontal matrix may fit in core. With a core memory lower than this value (*i.e.* the area on the left of the vertical bar in Figure 3.4), the *I/O* volumes presented are actually lower bounds on the effective *I/O* volume: they are computed as if we could process the out-of-core frontal matrices with a read-once write-once scheme. They however remain meaningful because the extra *I/O* cost due to the specific treatment of frontal matrices will be independent of the assembly scheme used. We first notice that the *last-in-place* assembly schemes strongly decrease the amount of *I/O* compared to the *classical* assembly scheme of Section 3.1. In fact, using an *in-place* assembly scheme is very useful in an out-of-core context: on most of our matrices, we observed that it divides the *I/O* volume by more than 2. With the *classical* assembly scheme we observe (on matrix `TWOTONE`) that `MinIO` and `MinMEM` produce the same *I/O* volume (their graphs are identical). Let us come back to Formula (2.3) to explain this behaviour. We have minimized $\max\left(\max_{j=1,n}(A_j + \sum_{k=1}^{j-1} cb_k), m + \sum_{k=1}^{n} cb_k\right)$ by minimizing the first member because the second one is constant; unfortunately on this particular matrix the second term is generally the largest and there is nothing to gain. In other words, the larger the frontal matrices (m in the formula) compared to the other metrics (contribution blocks cb_k and active memory requirements for the subtrees A_j), the lower the probability that reordering the children will impact the *I/O* volume. From our set of matrices, we have extracted four cases (one for each ordering strategy) for which the gains are significant and we report them in Figure 3.5(a). To better illustrate the gains resulting



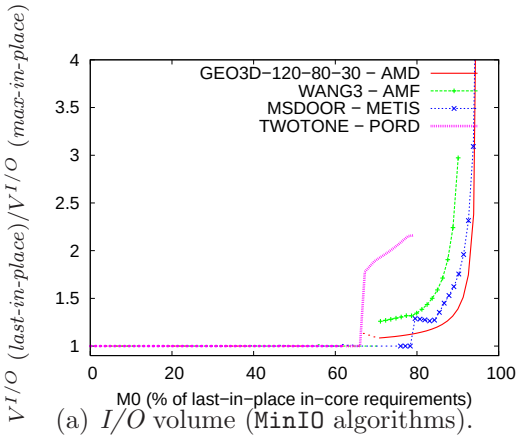
(a) *classical* case (non in-place) (b) *last-in-place* case

Figure 3.5: *I/O* volume obtained with `MinMEM` divided by the one obtained with `MinIO`. For each matrix/ordering, the filled (right) part of the curve matches the area where the amount of core memory is larger than the size of the largest frontal matrix, whereas the dotted (left) part matches the area where this amount is lower. For each matrix, we normalized the memory (x-axis) to the in-core minimum requirement (of the given assembly scheme). Note that the y-scales differ.

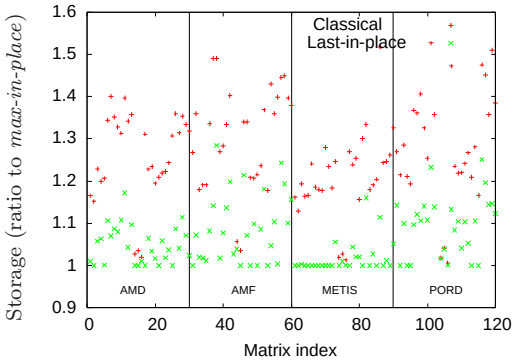
from the `MinIO` algorithm, we analyze the *I/O* ratios as a function of the amount of core memory available (in percentage of the core memory requirements). For instance, the point ($x = 80\%, y = 1.3$) (obtained for both `BCSSTK` and `BMWCRA`) means that `MinMEM` leads to 30% more *I/O* than `MinIO` when 80% of the in-core memory requirement is

provided. Values lower than 1 are not possible because MinIO is optimal.

We now focus on the *in-place* assembly scheme (described in Section 3.2). Rather than showing the graphs obtained for our whole collection of matrices, we again decided to present four cases in Figure 3.5(b) (one for each ordering strategy) for which MinIO was significantly more efficient than MinMEM: the *I/O* volume was for instance divided by more than 2 for a large range of core memory amounts on the MHD1-AMF matrix. An extensive study has shown that the largest profits from MinIO are obtained when matrices are pre-processed with orderings which tend to build irregular assembly trees:AMF, PORD and - to a lesser extent - AMD (see [45] for more information on the impact of ordering on tree topologies). This is because on such trees, there is a higher probability to be sensitive to the order of children. We show in Figure 3.6(a) by how much the MinIO algorithm



(a) *I/O* volume (MinIO algorithms).



(b) Storage requirements (MinMEM algorithms). Thirty test matrices are ordered with four ordering heuristics (AMD, AMF, METIS and PORD), separated by vertical bars.

Figure 3.6: Impact of *max-in-place* assembly scheme.

with a *max-in-place* assembly scheme improved the MinIO *last-in-place* one, again on four matrices of the collection (one for each ordering heuristic) for which we observed large gains. We observe in Figure 3.6(a) that the *last-in-place* and *max-in-place* MinIO schemes induce the same volume of *I/O* when the available core memory decreases: the ratio is equal to 1. This is because, in this case, the MinIO heuristic for the *max-in-place* assembly variant switches to the *last-in-place* scheme (as explained in Section 3.3) and has exactly the same behaviour since the switch happens very early.

Finally, Figure 3.6(b) shows that the peak of storage (critical for the in-core case) is significantly decreased. This allows us to interpret the right-extreme parts of the curves in Figure 3.6(a) which tend to (or are equal to) infinity: the *max-in-place* assembly scheme does *not* induce *I/O* but the *last-in-place* scheme *does*.

Assembly scheme	Algorithm	Objective function: Memory minimization	Objective function: **I/O minimization**
classical	`MinMEM`	• Optimum ([45], adapting[55])	• **Arbitrarily bad in theory** • **Reasonable in most cases**
	MinIO	• Not suited	• **Optimum**
last-in-place	`MinMEM`	• Optimum[55]	• **Arbitrarily bad in theory** • **Bad in practice on some irregular assembly trees**
	MinIO	• Not suited	• **Optimum**
max-in-place	**MinMEM**	• **Optimum**	• Not suited
	MinIO	• **Optimum**	• **Efficient heuristic**

Table 3.1: Summary. Contributions of this chapter are in bold.

3.6 Conclusion

Table 3.1 summarizes the contributions of this chapter. We have recalled the existing memory-minimization algorithms for the *classical* and *last-in-place* assembly schemes. We have shown that these algorithms are not optimal to minimize the *I/O* volume and that they can be arbitrarily bad. Therefore, we have proposed optimal algorithms for the *I/O* volume minimization and have shown that significant gains could be obtained on real problems (especially with the *last-in-place* assembly scheme). We have also presented a new assembly scheme (which consists in extending the largest child contribution into the frontal matrix of the parent) and a corresponding postorder which is optimal to minimize memory. This new assembly scheme leads to a very good heuristic when the objective is to minimize the *I/O* volume.

This work can be particularly important for large-scale problems (millions of equations) in limited-memory environments. It is applicable for shared-memory solvers relying on threaded `BLAS` libraries. In a parallel distributed context, it will help to limit the memory requirements and to decrease the *I/O* volume in the serial (often critical) parts of the computations. Orderings from tree rotations [56] form another important class of equivalent reorderings that might also be interesting to investigate: with tree rotations, an arbitrary node can become the root of the tree, modifying both the shape of the tree and the storage or *I/O* requirements.

In the next chapter, we will use these results to reconsider the problem of the minimization of the *I/O* volume in the context of the flexible allocation scheme, where the parent node is allowed to be allocated even when some children still have to be processed [44].

Chapter 4

Reducing the I/O volume in the multifrontal method: Flexible allocation scheme case

In the previous chapter we have studied the multifrontal method as it is implemented in most multifrontal solvers. Because the frontal matrix is allocated after all the children have been processed, the allocation scheme of that multifrontal method was said to be *terminal* . However, it is possible to improve its memory behaviour by modifying the moment when the frontal matrix is allocated. In the past, this freedom has been exploited to further decrease the storage requirement [44]. The allocation scheme of the frontal matrix is then said to be *flexible*. In this chapter, we briefly restate the results obtained in the previous chapter to take into account this terminology. They are then used as a basis to reduce the *I/O* volume in the flexible context. As we did in the previous chapter, we first focus on the *classical* assembly scheme (sections 4.1 to 4.5) before extending the results to the *in-place* cases (Section 4.6). In those sections, we consider the *I/O* minimization problem from a theoretical point of view, aiming at studying its complexity and possible variations. We then present an algorithm motivated by practical considerations which aims at limiting the *I/O* volume (Section 4.7). We finally discuss the reduction of the *I/O* volume due to our algorithm on real-life matrices (Section 4.8).

4.1 Restating results of Chapter 3

Considering a so-called *family* composed of a parent node, with a frontal matrix of size m, and its set of n children that produce contribution blocks of size $cb_i,\ i = 1, \ldots, n$, we have seen that the storage requirement to process the tree rooted at the parent with a *classical* assembly scheme is:

$$S^{terminal} = \max\left(\max_{j=1,n}(S_j^{terminal} + \sum_{k=1}^{j-1} cb_k), m + \sum_{k=1}^{n} cb_k\right) \tag{4.1}$$

(where $S_j^{terminal}$ is recursively the storage for the subtree rooted at child j) and can be minimized by sorting the children in decreasing order of $S_j^{terminal} - cb_j$. By applying this formula and this ordering at each level of the tree, we obtain the volume of I/O for the complete tree, together with the tree traversal. Starting from (4.1), we have shown in the previous chapter that for a given amount of available memory, M_0, the volume of I/O (=volume written=volume read) associated with the temporary storage of the multifrontal method is

$$V^{terminal} = \max\left(0, \max(\max_{j=1,n}(\min(S_j^{terminal}, M_0) + \sum_{k=1}^{j-1} cb_k), m + \sum_{k=1}^{n} cb_k) - M_0\right) + \sum_{j=1}^{n} V_j^{terminal} \quad (4.2)$$

which is minimized by sorting the children in decreasing order of

$$\min(S_j^{terminal}, M_0) - cb_j$$

at each level of the tree and gives an optimal tree traversal to minimize the I/O volume.

4.2 Flexible parent allocation

With the *terminal allocation* scheme, steps (MF-1), (MF-2) and (MF-3) (as presented in Section 1.1.3) for a parent node are only performed when all children have been processed. However, the main constraint is that the partial factorization (step (MF-3)) at the parent level must be performed after the assembly (step (MF-2)) of all child contribution blocks into the parent. Thus, the allocation of the parent node (step (MF-1)), and the assembly of the contribution blocks of some children can be performed (and the corresponding contribution block freed) without waiting that all children have been processed. This flexibility has been exploited by [44] to further reduce the storage requirement for temporary data. Let us assume that the parent node is allocated after p children have been processed, and that the memory for the p^{th} child overlaps with the memory for the parent. The storage required for a parent in this *flexible* scheme is then given by:

$$S^{flexible} = \max\left(\max_{j=1,\boxed{p}}(S_j^{flexible} + \sum_{k=1}^{j-1} cb_k), m + \sum_{k=1}^{\boxed{p}} cb_k, m + \max_{j=p+1,n} S_j^{flexible}\right) \quad (4.3)$$

When the parent is allocated, all the contribution blocks from its factored children are assembled and discarded. From that point on, each child that is factored sees its contribution block immediately assembled and its memory is released. [44] shows how to choose the point (*split point*) where the parent should be allocated and how to order the children so that the storage requirement $S^{flexible}$ is minimized.

Now, if the amount of storage $S^{flexible}$ is larger than the available core memory M_0, then some disk storage has to be used. In that case, rather than minimizing $S^{flexible}$, it

becomes more relevant to minimize the volume of I/O $V^{flexible}$ obtained with the flexible multifrontal method: this is the objective of the chapter. To limit the volume of I/O, minimizing $S^{flexible}$ can appear like a good heuristic. In [59], the authors have done so, adapting [44] with respect to some additional constraints imposed by their code. However, by computing the volume of I/O formally, we can show the limits of a memory-minimizing approach when aiming at decreasing the I/O volume: similarly to the terminal allocation case, minimizing the volume of I/O in the flexible allocation scheme is different from minimizing the storage requirement.

4.3 Volume of I/O in a flexible multifrontal method

The main difference compared to Formula (4.2) is that with a flexible allocation scheme, a child j processed after the parent allocation ($j > p$) may also generate I/O. Indeed, if this child cannot be processed in-core together with the frontal matrix of the parent, then part of that frontal matrix (or that whole frontal matrix) has to be written to disk in order to make room and process the child with a maximum of available memory. This possible extra-I/O corresponds to underbrace **(a)** of Formula (4.4). 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 rely on a simple property to find which rows of the contribution block, if any, can be assembled into the part of the frontal matrix available in memory (we will discuss this assumption in Section 4.6). 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 (4.4). All in all, and using the storage definition from Formula (4.3), the volume of I/O required to process the subtree rooted at the parent node is given by:

$$
\begin{aligned}
V^{flexible} = \max & \left(0, \max \left(\max_{j=1,\boxed{p}} \left(\min(S_j^{flexible}, M_0) + \sum_{k=1}^{j-1} cb_k \right), m + \sum_{k=1}^{\boxed{p}} cb_k \right) - M_0 \right) \\
& + \underbrace{\sum_{j=p+1}^{n} \left(\max(0, m + \min(S_j^{flexible}, M_0) - M_0) \right)}_{\textbf{(a)}} + \underbrace{\sum_{j=p+1}^{n} (\max(0, m + cb_j - M_0))}_{\textbf{(b)}} \\
& + \sum_{j=1}^{n} V_j^{flexible}
\end{aligned}
\tag{4.4}
$$

Again, a recursion gives the I/O volume for the whole tree.

With the terminal allocation scheme, the I/O volume (on a parent node and its n children) is minimized by sorting the children in an appropriate order (see Chapter 3).

With the flexible scheme, one should *moreover* determine the appropriate *split point*, *i.e.* the best value for p. In other words, the flexible *I/O* volume is minimized when together **(i)** the children processed *before* the parent allocation are correctly separated from the ones processed *after* and **(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 the `GUPTA3` matrix). However, we have shown in the previous chapter that an optimal order among the children processed before the parent allocation is obtained when they are sorted in decreasing order of $\min(S_j^{flexible}, M_0) - cb_j$. 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 **(ii)** is actually immediate when **(i)** is determined. Therefore we only have to determine to which set (before or after the parent allocation) each child belongs to. Solving the initial problem finally consists in reaching Objective 4.1 on the children of a family:

Objective 4.1. *Given an amount of available core memory M_0, our objective is to minimize the* I/O *volume on the contribution blocks by determining the children that should be processed before (and the ones that should be processed after) the parent allocation.*

However this still makes an exponential (2^n) number of possibilities to explore and motivates to further reduce the complexity.

4.4 Reducing the complexity (discrimination process)

To further reduce the complexity, we consider a family and we are interested in finding some children for which we can easily decide whether they should be processed before or after the parent allocation. To do so, we compare the impact on the *I/O* volume of processing a given child j before and after the allocation of the parent node. Because the *I/O* volume on its own subtree (the term $V_j^{flexible}$ in Formula (4.4)) is independent on the way of processing the considered family, we do not need to take this amount into account; we suppose it equal to 0 and we introduce the following definition:

Definition 4.1. *The* contribution *of a child to the* I/O *volume of a family is the additional* I/O *volume of the family due to the child compared to the case when the family does include that child.*

Property 4.1 identifies some children that should be processed before the parent allocation:

Property 4.1. *The children that satisfy $m + S_j^{flexible} \leq M_0$ should be processed* after *the parent allocation.*

Proof. We are considering a child j that can be processed in core memory along with the frontal matrix of its parent node ($m + S_j^{flexible} \leq M_0$). Ordering this child after the

parent allocation does not induce any additional *I/O* (**(a)** and **(b)** are both 0 in (4.4)). On the other hand, its contribution to the *I/O* volume might be positive if it is processed before the parent allocation. Clearly, we should thus process this child after the parent allocation (at least we lose nothing if we process it after the parent allocation). ■

We now aim at finding some children that should be processed before the parent allocation. Lemma 4.1 gives an upper bound on the contribution to the *I/O* volume of a child processed before the parent allocation. This upper bound is then used in properties 4.2 and 4.3 to identify some children that should be processed before the parent allocation.

Lemma 4.1. *The contribution (see Definition 4.1) to the* I/O *volume of a child j processed before the parent allocation is at most equal to cb_j.*

Proof. Computing the exact contribution of a child is not trivial. However an upper bound is easily obtained by assuming that the child is the first one processed: this might be suboptimal and thus constitutes an upper bound compared to the best possible configuration with the constraint to process the child before the parent allocation. We number the initial children from 1 to n and we renumber the additional child 0 ($j = 0$). According to the first term of Formula (4.4), the *I/O* volume on the children processed before the parent allocation V^{before} with this configuration satisfies:

$$
\begin{aligned}
V^{before} & \\
&= \max\left(0, \max\left(\max_{j=\boxed{0},p}\left(\min(S_j^{flexible}, M_0) + \sum_{k=\boxed{0}}^{j-1} cb_k\right), m + \sum_{k=\boxed{0}}^{p} cb_k\right) - M_0\right) \\
&= \max\left(0, \max\left(\max_{j=\boxed{1},p}\left(\min(S_j^{flexible}, M_0) + \sum_{k=\boxed{0}}^{j-1} cb_k\right), m + \sum_{k=\boxed{0}}^{p} cb_k\right) - M_0\right) \\
&= \max\left(0, \max\left(\max_{j=\boxed{1},p}\left(\min(S_j^{flexible}, M_0) + \sum_{k=\boxed{1}}^{j-1} cb_k\right), m + \sum_{k=\boxed{1}}^{p} cb_k\right) \boxed{+\, cb_0} - M_0\right) \\
&\leq \max\left(0, \max\left(\max_{j=\boxed{1},p}\left(\min(S_j^{flexible}, M_0) + \sum_{k=\boxed{1}}^{j-1} cb_k\right), m + \sum_{k=\boxed{1}}^{p} cb_k\right) - M_0\right) \boxed{+\, cb_0}
\end{aligned}
$$

Note that the first equality stands because $\min(S_0^{flexible}, M_0) - M_0$ is bounded by 0. ■

Property 4.2. *The children that satisfy $S_j^{flexible} \geq M_0 - m + cb_j$ should be processed* before *the parent allocation.*

Proof. (Proof by contradiction.) We assume to the contrary that there exists an optimum configuration which contains a minimum number of children after the parent allocation, including a child j that satisfies $S_j^{flexible} \geq M_0 - m + cb_j$. The contribution to the *I/O* volume of this child when processed after the parent allocation is given by underbraces **(a)** and **(b)** in Formula (4.4). Since the term related to underbrace **(b)** is nonnegative, underbrace **(a)** provides a lower bound of its contribution to the *I/O* volume if that child is processed after the parent allocation. That contribution is at least equal to m (if $S_j^{flexible} \geq M_0$) – which is greater than cb_j, or to $S_j^{flexible} - M_0 + m$ (if $S_j^{flexible} \leq M_0$) – which is also greater than cb_j. On the other hand, treating that child before the parent allocation would lead to a maximum additional volume of *I/O* equal to cb_j according

to Lemma 4.1. We can thus move it back to the set of children processed before the parent allocation without increasing the I/O volume, contradicting our assumption that the number of children after the parent allocation is minimal. ■

Property 4.3. *The children that both satisfy* $S_j^{flexible} \geq 2(M_0 - m)$ *and* $m + cb_j > M_0$ *should be processed* before *the parent allocation.*

Proof. (Proof by contradiction.) We assume to the contrary that there exists an optimum configuration which contains a minimum number of children after the parent allocation, including a child j that both satisfies $S_j^{flexible} \geq 2(M_0 - m)$ and $m + cb_j > M_0$. If this child is processed after the parent allocation, its contribution is equal to the sum of the amounts expressed by underbraces **(a)** and **(b)** in Formula (4.4). Noticing that the term related to underbrace **(b)** is now positive (since $m + cb_j > M_0$) the contribution is equal to: $(m + S_j^{flexible} - M_0) + (m + cb_j - M_0)$. We can rewrite this amount as $S_j^{flexible} - (2(M_0 - m)) + cb_j$ which is at least equal to cb_j since we have assumed that $S_j^{flexible} \geq 2(M_0 - m)$. According to Lemma 4.1, the contribution is thus larger than if the child is processed after the parent allocation. We can again move it back to the set of children processed before the parent allocation without increasing the I/O volume, which is a contradiction to the fact that the number of children after the parent allocation should be minimal. ■

All in all, as one can see in Algorithm 4.1, it is straightforward to decide whether a child has to be processed before or after the parent allocation as soon as one of the three above properties applies to it. Indeed, the verification of these conditions is independent of the metrics of the siblings. Definition 4.2 discriminates those children:

Definition 4.2. *A child is said to be* fixed *if one of the properties 4.1, 4.2 or 4.3 applies to it. It is* unfixed *otherwise. We call* discrimination *the process that classifies the children between* fixed *and* unfixed *ones.*

For a given family, if all the children are *fixed*, Objective 4.1 is reached. In all cases, any positive number of *fixed* children represents a partial answer to the question raised by that objective. In this sense, the *discrimination* process constitutes a straightforward first step of an algorithm that aims at reaching the above objective.

This step is actually very important since the decision problem associated with this minimization problem is NP-complete. In other words, given an arbitrary target amount of I/O $V_{I/O}^{target}$, 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 $V_{I/O}^{target}$ (except if $P = NP$). Compared to the exponential exploration needed to solve this NP-complete problem, the cheap step that the *discrimination* process represents is essential if it can discard some children. We present a proof of the NP-completeness of the decision problem (Section 4.5) followed by a heuristic (Section 4.7) based on the *discrimination* process and motivated by a study on real-life matrices.

Input: A family composed of a parent node and n children.
Output: An array `position()` of size n notifying whether a child j $(1 \leq j \leq n)$ should be processed before, after the parent allocation or whether it is still unfixed.

foreach *child* j *from* 1 *to* n **do**
  if $m + S_j^{flexible} \leq M_0$ **then**
    `position(j)` $\leftarrow before$; `% Property 4.1`
  else if $S_j^{flexible} \geq M_0 - m + cb_j$ **then**
    `position(j)` $\leftarrow after$; `% Property 4.2`
  else if $S_j^{flexible} \geq 2(M_0 - m)$ *and* $m + cb_j > M_0$ **then**
    `position(j)` $\leftarrow after$; `% Property 4.3`
  else
    `position(j)` $\leftarrow unfixed$;

Algorithm 4.1: Discrimination process. A child j which satisfies `position(j)` $= before$ or `position(j)` $= after$ is *fixed*.

Note that the study on real matrices presented in sections 4.7 and 4.8 can be read independently of (or before) the theoretical aspects discussed in sections 4.5 and 4.6.

4.5 Minimizing the I/O volume in the flexible multifrontal method is NP-complete

In this section we show that the decision problem related to Objective 4.1 is NP-complete.

4.5.1 Intuition

To have an intuition on the difficulty to solve Objective 4.1, we consider a family and a partition of its n children (not necessarily optimal). Let us imagine that the children that are ordered before the parent allocation are such that each of them has a larger contribution to the I/O volume (in the sense of Definition 4.1) than if it was processed after the parent allocation. This implies that moving anyone of those children after the parent allocation will decrease the I/O volume. However, it can happen that processing all of them after the parent allocation is not optimal. We suppose for instance that we are given a memory M_0, that the parent node of the family has a frontal matrix of size $m = \frac{M_0}{2}$ and that there are 4 children, each of them satisfying $cb_j = \frac{M_0}{5}$ and $S_j^{flexible} = \frac{6M_0}{10}$. We moreover suppose that all the children are processed before the parent allocation in the given initial partition. Because the maximum in the first term of Formula (4.4) is obtained

when the parent is allocated, the I/O volume related to this family is equal to:

$$V^{flexible} = V^{before} = m + \sum_{k=1}^{4} cb_k - M_0 = \frac{3M_0}{10} \tag{4.5}$$

The contribution to the I/O volume of any child j is thus here exactly equal to $cb_j = \frac{M_0}{5}$ whereas it would only be equal to $m + S_j^{flexible} - M_0 = \frac{M_0}{10}$ if it were processed after the parent allocation. It is thus interesting to move anyone of those children after the parent allocation. But moving all of them leads to a total I/O volume equal to $V^{flexible} = \frac{4M_0}{10}$ which is larger than the initial I/O volume expressed in Formula (4.5). The reason is that after we have moved one child, the I/O volume on the children processed before the parent allocation is now equal to $V^{before} = m + \sum_{k=1}^{3} cb_k - M_0 = \frac{M_0}{10}$. Moving a second child thus cannot decrease this amount of more than $\frac{M_0}{10}$ as V^{before} is nonnegative (see Formula (4.4): $V^{before} = \max(0, \ldots)$). If we move a second child, V^{before} becomes equal to 0 and there is no interest to move a third (or a fourth) child after the parent allocation. In fact, moving a third child after the parent allocation would induce an additional I/O volume of $\frac{M_0}{10}$ (the contribution of the child when processed after the parent allocation) and this amount would not be balanced anymore by a decrease of V^{before} (which is already at its minimum, 0).

All in all, this situation can be viewed as follows. We have some items (children initially ordered before the parent allocation) that we can embed into a knapsack (we move them after the parent allocation). Each item has a value (the decrease obtained on the total I/O volume if the corresponding child is moved alone after the parent allocation). But the knapsack has a limited weight (the I/O volume does not decrease anymore after we have moved too many children). The objective is to embed a maximum total value (to decrease as much as possible the total I/O volume) into the knapsack (by moving children after the parent allocation). This problem is exactly the Knapsack Problem which is known to be NP-complete [35]. However, in our case, we have to imagine that we can try to embed one more item that spills out from the knapsack. Indeed, if we come back to the above example, when moving a second child after the parent allocation, we can still decrease V^{before} of $\frac{M_0}{10}$ but not of $\frac{M_0}{5}$ anymore. The situation is like if the knapsack was almost full (V^{before} is not equal to 0 yet but it would be the case if we moved one more child); therefore, we would not fully benefit from the last item we embed. As an image, we can think that the part of it that spills out will be wasted (not useful), but still has to be carried and thus represents an inconvenient.

We formalize the problem of a knapsack that can spill out and show that it is NP-complete in Section 4.5.2. This preliminary result will be used in Section 4.5.3 to perform a natural reduction of the decision problem related to Objective 4.1 from it.

4.5.2 Preliminary result: the problem of a knapsack that can spill out is NP-complete

We consider the optimization problem of a knapsack that can spill out. It derives its name from the following maximization problem of the best choice of essentials that can fit into one bag to be carried on a trip. Given a set of items, each with a cost and a value, determine the number of each item to include in a collection so that the total cost is less than a given limit and the total value is as large as possible. Contrary to the classical knapsack problem, the cost is considered here as an inconvenient (an additional weight for instance) that we have to subtract from its value: we might want to carry less items if their cumulated interests measured by their values is not large enough compared to their cumulated inconvenient measured by their costs. Moreover, we can embed one more item that only partially fits in the bag. For this item we will only benefit from the part that fits in the bag (we will only count the corresponding part of its value) but we will have to carry the entire item (we will pay its entire cost). We assume that the value of each item is larger than its cost (otherwise such an object would never be subject to being included). The situation is formalized in Problem 4.1:

Problem 4.1 (Knapsack-Spill-Opt)**.** *We have n kinds of items $(1,\dots,n)$. Each item j has a value p_j and a cost c_j such that: $0 \leq c_j \leq p_j$. We moreover assume that the capacity V of the bag is limited: $0 \leq V \leq \sum_{i=1}^{n} p_i - \max_{i=1}^{n} p_i$. We aim at maximizing the algebraic benefit ΔB, whose expression is $\Delta B = \min(\sum_{j\in\mathcal{S}} p_j, V) - \sum_{j\in\mathcal{S}} c_j$, subject to $\mathcal{S} \subset \{1;\dots;n\}$.*

The decision problem form of the knapsack that can spill out is the question: *Can an algebraic benefit (value minus cost) of at least K be achieved?*

Problem 4.2 (Knapsack-Spill-Dec)**.** *We have n items $(1,\dots,n)$. Each item j has a value p_j and a cost c_j such that: $0 \leq c_j \leq p_j$. We moreover assume that the capacity V of the bag is limited: $0 \leq V \leq \sum_{i=1}^{n} p_i - \max_{i=1}^{n} p_i$. Can we achieve an algebraic benefit K? Or, formally, is the following assertion true: $(\exists \mathcal{S} \subset \{1;\dots;n\}) \left(\min(\sum_{j\in\mathcal{S}} p_j, V) - \sum_{j\in\mathcal{S}} c_j \geq K\right)$?*

Lemma 4.2. *Problem Knapsack-Spill-Dec is NP-complete.*

Proof. To prove this result, we consider the Partition problem – which is known to be NP-complete [35] – formulated as follows:

Problem 4.3 (Partition)**.** *Given n positive integer numbers $(x_1,\dots,x_n)$ of half-sum $X = \frac{\sum_{j=1}^{n} x_j}{2}$, is there a subset $\mathcal{S}$ such that $\sum_{j\in\mathcal{S}} x_j = X$?*

The idea of the proof is to show that Partition can be reduced in polynomial time to Knapsack-Spill-Dec. The details are provided in Appendix C (Section C.1). ■

4.5.3 Proof of the NP-completeness

We now show that the decision problem related to Objective 4.1 is NP-complete.

Problem 4.4 (Flex-MinIO-Dec)**.** *We consider a family composed of a parent node and n children numbered from 1 to n. We are given a core memory of size M_0. The parent node has a frontal matrix of size m that can fit in core memory: $m \leq M_0$. The size of the contribution block of each child j is lower than the size of the frontal matrix of its parent ($0 \leq cb_j \leq m$) and than the storage requirement of the subtree rooted at j ($cb_j \leq S_j^{flexible}$). Does there exist a subset $\mathcal{S}$ of the children such that ordering those children after the parent allocation and the other ones before induces an* I/O *volume $V^{flexible}$ (as expressed in Formula (4.4)) lower than or equal to a given value $V_{I/O}^{target}$.*

Theorem 4.1. *Problem Flex-MinIO-Dec is NP-complete.*

Proof. First, Flex-MinIO-Dec belongs to NP. If we are given a subset $\mathcal{S}$ of the children such that ordering those children after the parent allocation and the other ones before induces an *I/O* volume lower than or equal to $V_{I/O}^{target}$, we can check in polynomial time in the size of the instance that we can arrange the children such that they indeed induce an *I/O* volume less than or equal to $V_{I/O}^{target}$. To do so, we order the p children processed before the parent allocation according to their decreasing order of $\min(S_j^{flexible}, M_0) - cb_j$; we evaluate the total *I/O* volume and compare it to $V_{I/O}^{target}$. This verification requires a maximum number of operations proportional to $n \log(n)$.

To prove the NP-completeness of Flex-MinIO-Dec, we show that Knapsack-Spill-Dec can be polynomially reduced to Flex-MinIO-Dec. We consider an arbitrary instance I_1 of Knapsack-Spill-Dec composed of a bag of capacity V; n items numbered from 1 to n of respective values and costs equal to p_i and c_i, $1 \leq i \leq n$; an algebraic benefit K to achieve. We build an instance I_2 of Flex-MinIO-Dec as follows. We consider a core memory $M_0 = 2(\sum_{i=1}^{n} p_i - V)$ and a size for the frontal matrix of the parent equal to $m = \frac{M_0}{2}$. We define n children with characteristics $cb_i = p_i, S_i^{flexible} = c_i + m, 1 \leq i \leq n$. We define a target *I/O* volume $V_{I/O}^{target} = m + \sum_{i=1}^{n} cb_i - M_0 - K$. The construction of I_2 is polynomial (and even linear) in the size of I_1. I_2 is effectively an instance of Flex-MinIO-Dec since $m \leq M_0$ and $0 \leq cb_i \leq m, cb_i \leq S_i^{flexible}, 1 \leq i \leq n$. Indeed, the first inequality stands because $m = \frac{M_0}{2}$. To show that the two other inequalities are valid, we only need to show that $cb_i \leq m, 1 \leq i \leq n$ since $S_i^{flexible} \geq m$. This result is obtained by definition of Knapsack-Spill-Dec which states that $p_j \leq \sum_{i=1}^{n} p_i - V, 1 \leq j \leq n$ and which exactly means here that $cb_i \leq m, 1 \leq j \leq n$.

We now consider a child j ($1 \leq j \leq n$) and show that it satisfies the following supplementary properties:

1. $cb_j \leq M_0 - m$;
2. $S_j^{flexible} \geq M_0 - m$;
3. $S_j^{flexible} \leq M_0 - m + cb_j$;

4. $S_j^{flexible} \leq M_0$;

5. $S_j^{flexible} \leq m + cb_j$.

Indeed, by construction of I_2, Inequality 1 is equivalent to $p_j \leq \sum_{i=1}^{n} p_i - V$, Inequality 2 is equivalent to $c_i \geq 0$ and inequalities 3 and 5 are equivalent to $c_i \leq p_i$. They are thus true by definition of Knapsack-Spill-Dec. Inequality 3 implies Inequality 4 since $cb_j \leq m$.

These properties simplify the expression of the I/O volume and actually reflect the intuition presented in Section 4.5.1 as we now explain. Inequality 1 means that if a child is processed after the parent allocation, it will not induce I/O to assemble its contribution block in the parent node (underbrace **(b)** is zero in Formula (4.4)). Inequality 5 implies that the maximum in expression $\max\left(\max_{j=1,p}\left(\min(S_j^{flexible}, M_0) + \sum_{k=1}^{j-1} cb_k\right), m + \sum_{k=1}^{p} cb_k\right)$ is obtained with the second term $m + \sum_{k=1}^{p} cb_k$. Inequalities 2 and 4 imply that underbrace **(a)** in Formula (4.4) is simplified to $\sum_{j=p+1}^{n}\left(m + S_j^{flexible} - M_0\right)$. All in all, if we note $\mathcal{S}$ the subset of children that are processed after the parent allocation, the I/O volume on the family is equal to:

$$V^{flexible}(\mathcal{S}) = \max\left(0, m + \sum_{k\notin\mathcal{S}} cb_k - M_0\right) + \sum_{j\in\mathcal{S}}\left(m + S_j^{flexible} - M_0\right) \tag{4.6}$$

The contribution to the I/O volume of a child j processed before the parent allocation is thus equal to cb_j while $m + \sum_{k\notin\mathcal{S}} cb_k - M_0 \geq cb_j$. Its contribution if processed after the parent allocation is equal to $m + S_j^{flexible} - M_0$. We assume that initially all the children are processed before the parent allocation. While $m + \sum_{k\notin\mathcal{S}} cb_k - M_0 \geq cb_j$, the algebraic benefit to move this child after the parent allocation is thus equal to $cb_j + M_0 - m - S_j^{flexible}$. According to Inequality 3, this amount is positive, which means that moving this child after the parent allocation will decrease the I/O volume. The point is that we are not sure that we should move all the children since the total value we can save is bounded by the initial I/O volume where all the children are processed before the parent allocation and which is equal to $m + \sum_{k=1}^{n} cb_k - M_0$. Therefore we might have to make a choice and this represents the difficulty of the problem. Formally, we note $\Delta V^{flexible}(\mathcal{S})$ the I/O volume that we save by processing the children of $\mathcal{S}$ after the parent allocation compared to an initial configuration where all the children are processed before the parent allocation: $\Delta V^{flexible}(\mathcal{S}) = m + \sum_{k=1}^{n} cb_k - M_0 - V^{flexible}(\mathcal{S})$. This amount can be viewed as an algebraic benefit of moving the children of $\mathcal{S}$ after the parent allocation and can be rewritten as: $\Delta V^{flexible}(\mathcal{S}) = \min\left(\sum_{j\in\mathcal{S}} cb_j, m + \sum_{k=1}^{n} cb_k - M_0\right) - \sum_{j\in\mathcal{S}}(m + S_j^{flexible} - M_0)$.

Therefore, for a given subset $\mathcal{S}$, the assertion $V^{flexible}(\mathcal{S}) \leq V_{I/O}^{target}$ is equivalent to $\Delta V^{flexible}(\mathcal{S}) \leq m + \sum_{k=1}^{n} cb_k - M_0 - V_{I/O}^{target}$, thus to $\Delta V^{flexible}(\mathcal{S}) \leq K$, and finally to $\min\left(\sum_{j\in\mathcal{S}} p_j, V\right) - \sum_{j\in\mathcal{S}}(c_j) \leq K$. Thus, any subset $\mathcal{S}$ is a solution to I_1 if and only if it is a solution to I_2, which implies that I_1 has a solution if and only if I_2 has a solution. ■

4.5.4 Another difficulty which also makes the problem NP-complete

To prove that Problem Flex-MinIO-Dec is NP-complete in Section 4.5.3, we have built families for which it is interesting to process any single child after the parent allocation but not all of them and this implied a difficult choice. The point was that after a certain number of children were moved after the parent allocation, the children remaining before the parent allocation could be processed altogether in core. They did not contribute to the *I/O* volume anymore and there was thus no point to move them after the parent allocation. Hence, the difficulty was related to the fact that the treatment of the children ordered before the parent allocation changed from an out-of-core to an in-core management depending on the children that were moved.

In this section, we show that this is not the only difficulty. This result is important since the understanding of the difficulties conditions the development of heuristics more than the knowledge itself that the problem is NP-complete. To do so, we propose another proof to Theorem 4.1 in which the optimum configurations of the considered instances of Problem Flex-MinIO-Dec are known to require an out-of-core process. We exhibit that the versatility of the position at which the peak of storage is reached among the children processed before the parent allocation also represents a difficulty (that adds to the previous one). Another interest of this second proof is that it can be applied as it is both to the *classical* and *in-place* cases whereas the previous one is not immediately adaptable to the *in-place* case. Again, we reduce Knapsack-Spill-Dec to Flex-MinIO-Dec. The proof is based on the following lemma:

Lemma 4.3. *We are given a family processed in an optimum configuration (that minimizes the* I/O *volume) and we consider the evolution of the storage requirement before the parent is allocated. We assume that children satisfying Property 4.1 have been moved to be processed after the allocation of the parent. The peak related to this storage is obtained either on the last child p or on a child j which satisfies $S_j^{flexible} > cb_j + M_0 - m$.*

Proof. (Proof by contradiction.) We assume to the contrary that the peak of storage is reached on a child j_0 such that $j_0 < p$ and $S_{j_0}^{flexible} \leq cb_{j_0} + M_0 - m$. According to Formula (4.3), the peak is equal to $\max_{j=1,p}\left(S_j^{flexible} + \sum_{k=1}^{j-1} cb_k\right)$ and thus to $S_{j_0}^{flexible} + \sum_{k=1}^{j_0-1} cb_k$ because of our assumption. We now consider child j_0+1 which is also processed before the parent allocation since $j_0+1 \leq p$. Property 4.1 does not apply to it (otherwise it would be processed after the parent allocation in our optimum configuration); hence it satisfies: $S_{j_0+1}^{flexible} > M_0 - m$. Because moreover we have $M_0 - m \geq S_{j_0}^{flexible} - cb_{j_0}$ by assumption, we obtain by transitivity that $S_{j_0+1}^{flexible} > S_{j_0}^{flexible} - cb_{j_0}$ and thus that $S_{j_0+1}^{flexible} + \sum_{k=1}^{j_0} cb_k > S_{j_0}^{flexible} + \sum_{k=1}^{j_0-1} cb_k$, contradicting our assumption. ■

Lemma 4.3 provides an intuition of the difficulty related to the versatility of the peak of storage. Let us imagine that we have $n+1$ children ($1 \leq j \leq n+1$) that do *not* satisfy $S_j^{flexible} > cb_j + M_0 - m$ and that they are ordered after a child (which we number 0) that *does* satisfy this assertion. We moreover imagine that we know that children 0 and

$n+1$ should not be processed after the parent allocation in an optimum configuration and we wonder which children among the n other ones should be processed after the parent allocation. According to Lemma 4.3, the peak of storage can only be obtained on child 0 or on child $n+1$. We suppose that the peak is initially obtained on child $n+1$. As long as the peak remains obtained on that child, it may be interesting to move some children after the parent allocation since the peak of storage may still decrease. But as soon as the peak of storage is obtained on child 0, there is no point to move any additional child after the parent allocation. We thus have had to make a choice between these children; and this choice might represent a difficulty. We provide a new proof of the polynomial reduction of Knapsack-Spill-Dec to Flex-MinIO-Dec which formalizes this intuition.

New proof of Theorem 4.1. We consider an arbitrary instance I_1 of Knapsack-Spill-Dec composed of a bag of capacity V; n items of respective values and costs equal to p_i and c_i, $1 \leq i \leq n$; an algebraic benefit K to achieve. Even if it implies to renumber the items, we suppose that they are ordered according to their increasing value of $p_i - c_i$. We build an instance I_2 of Flex-MinIO-Dec as follows. We consider that the frontal matrix of the parent has a size $m = \sum_{i=1}^{n} p_i - V + \max_{i=1}^{n} c_i$. We define the amount of core memory $M_0 = 2m + p_n - c_n$. We consider $n+2$ children numbered from 0 to $n+1$ of respective contribution block of size cb_i and respective storage requirement of size $S_i^{flexible}$. We define $cb_i = p_i, S_i^{flexible} = c_i + M_0 - m, 1 \leq i \leq n$. We define $S_0^{flexible} = M_0$ and $cb_0 = \min_{i=1}^{n} p_i$; $S_{n+1}^{flexible} = \max_{i=1}^{n} c_i + M_0 - m$ and $cb_{n+1} = S_{n+1}^{flexible} - S_n^{flexible} + cb_n$. We consider a target I/O volume $V_{I/O}^{target} = S_{n+1}^{flexible} + \sum_{k=0}^{n} cb_k - M_0 - K$.

The construction of I_2 is polynomial in the size of I_1: its complexity is bounded by the renumbering step of the children which requires a maximum number of operations proportional to $n \log(n)$. We first prove that I_2 is effectively an instance of Flex-MinIO-Dec. The assertion $m \leq M_0$ stands because $p_n \geq c_n$ and $cb_i \leq m$ $(1 \leq i \leq n)$ comes from the two following inequalities: $p_j \leq \sum_{i=1}^{n} p_i - V$ and $\max_{i=1}^{n} c_i \geq 0$. Child 0 clearly satisfies the constraints of the multifrontal method. For child $n+1$, the assertion $0 \leq cb_{n+1} \leq m$ is not immediate and we show it later.

Children of index j such that $1 \leq j \leq n+1$ satisfy the five additional properties mentioned in the proof of Section 4.5.3 except that the last one is reversed and we note it 5':

5'. $S_j^{flexible} \geq m + cb_j$.

Indeed, Let j be a child such that $1 \leq j \leq n$. Inequality 1 is equivalent to: $p_j \leq \sum_{i=1}^{n} p_i - V + \max_{i=1}^{n} c_i + p_n - c_n$. Inequalities 2, 3 and 4 are established as in the proof of Section 4.5.3. Inequality 5' is equivalent to $p_j - c_j \leq p_n - c_n$ which is true according to the renumbering of the children. We now consider child $n+1$ and prove that the remaining assertion $0 \leq cb_{n+1} \leq m$ stands. First, we have $cb_{n+1} \geq 0$ since child n satisfies Inequality 3 and because $\max_{i=1}^{n} c_i \geq 0$. Second, we have $cb_{n+1} \leq p_n + \max_{i=1}^{n} c_i \leq m$ respectively because child n satisfies Inequality 2 and then because $\max_{i=1}^{n} p_i \leq \sum_{i=1}^{n} p_i - V$. We now show that the five inequalities also apply to child $n+1$. Inequality 1 stands as we have: $cb_{n+1} \leq m \leq M_0 - m$. Inequality 2 is immediate.

Inequalities 3 and 5' for child $n+1$ are respectively equivalent to inequalities 3 and 5' for child n by definition of cb_{n+1}. Inequality 4 comes from Inequality 3.

Again, these properties simplify the expression of the I/O volume. Inequality 5' (which is reversed compared to Section 4.5.3) implies that the maximum in expression $\max\left(\max_{j=0,p}\left(\min(S_j^{flexible}, M_0) + \sum_{k=0}^{j-1} cb_k\right), m + \sum_{k=0}^{p} cb_k\right)$ is now obtained with the first term of the outer maximum. We assume that children 0 and $n+1$ are not subject to be processed after the parent allocation (we justify this assumption later). According to Lemma 4.3, Inequality 3 moreover implies that this maximum is obtained on the last child processed before the parent allocation, $n+1$, or on child 0 (which does not satisfy Inequality 3). If we note $\mathcal{S}$ the subset of the children $\{1, \dots n\}$ that are processed after the parent allocation, the I/O volume on the family is thus equal to:

$$V^{flexible}(\mathcal{S}) = \max\left(0, S_{n+1}^{flexible} + \sum_{k\in\{0,\dots n\}\setminus\mathcal{S}} cb_k - M_0\right) + \sum_{j\in\mathcal{S}}\left(m + S_j^{flexible} - M_0\right) \quad (4.7)$$

We note $\Delta V^{flexible}(\mathcal{S})$ the I/O volume that we save by processing the children of $\mathcal{S}$ after the parent allocation compared to an initial configuration where all the children are processed before the parent allocation. This initial configuration would induce an I/O volume equal to $V^{flexible}(\emptyset) = S_{n+1}^{flexible} + \sum_{k=0}^{n} cb_k - M_0$. Therefore, we have: $\Delta V^{flexible}(\mathcal{S}) = \min\left(\sum_{j\in\mathcal{S}} cb_j, S_{n+1}^{flexible} + \sum_{k=0}^{n} cb_k - M_0\right) - \sum_{j\in\mathcal{S}}(m + S_j^{flexible} - M_0)$. For a given subset $\mathcal{S}$, the assertion $V^{flexible}(\mathcal{S}) \leq V_{I/O}^{target}$ is equivalent to $\Delta V^{flexible}(\mathcal{S}) \leq S_{n+1}^{flexible} + \sum_{k=0}^{n} cb_k - M_0 - V_{I/O}^{target}$, thus to $\Delta V^{flexible}(\mathcal{S}) \leq K$, and finally to $\min\left(\sum_{j\in\mathcal{S}} p_j, V\right) - \sum_{j\in\mathcal{S}}(c_j) \leq K$. Thus, any subset $\mathcal{S}$ is a solution to I_1 if and only if it is a solution to I_2, which implies that I_1 has a solution if and only if I_2 has a solution.

We end up the proof with the justification of the fact that children 0 and $n+1$ are not subject to be processed after the parent allocation. It is immediate for child 0 since Property 4.2 applies to it. We assume that child $n+1$ is processed after the parent allocation in an optimum configuration and we exhibit another configuration in which child $n+1$ is processed before the parent allocation without increasing the I/O volume. Indeed, if p is the last child processed before the parent allocation (thus $p \neq n+1$), we exchange p and $n+1$ (p is moved after the parent allocation and $n+1$ before). We can assume that $p > 0$ (otherwise it is immediate to verify that we can move back any child towards the set of children processed before the parent allocation without increasing the I/O volume); hence: $S_{n+1}^{flexible} \geq S_p^{flexible}$. This exchange leads to an increase of the storage required to process the children before the parent allocation of $S_{n+1}^{flexible} - S_p^{flexible}$ and thus to a *maximum* increase of $S_{n+1}^{flexible} - S_p^{flexible}$ on the I/O volume related to those children. On the other hand, the exchange induces a decrease on the I/O volume of the children that are processed after of exactly $S_{n+1}^{flexible} - S_p^{flexible}$. The total volume is thus lower than or equal to the original volume before exchange. ■

4.6 In-place assembly schemes

In this section, we show that the results obtained in the previous sections apply to the *in-place* assembly schemes.

4.6.1 In-place assembly before the parent allocation

As in the previous chapter, the expression of the storage requirement $S^{flexible}$ and of the induced volume of I/O $V^{flexible}$ are modified. Again, the difference comes from the child that is assembled in-place. In the *last-in-place* case, the term $\sum_{k=1}^{\boxed{\text{p}}} cb_k$ in formulas (4.3) and (4.4) becomes $\sum_{k=1}^{\boxed{\text{p-1}}} cb_k$. In the *max-in-place* case, this term becomes $\sum_{k=1}^{p} cb_k - \max_{k=1}^{p} cb_k$. We provide for instance the expression of $S^{flexible}$ and $V^{flexible}$ for the *last-in-place* assembly scheme:

$$S^{flexible} = \max \left(\max_{j=1,p}(S_j^{flexible} + \sum_{k=1}^{j-1} cb_k), m + \sum_{k=1}^{\boxed{\text{p-1}}} cb_k, m + \max_{j=p+1,n} S_j^{flexible} \right) \tag{4.8}$$

$$\begin{aligned} V^{flexible} = \max \Bigg(0, \max \Bigg(\max_{j=1,p} \Bigg(\min(S_j^{flexible}, M_0) + \sum_{k=1}^{j-1} cb_k \Bigg), m + \sum_{k=1}^{\boxed{\text{p-1}}} cb_k \Bigg) - M_0 \Bigg) \\ + \sum_{j=p+1}^{n} \Big(\max(0, m + \min(S_j^{flexible}, M_0) - M_0) \Big) + \sum_{j=p+1}^{n} (\max(0, m + cb_j - M_0)) \\ + \sum_{j=1}^{p} V_j^{flexible} \end{aligned} \tag{4.9}$$

For the two *in-place* assembly schemes, properties 4.1, 4.2 and 4.3 remain valid and they can be proved exactly as in Section 4.4. Nevertheless, we provide an intuition for that. Property 4.1 is valid since a child that can be processed in core memory along with the frontal matrix of its parent does not induce I/O when processed after the parent allocation; this is true for all the assembly schemes. Properties 4.2 and 4.3 provide sufficient conditions to identify some children that should be processed before the parent allocation. These conditions allow us to compare the contribution to the I/O volume depending on whether those children are processed before or after the parent allocation. With an *in-place* assembly scheme, the contribution to the I/O volume of a child processed before the parent allocation is decreased or remains equal compared to the *classical* assembly scheme. On the other hand, it does not change if that child is processed after the parent

allocation. Therefore, if these two properties are valid with the *classical* assembly scheme, they are "even more true" – if we may say so – with an *in-place* assembly schemes.

Theorem 4.1 remains valid for the two *in-place* assembly schemes. This means that Problem Flex-MinIO-Dec is NP-complete whichever assembly scheme is used. The proof presented in Section 4.5.3 is not immediate to extend to the *in-place* assembly schemes. In that latter proof, we could simplify the expression of the *I/O* volume thanks to the fact that Inequality 5 ensured that the peak of storage before the parent allocation was lower than (or equal to) the storage required at the moment of the allocation. However Inequality 5 is not sufficient to ensure this property with an *in-place* assembly scheme. Intuitively, the probability that the peak of storage is reached at the moment of the parent allocation is lower with an *in-place* scheme since the purpose of such a scheme is precisely to limit this amount. On the contrary, in the proof presented in Section 4.5.4, the storage requirement at the moment of the allocation is lower than the peak of storage before allocation. This time, if it is true with a *classical* assembly scheme, intuitively, it is also true with an *in-place* assembly scheme. Formally, Inequality 5' is sufficient to ensure this property independently of the assembly scheme used and the proof can be applied as it was presented in Section 4.5.4. Nevertheless, to be accurate, we moreover need to notice that the children processed before the parent allocation follow the same order as in the *classical* assembly scheme. It is obvious with a *max-in-place* assembly scheme. For the *last-in-place* assembly scheme, this is due to the fact that $\max(S_j^{flexible}, m) = S_j^{flexible}$ stands (which is immediate with the definition of $S_j^{flexible}$ in Instance I_2). Therefore, ordering the children that are processed before the parent allocation according to their decreasing value of $S_i^{flexible}$-cb_i (which is the case since the items of I_1 are ordered in increasing value of $p_i - c_i$) still leads to an optimum configuration.

4.6.2 In-place assembly after the parent allocation

An *in-place* assembly scheme consists in assembling *in-place* a contribution block (the last one produced or the largest one) into the frontal matrix of the parent at the moment of its allocation. In a flexible context, once a child j ordered after the parent allocation ($j > p$) has been processed, its contribution block is in memory ready to be assembled. However, we have assumed in Section 4.3 that we had to fully re-load the frontal matrix of the parent into memory before performing the assembly of the child. This operation generated extra-*I/O* on the contribution blocks which was counted in underbrace **(b)** of Formula (4.4). The reason for this assumption was that, in general, we cannot easily find which rows of the contribution block, if any, can be assembled into the part of the frontal matrix available in memory. However, in static codes where the sizes and the indices of the variables of the frontal matrices are known in advance, we can avoid the *I/O* volume corresponding to underbrace **(b)**. In the worst case, we can for example read the rows of the parent one-by-one. When reading row i of the parent, if the next row of the contribution block has to be assembled into row i, we assemble it and free it; otherwise, we read row $i+1$, $i+2$, ... until we can assemble and free the current row of the contribution block. The key properties for this to be possible are that:

– indices of the parent and of the contribution blocks are sorted in a compatible order;
– a frontal matrix can fit in core;
– a row of a contribution block is smaller than a row of a frontal matrix.

So, intuitively, even if the NCB rows of the contribution block must be assembled in the last NCB rows of the frontal matrix, we have enough memory to read rows of the parent without overwriting unassembled rows of the contribution block. (At worse, we may need a workspace corresponding to one row of the front.)

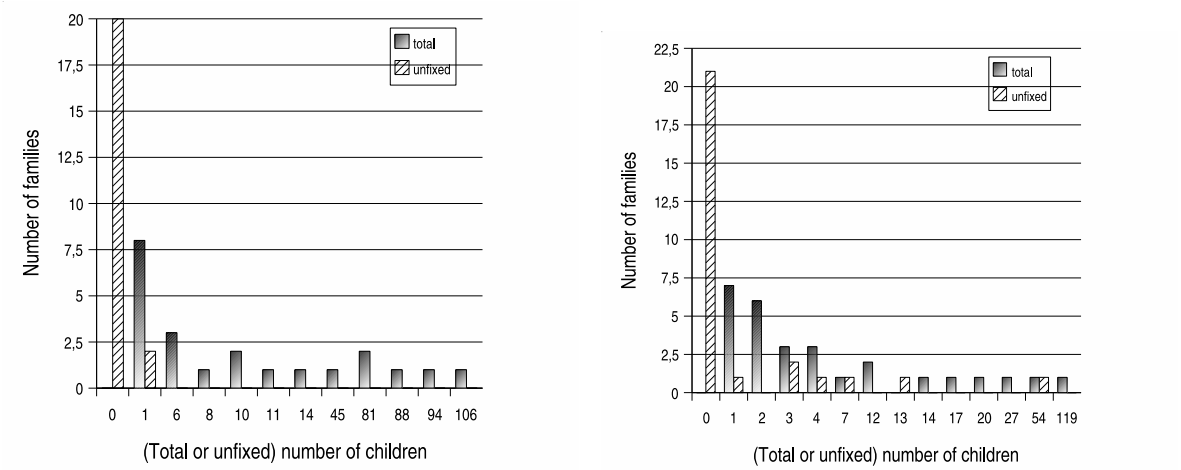
We name this mechanism *in-place-realloc* scheme. It can be combined with any of the *classical*, *last-in-place* and *max-in-place* assembly schemes. Without underbrace **(b)**, we let the reader check that Properties 4.1 and 4.2 still stand as they are. However, Property 4.3 does not apply anymore. Intuitively, we are more likely to process children after the parent allocation with the use of an *in-place-realloc* method. Finally, Theorem 4.1 applies to the *in-place-realloc* schemes. Indeed, in the proofs of both sections 4.5.3 and 4.5.4, the term related to underbrace **(b)** is already equal to zero as ensured by Inequality 1.

4.7 A Heuristic based on the discrimination process

We now study the I/O volume minimization from a pragmatic point of view. We focus on the *in-place* allocation since it is likely to reduce the I/O volume over the *classical* approach. More specifically, we study a *last-in-place* allocation scheme without *in-place-realloc* (introduced in Section 4.6.2) since *in-place-realloc* is not general enough (it does not easily apply to codes that handle a dynamic structure).

We have shown in Section 4.5 that the minimization of the I/O volume on a family requires an algorithm whose complexity may be exponential with the number of children (except if $P = NP$). In this context, the straightforward *discrimination* process presented in Section 4.4 is thus essential if it can discard enough children from the exploration required to reach Objective 4.1. To measure the relevance of this process, we have applied it to the matrices of Table 1.2 for different possible values of available memory M_0. We have noticed that, in practice, the number of *fixed* children is often large and that, for many matrices, most families have no (or almost no) *unfixed* children. We illustrate this typical behaviour with matrix `TWOTONE` in Figure 4.1(a). This matrix has families which contain up to 108 children. Performing an exhaustive exploration involves 2^{108} possible combinations for one such single family. However, after the *discrimination* process, *fixed* children are discarded and no family has more than one *unfixed* children. Therefore, for each family, at most only 2 (2^1) combinations actually have to be tested. In this case, representative of most of our experimental results, the *discrimination* process combined with a single test is thus sufficient to reach Objective 4.1.

Nonetheless, some matrices may have a few families with a large number of *unfixed* children. We illustrate this behaviour with a case that leads to families with many *unfixed* children, `GUPTA3`, as shown in Figure 4.1(b). Indeed, one family keeps having 54 *unfixed* children. For that family, it remains 2^{54} possible partitions. Although this number is far lower than the initial number of 2^{119} possibilities, an exhaustive exploration is not



(a) `TWOTONE` matrix - `PORD` ordering M_0=7572632 (b) `GUPTA3` matrix - `METIS` ordering M_0=684686

Figure 4.1: Distribution of the families as a function of their *total* and *unfixed* number of children. After the *discrimination* process (see Algorithm 4.1) has been applied, most families appear to have few (or no) *unfixed* children.

conceivable in practice and justifies the use of an approximation algorithm that will explore a subset of these combinations. However, among the 28 families inducing I/O, 21 families have no *unfixed* children and only 3 families have strictly more than 4 *unfixed* children (respectively 7, 13 and 54). If we consider that it is acceptable to explore up to 2^4 combinations for each family (which seems reasonable to us), this means that we can find the optimum combination for 25 of the 28 families that compose this matrix whereas the approximation algorithm will have to be used for only 3 families. Therefore, the quality of the approximation will not impact dramatically the total volume of I/O, the essential of the optimization being performed by the *discrimination* step. This is why we propose the following greedy heuristic to perform the approximation that only considers a number of combinations bounded by the number of *unfixed* children plus one. We start from a partition where the *fixed* children are sorted according to the *discrimination* step and where the *unfixed* children are positioned before the split point. In other words, the children that satisfy $m + S_j^{flexible} \leq M_0$ are positioned *after* the split point (according to Property 4.1) whereas all the other ones are positioned *before*. Then, we iteratively select the child that is responsible for the peak of storage before the split point and we move it after the split point until one move does not decrease the volume of I/O anymore. We name `Flex-MinIO` this two-step algorithm.

4.8 Experimental results

In order to evaluate the impact of this flexible allocation scheme on the volume of I/O, we compare the results of our heuristic (`Flex-MinIO`) both to the terminal allocation scheme with the IO-minimizing algorithm of Chapter 3 (that was named `MinIO` and

that we rename `Term-MinIO`) and to the flexible allocation scheme with the memory-minimizing algorithm of [44] (`Flex-MinMEM`).

The volumes of *I/O* were computed by instrumenting the analysis phase of `MUMPS`. We experimented several ordering heuristics and present results with both `METIS` [51] and `PORD` [69]. We present results related to four test problems extracted from Table 1.2 (see Section 1.4.3) for which we have observed significant gains. Figure 4.2 shows the evolution of the volume of *I/O* with the available memory. 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 amount of *I/O*): the possible extra-*I/O's* corresponding to underbraces **(a)** and **(b)** of Formula (4.4) are actually equal (or almost equal) to zero for those children.

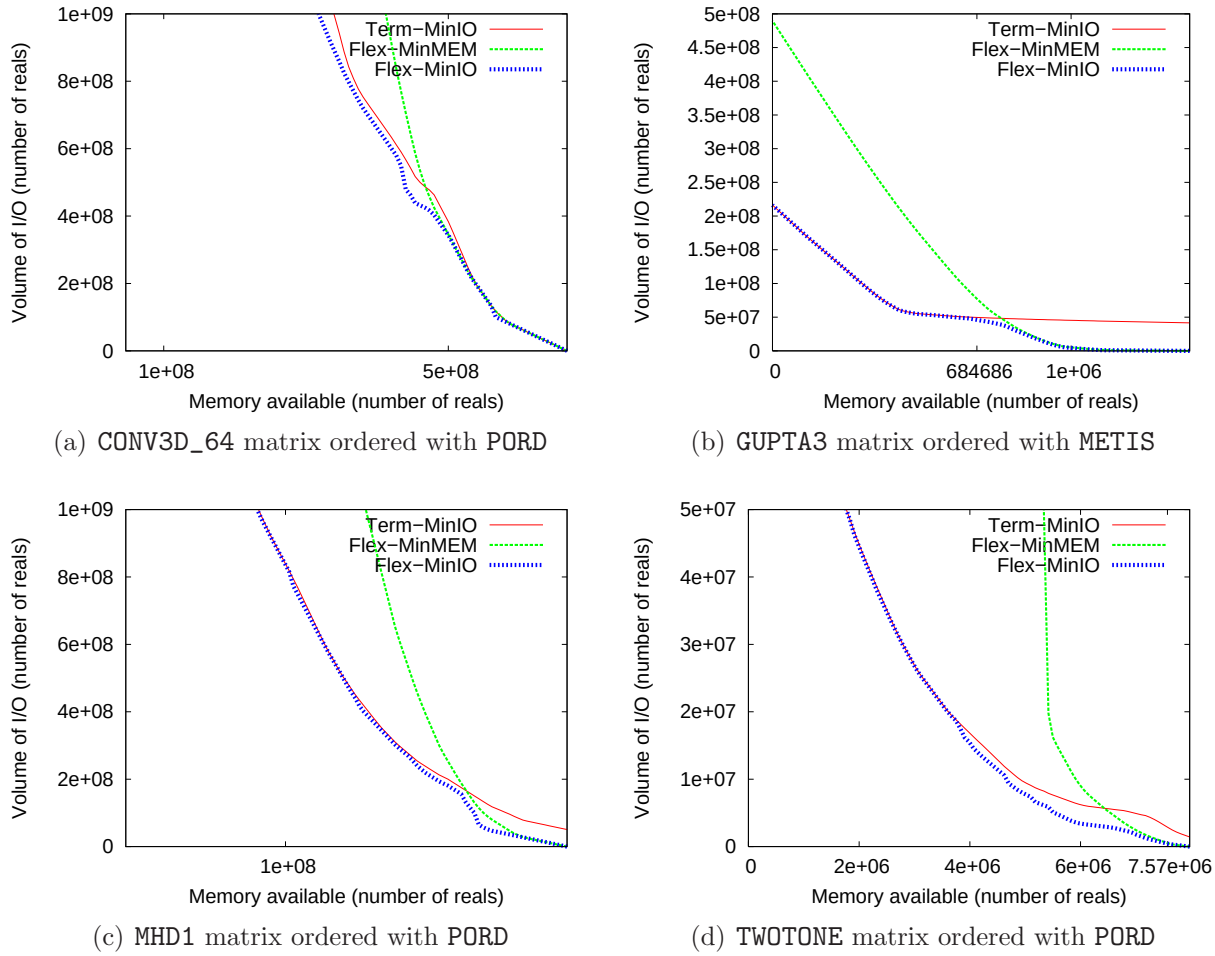
When the amount of available memory is small (left part of the graphs), the memory-minimizing algorithm (`Flex-MinMEM`) induces a very large amount of *I/O* compared to the *I/O*-minimization algorithms (both `Flex-MinIO` and `Term-MinIO`). Indeed, processing a child after the parent allocation may then induce a very large amount of *I/O* (M_0 is small in underbraces **(a)** and **(b)** of Formula (4.4)) 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 the two 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.

4.9 Conclusion

In this chapter, we have studied the *I/O* minimization problem in the context of the flexible multifrontal method and proved that it is NP-complete. In particular, it is interesting to notice that the *I/O* minimization problem is more complex than the minimization of the storage requirement which is polynomial [44]. However we have proposed an algorithm that provides a configuration which most of the time does minimize the *I/O* volume at no cost more than sorting the children of the family. We have shown that the practical impact on the *I/O* volume may be significant. In particular, an experimental study has shown that an algorithm which aims at minimizing the storage in the context of the multifrontal method can lead to dramatically huge *I/O* volumes. Said differently, it is even more critical to use an *I/O*-minimization algorithm in the flexible context than it was with a terminal allocation scheme compared to an algorithm that would aim at minimizing the storage.

An extension to the flexible allocation scheme could consist in allocating the frontal matrix of the parent more than once. Each time, a range of contribution blocks is assem-



(a) `CONV3D_64` matrix ordered with `PORD`

(b) `GUPTA3` matrix ordered with `METIS`

(c) `MHD1` matrix ordered with `PORD`

(d) `TWOTONE` matrix ordered with `PORD`

Figure 4.2: I/O volume on the stack of contribution blocks as a function of the core memory available for the three heuristics with the use of an *last-in-place* assembly scheme.

bled, and the frontal matrix is written back to disk when necessary. It is only read again when enough contribution blocks have been computed. However, this *multiple allocation scheme* does not allow us to decrease the *I/O* volume and is also NP-complete as we prove in Appendix C (Section C.2).

We have shown that the results presented in this chapter apply to all the considered assembly schemes (*classical*, *last-in-place* and *max-in-place* ones). We have furthermore discussed how to handle the *in-place* allocation in the flexible context and proposed a new scheme (named *in-place-realloc*) to further limit the *I/O* volume on the children processed after the parent allocation.

Chapter 5

Memory management schemes for multifrontal methods

In this chapter we aim at providing new memory management algorithms, adapted to the different multifrontal schemes presented in the previous chapters. We show that our models can lead to a reasonable implementation during the numerical factorization, without extra copies or complicated garbage collection mechanisms: we show that those can be avoided by relying on simple stack mechanism.

The different memory-minimization and *I/O*-minimization algorithms presented in chapters 3 and 4 compute a particular postorder traversal of the elimination tree. With a flexible allocation scheme (Chapter 4), they also compute the positions of the parent allocations. These algorithms can be applied during the analysis phase of a sparse direct solver, *i.e.* during a preliminary step performed before the numerical factorization. Then the numerical factorization relies on this traversal and should respect the forecast optimal metrics (memory usage, *I/O* volume). We suppose that a traversal has been given (thanks to one of the algorithms presented in the previous chapters) and we present memory management algorithms that match the different assembly schemes we have considered for both the terminal and flexible allocation schemes. Remember that we consider that the factors are written to disk on the fly. As soon as a block of the frontal matrix is factored it can be written to disk, possibly asynchronously. Thus we only have to store temporary frontal matrices and contribution blocks. We assume that those are 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 5.1.

The chapter is organized as follows. We first describe mechanisms corresponding to an in-core management of the contribution blocks in Section 5.1. Those mechanisms can be applied when the storage requirement is smaller than the available memory M_0. In Section 5.2, we generalize those mechanisms in the case of an out-of-core storage of the contribution blocks that do not fit in memory.

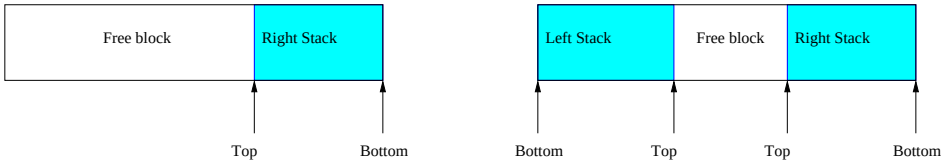


Figure 5.1: Subdivision of the main workarray, W, into one stack (left) or two stacks (right) of contribution blocks. The free block can be used to store the temporary frontal matrices.

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 5.1.1. In Section 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 5.1.3.

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 rest of the chapter. We have seen in Chapter 2 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 5.1. *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 5.2, 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 on the left of W, then cb_5 and cb_4 (in that order) are assembled into it and removed from the stack. Once the assembly of the parent is finished, the frontal matrix is factored, 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 as was illustrated in Figure 2.3(c). In the example of Figure 5.2, this means that if the free block on the left of the workarray 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.

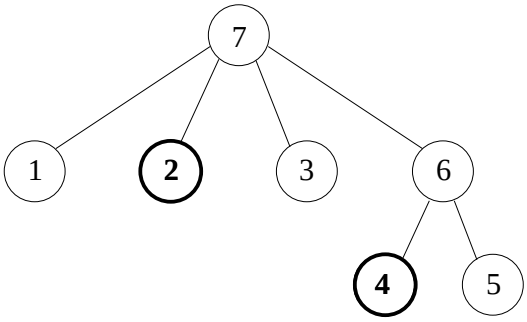


Figure 5.2: Example of a tree with 7 nodes. Nodes in bold correspond to the nodes with the largest contribution block among the siblings. (This property will only be used in Section 5.1.2.)

5.1.2 In-place assembly of the largest contribution block

We have introduced in Chapter 3 (Section 3.3) a new *in-place* assembly scheme and we now propose a memory management mechanism that matches this scheme. This *max-in-place* assembly scheme 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 contiguous 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 5.1), Property 5.1 also applies to the largest contribution blocks. Thus, when processing a parent node,

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

This is illustrated by the tree of Figure 5.2. 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.

We name `AlgoIC_maxinplace`() the corresponding memory management algorithm.

5.1.3 Flexible allocation of the frontal matrices

5.1.3.1 Classical and last-in-place assembly schemes

We now consider the flexible multifrontal method as discussed in Chapter 4. In this method, the frontal matrix of a parent is allowed to be allocated before all the children have been processed. This implies that several frontal matrices may be in core at the same time. We assume that a *classical* assembly scheme is used. On the example of Figure 5.3, 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)**.

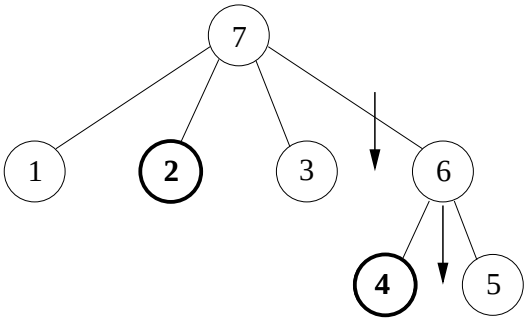


Figure 5.3: Reconsidering the example of Figure 5.2 with a flexible allocation. The arrows indicate the position at which the frontal matrices of the parents are allocated. Nodes in bold correspond to the nodes with the largest contribution block among the siblings processed before a parent allocation (this property will only be used in Section 5.1.3.2).

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 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.

We now illustrate the use of those two stacks on our example. After node **(7)** has been allocated, the contribution blocks of nodes **(1)**, **(2)** and **(3)** are assembled and released. Then, node **(4)** is factored and produces a contribution block. At this time, the workarray W contains:

f_7	Free block	cb_4

The frontal matrix of node **(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

The contribution block of node **(4)** is assembled into f_6 and released. Next; node **(5)** is processed; its contribution block is assembled into f_6 and released. Node **(6)** is factored; its contribution block is assembled into f_7 and released. Ultimately, node **(7)** factored.

5.1.3.2 Max-in-place assembly scheme

The mechanism of the previous section is not immediate to adapt to the *max-in-place* scheme. Indeed, as explained in Section 5.1.2, this latter scheme requires a supplementary stack for storing the largest contribution block among the siblings processed before the parent allocation of each family. All in all, three stacks would be required: one for the frontal matrices, one for the largest contribution blocks and one for the other contribution blocks. The difficulty is that it is not straightforward to manage three stacks within a single workarray. However, it is possible to merge the stack of the largest contribution blocks with the one of the frontal matrices into a single common stack as we now explain. We already know that both the largest contribution blocks and the frontal matrices follow a LIFO data access pattern. It remains to check that (i) a largest contribution block produced before a frontal matrix is released after that frontal matrix and that (ii) a largest contribution block produced after a frontal matrix 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 5.3, 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.

We name `AlgoIC_flex_maxinplace`() the corresponding memory management algorithm.

5.2 Out-of-core stacks

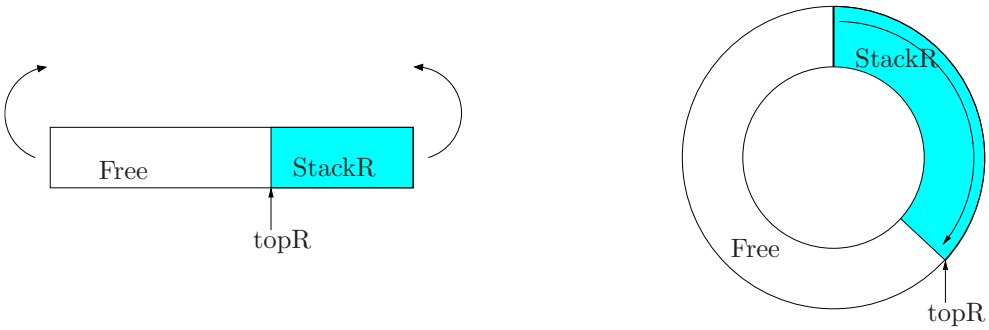
We now consider the out-of-core contexts of chapters 3 and 4. This assumes that the contribution blocks, and thus the different stacks, may be processed out-of-core (only the active frontal matrix has to be kept in core). When the free space vanishes, Property 2.1 suggests that the bottom of the stack(s) should be written to disk in priority. Therefore, the question of how to reuse the corresponding workspace arises. We give a first natural answer to this question in Section 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 sections 5.2.2 and 5.2.3 an original approach that greatly simplifies the memory management for all the considered assembly schemes.

5.2.1 Cyclic memory management

In the *classical* and *last-in-place* cases with a terminal allocation, 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 5.4. 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 3.1 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 5.5(a)).

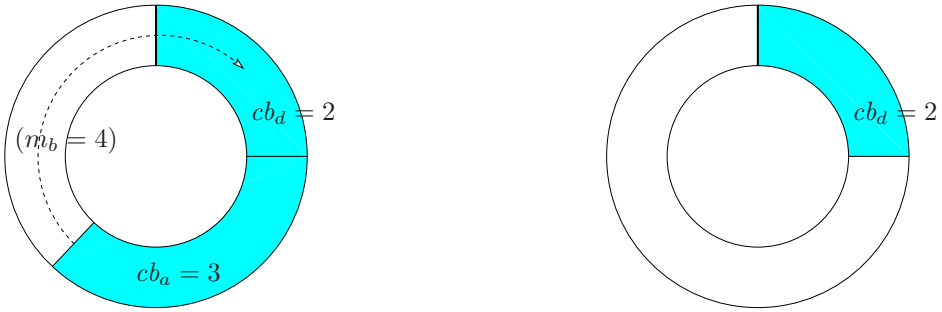
A significant drawback of this approach is that a specific management has to be applied to the border, especially when a contribution block or a frontal matrix is split on both sides of the memory area (as occurs for frontal matrix m_b in Figure 5.5(a)).

With a *max-in-place* scheme or with any assembly scheme based on a flexible allocation, the out-of-core extension is not as natural because of the existence of two stacks. One may decide to move the two stacks against each other as illustrated in Figure 5.6(a). As the bottom of the stacks is written to disk first, the corresponding freed space may be reused only if (at least) one stack is shifted in its whole. Such a memory copy may be a large overhead for efficiency. On the contrary, one may decide to move the two stacks in the same direction (see Figure 5.6(b)). In this case, *I/O* will be required as soon as the top of a stack reaches the bottom of the other one. But the relative speed of movement of



(a) Workarray (b) Cyclic workarray

Figure 5.4: Folding a linear workarray (left) into a cyclic workarray (right).

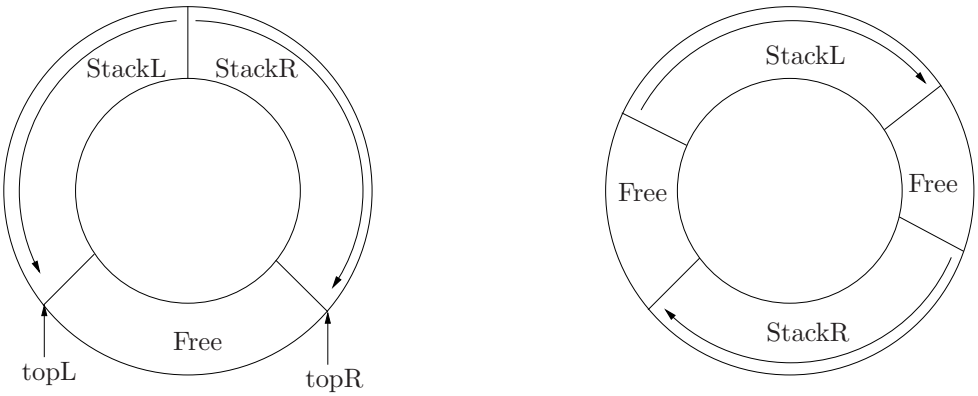


(a) Dynamic cyclic approach (b) Top-down approach

Figure 5.5: Memory state while processing the tree of Figure 3.1 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_d only before dealing with node **(b)**. With the approach of Section 5.2.2 (right), we know *a priori* that cb_d must be fully written to disk thanks to the analysis phase.

each stack within the circular memory area is not controlled (it is dictated by the given postorder tree traversal). As a consequence, one stack might reach the bottom of the other one and imply *I/O* on recently produced contribution blocks while keeping older contribution blocks in its own bottom. This would break the rule which states that the oldest contribution blocks are written to disk first and would prevent one to minimize the volume of *I/O*. To avoid this overhead on the volume of *I/O*, one should shift again one whole stack, which may be in turn an overhead for efficiency. In any case, this might imply a drop of efficiency.

In the next subsections, we propose another approach which avoids the use of a cyclic stack for the *classical* and *last-in-place* cases, and allows to efficiently handle the other cases where two stacks are required.



(a) Opposite directions (b) Same directions

Figure 5.6: Two possibilities for a cyclic out-of-core double stack mechanism.

5.2.2 Using information from the analysis: terminal allocation scheme

In order to minimize the *I/O* volume in the previous approach, a contribution block 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. We focus for the moment one the terminal allocation scheme.

Property 5.2. *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 forecasts done during the analysis phase. When considering a parent node with n child subtrees, the volume of *I/O* $V_{family}^{I/O}$ performed on the direct 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 (2.3) and (3.2) respectively for the *classical* and *in-place* cases. For example,

$$V_{family}^{I/O} = \max\left(0, \max(\max_{j=1,n}(A_j + \sum_{k=1}^{j-1} cb_k), m + \sum_{k=1}^{n} cb_k) - M_0\right) \tag{5.1}$$

in the *classical* assembly scheme with a terminal allocation. 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 fully remains in-core.

In the tree of Figure 3.1 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 5.3. *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 block is produced. This will induce the same overall* I/O *volume.*

Thanks to Property 5.3, we will assume in the following that:

Assumption 5.1. *We decide to write all the contribution blocks which have to be written as soon as possible.*

This is illustrated in Figure 5.5(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 may decide to write it as soon as possible, *i.e.*, before processing node **(a)**.

Property 5.4. *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 2.1. A slightly stronger property is the following:

Property 5.5. *If a subtree requires some* I/O*, then at the moment of processing the first leaf of that subtree, the memory 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 5.6. *When we stack a frontal matrix 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 to process this subtree, the contiguous free block of our workarray W is necessarily at least as large as S.*

It follows that by relying on Assumption 5.1 a cyclic memory management is not needed anymore: a simple stack is enough for the *classical* and *last-in-place* assembly schemes based on a terminal parent allocation, and a simple double stack is enough for all the other cases (*max-in-place* assembly scheme or any assembly scheme with a terminal allocation).

Application to the max-in-place terminal scheme

We now illustrate this approach on the out-of-core *max-in-place* variant of Section 3.3 with a terminal allocation. We recall that the heuristic we have proposed in Section 3.3 switches to a *last-in-place* approach for subtrees involving *I/O*. Therefore, a double stack will only be needed when processing in-core subtrees.

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 5.1. It is a top-down recursive formulation, more natural in our context, which starts with the application of `AlgoOOC_maxinplace_rec`() on the root of the tree. A workarray W of size M_0 is used. We rely on the in-core memory management algorithm described in Section 5.1.2 for the *max-in-place* assembly scheme that we named `AlgoIC_maxinplace`().

```
% W: workarray of size M_0
% n: number of child subtrees of tree T
for j = 1 to n 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 5.6
        AlgoIC_maxinplace(T_j) (Apply the max-in-place approach - see
        Section 5.1.2);
    else
        % Some I/O are necessary on this subtree, therefore W is empty
          (Property 5.5)
        % We do a recursive call to AlgoOOC_maxinplace_rec(), using all
          the available workspace
        AlgoOOC_maxinplace_rec(subtree T_j) ;
    Write cb_j to disk or stack it (decision based on Property 5.2 and
    Assumption 5.1);
Allocate frontal matrix of the parent node; it can overlap with cb_n;
for j = n 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);
Factorize the frontal matrix; this step produces a contribution block (except for
the root node);
```

Algorithm 5.1: `AlgoOOC_maxinplace_rec`(tree T).

5.2.3 Using information from the analysis: flexible allocation scheme

We now explain how to generalize the mechanism to a flexible allocation scheme. Similarly to what was done with the terminal allocation scheme, one can determine *a priori* if the contribution block cb_j of a child j processed before the parent allocation ($j < p$) must be written to disk. This is an adaptation of Property 5.2. In the flexible case, assuming this time (for example) that we have a *last-in-place* scheme before the parent allocation, $V^{I/O}_{family}$ can be computed as follows:

$$V^{I/O}_{family} = \max\left(0, \max(\max_{j=1,\boxed{p}}(\min(S_j, M_0) + \sum_{k=1}^{j-1} cb_k), m + \sum_{k=1}^{\boxed{p}-1} cb_k) - M_0\right) \quad (5.2)$$

As before, knowing that oldest contribution blocks must be written to disk in priority, cb_j will have to be written to disk, possibly partially, if $\sum_{i=1}^{j} cb_i < V^{I/O}_{family}$. In that case, we decide to write cb_j to disk as soon as possible. Therefore, Property 5.6 (and the previous ones) still hold. Our memory management algorithm will also rely on two new properties that we state below.

Property 5.7. *We consider a family involving some* I/O*, using either a a* classical *or* last-in-place *assembly scheme. Before processing the first subtree of this family, we know that the memory is empty (Property 5.5). 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 5.8. *Considering a family involving* I/O*, once the the contribution blocks from the children $j \leq p$ have been assembled in the frontal matrix of the parent, that frontal matrix is alone in memory.*

Property 5.8 is a direct consequence of Property 5.7.

We now consider a child j processed after the parent allocation ($j > p$). According to Property 4.2, we know that such a subtree can be processed in-core alone ($S_j^{flexible} < M_0$). However, if it does not fit together with the frontal matrix of its parent ($S_j + m > M_0$), a part of the frontal matrix equal to $S_j + m - M_0$ has to be written to disk in order to make room for the subtree rooted at j. This subtree is then processed in-core. If we are not considering an *in-place-realloc* scheme (see Section 4.6), then we might have to write part of the contribution block if it does not fit together with the frontal matrix of the parent (*i.e.* if $m + cb_j > M_0$). Finally the part of the frontal matrix that had been written to disk (if any) is read back and the contribution block is assembled into it (possibly requiring to read the part that had been written to disk).

A top-down recursive formulation of this flexible out-of-core approach is given in Algorithm 5.2. This algorithm assumes that the analysis phase has identified in-core subtrees and that those subtrees are processed with the combination (`MinMEM` + flexible +

max-in-place), named `AlgoIC_flex_maxinplace`() in Section 5.1.3.2). The analysis phase has also identified out-of-core subtrees and we decide to use the combination (`MinIO` + flexible + *last-in-place*), as this corresponds to the heuristic from Section 3.3. To simplify the presentation of the algorithm, we consider that we are in the *in-place-realloc* case; this implies that no I/O is required on the contribution blocks processed after the parent allocation. We first apply `AlgoOOC_flex_maxinplace_rec`() on the root of the tree, and a workarray W of size M_0 is used.

Note that only one stack (on the right of W) is manipulated in Algorithm 5.2, although more stacks are used temporarily when applying `AlgoOOC_flex_maxinplace_rec`(). This is because

1. the stack of frontal matrices is empty (thanks to Property 4.2);
2. the heuristic from Section 3.3 uses the *max-in-place* scheme only on in-core families, not out-of-core ones.

However, we still have freedom to use another stack on the left of W if needed, for example to isolate the largest contribution block. Although that child may need to be written to disk, depending on the other peaks, it could happen that the part of it that is still in memory at the moment of allocating the frontal matrix of the parent is larger than the contribution block of the last child. In such a case it would make sense to overlap the frontal matrix of the parent with that contribution rather than the one of the last child. This is prospective work since the previous chapters do not provide an optimal order of the children in that case. However, we just insist here on the fact that our memory management algorithms can handle such cases.

5.3 Conclusion

In this chapter, we have proposed memory management algorithms that fit the different multifrontal methods presented in the previous chapters. We have exhibited that all these schemes could be managed with simple stack mechanisms. Table 5.1 sums up how to organize the data in the in-core case depending on the variant of the multifrontal method considered.

We have then proposed models to extend these mechanisms in an out-of-core context. The originality of these out-of-core memory management algorithms is that they allow to avoid complicated garbage collections thanks to information computed during the analysis phase and can lead to a reasonable implementation. In particular, we have shown that a cyclic memory management is not necessary.

In static multifrontal solvers that *do* respect the forecast metrics, the algorithms we have described can be implemented as presented. In dynamic codes (allowing for dynamic pivoting, for example) that do *not* respect exactly the forecast metrics, a specific treatment (emergency I/O, ...) will be required when the storage effectively used by a subtree is larger than was forecast. Another possibility consists in relaxing the forecast metrics, although this implies extra, possibly unnecessary, I/O.

```
% 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 5.6
        AlgoIC_flex_maxinplace(T_j) ;
    else
        % Some I/O are necessary on this subtree, therefore W is empty
          (Property 5.5)
        % We do a recursive call to AlgoOOC_flex_maxinplace_rec(),
          using all the available workspace
        AlgoOOC_flex_maxinplace_rec(subtree T_j) ;
    Write cb_j to disk or stack it (decision based on Property 5.2 and
    Assumption 5.1, but using Formula (5.2)) at the right of W ;
% Thanks to Property 5.7:
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 alone in memory (Property 5.8)
for j = p + 1 to n do
    % We know that S_j ≤ M_0 thanks to Property 4.2
    if the subtree T_j rooted at child j can 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
    AlgoIC_flex_maxinplace(T_j) ;
    Assemble cb_j into the frontal matrix of the root of T (reading from disk 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 node) that we stack on the right of W ;
```

Algorithm 5.2: `AlgoOOC_flex_maxinplace_rec`(tree T). An *in-place-realloc* scheme is used after the parent allocation, and a *max-in-place* (resp. *last-in-place*) is used before the parent allocation for the in-core (resp. out-of-core) parts.

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

Table 5.1: 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*.

Chapter 6

A study of out-of-core supernodal sparse direct methods [1]

In this chapter we study out-of-core supernodal unsymmetric factorizations with partial pivoting. Their evaluation is done with respect to several metrics, including the volume of *I/O*, the spatial locality of disk accesses and the overhead due to indirections. Although many of the discussed algorithms have already been proposed and/or implemented by different authors, we aim at showing the impact of the algorithmic decisions on the different metrics with a formal comparative evaluation.

The purpose of this study is to exhibit a fast algorithm which computes an efficient schedule of the operations before starting the out-of-core numerical factorization. In Section 6.1, we present our assumptions and discuss our degrees of freedom. We then present schedules which minimize the amount of required core memory in Section 6.2. These methods are reused as building blocks for out-of-core factorizations in Section 6.3 where we aim at minimizing the *I/O* volume. *Out-of-core* methods require to group supernodes into subsets such that the supernodes of a same subset are processed together in core memory [41]. Such a subset will be named SuperPanel in this dissertation. This partitioning of the supernodes of the matrix (or equivalently of the nodes of the elimination tree) into SuperPanels significantly impacts the *I/O* volume. We present partitioning algorithms that aim at limiting that *I/O* volume in Section 6.4.

This study has led to the development of a prototype for an out-of-core extension of `SuperLU` that we present in Section 6.5.

1. The work presented in this chapter has been done during a stay at the Lawrence Bekeley National Laboratory under the direction of Xiaoye S. Li in the group of Esmond G. Ng.

6.1 Assumptions on models for out-of-core supernodal algorithms

We focus on *supernodal* algorithms at the exclusion of multifrontal methods as explained in the introduction (Section 1.1.4). We adopt the following definition:

Assumption 6.1. *Supernodal algorithms can only store and work on supernodes.*

We recall the freedom offered by supernodal methods in terms of scheduling of elementary operations. The elementary operations performed by supernodal methods are FACTO and UPDATE operations (see Section 1.1.3). ASSEMBLE-like operations are excluded because they use intermediate data-structures. We consider that the columns of the original matrix have been reordered (thanks to one of the algorithms presented in Section 1.1.1) and that we have to respect the induced dependencies. The dependencies between FACTO operations are provided through the elimination tree: any topological traversal [56] of the elimination tree respects the dependencies between FACTO operations. The choice of a topological traversal thus represents a first degree of freedom. The second degree of freedom is provided by the choice of the schedule of the UPDATE operations. *Left-looking* algorithms delay the UPDATE operations as late as possible whereas *right-looking* algorithms perform the UPDATE operations as soon as possible. However, UPDATE(A_i,A_j) only needs to be performed after FACTO(A_i) and before FACTO(A_j). This flexibility offers a lot of possible schedules among which *left-looking* and *right-looking* are only two particular cases.

In order to compare the behaviour of a supernodal method with another, we moreover consider that all the factors are written to disk during the factorization step (and thus read back during the solution step) as stated in Assumption 6.2.

Assumption 6.2. *All the factors have to be on disk after the factorization step. Therefore, the* I/O *volume related to the first write of the supernodes is constant and we do not count it.*

Writing the factors to disk also makes sense from an engineering point of view. Indeed, it allows a strict separation of the out-of-core factorization and solution steps, which would make it possible for the end-user to perform the solution step long after the factorization in a different `UNIX` process.

In addition to the *I/O* volume related to the first write of supernodes, some supplementary *I/O's* have to be performed when the updating supernode could not be kept in-core until the updated supernode was loaded in memory. Therefore, the schedule of the UPDATE operations impacts the *I/O* volume. However, remember that we focus in this chapter on unsymmetric factorizations with partial pivoting. In this context, neither the structure of the sparse factors nor the exact dependencies can be known without performing the actual numerical factorization. This is explained in sections 1.1.2 and 1.4.1. In particular, it is not possible to forecast the supernodes that a given supernode will update. Because the purpose is to decide of the schedule of the operations before starting

the numerical factorization, we need to rely on an approximation. An optimistic approximation would consist in betting that few rows will be swapped so that we may rely on the initial ordering. A pessimistic approximation would consist in considering the dependencies of A^TA which takes into account all potential numerical pivoting [40]. However, in both cases, the computation and the storage of the whole graph of dependencies might be costly whereas we want a fast algorithm. This is why we rely on a more compact structure, faster to traverse: the elimination tree. On the other hand, the elimination tree does not provide information on the explicit dependencies. We only know that two nodes that are not on the same root-to-leaf path will never update each other. Thus we make the following coarse approximation:

Assumption 6.3. *Each supernode updates all its ancestor supernodes.*

Of course a tighter approximation could be considered but, besides relying on a compact structure, this approach presents two other interests. First, it models the worst case in terms of computation, storage requirements and *I/O* volume and the study of the worst case seems interesting to us for a first approach. Second, this assumption can be relaxed during the numerical step so that only the necessary updates are actually performed. Indeed, recall that the purpose of our models is to schedule the operations before starting the actual factorization. Although we may dynamically adapt the initial forecast as done in [58] for the $\mathbf{LDL}^T$ case, it it is still helpful to compute a good initial schedule.

6.2 Minimum core problem - W1/R0 scheme

Objective 6.1. *Adapting the terminology of [27][2], our* minimum core *problem consists in determining the supernodal method that minimizes the amount of core memory required to perform the factorization of a given sparse matrix. We consider a* Write Once / Read Zero *scheme (*`W1/R0`*) where factors are on disk at the end of the factorization (Assumption 6.2). In particular, any factor written (and freed from memory) should not be accessed again.*

In order to reach Objective 6.1, we need to find a schedule of the Facto and Update operations which minimizes the amount of core memory with a `W1/R0` scheme. Note that [27] answers a closely related question: a complexity analysis and simulations on different classes of sparse matrices show that the *right-looking* method is usually the most efficient method for the *Write Once / Read Once* scheme (`W1/R1`) scheme; but the author does not focus on the `W1/R0` scheme as we do here. We formally show in the following that, under assumptions 6.1, 6.2 and 6.3, a *right-looking* method does minimize the *minimum core* problem whereas the *left-looking* method is not suited.

Proposition 6.1. *If the elimination tree is a chain, all the supernodes have to be together in memory at a given instant.*

2. In the context of Dobrian's approach, the *minimum core* problem is related to in-core processes.

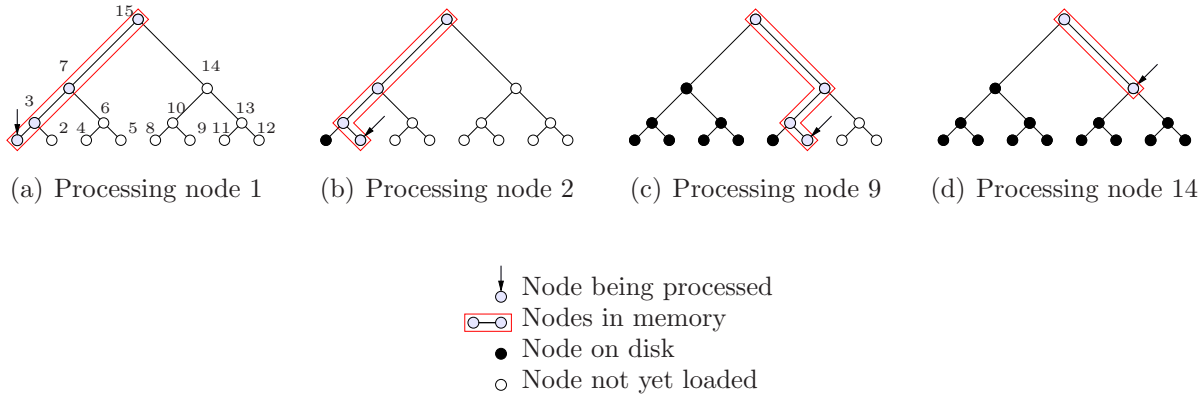
Proof. Otherwise there would exist two distinct supernodes i and j such that the allocation of j is performed after having written and released i from memory. Because a `W1/RO` scheme is used, i is not read again; thus i and j are never in core together. On the other hand Assumption 6.3 requires either that i updates j (if j is ancestor of i) or that j updates i (if i is ancestor of j). Both configurations require to have i and j together in memory, which is a contradiction. ■

Corollary 6.1. *The* minimum core *required for a general elimination tree is at least equal to the maximum storage of a root-to-leaf path.*

Proof. Proposition 6.1 applies to any root-to-leaf path (a chain from the root node to a leaf node) and in particular to the one that requires the maximum storage. ■

Proposition 6.2. *The* minimum core *for a* right-looking *method is equal to the maximum storage of all root-to-leaf paths.*

Proof. We exhibit a *right-looking* method which satisfies this criterion. The elimination tree with a postorder traversal. When a supernode is processed, its ancestors are loaded in memory if they are not loaded yet. As soon as all its ancestors have been updated, the current supernode is written to disk and freed from memory. Figure 6.1 illustrates the corresponding memory behaviour on a sample elimination tree. Nodes 1, 3, 7 and 15 (we



(a) Processing node 1 (b) Processing node 2 (c) Processing node 9 (d) Processing node 14

Figure 6.1: Different stages of a `W1/RO` *right-looking* factorization. At any time during the postorder traversal, no more than one root-to-leaf path has to be held in core.

use a postorder numbering) are loaded in memory. Node 1 is factored and updates its ancestors $\{3, 7, 15\}$ as shown in Figure 6.1(a). It is then written to disk and freed from core memory. Node 2 is loaded in memory (see Figure 6.1(b)) and updates its ancestors. It is written to disk and node 3 is processed. And so on. For instance, when node 9 is loaded in memory (see Figure 6.1(c)), nodes 1 to 8 are on disk; nodes 10, 14, and 15 are in memory; nodes 11, 12 and 13 have not been loaded yet. Note that a `free` operation could be delayed until a new node has to be loaded. For instance, node 2 could have been freed after node 3 has been processed. ■

Corollary 6.2. *The* right-looking *method (is the supernodal method which) minimizes the* minimum core *problem.*

Proof. Immediate from Corollary 6.1 and Proposition 6.2. ■

Proposition 6.3. *The* left-looking *method requires an amount of memory equal to the storage of all supernodes in the tree.*

Proof. The root is an ancestor of all the other nodes of the elimination tree. According to Assumption 6.3, all those nodes have thus to update it. With a *left-looking* method, those updates have to be performed when the root node is processed, *i.e.*, last. Therefore, no node could have been freed from memory until the process of the root. At that moment, all the nodes are together in memory. Figure 6.2 illustrates the corresponding memory

Figure 6.2: Memory state when the root node is being processed in a `W1/R0` *left-looking* factorization. All the nodes have been loaded but none of them could be freed from memory.

state on a sample elimination tree when processing the root node of the tree. ■

Note that, if we do not rely on Assumption 6.3 anymore, the *right-looking* method is still natural to apply with a `W1/R0` scheme: the same memory management mechanism as the one proposed in the proof of Proposition 6.2 and illustrated in Figure 6.1 can be used. The `W1/R0` schemes studied in this section are reused in the following as building blocks for hybrid out-of-core schemes.

6.3 Combining left-looking and right-looking methods to limit the I/O volume

As already discussed, in an out-of-core context, limiting the volume of *I/O* is crucial. In [27], the author studies the minimization of the *I/O* volume for different classes of matrices. Simulations show that when the core memory available becomes small, the *right-looking* method usually induces far more *I/O* than both *left-looking* and *multifrontal* methods. This means that there is a gap between the case where a large amount of memory is available - for which the *right-looking* method has a very good behaviour and is even optimum if we assume that each node updates its ancestors - and the case where

the available memory becomes small - for which a pure *right-looking* method seems to be inadequate. In other words, locally, the *right-looking* method is efficient but not globally (if the available memory is small compared to the storage required).

To handle this gap, Dobrian suggested in the conclusion of [27] to implement different out-of-core methods (*left-looking*, *right-looking* and *multifrontal*). An analysis step would then automatically decide which algorithm to apply, depending on the characteristics of the matrix and on the available memory. However, such an automatic algorithm might not be as adaptative as it appears. Indeed the algorithm finally applied is *either* a pure *left-looking or* a pure *right-looking* approach but does not benefit from the advantages of both methods. This remark motivates the construction of a hybrid algorithm locally based on the *right-looking* method but using a *left-looking* approach at a higher level. Such a hybrid algorithm has been separately introduced and implemented for unsymmetric factorizations in [41]. The main drawback of that approach, as stated by its authors and recalled in Section 1.2.4, is that updates are not actually performed between supernodes but between columns. In [63], the authors implemented that algorithm for the Cholesky factorization, enhanced thanks to the use of supernodes.

Out-of-core hybrid algorithms such as [41, 63] rely on a partition of the elimination tree into disjoint subgraphs so that each subgraph is a connected subgraph which could be processed with a `W1/R0` approach if it were alone. We name SuperPanel [3] such a subgraph. In [63], the authors consider that the updates between SuperPanels are performed with a *left-looking* method. They study the combination of those *left-looking* updates between SuperPanels with two inner methods: either a *left-looking* or a *right-looking* approach is applied within the SuperPanels. The authors have compared the impact on the *I/O* volume and on the efficiency of their methods. Besides the practical impact, their work constitutes an important contribution to model the *I/O* volume in the context of supernodal methods. The purpose of this section is to propose a model to study the *I/O* volume when updates between SuperPanels are performed with a *right-looking* approach and to compare the effects on *I/O* volume to the *left-looking* approach. We also consider that the inner factorization of a SuperPanel can be performed either with a *left-looking* or with a *right-looking* method. All in all, still relying on assumptions 6.1, 6.2 and 6.3, we aim at addressing Objective 6.2:

Objective 6.2. *How can we combine* left-looking *and* right-looking *methods to minimize the volume of* I/O*?*

We will consider four variants, following a terminology "outer method / inner method". The inner method (*i.e.* the method applied within a SuperPanel) will always refer to one of the `W1/R0` scheme presented in Section 6.2. The method that we call "inner *right-looking* method" refers to the one described in Figure 6.1 and allows for large SuperPanels that do not have to be fully kept in core (only a root-to-leaf path of the SuperPanel has to remain in core). The method that we call "inner *left-looking* method" refers to the one described in Figure 6.2 and requires to keep the entire SuperPanel in memory. In

3. The term *subtree* or *cold subtree* was used in [63].

the following subsections we first consider a *right-looking* approach between SuperPanels (*right-looking/right-looking* and *right-looking/left-looking* methods), then a *left-looking* one (*left-looking/right-looking* and *left-looking/left-looking* methods) and finally we compare them. Note that, besides the volume of *I/O*, we will also take into consideration the spatial locality of the *I/O's*.

6.3.1 Right-looking approach between SuperPanels

With a *right-looking* approach between SuperPanels, the processing of a SuperPanel consists in:

(i) loading this SuperPanel in memory;
(ii) factorizing it;
(iii) updating the ancestor nodes which are not part of this SuperPanel.

We name *outer ancestors* the ancestor nodes involved in step (iii).

Definition 6.1. *An* outer ancestor *of a SuperPanel is a node on the path from the SuperPanel up to the root of the elimination tree which is not part of the SuperPanel. This definition more generally applies to any connected subgraph.*

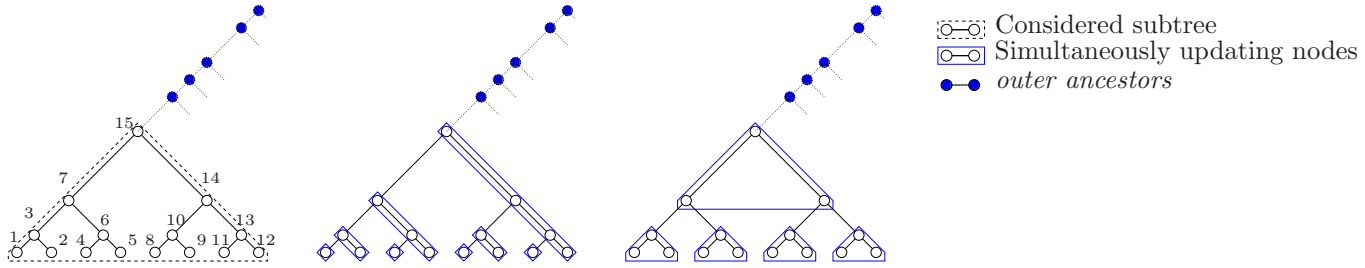
An *outer ancestor* is updated as follows. First it is temporarily loaded in memory. Note that we neglect the corresponding *I/O* the first time it is updated because we consider that it is small compared to the rest in that case (it only contains parts of the initial matrix). Then the numerical updates are applied to it, and it is written back to disk. To decrease the *I/O* volume, each *outer ancestor* receives the contributions of all the updating nodes currently in memory in the SuperPanel and already factored. Of course, in general, all the *outer ancestors* cannot fit in core together with the current SuperPanel.

Right-looking approach inside SuperPanels (`RL-RL`)

Let us consider the case where a SuperPanel is processed with a (`W1/RO`) *right-looking* scheme. Following the terminology "outer method/ inner method", the overall algorithm corresponds to a *right-looking/right-looking* (`RL-RL`) approach. Because a SuperPanel is processed with a `W1/RO` *right-looking* scheme, not all the nodes of the SuperPanel are simultaneously in core (as discussed in Section 6.2). Thus each *outer ancestor* has to be updated before an updating factored node can be freed from memory. Noticing that two leaves of the same SuperPanel are never together in core in the case of a `W1/RO` *right-looking* inner factorization, we conclude that an updated node has to be read and written back to disk as many times as there are leaves in the SuperPanel. This is stated as Property 6.1.

Property 6.1. *During the treatment of a SuperPanel with a* `RL-RL` *scheme, each* outer ancestor *has to be read and written back to disk as many times as there are leaves in the SuperPanel.*

We illustrate this property on a sample tree: we now consider the elimination tree previously given in Figure 6.1 as a subtree of a larger elimination tree, as shown in Figure 6.3(a). We suppose that 4 nodes can fit in a SuperPanel, which corresponds to



(a) Elimination tree (b) RL-RL approach (c) RL-LL approach

Figure 6.3: Partitioning of a subtree of an elimination tree into connected subgraphs that update simultaneously some *outer ancestors*, when 4 nodes can fit in a SuperPanel. With a RL-RL approach, as many such subgraphs as leaves are required (even though the whole subtree fits in a single SuperPanel). In the RL-LL case, the subgraphs exactly match the SuperPanels (but the SuperPanels are smaller).

the largest root-to-leaf path of the considered subtree. Thus, the subtree constitutes a single SuperPanel (as we use a W1/RO *right-looking* approach within the SuperPanel). After node 1 (in the postorder) is factored, it has to be written to disk and freed from memory to make room for loading node 2. At this moment, node 1 is the only node fully factored. It has updated its ancestors within the SuperPanel (nodes 3, 7, and 15, which are kept in memory) but not the *outer ancestors* yet. However, it has to update them too before being freed from memory because a *right-looking* updating scheme is also used at the outer level. Therefore these *outer ancestors* are loaded in memory (possibly blocks by blocks), updated by node 1 and written back to disks. Afterwards, nodes 2 and 3 are factored. As they only have to be freed from memory when the next leaf is loaded (node 4), they can update the *outer ancestors* together. Then node 4 alone has to update the *outer ancestors* since it has to make room for node 5 as soon as processed. Nodes 5, 6 and 7 can then be factored and update together the *outer ancestors*. And so on. As illustrated in Figure 6.3(b) here is the partition of the subtree into subgraphs: $\{\{1\},\{2,3\},\{4\},\{5,6,7\},\{8\},\{9,10\},\{11\},\{12,13,14,15\}\}$. Finally the subtree is partitioned into 8 connected subgraphs (one per leaf node), each of them simultaneously updating the *outer ancestors*. Suppose that there are 100 *outer ancestors* to update, it implies a volume of *I/O* on the *outer ancestors* equal to 800 (reads and writes of a node) during the process of the considered subtree.

Left-looking approach inside the SuperPanel (RL-LL)

We now consider the case where a SuperPanel is processed with a W1/RO *left-looking* approach (the one of Section 6.2). The global scheme corresponds to a *right-looking/left-*

looking (RL-LL) algorithm. As stated in Section 6.2, with the inner *left-looking* method, all the nodes of the SuperPanel are factored and are altogether in core at the moment of processing the root node (see Figure 6.2). Thus each *outer ancestor* is updated only once during the treatment of the SuperPanel. We state this result in Property 6.2.

Property 6.2. *During the treatment of a given SuperPanel with a* RL-LL *scheme, each* outer ancestor *is read and written back to disk once.*

On the other hand, the SuperPanels obtained with a W1/RO *left-looking* inner method are smaller (all the nodes of the SuperPanel have to fit in core together). That might lead to perform more *I/O*. For instance, if the available memory allows the storage of 4 nodes, the sample tree given in Figure 6.1 can be processed in core with a *right-looking* approach but requires an out-of-core process if a *left-looking* approach is applied. In the latter case, it will have to be partitioned into several SuperPanels - each one holding a maximum of 4 nodes. We now consider this tree as a subtree of a larger elimination tree as in Figure 6.3(a) and we measure the volume of *I/O* on the *outer ancestors*. One may notice that a possible (naive) partition is the one resulting from the *right-looking/right-looking* partitioning into connected subgraphs that simultaneously update the *outer ancestors*. On our sample tree, this means that the partition of Figure 6.3(b) (which determines the volume of *I/O* on the *outer ancestors* of a RL-RL method) can be used as a partition into SuperPanels for the RL-LL method. Note that, in that case, both the RL-LL and RL-RL schemes lead to the same volume of *I/O*. Therefore, we have:

Property 6.3. *Processing a subtree with a* RL-LL *scheme leads to a volume of* I/O *on the* outer ancestors *of this subtree equal to or smaller than the one induced with a* RL-RL *scheme.*

Another possible partitioning of this subtree into SuperPanels (with a *left-looking* inner method) is given in Figure 6.3(c): $\{\{1,2,3\},\{4,5,6\},\{8,9,10\},\{11,12,13\},\{7,14,15\}\}$. Here the subtree is partitioned into 5 SuperPanels, each of them simultaneously updating the *outer ancestors*. Suppose again that there are 100 *outer ancestors* to update, it implies a volume of *I/O* on the *outer ancestors* equal to 500 (reads and writes of a node) while processing the considered subtree. To be comparable to the *right-looking* inner scheme, the volume of *I/O* furthermore has to include the *I/O* due to the fact that the subtree is constituted of several SuperPanels. This overhead represents an extra volume equal to 8 units of *I/O* performed on SuperPanel 5 (which is composed of nodes $\{7,14,15\}$) while processing the leaf SuperPanels. For instance, when processing the leaf SuperPanel $\{8,9,10\}$, nodes 14 and 15 have to be updated because they are not in core at this moment. All in all, it leads to a volume of *I/O* equal to 508 which is far smaller than the 800 units for a *right-looking/right-looking* factorization. As a conclusion, the *right-looking/left-looking* approach locally generates a large amount of *I/O* but induces a smaller asymptotic volume of *I/O* than the *right-looking/right-looking* approach.

Spatial locality of disk accesses - outer right-looking approaches

Finally, let us focus on the spatial locality of disk accesses. The pattern of these accesses is independent of the inner factorization since they are performed above the current SuperPanel. The following statements are thus valid for both `RL-RL` and `RL-LL` algorithms. Let us consider Figure 6.4. When processing the SuperPanels within

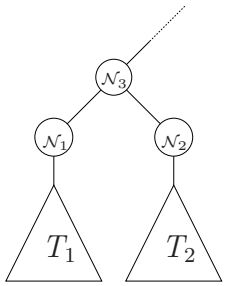


Figure 6.4: Spatial locality of disk accesses with a `RL` outer scheme.

Subtree T_1, nodes $\mathcal{N}_1$ and $\mathcal{N}_3$ are updated; they thus have to be read from disk and written back, possibly several times. When processing the SuperPanels within T_2, this time, nodes $\mathcal{N}_2$ and $\mathcal{N}_3$ are updated and have to be read and written back. Which node among $\mathcal{N}_1$ and $\mathcal{N}_2$ should be the predecessor of $\mathcal{N}_3$ on disk ? If it is $\mathcal{N}_1$, then there is no locality for the disk accesses between $\mathcal{N}_2$ and $\mathcal{N}_3$, which is likely to slow down the process of T_2. On the other hand, if it is $\mathcal{N}_2$, there is no locality between $\mathcal{N}_1$ and $\mathcal{N}_3$, which is likely to slow down the process of T_1. Some dynamic schemes of allocation of data on disks might improve this I/O behaviour, but there is no obvious - to us - way to get a good spatial locality of disk accesses with an outer *right-looking* scheme.

In a nutshell, first, with a *right-looking* approach between SuperPanels, an inner `W1/R0` *right-looking* factorization of the SuperPanels is likely to induce a large amount of I/O. This is especially the case when the number of nodes that can fit in a SuperPanels is small compared to the number of nodes of the elimination tree, *i.e.*, when the available memory is small compared to the memory required to store the sparse factors. An inner *left-looking* approach allows for a better scalability in terms of volume of I/O. However, the subsequent SuperPanels are smaller and this also represents a non negligible amount of I/O either. Second, in both cases, there is no possible organization of the data on disk that allow contiguous disk accesses.

6.3.2 Left-looking approach between SuperPanels

With a *left-looking* approach between SuperPanels, the treatment of a SuperPanel consists in:

(i) loading the SuperPanel in memory;
(ii) updating it with the nodes of the subtrees below this SuperPanel;
(iii) factorizing the SuperPanel, writing it to disk and releasing it from memory.

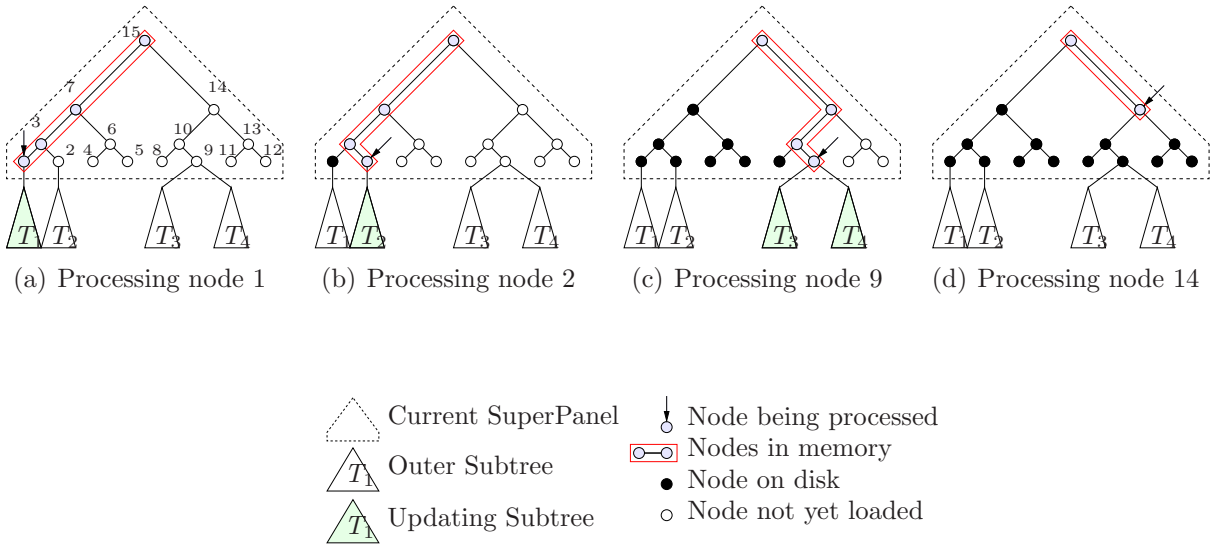
We name *outer subtrees* the subtrees involved in step (ii).

Definition 6.2. *An* outer subtree *of a SuperPanel is a subtree whose root is a child of a leaf node of the SuperPanel. This definition more generally applies to any connected subgraph.*

Notice that we will not consider the *I/O* involved in step (i) because the first time a SuperPanel is accessed, it only contains parts of the initial matrix, which is either already in-core, or is small compared to the factors.

Right-looking approach inside SuperPanels (`LL-RL`)

Let us consider the case where a SuperPanel is itself processed with a (`W1/RO`) *right-looking* scheme. The overall algorithm thus corresponds to a *left-looking/right-looking* (`LL-RL`) approach. Because a SuperPanel is processed with a `W1/RO` *right-looking* scheme, not all the nodes of the SuperPanel are simultaneously in core (see again Figure 6.1). Thus (i), (ii) and (iii) are interleaved. The SuperPanel is processed with an inner postorder traversal. Each time a leaf of the SuperPanel is processed, it is loaded in memory along with the nodes on the path up to the root of the SuperPanel (partial step (i)). When the node processed is the parent of an *outer subtree*, nodes within the *outer subtree* update the SuperPanel (partial step (ii)). This update step consists in loading the nodes of the subtree (possibly block by block) into memory, updating ancestor nodes within the SuperPanel and releasing the updating nodes from memory (they do not have to be written back to disk since they have not been modified). The currently processed node is then factored, updates its ancestors within the SuperPanel, is written to disk and released from memory (partial step (iii)). We illustrate this process on a sample tree: we consider the elimination tree of Figure 6.1 as a subgraph of a larger elimination tree on top of several subtrees, as in Figure 6.5. The subtrees have already been factored (*left-looking* outer scheme) and are now on disk (a `W1/RO` inner approach was applied in the subtrees). A `W1/RO` *right-looking* factorization is performed within the SuperPanel. The SuperPanel is processed in postorder. Leaf node 1 is loaded in memory along with its root-to-leaf path (nodes 3, 7 and 15). The nodes of subtree T_1 are loaded in memory, block by block. Each time such a block is in core, it updates nodes $\{1, 3, 7, 15\}$. When all the blocks of the subtree have performed their update on the current root-to-leaf path of the SuperPanel, node 1 is factored and updates in turn its ancestors $\{3, 7, 15\}$. Figure 6.5(a) illustrates this step. It is then written to disk and freed from core memory. Node 2 is then loaded in memory (see Figure 6.5(b)) and the same process is applied with Subtree T_2. And so on. For instance, when node 9 is loaded in memory (see Figure 6.5(c)) (nodes 1 to 8 are on disk), subtrees T_3 and T_4 perform their out-of-core updates onto nodes 9, 10, 14, and 15. All in all, the subtrees have been read exactly once. Property 6.4 generalizes this result.



(a) Processing node 1 (b) Processing node 2 (c) Processing node 9 (d) Processing node 14

Figure 6.5: Different stages of the process of a SuperPanel with an out-of-core *left-looking/right-looking* factorization.

Property 6.4. *During the treatment of a given SuperPanel with an outer* left-looking *scheme, the* outer subtrees *have to be read exactly once.*

Left-looking approach inside SuperPanels (`LL-LL`)

We consider now the case where a SuperPanel is itself processed with a (`W1/RO`) *left-looking* scheme. The overall algorithm thus corresponds to a *left-looking/left-looking* (`LL-LL`) approach. In this case, steps (i), (ii) and (iii) do not require to be interleaved anymore. Indeed, after step (i), all the nodes of the SuperPanel are in core and can thus be updated by a node in any subtree of the SuperPanel (ii). When all the nodes of the SuperPanel have been updated by the nodes in the *outer subtrees*, the inner factorization is performed and finally the SuperPanel is written to disk (iii). We compare this approach to the *left-looking/right-looking* one on the sample tree of Figure 6.5. Because all the nodes of a SuperPanel have to simultaneously fit in core (Proposition 6.3), if the available memory only allows for the storage of 4 nodes, the subgraph considered in Figure 6.6(a) now has to be partitioned into five SuperPanels (whereas it could be held in a single SuperPanel with a *left-looking/right-looking* approach). We number them according to their postorder traversal. These five SuperPanels are processed with a *left-looking* approach between themselves. SuperPanel 1 is updated by subtrees T_1 and T_2 (see Figure 6.6(b)). SuperPanel 2 is then processed as in the `W1/RO` case and thus does not induce *I/O* (it does not have out-of-core subtrees). SuperPanel 3 is updated by subtrees T_3 and T_4 (see Figure 6.6(c)). SuperPanel 4 does not induce *I/O*. Finally SuperPanel 5 is processed, receiving out-of-core contributions of all the *outer subtrees* (that thus have to be read a second time) as well as the ones of SuperPanels 1 to 4 (see Figure 6.6(d)). All

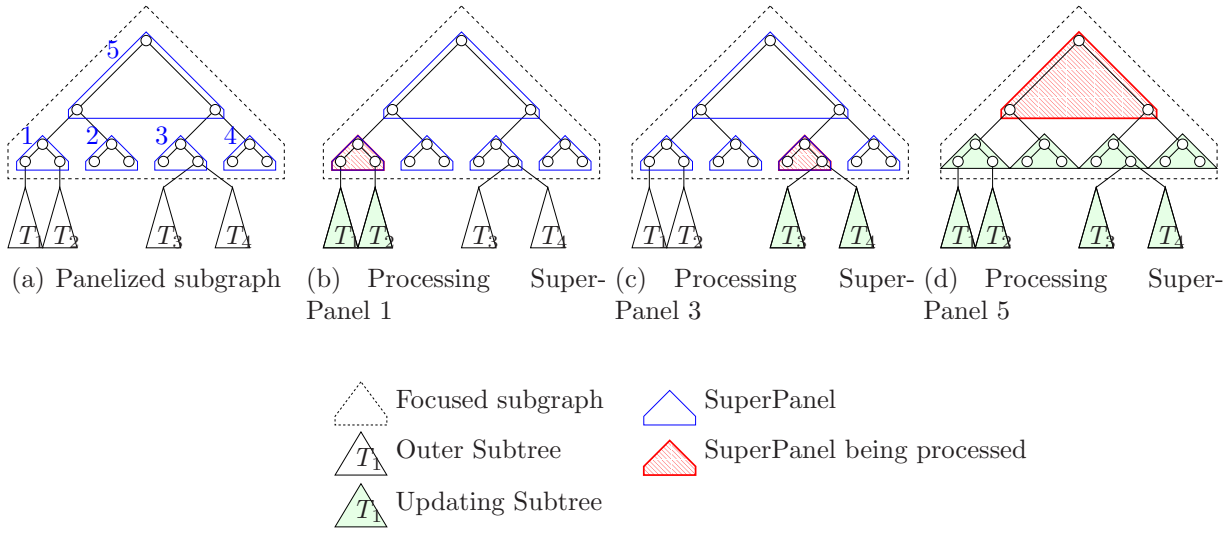


Figure 6.6: Different stages of the process of a subgraph (corresponding to five different LL-LL SuperPanels) with an out-of-core *left-looking/left-looking* factorization.

in all, to process the same subgraph, the *left-looking/left-looking* method induces twice as much *I/O* as the *left-looking/right-looking* one (bad scalability) plus some *I/O* inside the subgraph (bad local behaviour) - from SuperPanels 1, 2, 3 and 4 to SuperPanel 5. Property 6.5 states this result.

Property 6.5. *The out-of-core* left-looking/left-looking *factorization induces at least as much* I/O *as the out-of-core* left-looking/right-looking *factorization.*

Spatial locality of disk accesses - outer left-looking approaches

We now show that the disk accesses related to the treatment of each SuperPanel can be performed sequentially in the context of an outer *left-looking* approach. Indeed, SuperPanels are processed one after the other according to a postorder traversal between SuperPanels. After a SuperPanel has been processed, it is written to disk. Therefore, SuperPanels are written in the same order as they are processed. Let us consider a given SuperPanel. With a postorder traversal between SuperPanels, its *outer subtrees* are constituted of the most recently processed SuperPanels; their corresponding supernodes are thus the most recently written data. Therefore, they represent contiguous data on disk and can thus be accessed sequentially. Under Assumption 6.3, those data are exactly the ones that have to update the current SuperPanel.

In a nutshell, with a *left-looking* approach between SuperPanels, an inner `W1/R0` *left-looking* factorization of the SuperPanels is likely to induce a large amount of *I/O*. On the contrary, the (`W1/R0`) *right-looking* inner factorization is natural to apply in this case and induces a limited amount of *I/O*. Moreover, in both cases, the disk accesses can be performed sequentially.

6.3.3 Comparison

Outer method	Inner method	Size of SuperPanels	Volume of *I/O*	Spatial locality of disk accesses
`RL`	`RL`	Large	Bad scalability	No obvious locality
	`LL`	Small	Bad local behaviour	
`LL`	`RL`	Large	Good	Sequential (for each SuperPanel)
	`LL`	Small	Bad local behaviour and scalability	

Table 6.1: Comparison of different out-of-core factorization schemes.

The main characteristics of the different out-of-core factorization schemes are summarized in Table 6.1. It appears that hybrid methods (`RL-LL` and `LL-RL`) induce a more scalable volume of *I/O* than methods which use the same kernel for the inner and outer factorizations (`RL-RL` and `LL-LL`). A natural question is thus the following one: *are the* `RL-LL` *and* `LL-RL` *methods equivalent ?* The answer is *no* for at least three reasons:

- the `LL-RL` allows for large SuperPanels (since it is based on an inner `W1/R0` *right-looking* method);
- the `LL-RL` method is a *Write Once / Read Many* scheme (`W1/RM`) scheme whereas the `RL-LL` one follows a *Write Many / Read Many* scheme (`WM/RM`);
- only the `LL-RL` method provides a natural spatial locality for disk accesses.

These remarks in turn lead to wonder whether `LL-RL` *induces less* I/O *than* `RL-LL`*?* Again the answer is *no* and Figure 6.7 provides a counterexample. We consider a subtree composed of two supernodes: a child node of size 100 and a parent node of size 10. We assume that the amount of available memory is equal to $M_0 = 100$, which thus represents the maximum size of a SuperPanel. Therefore, the child node occupies all the memory and a SuperPanel can include at most one node. With a `RL-LL` method, the child node is loaded in memory and factored. The parent node is then updated. It is loaded block by block in memory. Each time a block is loaded, it is updated by the child node and written to disk. When all the blocks have been updated, the child node is written to disk. The parent node is read back from disk, factored and written back to disk. This makes an *I/O* volume of 10 units of read and 10 units of write. On the contrary, with

a LL−RL method, the child node is loaded in memory, factored and directly written to disk. Then, the parent node is loaded in memory. The child node is read back block by block to update the parent node. When all the updates have been performed, the parent node is factored and written to disk. In this case, an amount of 100 units of I/O (read operations) has been performed.

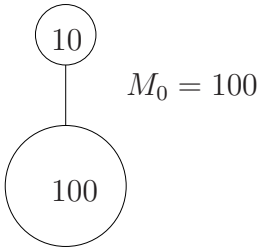


Figure 6.7: $V^{I/O}(\texttt{RL} - \texttt{LL}) = 20$. $V^{I/O}(\texttt{LL} - \texttt{RL}) = 100$.

However, in practice, the size of a column is small compared to the size of a SuperPanel. For instance, five thousands columns, such that each one contains one hundred thousands nonzero values, would fit in 4 GB of core memory. Furthermore the number of columns that a supernode include can be limited by splitting large supernodes (this is what we do in practice as we will see in Section 6.5). Therefore, we can do the following assumption:

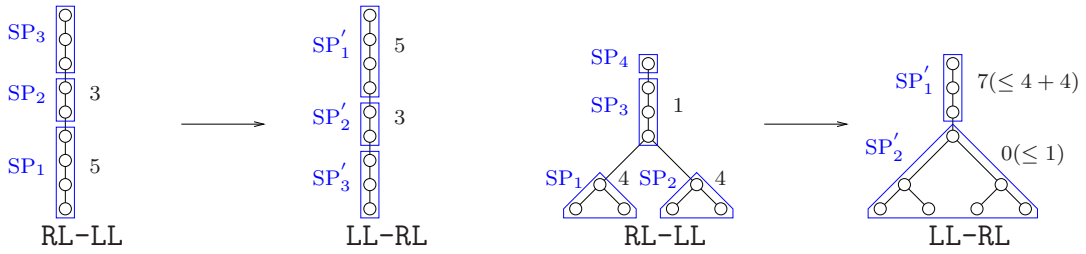
Assumption 6.4. *The size of a supernode is small compared to the size of a SuperPanel.*

With this assumption, a good approximation to compute the I/O volume, consists in considering that a node cannot be split into two different SuperPanels. Indeed, in the case of a SuperPanel of 5000 nodes of the same size, this approximation would impact the I/O volume related to the SuperPanel with a maximum relative error equal to 1/5000. An equivalent approximation consists in counting the number of nodes rather than summing the sizes of the supernodes, even if it means that a supernode represents as many nodes as it has nonzero values. With this assumption, we have the following result:

Property 6.6. LL−RL *induces at most half the* I/O *volume as* RL−LL.

Proof. Rather than a formal proof, we give the intuition and explain it on two examples. The idea is that, if we are given an elimination tree with a RL−LL partition, we can reverse the partition to form a LL−RL partition which decreases the I/O volume by at least a factor 2. Recall that we do not count the *I/O's* related to the first write (see Assumption 6.2) of a SuperPanel. We do not count either the *I/O's* when a panel is loaded in memory for the first time since that only involves parts of the initial matrix (see the related discussion at the beginning of Section 6.3.2). Both cases will be mentioned below as "not counted".

We first consider an elimination tree which is a chain and a RL−LL partition of it as illustrated in Figure 6.8(a)(left). The chain is composed of 9 nodes. From bottom to



(a) Chain. (b) General elimination tree.

Figure 6.8: Reversing a RL-LL partition into a LL-RL partition. Numerical values represent the number of nodes that are written due to a SuperPanel.

top, the three SuperPanels, SP_1, SP_2 and SP_3, are respectively composed of 4, 2 and 3 nodes. With a RL-LL factorization, SP_1 is factored first. It has to update SP_2 and SP_3; therefore those SuperPanels are loaded in memory (block by block), updated, and written to disk. This requires writing 5 nodes (the corresponding read operation are not counted, see above). SP_1 is written to disk (not counted) and released from memory. SP_2 is read back from disk - 2 nodes are read -, loaded in memory and factored. SP_2 has to update SP_3. To do so, SP_3 is read (block by block) and written back. This step requires reading and writing 3 more nodes. SP_2 is written to disk and released. SP_3 is loaded in memory - 3 nodes are read -, factored and written to disk (not counted). Note that, though the first read operation on SP_3 occurs later, it is due to the fact that SP_1 initially had updated it. Therefore, that read operation is related to the process of SP_1. With this reasoning, one can see that SP_1 requires to read and write 5 nodes and that SP_2 requires reading and writing 3 nodes: the number of read and written nodes due to a SuperPanel is equal to the number of its *outer ancestors*. All in all, the total *I/O* volume is equal to $V^{I/O}(\texttt{RL} - \texttt{LL}) = 16$ (8 for read operations and 8 for write operations). On the other hand, we can reverse the partition to form a LL-RL partition as illustrated in Figure 6.8(a)(right). With SP_1, we associate $SP_1^{'}$ with the same number of nodes, but which includes the nodes of the top of the chain this time. From top to bottom, we then associate $SP_2^{'}$ with SP_2, with the same number of nodes, and $SP_3^{'}$ with SP_3, with the same number of nodes. The point is that the number of read operations implied by $SP_i^{'}$ $(1 \leq i \leq 3)$ is equal to the number of read and write operations implied by SP_i. Indeed, with a LL-RL factorization, $SP_3^{'}$ is processed first. It is loaded in memory (not counted), factored, written to disk (not counted) and released from memory. $SP_2^{'}$ is then loaded in memory (not counted). It is updated by $SP_3^{'}$ which requires to be read back from disk (block by block). A number of 3 nodes are read. $SP_2^{'}$ is factored, written to disk (not counted) and released from memory. $SP_1^{'}$ is loaded in memory (not counted) and has to be updated by $SP_3^{'}$ and $SP_2^{'}$. Both these SuperPanels are on disk and need to be read back (block by block), which represents a read of 5 nodes. When all the updates are performed, $SP_1^{'}$ is factored and written to disk (not counted). One can see that the number of read operations due to a SuperPanel is equal to the number of nodes in its *outer subtrees*. All in all, 8 nodes have been read: $V^{I/O}(\texttt{LL} - \texttt{RL}) = 8$.

We now consider a more general elimination tree and a RL-LL partition of it, as

illustrated in Figure 6.8(b)(left). SP_1 requires reading and writing 4 nodes (number of *outer ancestors*), and SP_2 also. SP_3 requires reading and writing 1 node (it has only one *outer ancestor*). All in all, $V^{I/O}(\texttt{RL} - \texttt{LL}) = 18$ (9 for read operations and 9 for write operations). Let us reverse the partition into a `LL-RL` partition as illustrated in Figure 6.8(b)(right). We associate with both SP_1 and SP_2 a single SuperPanel $SP_1^{'}$. The number of nodes (7) in the *outer subtree* of $SP_1^{'}$ is lower than or equal to the sum of the number of *outer ancestors* of SP_1 and SP_2 $(4 + 4)$. Therefore, the number of read operations (7) due to $SP_1^{'}$ is lower than or equal to the number of read and write operations of the `RL-LL` approach due to both SP_1 and SP_2 ($4 + 4$ read operations, $4 + 4$ write operations). We associate $SP_2^{'}$ with SP_3. $SP_2^{'}$ has a no *outer subtree* and therefore will not induce *I/O* with a `LL-RL` scheme. All in all, $V^{I/O}(\texttt{LL} - \texttt{RL}) = 7$. Note that we could reverse the partition because $SP_2^{'}$ is processed with an inner `W1/RO` *right-looking* scheme: therefore, only 3 (the height of the SuperPanel) nodes need to fit together in memory. ■

In the following, we discard *right-looking* outer factorizations and focus on `LL-RL` and `LL-LL` methods.

6.4 I/O volume reduction partitioning algorithms

In the previous section, we have shown that *left-looking* methods (especially the `LL-RL` method) are the most suitable when aiming at limiting the *I/O* volume in our context. However, the *I/O* volume depends on the partition of the elimination tree into SuperPanels. In this section, we rely on assumptions 6.1, 6.2, 6.3 and have the following objective:

Objective 6.3. *Given an amount of core memory M_0, our purpose is to find the partition of the elimination tree that minimizes the* I/O *volume.*

We first formalize the expression of the *I/O* volume with a *left-looking* outer approach in Section 6.4.1. We then minimize the *I/O* volume when the elimination tree is a chain in Section 6.4.2. Next, we present an optimum partitioning algorithm for the `LL-RL` method in Section 6.4.3. We finally study the `LL-LL` method in Section 6.4.4.

6.4.1 I/O volume induced with a left-looking approach between SuperPanels

We recall that, according to Assumption 6.2, all the nodes are at least written to disk once. We do not count the *I/O* volume due to the first write of a node but only the extra-*I/O* volume with respect to this amount. With a *left-looking* outer method, those *I/O's* are only due to read operations. We consider a node $\mathcal{N}$ and we note $|\mathcal{N}|$ its number of nonzero values. We denote by $\#\text{SP}(\mathcal{N})$ the number of SuperPanels above $\mathcal{N}$. For each of those SuperPanels, $\mathcal{N}$ has to be read once, as we have seen when discussing Figure 6.8(a)(right).

The volume of I/O related to node $\mathcal{N}$ is thus equal to: $V_{\mathcal{N}}^{I/O} = |\mathcal{N}| \times \#\mathrm{SP}(\mathcal{N})$. This amount represents the number of nonzero entries of node $\mathcal{N}$ that have to be read from disk during the whole factorization. The total I/O volume is obtained by summing those amounts for each node in the elimination tree:

$$V^{I/O} = \sum_{\mathcal{N} \in \mathtt{etree}} V^{I/O}(\mathcal{N}) = \sum_{\mathcal{N} \in \mathtt{etree}} |\mathcal{N}| \times \#\mathrm{SP}(\mathcal{N}). \tag{6.1}$$

6.4.2 Partitioning a chain

We assume that the elimination tree is a chain. Let us consider the following greedy partitioning algorithm. Initially, we consider a partition $\mathcal{P}$ composed of a single empty SuperPanel SPcurrent. We traverse the chain from top to bottom. When traversing a node $\mathcal{N}$, we include it in the current SuperPanel SPcurrent if it can fit in core memory together with the other nodes of SPcurrent. Otherwise, $\mathcal{N}$ is put into a new SuperPanel which is added to the partition and becomes the current SuperPanel. Algorithm 6.1 describes this partitioning algorithm.

Input: A chain representing the elimination tree; an amount M_0 of core memory.
Output: A partition $\mathcal{P}$ of the chain into SuperPanels.
`% Init: The partition` $\mathcal{P}$ `is composed of a single empty SuperPanel SPcurrent (thus of size 0).`
SPcurrent $\leftarrow \emptyset$;
SizeSPcurrent $\leftarrow 0$;
$\mathcal{P} \leftarrow \{$SPcurrent$\}$;
foreach *node* $\mathcal{N}$ *in the chain, from top to bottom* **do**
  if $|\mathcal{N}| +$ *SizeSPcurrent* $\leq M_0$ **then**
    SPcurrent $\leftarrow$ SPcurrent $\cup \{\mathcal{N}\}$;
    SizeSPcurrent $\leftarrow$ SizeSPcurrent $+ |\mathcal{N}|$;
  else
    SPcurrent $\leftarrow \{\mathcal{N}\}$;
    SizeSPcurrent $\leftarrow |\mathcal{N}|$;
    $\mathcal{P} \leftarrow \mathcal{P} \cup \{$SPcurrent$\}$;

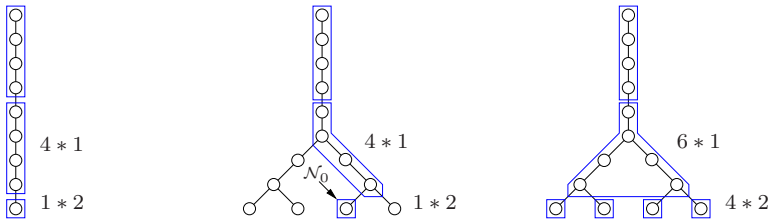
Algorithm 6.1: Partitioning of a chain with an outer *left-looking* method.

We consider a node $\mathcal{N}$. One may notice that the number of SuperPanels on top of $\mathcal{N}$, $\#\mathrm{SP}(\mathcal{N})$, is minimized when Algorithm 6.1 is applied. As furthermore Algorithm 6.1 is independent of the node considered, it minimizes Formula (6.1).

Property 6.7. *Algorithm 6.1 finds the partition of a chain which minimizes the volume of* I/O.

We illustrate our algorithm on the chain of Figure 6.8(a)(right). We assume that 4 nodes can fit together in core and we apply Algorithm 6.1. Initially, the partition

contains a single empty SuperPanel. We traverse the chain from top to bottom. The first four nodes are added to the current SuperPanel. The fifth node cannot fit in the current SuperPanel. Therefore, a second SuperPanel is created, which will fit the four next nodes. A third SuperPanel has to be created to fit the ninth node. Figure 6.9(a) represents the resulting partition. With such a partition, the I/O volume (read) is equal to $V^{I/O} = 5$: the node at the bottom will have to be read twice and the four nodes in the second SuperPanel will have to be read once.



(a) Chain (b) `LL-RL` (partial partition) (c) `LL-RL`

Figure 6.9: Partitioning of an elimination tree into SuperPanels. Numerical values represent the volume of I/O (read) due to the nodes of a SuperPanel. The memory is equal to $M_0 = 4$ (nodes).

6.4.3 Left-looking/right-looking approach

We now consider a general elimination tree processed with a `LL-RL` method. Let $\mathcal{N}_0$ be a node of the elimination tree. The number of SuperPanels on top of $\mathcal{N}_0$, $\#\mathrm{SP}(\mathcal{N}_0)$, is minimized when Algorithm 6.1 is applied to the chain composed of the nodes above $\mathcal{N}_0$. For instance, we can apply this mechanism to node $\mathcal{N}_0$ of Figure 6.9(b) with a core memory of $M_0 = 4$ (nodes). Such a construction, of course, depends on the node considered ($\mathcal{N}_0$ in our example). Intuitively, we are building SuperPanels in priority along the path from the root of the elimination tree down to node $\mathcal{N}_0$ and this path is favoured. However, with an inner `W1/R0` *right-looking* method, we recall that not all the nodes of a SuperPanel need to fit together in memory but only the nodes along a single root-to-leaf path (see Figure 6.5). Therefore, with a top-down partitioning, a node can be added to the SuperPanel of its parent if and only if it can fit together with the nodes on the path from itself up to the root of the SuperPanel. In particular, this decision is independent of the possible inclusion of its siblings.

Algorithm 6.1 can thus be adapted as we now explain. Initially, we consider an empty partition, $\mathcal{P}$. We traverse the elimination tree in reverse postorder (top-down traversal). A node $\mathcal{N}$ is added to SuperPanel SPparent to which belongs its parent if it can fit in core together with the nodes along the path from $\mathcal{N}$ to the root of SPparent. Otherwise, $\mathcal{N}$ is included in a new SuperPanel and this SuperPanel is added to the partition. Algorithm 6.2 describes this partitioning algorithm. Note that we use an explicit mapping of the nodes to their SuperPanel because we need to access to the mapping of a parent node.

Input: The parent structure (parent() array) of the elimination tree; an amount M_0 of core memory.
Output: The mapping of the nodes (mapping() array) of the elimination tree to the index of their SuperPanel.
Data: mempath() array: mempath($\mathcal{N}$) is the amount of memory required to fit the nodes along the path from $\mathcal{N}$ up to the root of the SuperPanel containing $\mathcal{N}$.

```
% Init: The root node N_root of the elimination tree is mapped to a
  SuperPanel of index equal to 0. The current number of SuperPanels,
  numberSP, is initially equal to 1.
```

mapping($\mathcal{N}_{root}$) $\leftarrow 0$;
mempath($\mathcal{N}_{root}$) $\leftarrow |\mathcal{N}_{root}|$;
numberSP $\leftarrow 1$;
foreach *node* $\mathcal{N}$ *different from the root in reverse postorder (top-down traversal)* **do**
    myparent $\leftarrow$ parent($\mathcal{N}$);
    if $|\mathcal{N}| + mempath(myparent) \leq M_0$ **then**
        mapping($\mathcal{N}$) $\leftarrow$ mapping(myparent);
        mempath($\mathcal{N}$) $\leftarrow |\mathcal{N}|$ + mempath(myparent);
    else
        mapping($\mathcal{N}$) $\leftarrow$ numberSP;
        mempath($\mathcal{N}$) $\leftarrow |\mathcal{N}|$;
        numberSP $\leftarrow$ numberSP + 1;

Algorithm 6.2: Partitioning of an elimination tree into SuperPanels with a `LL-RL` method.

If we consider any node $\mathcal{N}$ of the elimination tree, the number of SuperPanels on top of $\mathcal{N}$, $\#\mathrm{SP}(\mathcal{N})$, is equal to the number of nodes that we would have got by applying Algorithm 6.1 to the chain from the root node of the elimination tree down to $\mathcal{N}$. Therefore, Algorithm 6.2 minimizes $\#\mathrm{SP}(\mathcal{N})$. As furthermore this latter algorithm is independent of node $\mathcal{N}$, it minimizes Formula (6.1).

Property 6.8. *Algorithm 6.2 provides the partition of a general elimination tree which minimizes the volume of* I/O *in a* left-looking/right-looking *context.*

Figure 6.9(c) illustrates Algorithm 6.2 when assuming that 4 nodes can fit together in core. With the resulting partition, the *I/O* volume (read) is equal to $V^{I/O} = 10$: the node at the bottom will have to be read twice and the nodes in the intermediate SuperPanel will have to be read once.

6.4.4 Left-looking/left-looking approach

Minimizing the I/O volume with a left-looking/left-looking method is NP-complete

With an inner `W1/R0` *left-looking* method, all the nodes of a SuperPanel have to fit together in memory (remember Figure 6.6). Therefore, the inclusion of a node into a SuperPanel depends on the inclusion of its siblings. We may have to choose which siblings are added to the SuperPanel. Such a choice may be difficult if one aims at minimizing the *I/O* volume. We prove that the associated decision problem is NP-complete in the special case of an elimination tree of height equal to 1 (and the general problem is thus NP-complete too). In this case, *I/O's* are exactly performed when the root node is updated by nodes that are not in its SuperPanel. If we note $\mathcal{S}$ the subset of children that are in the SuperPanel of the root node, the amount of data read is then equal to $V^{I/O} = \sum_{i \notin \mathcal{S}} s_i$. We name Panelize-LLLL-H1-Dec this decision problem where H1 denotes a height equal to 1.

Problem 6.1 (Panelize-LLLL-H1-Dec)**.** *We consider an elimination tree composed of a root node and n children numbered from 1 to n. We are given a core memory of size M_0 (the number of nonzero values that can fit in a SuperPanel). Node i ($1 \leq i \leq n$) has a nonnegative size s_i (the number of nonzero values). Does there exist a subset $\mathcal{S}$ of children such that all these children can fit in the same SuperPanel as the parent ($s_0 + \sum_{i \in \mathcal{S}} s_i \leq M_0$) and that the subsequent* I/O *volume is lower than or equal to $V_{I/O}^{target}$ ($\sum_{i \notin \mathcal{S}} s_i \leq V_{I/O}^{target}$) ?*

This problem can be reduced in polynomial time to the Subset-Sum decision problem formulated as follows:

Problem 6.2 (Subset-Sum-Dec)**.** *We have n items $(1, \ldots, n)$. Each item i has a nonnegative size x_i. Is there a subset $\mathcal{S}$ of these items such that their sum is at least equal to a positive integer L and at most equal to a positive integer U ($L \leq \sum_{i \in \mathcal{S}} x_i \leq U$)?*

Note that Subset-Sum is usually formulated in the special case where $L = U$ for which it remains NP-complete [35].

Property 6.9. *Panelize-LLLL-H1-Dec is NP-complete.*

Proof. First, Panelize-LLLL-H1-Dec belongs to NP. Let us assume we are given a subset $\mathcal{S}$ of the children such that all these children can fit in the same SuperPanel as the parent and that the subsequent I/O volume is lower than or equal to $V_{I/O}^{target}$. The amounts $s_0 + \sum_{i \in \mathcal{S}} s_i$ and $\sum_{i \notin \mathcal{S}} s_i$ can be computed in time linear with the size of the instance. Therefore, one may verify that $\mathcal{S}$ is a solution in linear time too.

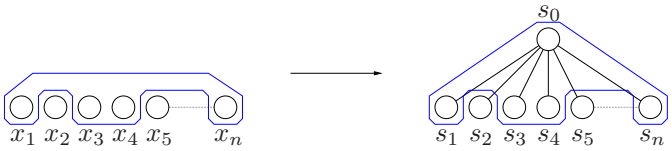


Figure 6.10: Instancing Panelize-LLLL-H1-Dec (right) from Subset-Sum-Dec (left). The surrounded area represents a possible solution.

For proving the NP-completeness of Panelize-LLLL-H1-Dec, we show that Subset-Sum-Dec can be polynomially reduced to Panelize-LLLL-H1-Dec. We consider an arbitrary instance I_1 of Subset-Sum-Dec composed of n items $(1, \ldots, n)$ such that each item i has a nonnegative size x_i. We are also given a lower bound L and an upper bound U. We build an instance I_2 of Panelize-LLLL-H1-Dec as follows. We consider an elimination tree composed of a root node and n children. The root node has a size s_0 equal to any positive value (for instance $s_0 = 100$). The children are of size $s_i = x_i$ $(1 \leq i \leq n)$. The amount of core memory is equal to $M_0 = s_0 + U$. We define an upper bound on the I/O volume equal to $V_{I/O}^{target} = \sum_{i=1}^{n} s_i - L$. The construction of I_2 is polynomial (and even linear) in the size of I_1. We now show that any subset $\mathcal{S}$ of $\{1; ...; n\}$ is a solution to I_1 if and only if it is a solution to I_2 (which implies that I_1 has a solution if and only if I_2 has a solution). Let $\mathcal{S}$ be a subset of $\{1; ...; n\}$. The assertion $L \leq \sum_{i \in \mathcal{S}} x_i \leq U$ is equivalent to $\sum_{i=1}^{n} s_i - V_{I/O}^{target} \leq \sum_{i \in \mathcal{S}} s_i \leq M_0 - s_0$ and thus to: $\sum_{i \notin \mathcal{S}} x_i \leq V_{I/O}^{target}$ and $s_0 + \sum_{i \in \mathcal{S}} x_i \leq M_0$. ■

Approximation

The problem can again be simplified with the approximation related to Assumption 6.4. As explained in Section 6.3.3, under this assumption, a good approximation indeed consists in counting the number of nodes rather than summing their sizes, even if it means that a supernode represents as many nodes as it has nonzero values. With this approximation, Panelize-LLLL-H1-Dec can be solved in time linear with the size of the instance. The volume of I/O is minimized with any subset of child nodes of cardinality equal to $M_0 - s_0$ that form a connected subgraph when put together with the root node. Indeed, the number of nodes read is then equal to the number of children that are not part of the root SuperPanel: $V^{I/O} = \sum_{i=1}^{n} s_i - M_0 + s_0$. Figure 6.11 illustrates this result.

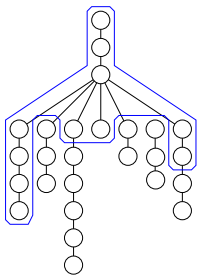
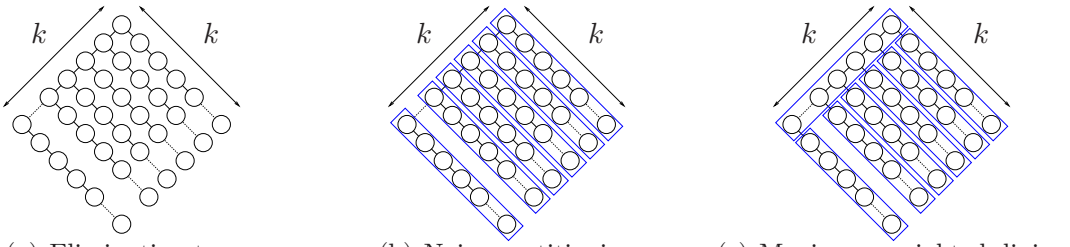


Figure 6.11: Instance of Panelize-LLLL-H1 under Assumption 6.4 with $n = 7$, $(s_i)_{i=0,1,\dots,n} = (3, 4, 3, 6, 1, 2, 3, 4)$ and $M_0 = 11$. Any root SuperPanel that holds 8 child nodes (for instance the one represented by the surrounded area) minimizes the *I/O* volume: 15 nodes have to be read.

Heuristic

However, Panelize-LLLL-H1 is related to an elimination tree of height equal to 1. We now explain that the *I/O* minimization problem is not immediately solved when considering a general elimination tree, even with Assumption 6.4. We consider the elimination tree of Figure 6.12(a) and the related naive partition of Figure 6.12(b). With this partition, the *I/O* volume is equal to: $V^{I/O}(k) = \sum_{i=1}^{k} k(i-1) = k^2(k-1)/2 = \Theta(k^3)$. Figure 6.12(b) provides a better partition which leads to a volume of *I/O* equal to: $V^{I/O}(k) = (k-1)k = \Theta(k^2)$. The impact of the partitioning algorithm is thus critical when aiming at limiting the *I/O* volume.



(a) Elimination tree (b) Naive partitioning (c) Maximum weighted diving

Figure 6.12: Different top-down LL-LL partitions of the same elimination tree with $M_0 = k$ nodes.

Both those partitions can be obtained with a top-down partitioning algorithm as we now explain. Initially, the partition is formed of a single SuperPanel under construction. Then, one of its children can be added to the SuperPanel. In the naive case (Figure 6.12(b)), its right child is (arbitrarily) chosen. On the contrary, the *maximum weighted diving* algorithm (Figure 6.12(c)) chooses the right child because it has itself more descendants. Therefore, by selecting that child, we are likely to reduce the number of ancestor SuperPanels of many nodes and thus the *I/O* volume. Whereas the naive partitioning algorithm chooses an arbitrary node at each step, the maximum weighted

diving algorithm extends the current SuperPanel with the node which has the largest number of children (see Figure 6.13).

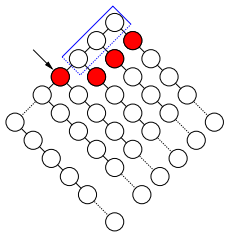


Figure 6.13: On-going top-down LL-LL partitioning of the elimination tree of Figure 6.12. The surrounded area represents the first SuperPanel under construction. At this step, the SuperPanel can be extended with one of the four colored nodes. The arrow points to the node that will be selected with the *maximum weighted diving* algorithm since it has more descendants than its competitors.

We now summarize our results on LL-LL partitioning. From Section 6.4.2, we know that top-down traversals should be applied to outer *left-looking* partitioning algorithms. We have then exhibited that on a given level of the elimination tree, the choice of the children nodes that are added to the parent SuperPanel is difficult (the related decision problem is NP-complete). However, we have noticed that a random greedy choice actually leads to a quasi-optimum *I/O* volume on a tree of height one. On general elimination trees, we have exhibited that the good question is: "how to dive". We have shown that a "bad" diving criterion can lead to an *I/O* volume arbitrarily larger than the one obtained with a "good" diving. This consideration has led us to propose a heuristic which consists in greedily selecting the node which has the largest number of descendants. In practice, we have to take into consideration the size of the nodes. We define the size of a subtree as the sum of the sizes of the nodes that compose the subtree. The selected node in our greedy algorithm thus becomes the one which is the root of the subtree of largest size. The heuristic is composed of two steps. A first step computes the weight of the nodes with a bottom-up traversal of the elimination tree: the weight of a node is the size of the subtree rooted at it. The weight() array is then used in the second step to perform a greedy maximum weighted top-down traversal which builds the SuperPanels. Algorithm 6.3 describes this second step. The first step is clearly linear with the number n of nodes. The second step is bounded by a number of operations proportional to $n \log(n)$ operations: n nodes are inserted into an ordered list of size bounded by n.

6.5 Preliminary prototype implementation

This study led us to develop a prototype implementation for an out-of-core extension of SuperLU. The purpose was to confirm that the in-core kernel of computation of SuperLU could effectively be extended to the models discussed in this chapter. Because SuperLU is based on a *left-looking* method, we have implemented a *left-looking/left-looking* method.

Input: The children structure (children() array) of the elimination tree (retrieves the set of child nodes of a given node); the weight (weight() array) of the nodes of the elimination tree; an amount M_0 of core memory.
Output: A partition $\mathcal{P}$ of the elimination tree into SuperPanels.
Data: A stack holds the roots of the SuperPanels not traversed yet. This stack is managed with the usual push (`push_root()`) and pop (`pop_root()`) operations. The `is_roots_stack_empty()` operation checks whether the stack is empty.
Data: An ordered list holds the nodes that are candidate to be added to the current SuperPanel. The nodes are maintained in decreasing order of their respective weight(). The `insert_cands()` operation inserts a set of nodes into that list. The first element of the list (the one with the maximum weight) can be retrieved with the `extract_first_cand()` operation. The `is_candidates_list_empty()` operation checks whether the list is empty.

```
% Init: The partition P is initially empty and the root node N_root
  of the elimination tree is pushed into the stack of the roots of
  future SuperPanels.
P ← ∅;
push_root(N_root) ;
while ! is_roots_stack_empty() do
    insert_cand(pop_root()) ;
    SPcurrent ← ∅;
    SizeSPcurrent ← 0;
    P ← P ∪ {SPcurrent};
    while ! is_candidates_list_empty() do
        N ← extract_first_cand()
        if |N| + SizeSPcurrent ≤ M_0 then
            SPcurrent ← SPcurrent ∪ {N};
            SizeSPcurrent ← SizeSPcurrent + |N|;
            insert_cands(children(N))
        else
            push_root(N) ;
```

Algorithm 6.3: Partitioning of an elimination tree with a LL-LL method with a maximum weighted diving traversal.

This prototype does not perform actual *I/O's* but memory copies to a special area instead. The data access pattern of our prototype being the same as the one of an actual out-of-core *left-looking/left-looking* method, this allowed us to valid our approach. The prototype clearly constitutes a preliminary validation step before incorporating an *I/O* layer.

To increase the number of nodes that they can contain, SuperPanels only store nonzero values in a compressed structure. However, the columns of the SuperPanels need to be scattered into dense arrays when they are updated or factored (with respect to the in-core scheme of `SuperLU`). We explain in Section 6.5.1 how we have handled those scatter/gather operations. We then explain in Section 6.5.2 how we have adapted the symbolic factorization of `SuperLU` to our out-of-core context. Finally, Section 6.5.3 presents the numerical factorization step.

6.5.1 Reducing the scatter-gather overhead - HyperNodes

To exploit memory locality, `SuperLU` factors several (say w of them) columns at a time so that one updating supernode $(r : s)$ (from column r to column s) can be used to update as many of the w columns as possible. The authors refer to these w consecutive columns as a *panel*. The row structure of these columns may not be correlated in any fashion, and the boundaries between panels may be different from those between supernodes [26]. Two types of updates occur. If the supernode is not part of the panel, supernode-panel updates are performed. Otherwise (within the panel), supernode-column updates are performed (sup-col algorithm). A data structure called sparse accumulator (SPA for short) is used to scatter the updates and hold the w columns of the active panel. It consists of an n by w full array (where n is the order of the matrix) and allows random access to the entries of the active panel.

In the in-core version of `SuperLU`, the factorization of a panel is processed all at once: the columns corresponding to the panel are scattered into the SPA, all required updates are performed and finally the data in the SPA are gathered into storage for L and U. In an out-of-core context, the updating supernodes may have been written to disk and released from memory if they are not part of the same SuperPanel. Therefore, they have to be read back from disk to perform the updates. If they cannot fit altogether in core memory with the current SuperPanel, they have to be read block by block. Each time a block is loaded in memory, the corresponding updates are performed on the columns of the SuperPanel which have to be alternately scattered and gathered into the SPA. If the memory attributed for the updating blocks is small, the number of scatter/gather operations may become large and represent a critical overhead on the efficiency. We denote HyperNode (HN for short) a group of supernodes that are loaded together in memory to perform those updates. Note that the boundaries between HyperNodes may be different from those between SuperPanels and there is a balance to find between the memory allocated for the current SuperPanel and the one for the current HyperNode. The first one will impact the *I/O* volume (as discussed in sections 6.3 and 6.4) whereas the second one will impact the extra number of scatter/gather operations.

6.5.2 Symbolic factorization - out-of-core depth-first-search

The symbolic factorization is the process that determines the nonzero structure of the triangular factors L and U from the nonzero structure of the matrix A. In turn, this process determines which columns of L will update each column j of the factors: those columns r for which $u_{rj} \neq 0$. As already discussed, partial pivoting requires interleaving numerical and symbolic factorization.

In the in-core version of `SuperLU`, a symbolic factorization occurs for each (core) panel [26]. In an out-of-core context, a partial symbolic factorization of the panel has to be performed each time a HyperNode is loaded in memory. We have adapted the symbolic factorization so that it limits its search of updating supernodes to the current HyperNode. Once the partial symbolic factorization related to a given HyperNode is finished, the subscripts of the nonzeros values of the columns of the current panel are gathered. They will be reused to initiate the next partial symbolic factorization of the panel with the next HyperNode.

6.5.3 Out-of-core numerical factorization - left-looking/left-looking method

We have developed a *left-looking/left-looking* factorization based on `SuperLU` as follows. Initially, we apply Algorithm 6.3 to partition the elimination tree into SuperPanels. The numerical factorization then processes the SuperPanels, one after the other. When a SuperPanel is processed, its sparse compressed data-structure is loaded in memory. The SuperPanel has to receive contributions from *outer subtrees*. Because those *outer subtrees* may not altogether fit in-core, they are read block by block. Such a block (called HyperNode, see Section 6.5.1) is constituted of several supernodes. For each HyperNode, we investigate all the columns of the current SuperPanel, (core) panel by (core) panel. The current panel is scattered into a sparse accumulator (SPA); a partial symbolic factorization (see Section 6.5.2) is performed and the supernodes of the HyperNode that require to update some columns of the panel perform their update as in the in-core case. If the HyperNode overlaps with the SuperPanel, the inner factorization of the panel is performed. Otherwise, nonzero values of the sparse accumulator are gathered back into the compressed structure of the SuperPanel to make room in the SPA for the next panel. This method corresponds to Algorithm 6.4.

6.5.4 Preliminary validation

We have validated our prototype on several matrices of intermediate size of Table 1.2. Note that we have had to turn off some features of the in-core version of `SuperLU` [26] (relaxed supernodes, symmetric pruning). Also, we recall that the prototype does not explicitly perform the *I/O's* but does memory copies instead.

```
foreach SuperPanel SP in order do
    Push the (forecast) sparse nonzero structure of SP into memory;
    foreach HyperNode HN descendant of SP do
        Read HN from disk;
        for j = first_column(SP) to last_column(SP) step w do
            if HN is the first HyperNode processed for SP then
                Scatter j : j + w − 1 columns from A into the SPA ;
            else
                Scatter j : j + w − 1 columns from SP into the SPA ;
            Symbolic factor: determine which supernodes of HN will update any of
            A(:,j:j+w-1);
            foreach updating supernode (r : s) of HN ((r : s) < j) in topological
            order do
                Apply triangular solves to A(r:s,j:j+w-1) using L(r:s,r:s);
                foreach row block B in L(s+1:n,r:s) do
                    for jj = j to j + w − 1 do
                        Multiply B.U(r:s,jj), and scatter into SPA(:,jj);
            if HN overlaps SP then
                Inner factor: Apply the sup-col algorithm on columns and
                supernodes within the SPA ;
                Write the SPA to disk;
            else
                Gather the SPA into SP
        Release HN from memory;
    Release SP from memory;
```

Algorithm 6.4: *Out-of-core left-looking/left-looking* algorithm. A is the on-going updated matrix.

We have also implemented Algorithm 6.2. Clearly, that latter algorithm is specifically designed for a *left-looking/right-looking* method whereas we have implemented a *left-looking/left-looking* approach. However, the partition of the elimination tree into SuperPanels that it computes can be used with a *left-looking/left-looking* factorization if memory usage allows for it. This should allow us to compare the practical impact on *I/O* volume of *left-looking/left-looking* and *left-looking/right-looking* methods when top-down partitioning algorithms are applied ([63] presents such a comparison for bottom-up partitioning algorithms).

6.6 Conclusion

In this chapter, we have proposed models to study the problem of the minimization of the *I/O* volume in the context of supernodal methods. We could generalize to the sparse case - with our assumptions - a result known in the context of dense factorization: *left-looking* algorithms allow us to perform significantly less *I/O* than *right-looking* algorithms. To prove this result, we have considered an existing hybrid method (*left-looking/right-looking*) proposed and implemented in [41, 63] and compared it with other possible combinations of *left-looking* and *right-looking* methods. We have then addressed the problem of *I/O* minimization for two specific methods (*left-looking/right-looking* and *left-looking/left-looking*); we have exhibited an optimum algorithm for the first one (under our specific assumptions) and proposed a heuristic for the second one after showing that it is NP-complete.

We have had to rely on assumptions that are clearly strong. It would be interesting to pursue this study with weaker assumptions in order to have more general results. However, note that although our models are based on strong assumptions, the proposed algorithms apply to (and have been implemented for) the general case for which they remain interesting heuristics. In particular, it would be interesting to compare in practice the effects on the volume of *I/O* and on the efficiency of the factorization of top-down partitioning algorithms to bottom-up approaches that have been used in the past [63].

This study has led us to develop a prototype out-of-core extension of `SuperLU`. This prototype is based on a method (*left-looking/left-looking*) that had already been successfully implemented by different authors [41, 63]. However, our approach is novel in several respects. HyperNodes might help to limit the scatter/gather overhead due to the use of compressed SuperPanels. The out-of-core depth first-search algorithm extends in a natural way the symbolic factorization of `SuperLU`. The data access pattern of the *left-looking/left-looking* method has been adapted to respect the constraints of a complex in-core solver, `SuperLU`. Currently, our prototype only performs memory copies instead of *I/O's*. This has allowed us to validate all the proposed mechanisms. We plan to incorporate a robust *I/O* layer in the prototype in order to assess the efficiency of our approach.

Part II

A parallel out-of-core multifrontal solver

Chapter 7

Preliminary study

In Part I, we have proposed models and algorithms for sparse direct methods in the serial case. We have studied several models for different methods and presented their respective advantages and limits. In this part, we extend this work to the parallel case. However, instead of considering the large panel of models studied in the serial case, we focus on a particular method that we aim at studying as far as possible in order to process problems as large as possible. Indeed, we focus on the distributed memory multifrontal method. A distributed memory environment allows the use of a large amount of memory distributed among several processors. Contrary to the serial case, in this context, frontal matrices can be scattered over the processors and do not represent a major bottleneck for memory. This consideration and the fact that the multifrontal method can be an efficient parallel approach [11] motivate the development of an out-of-core parallel multifrontal solver in a distributed environment. We illustrate our study with a fully asynchronous parallel multifrontal solver for distributed memory architectures, `MUMPS` [9, 10].

As already discussed, in the multifrontal method, the factors produced during the factorization step are not re-used before the solution step. It is thus natural to write them on disk as in the serial case. In this chapter, we present a preliminary study which aims at evaluating by how much the core memory usage can be reduced by writing the factors to disk during the numerical factorization step. We first describe the parallel multifrontal method as it is implemented in `MUMPS`. We then explain how we have instrumented this software to perform our study and we present experimental results.

7.1 Parallel scheme for the multifrontal method

`MUMPS` is based on a parallel multifrontal method which applies to matrices with symmetric (or symmetrized) structure. As explained in Chapter 1, the symmetric structure allows the natural use of an assembly tree, obtained by merging nodes of the elimination tree whose corresponding columns belong to the same supernode [17]. We recall that in the multifrontal approach, the factorization of a matrix is done by performing a succession of partial factorizations of small dense matrices called *frontal matrices*, associated with

the nodes of the tree and that the tree is processed from bottom to top. Once a partial factorization is done, a *contribution block* is passed to the parent node. When factors are kept in core, the elimination algorithm uses three areas of storage that can overlap in time: one for the factors, one for the contribution blocks, and another one for the current frontal matrix [7]. Whereas a postorder traversal of the assembly tree is applied for serial executions and ensures that the contribution blocks can be accessed with a stack mechanism (as seen in Chapter 2, Section 2.1); for parallel executions, task scheduling does not allow a strict postorder and the management of the contribution blocks differs from a pure stack mechanism. During the tree traversal, the amount of storage required for the factors always grows whereas the amount of active storage required varies; when the partial factorization of a frontal matrix is performed, a contribution block is stacked, increasing the size of the stack; on the other hand, when the frontal matrix is formed and assembled, the contribution blocks of the children nodes are discarded and the size of the stack decreases. We still call *active storage* the area containing both the active frontal matrices [1] and the contribution blocks waiting to be assembled. The *active memory* is the part of the active storage held in core memory.

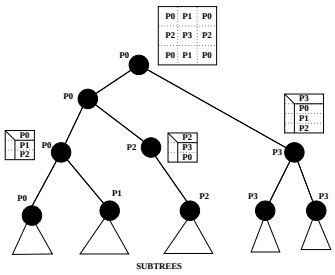


Figure 7.1: Example of the distribution of an assembly tree over four processors.

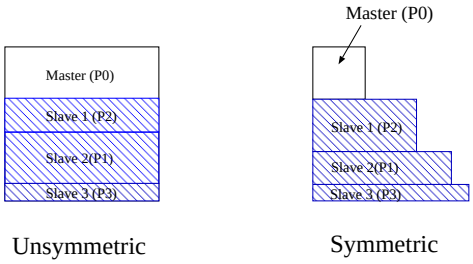


Figure 7.2: Distribution of the processors on a parallel node in the unsymmetric and symmetric cases.

From the parallel point of view, the parallel asynchronous multifrontal method as implemented in `MUMPS` uses a combination of static and dynamic scheduling approaches. Indeed, a first partial mapping is done statically (see [12]) to map some of the tasks to the processors. Then, for parallel tasks corresponding to large frontal matrices of the assembly tree, a *master task* corresponds to the elimination of the so-called fully summed rows, while dynamic decisions are done at runtime to select the *slave* processors in charge of updating the rest of the frontal matrix (see Figures 7.1 and 7.2). Once the slave processors are chosen, the elimination process uses a 1D pipelined factorization, in which the master repeatedly updates a block of rows, then sends it to the (slave) processors in charge of updating their share of the frontal matrix. Therefore, the fully summed part of the frontal matrix is only treated by one processor, the one responsible for the master task whereas the rest of the frontal matrix - and thus the associated memory - is scattered among several processors, each of them receiving a *slave task* to process. Note that those dynamic scheduling decisions are taken to balance workload, possibly under

1. In parallel, it may happen that more than one matrix is active on a given processor.

memory constraints [13]. In order to limit the amount of communication, the nodes at the bottom of the tree are statically merged into *subtrees* (see Figure 7.1), each of them being processed sequentially on a given processor. There are usually slightly more subtrees than processors. Finally, the root of the tree must be factored completely and is processed thanks to a static 2D cyclic factorization [18]. Because of the dynamic and asynchronous nature of the approach, it often happens that at a given time, a processor has been selected to work on slave tasks of two (or more) different nodes in the tree. The processor will then alternate between work corresponding to the different active slave tasks depending on the messages it receives. Note that these dynamic aspects make the behaviour of the application complex and difficult to forecast without performing actual executions.

7.2 Instrumentation

Because the code is dynamic and asynchronous, the storage requirement of the factorization cannot be forecast. To make sense, such a simulation thus needs to be performed during a real execution of the numerical factorization step rather than during the symbolic factorization step. However, the factorization of the matrices we wish to process precisely requires a large amount of memory possibly larger than the amount available on the computer used (otherwise we could systematically process them in-core and this study would not be very relevant). Therefore, we simulated an out-of-core treatment of the factors in `MUMPS`: we free the corresponding memory as soon as each factor is computed. Of course the solution step cannot be performed as factors are definitively lost, but freeing them allows us to analyze real-life problems on a wide range of processors.

We measure the size of the new peak of memory (which actually corresponds to the peak of *active storage*) and compare it to the one we would have obtained with an in-core factorization (*i.e.* the *total storage peak*). In a distributed memory environment, we are interested in the maximum peak obtained over all the processors as this value represents the memory bottleneck.

7.3 Experimental results

In Figure 7.3, we present the typical storage requirements observed of the parallel multifrontal method for a large sparse matrix of 943695 equations, called `AUDIKW_1` (see Table 1.2). We have also reported the peak of storage for the factors. We notice that it is often close to the peak of total storage. This justifies the fact that it is not relevant to treat only the active storage out-of-core since it would not lead to large memory gains.

For a small number of processors, we observe that the active storage is much smaller than the total storage. In other words, if factors are written to disk as soon as they are computed, only the active storage remains in-core and the memory requirements decrease significantly (up to 80 % in the sequential case).

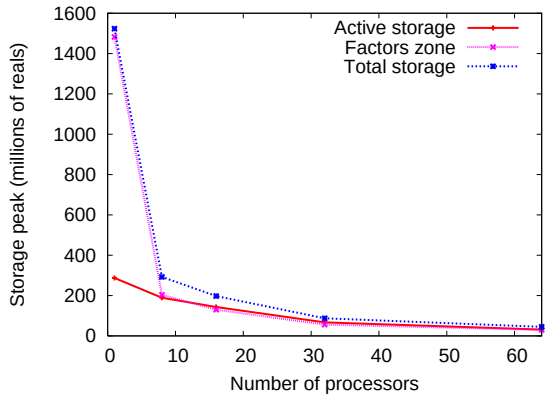


Figure 7.3: Typical storage requirement (`AUDIKW_1` matrix) with `METIS` on different numbers of processors.

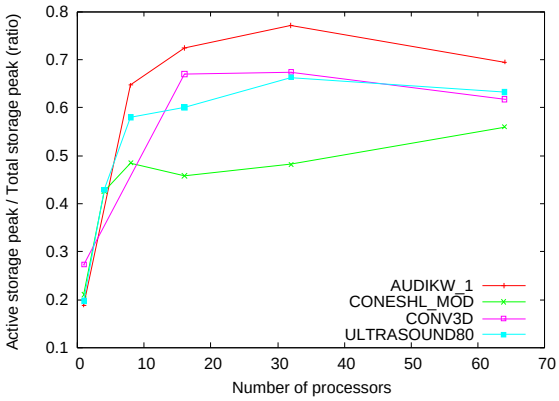


Figure 7.4: Ratio of active and total storage peak on different number of processors for several large problems reordered with `METIS`.

On the other hand, when the number of processors increases, the required amount of active storage decreases more slowly than the required amount of total storage as shown in Figure 7.4 for our four main test problems. For example, on 64 processors, the active storage peak reaches between 50 and 70 percent of the peak of total storage. In conclusion, on platforms with small numbers of processors, an out-of-core treatment of the factors will allow us to process significantly bigger problems; the implementation of such a mechanism is the object of Chapter 8. Nevertheless, either in order to further reduce memory requirements on platforms with only a few processors or to have significant memory savings on many processors, we may have to treat both the factors and the active storage with an out-of-core scheme. We will study this possibility in Chapter 8.

Note that we have been focussing in this discussion on the number of real entries in the factors, in the active storage, and in the total storage. The ratios presented in Figure 7.4 only consider the number of reals used for the numerical factorization. To be more precise, we should also take care of the amount of storage due to the integer workspace (indices of the frontal matrices, tree structure, mapping information,...) and the communication buffers. Table 7.1 provides the size in Megabytes of the different storage areas in the multifrontal code MUMPS, for 1 processor and 32 processors: integers for active storage and factors, integer arrays to store the tree, the mapping and various data structures, communication buffers at the application level to manage asynchronous communications.

We observe that communication buffers, that depend on the largest estimated message sent from one processor to another, also use a significant amount of memory in parallel executions. Once factors will be processed out-of-core, the memory required for this buffer will become a bottleneck to process very large problems. To overcome this limit, we will show in Chapter 10 (Section 10.5) the effects of subdividing large messages into series of smaller ones.

Matrix		Factors and active storage		Other data structures	Comm. buffers	Initial matrix (1)	Total
		Integers	Reals	(tree, ...)			
AUDIKW_1	1P	98	11839	26	0	479	12443
	32P	8	758	33	264	33	1097
CONESHL_MOD	1P	107	7160	34	0	526	7828
	32P	9	314	44	66	24	458
CONV3D64	1P	83	(2)	(2)	0	157	(2)
	32P	7	927	32	286	9	1260
ULTRASOUND80	1P	51	8858	16	0	401	9326
	32P	4	348	19	75	19	464

Table 7.1: Average memory (MegaBytes) per processor for the different memory areas. Those numbers are the ones estimated during the analysis step of the solver, and they are used to allocate the memory at the factorization step. (1) This corresponds to a copy of the initial matrix that is distributed (with some redundancy) over the processors. (2) For these values, an integer overflow occurred in the statistics computed by MUMPS.

Although the memory for the integer indices corresponding to active storage and factors is small compared to the memory for real entries, the algorithms presented in this dissertation could also be applied to the integers, and processing them out-of-core is also a possibility. A copy of the initial matrix is distributed over the processors in a special format (so called *arrowhead* format) for the assemblies occurring at each node of the tree. Some parts of this initial matrix are replicated on several processors to allow some tasks to be mapped dynamically. Once a node is assembled, the corresponding part of the initial matrix could be discarded. This is another possible improvement. However, we will not deal with these two possible improvements in the dissertation since they are not critical enough (for the moment).

In any case, for the four large matrices studied, we observe that the storage corresponding to real entries for factors and active storage is predominant, and that reducing it is a priority. This is the objective of the following chapter.

Chapter 8

A robust parallel code with factors on disk

We present in this chapter a new robust out-of-core code based on MUMPS in which computed factors are stored on disk during the factorization step. The motivation for storing the factors on disk is that, in the multifrontal method, factors produced are not reused before the solution step. According to the study presented in our preliminary study (Chapter 7), we expect to significantly decrease the memory requirements on a small number of processors and to obtain a reasonable memory reduction on larger numbers of processors.

The efficiency of low-level *I/O* mechanisms directly affects the performance of the whole application. First, we discuss direct and buffered I/O mechanisms at the system level (Section 8.1) and we present synchronous and asynchronous approaches that we have implemented at the application level (Section 8.2). We then compare the behaviour of those *I/O* mechanisms on several test matrices in the sequential case (Section 8.3) and in the parallel case (Section 8.4). Throughout this study, we aim at showing that we can reach the expected memory reductions (from our preliminary study) with a high performance of the factorization by managing asynchronism of the *I/O's* at the application level. We will also show that *I/O* buffering at the operating system level makes performance results difficult to reproduce and to interpret. More generally our study illustrates several drawbacks from default *I/O* mechanisms that have not been taken into consideration in the past.

In this chapter, the factors are written as soon as they are computed (possibly via a buffer) and only the active storage remains in-core. We recall that we use the C *I/O* library to perform the *I/O's* between core memory and disks (see Chapter 1, Section 1.3). When reporting memory usage, we focus on real data (factors, temporary active memory), excluding storage for integers and symbolic datastructures (which is comparatively negligible). Parallel executions (Section 8.4) rely on the dynamic scheduling strategy proposed in [13] (see Section 7.1). The results presented in this chapter have been obtained using METIS (see Section 1.1.1) - if not stated otherwise.

8.1 Direct and buffered (at the system level) I/O mechanisms

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 analyzing the performance of an out-of-core code, one wants to be sure that *I/O's* 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 rarely been done in other studies related to sparse out-of-core solvers. The only authors we are aware of who have taken these types of issues into account are Rothberg and Schreiber [61]: 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/O's* 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/O's* 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/O's* and computations.

8.2 Synchronous and asynchronous approaches (at the application level)

The purpose of this study 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 reach this objective, 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 have designed several *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 us to overlap *I/O* operations with computations[1]. 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. 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 of the first one. The whole synchronization mechanism is illustrated in Figure 8.1. Note that we limited our description to the case where only one *I/O* thread is attached to each computational thread. It could be interesting to use multiple *I/O* threads to improve overlapping

1. 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.

on machines with several hard disks per processor, or with high performance parallel filesystems.

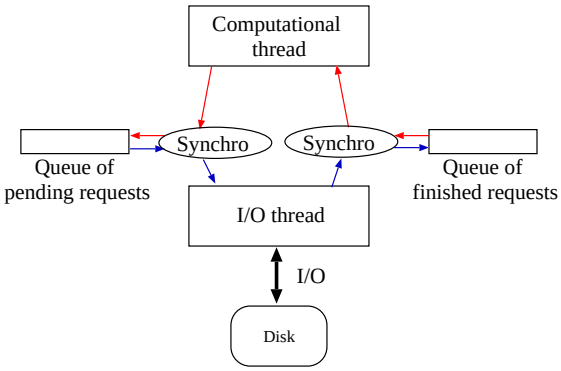


Figure 8.1: Asynchronous *I/O* scheme.

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.

8.3 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 1.4.2. Because this machine has a smaller memory, the factorization of some of the largest test problems swapped or ran out of memory. We first present results concerning relatively small problems 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 8.1 reports the results.

Matrix	Direct *I/O* Synch.	Direct *I/O* Asynch.	P.C. Synch.	P.C. Asynch.	IC
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 8.1: 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 8.1), 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 8.1.

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 that we have 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

Direct *I/O* Asynch.	P.C. Asynch.
1674	[2115]

Table 8.2: Elapsed time (seconds) for the factorization of matrix `AUDIKW_1` when the ordering strategy `PORD` is used. Platform is PSMN/FLCHP. [2115] Swapping occurred.

asynchronous 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` [69] ordering (see Table 8.2), which reduces the memory requirements in comparison to `METIS` for matrix `AUDIKW_1`. 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 8.1).

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 8.1) 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 8.3 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 *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.) is usually not enough. 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

Matrix	Direct *I/O* Synch.	Direct *I/O* Asynch.	P.C. Synch.	P.C. Asynch.	IC
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 8.3: 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.

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.

8.4 Parallel Performance

Table 8.4 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 has again serious difficulties to ensure efficiency (second and third drawbacks).

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.

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

Matrix	#P	Direct *I/O* Synch.	Direct *I/O* Asynch	P.C. Synch	P.C. Asynch	IC
CONESHL_MOD	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
ULTRASOUND80	4	373.2	[399.6]	349.5	[529.1]	(*)
	8	310.7	260.1	275.6	256.7	(*)
QIMONDA07	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 8.4: 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.

show in Table 8.5 (for matrix ULTRASOUND80) that we can achieve high performance using direct *I/O's* with an asynchronous scheme. When the number of processors becomes very

I/O mode	Scheme	1	2	4	8	16	32	64	128
Direct *I/O*	Synch.	1398.5	1247.5	567.1	350.9	121.2	76.9	44.6	36.5
Direct *I/O*	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 8.5: 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.

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 8.6, 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/O mode	Scheme	1	2	4	8	16	32	64	128
Direct *I/O*	Synch.	983.7	595.3	361.3	158.2	69.8	41.6	26.9	21.5
Direct *I/O*	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 8.6: 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.

8.5 Discussion

Overlapping of *I/O's* and computations allows us 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 [8]. 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 chapter, 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 8.7). When the number of processors increases (16 or more), an out-of-core execution usually allows us to save between 40% and 50% of memory, as reported in Table 8.7. Note that in some cases, the amount of memory saved can be much larger, as illustrated by the `QIMONDA07` matrix.

In the rest of this dissertation, the code presented in this chapter will be referred to as `Factors-on-disk`. This corresponds to the nomenclature previously introduced in Chapter 2 (Section 2.6).

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 8.7: 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).

Chapter 9

Effects of a parallel out-of-core stack on memory

In the previous chapter we have presented a scheme in which factors are processed out-of-core. Either to reduce more efficiently the memory requirements on large numbers of processors, or to process even larger problems, one may also need to process the active storage on disk. In this chapter, we evaluate the interest of storing the contribution blocks out-of-core. Our motivation is that the problem of managing the active storage out-of-core in a parallel 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 extend the models introduced in Chapter 2 (Section 2.6) to the parallel case. We use these models to better understand the memory limitations 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 completely different issue from achieving good performance (as discussed in the previous section).

9.1 Models to manage the contribution blocks on disk in a parallel context

We extend to the parallel context the different out-of-core models of assembly schemes introduced in Chapter 2 (Section 2.6, Figure 2.7): `All-CB`, `One-CB`, `Only-Parent`. As in the serial case, we rely on Assumption 2.1 from Chapter 2. In particular, because we assume that frontal matrices can be held in-core (but they can be scattered over several processors), we maintain the write-once/read-once property. However, in parallel, the stack mechanism to manage the contribution blocks is not maintained anymore since the dynamic and asynchronous behaviour of the factorization does not guarantee that each

processor performs a postorder traversal of the assembly tree.

Our aim is to evaluate the memory reduction allowed by our models. To assess the memory requirements, we have instrumented our parallel solver (the one from the previous chapter, `Factors-on-disk`) with a layer that simulates *I/O*s 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 about 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 performed 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. And contrary to the serial case, they cannot be forecast with a symbolic factorization. Clearly our main goal is to study the potential of a parallel out-of-core multifrontal method that stores temporary active storage 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.

9.2 Analysis of the memory needs of the different schemes

We report in Figure 9.1 a comparison of the memory peaks obtained when using our different assembly schemes for several large test problems (two symmetric ones and two unsymmetric ones). These test problems are representative of the behaviour we observed on the other matrices from Table 1.2. The top curve (`Factors-on-disk`), used as a reference, corresponds to the memory requirements of the code from the previous chapter, where factors are processed out-of-core but where the whole active storage is kept in core memory; the other ones have been 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 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 of the Factors-on-disk code. 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, see Section 7.1) that have been allocated to the processor responsible for 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.

9.3 Analysing how the memory peaks are obtained

We now analyze in more detail which types of tasks cause the peaks for each strategy. Table 9.1 shows the state of the memory when the peak is reached on the processor responsible for the peak, in the case of an execution on 64 processors for the AUDIKW_1 problem. Note that, based on load balancing criteria, the dynamic scheduler may allocate several tasks to one processor (each type of task is defined in Section 7.1). We notice that for the Only-Parent and One-CB out-of-core schemes as well as for the active memory in-core case, the memory peak 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 scheduler has allocated simultaneously too many slave tasks (3) to one processor, reaching together 42.97% of its memory. Note that it was also responsible for a master task but its size is less important (5.93%). Similarly to matrix AUDIKW_1, we have indeed studied the memory state for almost all the problems presented in Table 1.2, on various numbers of processors. Rather than presenting all the results, we preferred to only present here the main phenomena observed on a representative example. We nevertheless give another example for an unsymmetric matrix (CONV3D64) in Table 9.2.

For the symmetric problems (AUDIKW_1, but also SHIP_003, CONESHL2, CONESHL_MOD, for example), between 8 and 128 processors, the peak is reached [1] when the root of a sequential subtree is assembled; this occurs for all out-of-core schemes. Sometimes a slave

1. Except that (i) for the CONESHL_MOD problem on 64 processors the peak for the Only-Parent scheme arises when the root of the overall tree is processed; and (ii) for the AUDIKW_1 problem on 64 processors, the peak for the All-CB scheme is reached early in the factorization process (22% of the factorization time is then elapsed) while one single processor is simultaneously responsible for three slave tasks.

Scheme	Memory ratio of the active tasks			Memory ratio of
	master tasks	slave tasks	sequential subtrees	the contribution blocks
Stack in-core	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%	$48,32\%$	$51,63\%^*$	$0,04\%$

Table 9.1: 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 stack in-core case, on the `AUDIKW_1` problem with 64 processors. Symbol * in a column refers to the last task activated before obtaining the peak, which is thus *responsible* for it. When a sequential subtree is responsible for the peak, we observed that it is (here) at the assembly step of its root; so the numerical value reported in the corresponding column represents the amount of memory of the frontal matrix of the root of this subtree.

Scheme	Memory ratio of the active tasks			Memory ratio of
	master tasks	slave tasks	sequential subtrees	the contribution blocks
Stack in-core	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 9.2: 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 stack in-core case, on the `CONV3D64` problem with 64 processors. Symbol * in a column refers to the last task activated before obtaining the peak, which is thus *responsible* for it.

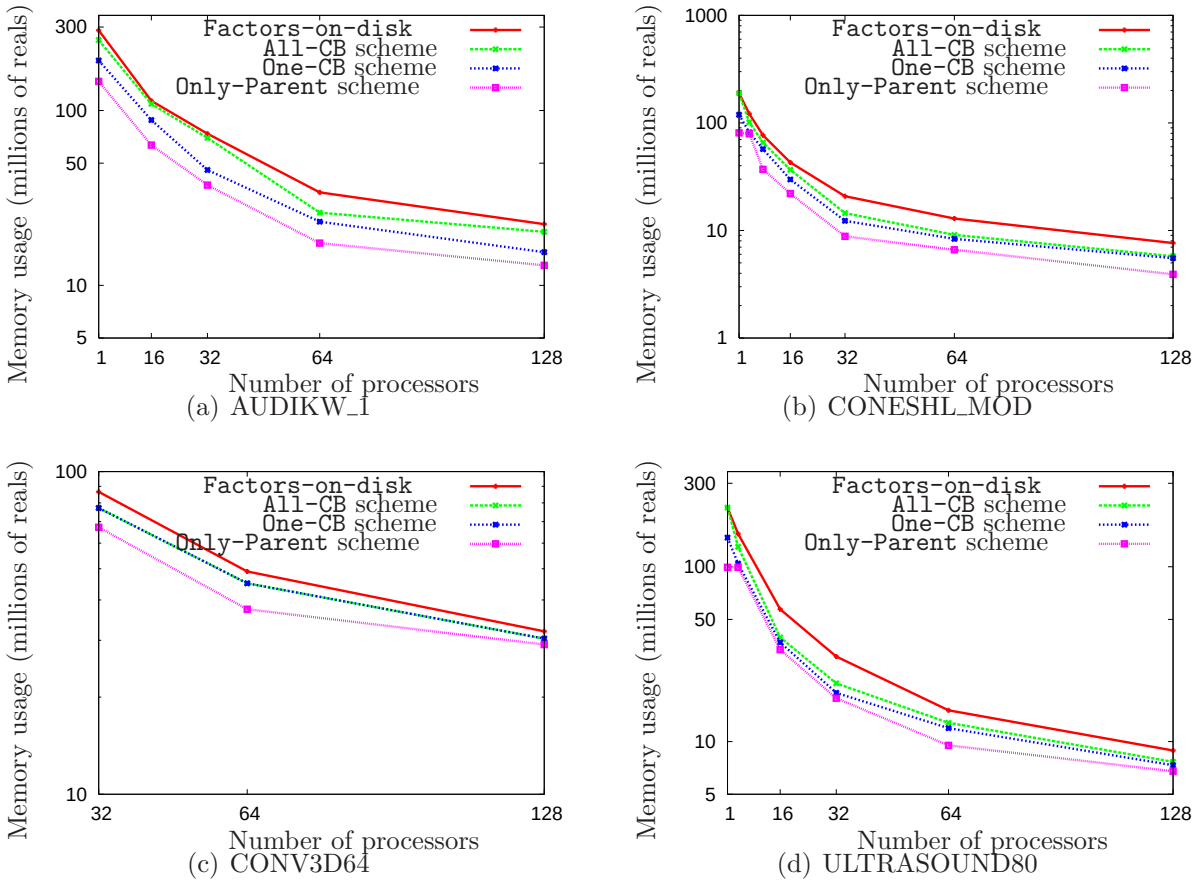


Figure 9.1: Memory behaviour (maximum memory requirement per processor) with different assembly schemes and various numbers of processors for several large problems (`METIS` is used as reordering technique).

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). For `CONESHL2`, a smaller symmetric problem, this behaviour remains globally true but it is less systematic (except for the `Only-Parent` scheme for which it remains systematic). Indeed, the main reason is that the memory of the active tasks is low compared to the one of the contribution blocks; the memory peak may thus arise just because we get one or several large contribution blocks.

For the unsymmetric problems (`CONV3D64`, `ULTRASOUND80`), on many processors (from 16 to 128), the peak is generally obtained because of a large master task. This is increasingly true when we tend to the `Only-Parent` scheme. With fewer processors (less than 8), the assembly of a root of a subtree is more often responsible for the peak. Nevertheless, these effects are sometimes hidden when many (2 up to 6) tasks are simultaneously active. For example, on 64 processors with the `All-CB` scheme, for the `CONV3D64` problem, the peak is obtained while a processor has four slave tasks simultaneously in memory.

Thanks to parallelism, memory needs of a particular task can be parcelled out over many processors. Ideally, platforms with an arbitrarily large number of processors should thus enable the factorization of arbitrarily large problems. However, in order to be efficient, some tasks are sequential and become the memory bottleneck when the other ones are parallelized.

First, in MUMPS, to bound the number of communications, the nodes at the bottom of the tree are aggregated into subtrees which are treated sequentially (see Figure 7.1). Such a subtree may then be critical in terms of memory, its peak (usually arising when its root is performed) being the memory bottleneck of the whole factorization step. We observed it was particularly true for symmetric problems.

Next, the processor responsible for a master task treats sequentially all the fully summed rows of the corresponding frontal matrix (only the blocks matching the Schur complement can be distributed over several processors). This way, with a large number of processors, their treatment becomes critical. Figure 7.2 shows that the memory needs corresponding to master tasks are more important for unsymmetric cases than for symmetric ones. On the range of processors used, the limiting factor observed is indeed the treatment of master tasks for unsymmetric problems and the one of the subtrees in the symmetric case.

9.4 Decreasing the memory peaks

It results from the previous section that in order to decrease the memory needs, the size of the master tasks has to be limited for the unsymmetric problems whereas the size of the subtrees has to be diminished for the symmetric ones. Furthermore, applying together these two approaches could further improve scaling. On a limited number of processors, the number of simultaneous active tasks should moreover be bounded.

Concerning large master tasks, we can use the splitting algorithm of [9]. Since the factorization of the pivot rows of a frontal matrix is performed by a single (master) processor, we replace the frontal matrix in the assembly tree by a chain of frontal matrices with less pivot rows, as illustrated in Figure 9.2. This limits the granularity of master tasks, at the cost of increasing the cost of assemblies from children to parents.

Concerning the size and the topology of the subtrees, they are currently based on load balancing criteria. For the symmetric problems, we have modified the corresponding threshold by hand to diminish the size of the subtrees. As shown in Figure 9.3(a) for the AUDIKW_1 problem, we can save up to more than 40% on large symmetric problems. In particular, the One-CB scheme, which (as shown above) is a good balance between performance and memory, saves more than 20% at every execution on the range of processors used (8 - 64). Note that decreasing the size of the subtrees allows us to decrease the global memory peak not only because (in most cases) it was obtained when processing a root of a subtree but also (sometimes) because decreasing the granularity of these sequential tasks allows a better load balancing when processing the tasks just above these in the

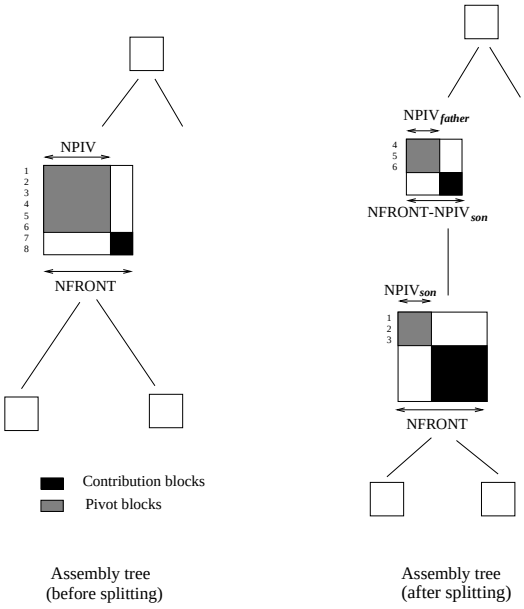


Figure 9.2: Tree before and after the subdivision (or splitting) of a frontal matrix with a large pivot block.

assembly tree. For the AUDIKW_1 problem on 64 processors, this second reason (and only this one) explains the saving of 23% of memory with the All-CB scheme (Figure 9.3(a)).

For the unsymmetric matrices, we have split the largest master tasks. The corresponding node is replaced by a chain of nodes. Figure 9.3(c) shows that for the CONV3D64 problem we get important savings (from 8.5 up to 27.8%) except on 64 processors for which we did not manage to get a good tuning.

When we split master tasks of unsymmetric matrices, we have observed that the new memory peak then sometimes arises when a subtree is processed. Thus, we have tried to both split large master tasks and reduce the size of the subtrees. For the CONV3D64 problem, with the One-CB strategy on 32 processors, splitting the master tasks allows us to process the problem with a memory of 69 million reals per processor, that is a 8.5% saving (see Figure 9.3(c)); but additionally decreasing the size of the subtrees makes it possible to treat it with 62 millions of reals which represents this time a 17.8% saving. Reciprocally, splitting the master tasks of the symmetric problems after reducing their subtrees sizes allowed us to increase the memory savings in several cases. The AUDIKW_1 problem illustrates this phenomenon for which on 32 processors 23.2% of memory is then saved instead of 12.4% without splitting. Nevertheless, with these memory improvements a new problem arises: the elapsed time for the factorization step increases. For example, for the CONV3D64 problem on 32 processors with the Only-Parent strategy, splitting does allow us to save 23% but we then observed an overhead of 20% on the elapsed factorization time (418.8s $\rightarrow$ 496.9s). We face a key point of the future work: decreasing memory requirements while keeping good performance. Indeed, in the current scheme implemented, the mapping of the chain of nodes built when splitting nodes with a large

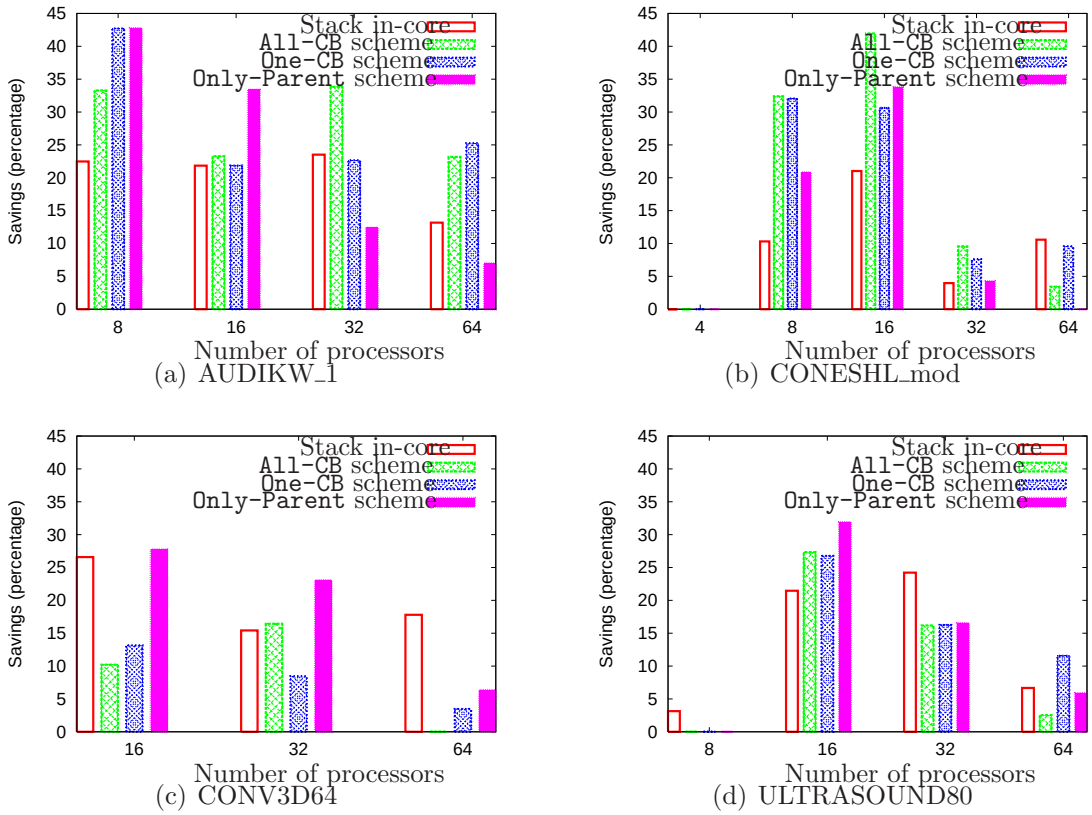


Figure 9.3: Memory savings for two symmetric problems, 9.3(a) and 9.3(b) cases (resp. for two unsymmetric problems, 9.3(c) and 9.3(d) cases), obtained by decreasing the size of the subtrees (*resp.* by splitting the master tasks), for several stack memory management schemes, on various numbers of processors. `METIS` is used to permute the matrices.

master task implies a communication overhead.

9.5 Conclusion

In this chapter, we have studied the effect on the core memory of storing the contribution blocks to disk. The results show some potential of the parallel out-of-core multifrontal method: it seems that the intrinsic limits of the sequential multifrontal method become much less critical thanks to parallelism. In the two cases (symmetric and unsymmetric), we have modified thresholds of load balancing constraints to save memory. The thresholds have been tuned specifically for each case. New criteria and algorithms based on memory constraints now have to be designed to determine the size of the subtrees and granularity of tasks. Furthermore, all scheduling decisions should be adapted to fit the out-of-core scheme and avoid too many simultaneously active tasks.

In the next chapter, we will tackle the problem of memory scalability from a different

point of view. Rather than modifying some parameters of an existing mapping, we will rethink the mapping and scheduling aspects to focus on memory scalability, even when the stack is processed in core.

Chapter 10

Improving the memory scalability

In the previous chapter, we have assessed the potential of a parallel method that processes the contribution blocks out-of-core. Another approach consists in keeping the contribution blocks in-core and limiting the peak active storage (contribution blocks and active frontal matrices). Even if the contribution blocks were processed out-of-core, reducing the peak of storage would bound the *I/O* volume. We already know the postorder traversal that minimizes this peak in the serial case (see [45], [55] or Section 3.1.2). In the parallel case, we would ideally like to divide this amount of memory by the number p of processors used to perform the factorization. We formalize that intuition with the notion of *memory efficiency*. Let S_{seq} be the amount of storage required to process a given matrix in sequential mode, $S_{avg}(p)$ the average amount of storage (among all the processors) required to process that matrix with p processors, $S_{max}(p)$ the maximum amount (among all the processors) of storage required to process that matrix with p processors.

Definition 10.1. *The* memory efficiency *on p processors is defined as: $e(p) = \frac{S_{seq}}{p \times S_{max}(p)}$; the* average memory efficiency *is defined as: $e_{avg}(p) = \frac{S_{seq}}{p \times S_{avg}(p)}$*

A perfect memory scalability on p processors corresponds to a memory efficiency equal to $e(p) = 1$. It is obtained when (i) the average memory efficiency is equal to $e_{avg}(p) = 1$ and (ii) the memory peaks on the different processors are equal.

10.1 Proportional mapping and memory scalability

10.1.1 Practical study on real-life matrices

In practice, the average memory efficiency of the `Factors-on-disk` code from Chapter 8, based on `MUMPS`, is significantly lower than the ideal value of 1 as illustrated in Table 10.1. One obvious reason for this suboptimal scalability is that the dynamic scheduler allows several tasks to be active simultaneously [13] on a given processor: those tasks occupy some memory and the processor works on them alternatively, depending on

Number p of processors	16	32	64	128
AUDI_KW_1	0.16	0.12	0.13	0.10
CONESHL_MOD	0.28	0.28	0.22	0.19
CONV3D64	0.42	0.40	0.41	0.37
QIMONDA07	0.30	0.18	0.11	-
ULTRASOUND80	0.32	0.31	0.30	0.26

Table 10.1: *Average memory efficiency* e_{avg} (p) of the `Factors-on-disk` code from Chapter 8. `METIS` ordering was used. These results have been derived from Table 8.7 (out-of-core part).

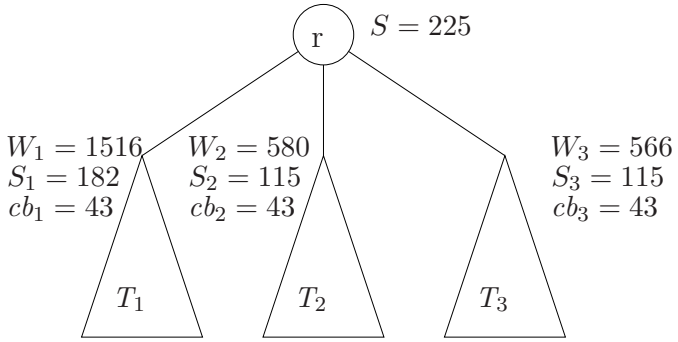


Figure 10.1: Zoom on the top of the elimination tree of matrix `ULTRASOUND80`. The root r has 3 children. Each child i ($1 \leq i \leq 3$) is root of a subtree T_i. Each subtree T_i represents W_i Gflops. The other numeric values are in millions of reals. They represent the sequential storage requirements for the whole elimination tree (S) or for the subtrees ($S_i, 1 \leq i \leq 3$); the frontal matrices associated with the roots of each subtree T_i produces a contribution block of size cb_i.

message receptions. However, we have constrained the dynamic schedulers to avoid such situations, and have observed that the average memory requirement was not significantly improved when the tasks are processed one after another. For instance, on 32 processors, matrix `ULTRASOUND80` would require 139 MB per processor doing so. It is lower than the original 178 MB but still represents a low average memory efficiency ($e_{avg}(32) = 0.39$).

We now explain this gap. To ensure coarse-grain parallelism and achieve high performance, parallel multifrontal methods partly rely on *proportional mapping* [36]. That top-down algorithm is performed during the analysis step and aims at mapping the processors onto the nodes of the elimination tree. Even if the proportional mapping in a code like `MUMPS` is relaxed, let us consider what happens with a strict proportional mapping. Initially, all the processors are mapped on the root of the elimination tree. The children then recursively inherit a subset of the processors that are mapped on their parent. The number of processors mapped on a child is proportional to the amount of work related to the subtree rooted at this child. The root of the elimination tree associated with matrix `ULTRASOUND80` has 3 children as illustrated in Figure 10.1. Initially, $p = 32$ pro-

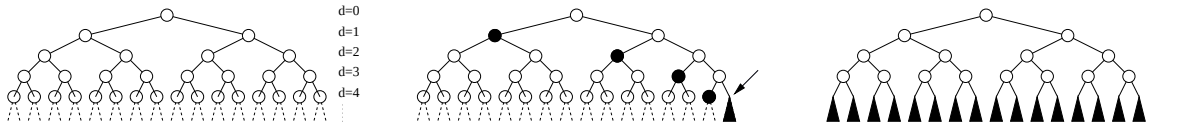
cessors are mapped on that root. A maximum average memory efficiency corresponds to an average memory peak equal to 7.0 million reals per processor (serial peak of storage, S, divided by p). According to the proportional mapping algorithm, the three children respectively inherit $p_1 = 18.4$, $p_2 = 6.9$ and $p_3 = 6.7$ processors (the sum is equal to 32), proportionally to the workload W_i of the subtree T_i whose child i is root. Subtree T_1 has a sequential peak of storage equal to $S_1 = 182$ million reals. At best, if T_1 is processed with an optimum memory efficiency with respect to that subtree, the average memory peak will be equal 9.9 million reals per processor (S_1/p_1). This amount is greater than the 7.0 million reals per processor required to process the whole tree with an optimum average memory. Similarly, at best, subtrees T_2 and T_3 can be respectively processed with 17.7 and 17.1 million reals per processor. Therefore, the average peak of storage per processor on the whole tree will be far greater than 7.0 million reals per processor and the optimum average memory efficiency cannot be reached.

One may think the reason is that proportional mapping is performed proportionally to workload but not to storage requirement. Let us apply this latter variant of proportional mapping to matrix `ULTRASOUND80` with 32 processors. Each subtree T_i inherits a number of processors proportional, this time, to its storage requirement S_i. We obtain $p_1 = 14.1$, $p_2 = 9.0$ and $p_3 = 8.9$. Therefore, at best, each subtree can be processed with 12.9 million reals per processor (S_i/p_i). This amount is equal for each subtree but remains significantly greater than the amount of 7.0 million reals per processor required to process the whole tree with an optimum average memory. We conclude that the partitioning of the processors mapped on the parent into disjoint subsets of processors is in itself a drawback to memory efficiency.

On the other hand, let us assume that all the processors are mapped on each subtree and that we process the subtrees one after the other. All the processors are mapped on subtree T_1 ($p_1 = 32$). It can thus optimally be processed with an average memory per processor equal to 5.7 million reals (S_1/p_1). Then, its contribution block is scattered on those 32 processors when processing T_2, again with all the processors ($p_2 = 32$). When the second subtree is processed, the contribution block of the first subtree has to be kept in memory. It can thus optimally be processed with an average memory per processor equal to 4.9 million reals ($(cb_1 + S_2)/p_2$). Similarly, T_3 is processed with $p_3 = 32$ processors with the contribution blocks of the first two subtrees distributed in the memories of the different processors. Subtree T_3 can thus optimally be processed with an average memory per processor equal to 6.3 million reals ($(cb_1 + cb_2 + S_3)/p_3$). At no time, the critical value of 7.0 million reals has been exceeded. That amount is finally reached when the root is processed and a memory efficiency equal to 1 is obtained.

10.1.2 Quantification of the sub-optimality of proportional mapping

Let us consider an elimination tree similar to a perfect binary tree, that is a tree in which every node has two children except leaf nodes which are all at the same depth (see Figure 10.2(a)). We assume that the tree is processed with $p = 2^k$ processors. We assume that the nodes that are at a depth lower than or equal to k have contribution



(a) Elimination tree (b) Postorder traversal (c) Traversal induced by proportional mapping

Figure 10.2: A perfect binary elimination tree (a) and possible traversals (b,c). Data in memory when the peak is reached during the tree traversal are colored. There are p of them in (c) and only $\log(p)$ of them in (b). The arrow points to the node on which the peak is reached with a postorder traversal (b).

blocks of the same size, cb, and frontal matrices of same size, $m = 2 \times cb$. 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 10.2(b)).

If all the processors are mapped on each node and that they synchronously follow a postorder traversal, the memory efficiency is equal to 1. 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 10.2(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 at the top of the elimination tree. Therefore, the proportional mapping may not induce such a poor memory efficiency in all cases.

10.2 A memory-aware mapping algorithm

In the previous section we have shown that proportional mapping was usually a drawback to reach a high memory efficiency and that an optimum average memory efficiency is obtained with a postorder traversal of the elimination tree. This postorder traversal requires all the processors to be mapped on each node. Each node is processed in parallel but the nodes are processed one by one. The structure of the elimination tree is not used to ensure coarse-grain parallelism anymore. Clearly the exclusive use of such a fine-grain parallelism is a major bottleneck to achieve reasonable performance.

We now assume that we are given an amount M_0 of memory per processor. If that amount is greater than or equal to S_{seq}/p, the matrix can be processed with p processors by following the same postorder traversal as in the sequential case. But we can also use the available memory to locally perform some proportional mapping when possible. The purpose is to favour coarse-grain parallelism (between paths of the elimination tree) to fine-grain parallelism (on a single node). An equivalent objective is to limit the number

of processors per node as long as the memory constraint is respected and the workload is balanced.

We adapt the proportional mapping algorithm as follows. We assume that a preliminary bottom-up traversal of the elimination tree has computed the storage requirement S_i and the number of floating points operations W_i of each subtree T_i ($1 \leq i \leq n_{sbtr}$). Our *memory-aware mapping algorithm* is then applied before starting the numerical factorization. It traverses the nodes of the elimination tree from top to bottom. Initially, all the nodes are mapped on the root of the elimination tree. The children then recursively inherit a subset of the p processors that are mapped on their parent. We first try to perform a proportional mapping step: the number of processors mapped on a child is proportional to the amount of work of the subtree rooted at this child. We then check whether each subtree T_i respects the memory constraints ($S_i/p_i \leq M_0$):

- If all the subtrees respect the memory constraints, we partition the set of processors mapped on the parent into disjoint subsets ($\sum_{i=1}^{n_{sbtr}} p_i = p$), according to the proportional mapping. The subtrees will be processed in parallel during the numerical factorization and will inherit the same memory constraint M_0.
- Otherwise, each subtree inherits *all* the processors of its parent ($\forall i, p_i = p$) and the subtrees will be processed one by one during the numerical factorization, following a local postorder. In that case, when a subtree T_j is processed during the numerical factorization, the contribution blocks of the previous siblings ($1 \leq i < j$) have to be kept in memory. We take into account this memory consumption by modifying M_0 for the next levels: $M_0 \leftarrow M_0 - (\sum_{i=1}^{j-1} cb_i)/p$. This means that each one of the p processors inherits on average a part cb_i/p of its previous sibling contribution blocks. The new value of M_0 will be recursively used for the tests at the lower levels of subtree T_j.

This algorithm ensures that, for a given an amount of memory $M_0 \geq S_{seq}/p$, we obtain a mapping such that the effective average storage used in the actual factorization satisfies: $S_{avg}(p) \leq M_0$. However, at this point, we have no control on the memory unbalance related to the size of the master tasks (see Section 7.1) which means that $S_{max}(p)$ may be greater than M_0. Therefore, we take into account a possible unbalanced memory distribution with a tolerance parameter, $t > 1$: the actual check consists in verifying whether $S_i/p_i \leq M_0/t$ is true. In practice, we must provide $M_0 \geq t \times S_{seq}/p$. If the effective memory imbalance ($S_{max}(p)/S_{avg}(p)$) is lower than t, we will use less memory than M_0 ($S_{max}(p) < M_0$). The parameter t will be determined experimentally (see below).

10.3 First experiments

We have implemented the memory-aware mapping algorithm described above within `MUMPS`. We have also modified the dynamic schedulers of `MUMPS` to respect the constraints imposed by our mapping algorithm. Indeed, the subtrees that inherit *all* the processors of their parent in the *memory-aware* algorithm have to be processed according to a local

postorder traversal. Therefore, we have prevented the activation of more than one node at a time on a given processor and we have implemented dynamic mechanisms (slave selection strategy, task selection) that force us to follow the expected traversal. We illustrate the effects of our algorithm on memory usage with matrix `ULTRASOUND80` on 32 processors. The results presented in this section have been obtained on the Borderline machine (see Section 1.4.2 of Chapter 1).

In a first experiment, we have used a tolerance parameter $t = 3.0$ and a memory $M_0 = 5 \times S_{seq}/p = 35.2$ million reals. The factorization successfully ran with an average effectively used memory equal to 11.8 million reals. However, the memory usage of all processors was unbalanced: the most loaded processor used 34.8 million reals. The ratio $S_{max}(p)/S_{avg}(p) = 2.95$ remained lower than the tolerance parameter $t = 3.0$. The factorization would have failed with a tolerance parameter $t < 2.95$. Indeed, the unbalanced distribution of memory usage is due to large master tasks on which our algorithm has no control. As discussed before, the algorithm succeeds if the effective ratio between the maximum memory usage and the average memory usage is lower than the tolerance parameter; we need: $S_{max}(p)/S_{avg}(p) < t$. Said differently, our algorithm maximizes the average memory efficiency $e_{avg}(p)$ but the memory efficiency itself, $e(p)$, will be bounded by $1/2.95$. This is not very satisfactory since, in practice, $e(p)$ is the critical parameter.

To overcome this drawback, we have split large frontal matrices into chains of nodes (see Figure 9.2 and the related discussion in Section 9.4). We have ensured that the size of the master part of each resulting node of a chain is bounded by the value: $1/3 \times S_{seq}/p$. This is is clearly a strong constraint but it is used here to assess whether the splitting of master tasks can resolve the problem of memory balancing. We now apply our algorithm after that preliminary splitting step. To take into account the expected improvement on the memory balancing, we use a lower tolerance parameter $t = 2.5$. We also expect to be able to go further in the memory reduction and take in entry an amount of memory $M_0 = 3 \times S_{seq}/p = 21.1$ million reals. The factorization step successfully ran. An average effective space of 8.79 million reals was used and the maximum effective space used was equal to 8.96 million reals. We obtained an almost perfect memory balance as illustrated by the distribution of the memory peaks in Figure 10.3. All the processors have reached their peaks when processing the same node, except processor 1 which was master of the node. Thanks to the initial splitting, master tasks are not a bottleneck for memory anymore.

The maximum memory peak of 8.96 million reals corresponds to a memory of 72 MB and a memory efficiency equal to $e(32) = 0.8$. This amount of memory is much lower than the 178 MB obtained with the `Factors-on-disk` code of Chapter 8 (see Table 8.7). Furthermore, the relaxation parameter $t = 2.5$ can be decreased since in practice the ratio $S_{max}(p)/S_{avg}(p)$ is almost equal to 1.

Figure 10.4 presents the impact of the available memory M_0 on the average number of processors per node. We recall that coarse-grain parallelism is obtained when that number is small. For each node, we divide the number of processors of the new mapping by the number of processors of the proportional mapping. We compute this ratio for all the nodes at a given depth and report the average value obtained at each depth.

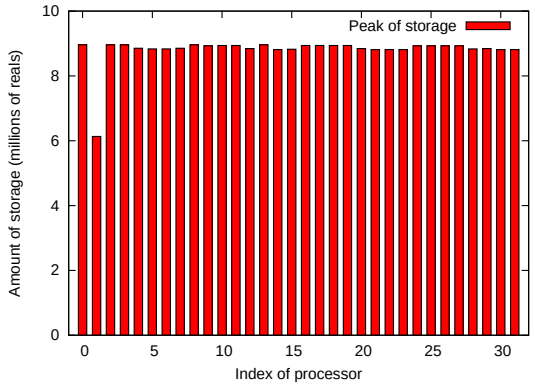


Figure 10.3: Distribution of the memory peaks with matrix ULTRASOUND80 on 32 processors.

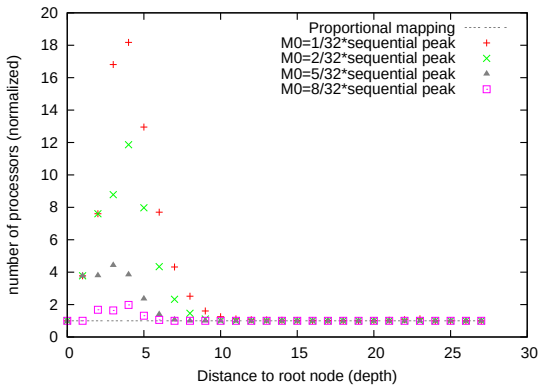


Figure 10.4: Number of processors per node (normalized to the proportional mapping case) with matrix ULTRASOUND80 on 32 processors.

Clearly, we see that our *memory-aware* algorithm exploits the available memory to limit the average number of processors per node. One interesting point of Figure 10.4 is the square of coordinates $(1, 1)$. It means that with an available memory of $M_0 = 8/32 \times S_{seq}$, we can apply a local proportional mapping on the root r of matrix ULTRASOUND80. Therefore, the 3 child subtrees of r (see Section 10.1.1) can be processed in parallel. At depth 2 (square of coordinates $(2, 1.9)$), some nodes then require more processors than the proportional mapping because of the memory constraints. The point corresponding to the "plus symbol" of coordinates $(4, 18.1)$ shows that with a very limited memory ($M_0 = 1/32 \times S_{seq}$), our algorithm strongly diverges from the proportional mapping on the top of the elimination tree (depth 4). Finally let us notice that the bottom of the elimination tree (depth $d > 10$) can be processed with the same degree of parallelism in all cases: the *memory-aware* algorithm has exploited the fact that, on that matrix, the bottom of the tree is not critical for memory.

10.4 Discussion

We have presented an algorithm which builds and follows the sequence that allows as much coarse-grain parallelism as possible while respecting the memory constraints. This sequence is built at the analysis step and is strictly respected during the parallel numerical factorization step. Nevertheless, the memory usage varies in time (see Figure 2.5 in Chapter 2). Therefore, we could temporarily exploit the free memory to deviate from the initial sequence and dynamically balance the workload. For example, we may start some tasks if memory allows it and we are sure that we do not take too many risks: decisions that enable to respect present memory constraints may lead the execution to run out-of-memory later on. This situation is similar to deadlock avoidance algorithms [64] except that in our context the critical resource is the memory. The sequence that we have

exhibited represents in deadlock avoidance algorithms what is called a *safe sequence*.

10.5 Other memory reductions

The out-of-core functionality has strongly reduced the memory requirements for the real space used in MUMPS. Therefore, some other workspace that used to be small compared to the real space became significant and their size had to be reduced. The work presented in this section is joint work with all the MUMPS development team.

I/O buffers

We have mentioned in Chapter 8 that our `Factors-on-disk` code requires large buffers to perform asynchronous *I/O's*. The drawback of that approach was that the size of the *I/O* buffers was dependent on the size of the frontal matrices and could be huge (see column *Factor block* of Table 10.2). Initially, the elementary written data was the factor part of each matrix. We have divided that factor part into panels of fixed size. The work consisted in rethinking the *I/O* layer to manage panels. We have obtained three advantages from that:

- The buffer size has been strongly reduced when asynchronism is managed at the application level (see column *Panel* of Table 10.2);
- *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.
- The L and U factors can be written to independent files. This allows a better data access during the solution step and strongly reduces the efficiency of that step which is even more sensitive to *I/O's* than the factorization phase.

		Out-of-core elementary data	
Matrix	#procs	*Factor block*	*Panel*
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 10.2: Size of the *I/O* buffers (MB) with an asynchronous factorization.

Limiting the size of the communication buffers

We have noticed in our preliminary study (see Table 7.1) that communication buffers became critical in a parallel out-of-core context. For instance, matrix CONV3D64 used

to require 1260 MB (all included) per processor with the in-core version of MUMPS on 32 processors. Among those 1260 MB, 286 MB were dedicated to the communication buffers. That amount was defined with respect to the largest message that could possibly be sent during the factorization. It was thus dependent on the size of the problem and on the number of processors. With the Factors-on-disk code from Chapter 8, the total space required to perform the factorization has been decreased to 800 MB per processor. The size of the communication buffers thus represents 35% of the total memory used.

The communication messages have been split into packets of smaller messages of fixed size. Table 10.3 shows the effects on memory. This communication scheme has required the introduction of some new synchronizations. However, experimental results have shown that they do not induce a significant overhead on performance.

	Communication scheme	
Matrix	*Original scheme*	*New scheme*
AUDIKW_1	264	4.2
CONESHL_MOD	66	3.7
CONV3D64	286	16.1
ULTRASOUND80	75	8.2

Table 10.3: Size of the communication buffers (MB) on 32 processors.

In-place allocation in the serial subtrees

We have also implemented a *last-in-place* scheme (see Section 2.1.2) where the last child is allowed to overlap with the frontal matrix of the parent. This allowed us to decrease the memory requirements for the serial subtrees by an amount usually approximately equal to 15% on average.

10.6 Conclusion

To conclude, we have proposed an algorithm that allows us to scale the memory space used for the active storage. Together with the other memory reductions presented in Section 10.5, our new code is ready to process very large matrices. The novelty of our algorithm is that it exploits the available memory to allow for as much coarse-grain parallelism as possible. This coarse-grain parallelism should lead to good performance by limiting the amount of communication.

Currently, our algorithm induces an average overhead on performance of a factor 3 compared to the original Factors-on-disk code. We need to further investigate and understand the reasons for this slow down. In all cases, we think that the splitting of master tasks is responsible for a large part of that overhead. In the current code, splitting is performed before our *memory-aware mapping* algorithm. One possible improvement would consist in interleaving mapping and splitting to limit the number of split nodes as

a function of the number of mapped processors. Another improvement (that could be in addition to the previous one) would consist in rethinking the communication scheme of split nodes. The idea would consist in mapping the same processors to the same rows of each node of the chain.

A detailed performance study then needs to be carried out knowing that there are also possibilities for improvement with respect to the discussion of Section 10.4. This could lead to improvements in the mapping algorithm and in the dynamic schedulers. All in all, even if the performance could be improved, this approach already provides a very good memory scalability and should allow the solution of very large problems. This is a very important result from the point of view of applications.

Conclusion

In this dissertation we have addressed both theoretical (Part I) and practical (Part II) issues related to out-of-core sparse direct methods. These parts have progressed in parallel.

Part I has investigated the difficulties of storing temporary data on disk. To study the out-of-core multifrontal method, we had to start by modeling the *I/O* volume and understanding its behaviour. We have then shown that minimizing the *I/O* volume is different from minimizing the peak storage, even in the most simple cases where the data access pattern follows a stack mechanism. We have proposed algorithms to minimize the *I/O* volume for many variants of the multifrontal method, including terminal and flexible allocations, classical and *in-place* assembly schemes, as well as new assembly schemes (*max-in-place*, *in-place-realloc*). We have provided all the ingredients to design a serial out-of-core multifrontal solver that minimizes the *I/O* volume related to the stack of contribution blocks; this includes memory management algorithms that should allow an efficient implementation. Furthermore, the proposed algorithms can improve the serial parts of parallel solvers based on a multifrontal method such as `MUMPS`.

Concerning supernodal methods, we have generalized to the sparse case (under specific assumptions) a well known result from dense factorization: *left-looking* algorithms allow us to perform significantly less *I/O* than *right-looking* algorithms. To prove this result, we have considered an existing hybrid method (*left-looking/right-looking*) proposed and implemented in [41, 63] and compared it with other possible combinations of *left-looking* and *right-looking* methods. We have then addressed the problem of *I/O* minimization for two specific methods (*left-looking/right-looking* and *left-looking/left-looking*); we have exhibited an optimum algorithm for the first one (under our specific assumptions) and proposed a heuristic for the second one after showing that it is NP-complete. This study has led us to develop a prototype out-of-core extension of `SuperLU`. The purpose has been to build an out-of-core-friendly data access pattern that respects as much as possible the original in-core version of `SuperLU`. Currently, our prototype performs memory copies instead of *I/O's*. We plan to incorporate a robust *I/O* layer in the prototype in order to assess the actual efficiency of our method.

All in all, the results presented in Part I tend to show that the *I/O*-minimization problem is more complex than the memory minimization problem: in contexts where the memory could be minimized with greedy algorithms, we have had to employ heuristic approaches in order to limit the volume of *I/O* since the associated decision problem was proved to be NP-complete. Although we have shown that the *I/O* minimization was NP-complete both in the context of the (flexible) multifrontal methods and the one of the supernodal methods (*left-looking/left-looking* approach), we think that the practical difficulty for reducing the *I/O* volume is not the same for these two classes of direct methods. Indeed, for the multifrontal methods, we could exhibit algorithms that are optimum in (most) practical cases. On the contrary, supernodal methods provide more freedom in the data access pattern and we could only address the *I/O* minimization problem under specific assumptions; the minimization of the *I/O* volume without those restrictive assumptions remains an open problem. In all cases, the proposed algorithms apply in (and have been implemented for) the general case for which they remain interesting heuristics. In particular, it seems to us that top-down partitioning algorithms of the elimination tree into SuperPanels should be preferred to bottom-up approaches that have been used so far [63].

In Part II, we have presented a robust parallel out-of-core direct solver that stores computed factors on disk. It allows us to handle problems significantly larger than an in-core solver. The out-of-core factorization achieves a high efficiency: we have shown that it could be almost as fast as the in-core one on our platforms. We have also highlighted several drawbacks of the *I/O* mechanisms generally used in other out-of-core sparse direct solvers (which in general implicitly use system buffers): memory overhead that can result in excessive swapping activity, extra cost due to 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 us to solve much larger problems. With more processors (16 to 128), because the active storage does not scale as well as the factors, the core memory usage is only reduced by a factor of two, on average. 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, especially when the number of processors is limited. 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 [8], ...). In order to go further in the memory reduction, we have then studied two complementary aspects: storing the contribution blocks on disk and totally rethinking the mapping and the scheduling of the tasks.

To study a possible out-of-core storage of the contribution blocks, we have proposed

in Chapter 9 several models for the assembly step of parallel multifrontal methods. 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 would be the most useful to implement. We have also identified some key parameters related to the management of parallelism (granularity of subtrees and of master tasks) that can impact the memory usage.

Based on those results, we have decided in Chapter 10 to first rethink the mapping and scheduling strategies. Our purpose has been to achieve a good scalability of the storage associated with contribution blocks. This is useful in the existing code that stores factors on disks and would reduce the *I/O* volume when contribution blocks are on disk. We have first shown that maximizing coarse-grain parallelism (ensured by proportional mapping) and minimizing the memory usage are two contrary objectives. We have then proposed an algorithm that achieves a very high memory scalability at the cost of being restricted to fine-grain parallelism. Finally, we have proposed an algorithm that exploits the available memory to maximize coarse-grain parallelism as long as it respects the memory constraints. Combined with other memory reductions (sizes of the *I/O* and communication buffers, size of the serial subtrees), our *memory-aware* algorithm has proved to significantly reduce the memory usage on large matrices and is ready to process very large ones. Our scalable out-of-core direct solver particularly fits the requirements of architectures which have a huge number of cores but a limited amount of memory per core (*e.g.* Blue Gene).

All in all, Part II has shown that parallel multifrontal methods could allow us to process large problems with a high efficiency. In particular, we have seen that parallelism allows us to efficiently handle frontal matrices - that are a major drawback to serial out-of-core multifrontal methods [61, 63]. Our main perspectives are related to the work of Chapter 10. Now that first experiments have validated the interests of our approach (scalability, usage of extra-memory for coarse-grain parallelism), we plan to design new dynamic schedulers that exploit the actual variations of memory consumption. In periods of low memory usage, they would be authorized to diverge from the initial schedule in order to balance the load whereas they would strictly follow that schedule when memory becomes fully used. We expect that, in an out-of-core context, load balancing will have to take into consideration the possible delays due to *I/O's*. One advantage of the initial schedule is that it has been computed to guarantee the respect of the memory constraints. If we want to maintain that property for asynchronous dynamic executions, we need to employ techniques similar to deadlock avoidance algorithms (where the critical resource would be the memory). Finally, an extensive performance analysis will then have to be performed. After that, we hope to provide a robust software version of that work to `MUMPS` users.

The study presented in Chapter 10 has also shown that the lack of memory requires to schedule the tasks of a parallel execution with respect to a sequence that is close to the one followed in the serial case. Therefore, we hope to be able to adapt other results from Part I to the parallel factorization. More generally, this thesis has shown that memory can be used to limit the *I/O* volume, to ensure coarse-grain parallelism and to dynamically

balance the workload. As a longer term perspective, based on the study of Chapter 9, all these results have to be combined to extend the code of Chapter 10 to the out-of-core management of the stack of contribution blocks.

At last, now that the numerical out-of-core factorization allows us to treat very large problems with a good memory scalability, the memory bottleneck on a large number of processors may be the analysis step which remains serial. To overcome this limit, that step has to be performed in parallel too [4, 43].

Appendix A

Bibliography

Bibliography

[1] The BCSLIB Mathematical/Statistical Library. http://www.boeing.com/phantom/bcslib/. 19

[2] 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. 6

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA., 1983. 6

[4] Patrick R. Amestoy, Alfredo Buttari, and Jean-Yves 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). 160

[5] Patrick R. Amestoy, T. A. Davis, and Iain S. Duff. An approximate minimum degree ordering algorithm. Technical Report TR-94-039, CIS Dept., Univ. of Florida, 1994. Appeared in SIAM J. Matrix Analysis and Applications. 5

[6] Patrick R. Amestoy, T. A. Davis, and Iain S. Duff. Algorithm 837: AMD, an approximate minimum degree ordering algorithm. *ACM Transactions on Mathematical Software*, 33(3):381–388, 2004. 49

[7] Patrick R. Amestoy and Iain S. Duff. Memory management issues in sparse multifrontal methods on multiprocessors. *Int. J. of Supercomputer Applics.*, 7:64–82, 1993. 122

[8] Patrick R. Amestoy, Iain S. Duff, Abdou Guermouche, and Tzvetomila Slavova. Analysis of the out-of-core solution phase of a parallel multifrontal approach. Research report RT/APO/07/3, ENSEEIHT, April 2007. Also appeared as CERFACS and INRIA technical report. 2, 135, 158

[9] Patrick R. Amestoy, Iain S. Duff, J. Koster, and Jean-Yves L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001. 8, 13, 121, 142

[10] Patrick R. Amestoy, Iain S. Duff, and Jean-Yves L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Comput. Methods Appl. Mech. Eng.*, 184:501–520, 2000. 8, 13, 121

[11] Patrick R. Amestoy, Iain S. Duff, Jean-Yves L'Excellent, and Xiaoye S. Li. Analysis, tuning and comparison of two general sparse solvers for distributed memory computers. Technical Report LBNL-45992, NERSC, Lawrence Berkeley National

Laboratory, June 2000. Also France-Berkeley Project final report and ENSEEIHT-IRIT report RT/APO/00/2 and CERFACS report TR/PA/00/72, shortened version appeared in *ACM Trans. Math. Softw.* 121

[12] Patrick R. Amestoy, Iain S. Duff, and Christof Vömel. Task scheduling in an asynchronous distributed memory multifrontal solver. *SIAM Journal on Matrix Analysis and Applications*, 26(2):544–565, 2005. 122

[13] Patrick R. Amestoy, Abdou Guermouche, Jean-Yves L'Excellent, and Stéphane Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel Computing*, 32(2):136–156, 2006. 49, 123, 127, 147

[14] Patrick R. Amestoy and Chiara Puglisi. An unsymmetrized multifrontal LU factorization. *SIAM Journal on Matrix Analysis and Applications*, 24:553–569, 2002. 11, 29

[15] Cleve Ashcraft and Roger G. Grimes. SPOOLES: An object oriented sparse matrix library. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, Texas, March 22–24, 1999. 13

[16] Cleve Ashcraft, Roger G. Grimes, and John G. Lewis. Accurate symmetric indefinite linear equation solver. *SIAM Journal on Matrix Analysis and Applications*, 20:513–561, 1998. 19

[17] Cleve Ashcraft, Roger G. Grimes, John G. Lewis, Barry W. Peyton, and Horst D. Simon. Progress in sparse matrix methods for large linear systems on vector computers. *Int. Journal of Supercomputer Applications*, 1(4):10–30, 1987. 12, 121

[18] 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. *Computer Physics Communications*, 97:1–15, 1996. 123

[19] T. H. Cormen, E. Riccio Davidson, and Siddhartha Chatterjee. Asynchronous buffered computation design and engineering framework generator (abcdefg). In *19th International Parallel and Distributed Processing Symposium (IPDPS'05)*, 2005. 21

[20] O. Cozette. *Contributions systèmes pour le traitement de grandes masses de données sur grappes*. PhD thesis, Université de Picardie Jules Vernes, 2003. 15

[21] Olivier Cozette, Abdou Guermouche, and Gil Utard. Adaptive paging for a multifrontal solver. In *Proceedings of the 18th annual international conference on Supercomputing*, pages 267–276. ACM Press, 2004. 15

[22] Randy H. Katz David A. Patterson, Garth Gibson. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data, p.109-116, June 01-03, 1988, Chicago, Illinois, United States*. 20

[23] T. A. Davis and Iain S. Duff. An unsymmetric-pattern multifrontal method for sparse LU factorization. *SIAM Journal on Matrix Analysis and Applications*, 18:140–158, 1997. 11, 13

[24] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G.-Y. Ng. A column approximate minimum degree ordering algorithm. Technical Report TR-00-005, Computer and Information Sciences Department, University of Florida, Gainesville, FL, 2000. 5

[25] James W. Demmel, S. C. Eisenstat, J. R. Gilbert, Xiaoye 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. 9, 13, 18

[26] James W. Demmel, S. C. Eisenstat, J. R. Gilbert, Xiaoye 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. 114, 115

[27] F. Dobrian. *External Memory Algorithms for Factoring Sparse Matrices*. PhD thesis, Old Dominion University, 2001. 19, 91, 93, 94

[28] F. Dobrian and Alex Pothen. Oblio: a sparse direct solver library for serial and parallel computations. Technical Report xx, Old Dominion University, 2000. 13

[29] F. Dobrian and Alex Pothen. The design of I/O-efficient sparse direct solvers. In *Proceedings of SuperComputing*, 2001. 9

[30] Iain S. Duff and John K. Reid. MA27—a set of Fortran subroutines for solving sparse symmetric sets of linear equations. Technical Report R.10533, AERE, Harwell, England, 1982. 19, 29, 45

[31] Iain S. Duff and John K. Reid. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Transactions on Mathematical Software*, 9:302–325, 1983. 10

[32] Iain S. Duff and John K. Reid. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Transactions on Mathematical Software*, 9:302–325, 1983. 12

[33] Iain S. Duff and John K. Reid. A note on the work involved in no-fill sparse matrix factorization. *IMA Journal of Numerical Analysis*, 18:1145–1151, 1983. 41

[34] Iain S. Duff and John K. Reid. The multifrontal solution of unsymmetric sets of linear systems. *SIAM Journal on Scientific and Statistical Computing*, 5:633–641, 1984. 10

[35] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1991. 60, 61, 110

[36] A. Geist and E. G. Ng. Task scheduling for parallel sparse Cholesky factorization. *Int J. Parallel Programming*, 18:291–314, 1989. 148

[37] A. George and J. W. H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Englewood Cliffs, NJ., 1981. 5

[38] J. A. George. Nested dissection of a regular finite-element mesh. *SIAM J. Numer. Anal.*, 10:345–363, 1971. 5

[39] 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. 6, 8

[40] J. R. Gilbert and E. G. Ng. Predicting structure in nonsymmetric sparse matrix factorizations. In J.R. Gilbert A. George and J.W.H. Liu, editors, *Graph Theory and Sparse Matrix Computations*, pages 107–140. Springer-Verlag NY, 1993. 8, 9, 18, 91

[41] John R. Gilbert and Sivan 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). 17, 18, 19, 89, 94, 117, 157

[42] G. H. Golub and C. F. Van Loan. *Matrix Computations. 3rd ed.* The Johns Hopkins University Press, Baltimore and London, 1996. 8

[43] Laura Grigori, James W. Demmel, and Xiaoye S. Li. Parallel symbolic factorization for sparse LU with static pivoting. *SIAM J. Sci. Comput.*, 29(3):1289–1314, 2007. 160

[44] Abdou Guermouche and Jean-Yves L'Excellent. Constructing memory-minimizing schedules for multifrontal methods. *ACM Transactions on Mathematical Software*, 32(1):17–32, 2006. 1, 16, 52, 53, 54, 55, 71, 178

[45] Abdou Guermouche, Jean-Yves L'Excellent, and G. Utard. Impact of reordering on the memory of a multifrontal solver. *Parallel Computing*, 29(9):1191–1218, 2003. 45, 51, 52, 147

[46] A. Gupta. WSMP: Watson Sparse Matrix Package part i - direct solution of symmetric sparse systems version 1.0.0. Technical Report TR RC-21886, IBM research division, T.J. Watson Research Center, Yorktown Heights, 2000. 13

[47] A. Gupta, F. Gustavson, M. Joshi, G. Karypis, and V. Kumar. Pspases: An efficient and scalable parallel sparse direct solver. Technical report, Department of Computer Science, University of Minnesota and IBM T.J. Watson Research center, 1999. 13

[48] Pascal Hénon, Pierre Ramet, and Jean Roman. PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems. *Parallel Computing*, 28(2):301–321, January 2002. 13

[49] HSL. A collection of Fortran codes for large scale scientific computation, 2000. 13

[50] HSL. A collection of Fortran codes for large scale scientific computation, 2004. 29

[51] 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, September 1998. 5, 71

[52] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–302, 1999. 49

[53] Xiaoye S. Li and James W. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Transactions on Mathematical Software*, 29(2), 2003. 8, 13

[54] J. W. H. Liu. Modification of the minimum degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software*, 11(2):141–153, 1985. 5

[55] 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. 16, 34, 41, 42, 45, 46, 52, 147

[56] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11:134–172, 1990. 6, 7, 9, 41, 52, 90

[57] Esmond G. Ng and Padma Raghavan. Performance of greedy heuristics for sparse Cholesky factorization. *SIAM Journal on Matrix Analysis and Applications*, 20:902–914, 1999. 5, 49

[58] S. Toledo O. Meshar, D. Irony. An out-of-core sparse symmetric-indefinite factorization method. *ACM Transactions on Mathematical Software*, 32(3):445–471, 2006. 18, 19, 91

[59] John K. Reid and Jennifer A. Scott. An out-of-core sparse Cholesky solver. Technical Report RAL-TR-2006-013, Rutherford Appleton Laboratory, 2006. Revised March 2007. 15, 55

[60] John K. Reid and Jennifer A. Scott. HSL_OF01, a virtual memory system in Fortran. Technical report, Rutherford Appleton Laboratory, 2006. 15

[61] E. Rothberg and R. Schreiber. Efficient methods for out-of-core sparse Cholesky factorization. *SIAM Journal on Scientific Computing*, 21(1):129–144, 1999. 15, 16, 19, 128, 159

[62] Edward Rothberg and Stanley C. Eisenstat. Node selection strategies for bottom-up sparse matrix ordering. *SIAM Journal on Matrix Analysis and Applications*, 19(3):682–695, 1998. 5

[63] 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. 18, 19, 94, 117, 157, 158, 159

[64] 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

[65] Olaf Schenk and Klaus Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Journal of Future Generation Computer Systems*, 20(3):475–487, 2004. 13

[66] Olaf Schenk and Klaus Gärtner. On fast factorization pivoting methods for sparse symmetric indefinite systems. *Electronic Transaction on Numerical Analysis*, 23:158–179, 2006. 13

[67] 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. 23

[68] R. Schreiber. A new implementation of sparse Gaussian elimination. *ACM Transactions on Mathematical Software*, 8:256–276, 1982. 6

[69] J. Schulze. Towards a tighter coupling of bottom-up and top-down sparse matrix ordering methods. *BIT*, 41(4):800–841, 2001. 5, 49, 71, 132

[70] 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. 20

[71] S. Toledo and A. Uchitel. A supernodal out-of-core sparse gaussian elimination method. In *Proceedings of PPAM 2007*, 2007. 18, 19

[72] M. Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 2:77–79, 1981. 5

Appendix B

On the shape of the I/O graphs: Formalization

In this appendix we prove Property 2.2 that has been presented and discussed in Chapter 2.

Let us first consider the more general context of any out-of-core application where data is produced and consumed with a stack mechanism (last data produced is consumed first). We use the term *memory* for all data relative to the application (that data may be either in core memory or on disk); and we define a *memory access* as an access to either the core memory or the disk (which implies in this second case a given amount of *I/O*). We have the following result:

Theorem B.1. *Given an out-of-core application which accesses the memory as a stack which is empty both initially and eventually; given a sequence of memory accesses, the optimum volume of* I/O $V^{I/O}$ *as a function of the available memory* M_0 *(*$V^{I/O} = f(M_0)$*) is a piecewise affine function; the steepness of each piece is an integer multiple of* -1 *whose absolute value decreases when the value of* M_0 *increases.*

Proof. The hypothesis that the stack is empty both initially and eventually implies that all data are reused; so any data written to disk will have to be read back. Therefore the volume of writes is equal to the volume of reads and we only count the write operations.

Let us focus on the evolution of the amount of memory (M) relative to the amount of memory accesses ($M_{accessed}$). At the beginning, the amount of memory is zero (stack initially empty). When (say) 1 MB of data is pushed, both the amount of data accessed $M_{accessed}$ (x axis) and the amount of memory M (y axis) increase by 1 MB. When (say) 1 MB of data is popped, the amount of data accessed still increases by 1 MB while the amount of memory decreases by 1 MB. Geometrically, the function $M = f(M_{accessed})$ is a piecewise affine function for which each piece has a steepness equal to 1 (pushes) or -1 (pops); its graph is composed of a succession of peaks and hollows as one can see in Figure B.1. At the end, the amount of memory is zero (stack eventually empty).

A memory access may be defined as a pair (T, Q) where T is the type of access (*push* or *pop*) and Q is the amount of data involved (in MB). From a memory point of view, if n is the number of accesses, such an application is then exactly defined by a sequence $S = ((T_i, Q_i))_{i\in\{1;...;n\}}$ that verifies the two following properties:

$$(\forall j \in \{1; ...; n\})(\sum_{i\in\{1;...;j\}|T_i=push} Q_i \geq \sum_{i\in\{1;...;j\}|T_i=pop} Q_i) \tag{B.1}$$

$$\sum_{i\in\{1;...;n\}|T_i=push} Q_i = \sum_{i\in\{1;...;n\}|T_i=pop} Q_i \tag{B.2}$$

Moreover, even if it means packing consecutive accesses of same type we may suppose without loss of generality that pushes and pops are alternated. Then, we can define a *local peak* P_i (*resp.* a *local hollow*) as two successive memory accesses $(push, Q_{push}), (pop, Q_{pop})$ (*resp.* $(pop, Q_{pop}), (push, Q_{push})$), in this order. We define $\mathcal{P}$ as the (ordered) set of peaks. Note that $\mathcal{P}$ also defines the sequence S.

For a given amount of available physical memory M_0, the (minimum) volume of *I/O* can be directly computed with a greedy algorithm on the sequence S as shown in Algorithm B.1. Each time the memory required exceeds M_0 (after $T_i = push$), we write the bottom of the stack to disk. When a *pop* operation is performed, we read the bottom of the stack only when needed. As earlier, note that since the volume written and read are equal, we only take write operations into account, so that $V^{I/O}$ represents the volume of data *written* to disk.

Input: $S = ((T_i, Q_i))_{i\in\{1;...;n\}}$: Sequence of memory accesses
Input: M_0: Memory available
Output: $V^{I/O}$: *I/O* volume
`% Initialization:`
$current\_mem \leftarrow 0$;
$i \leftarrow 1$;
while $i \leq n$ **do**
  if $T_i = push$ **then**
    `% Memory required is` $current\_mem + Q_i$ `but only` M_0 `is available`
    `% Write the overhead to disk`
    $V^{I/O} \leftarrow V^{I/O} + \max(current\_mem + Q_i - M_0, 0)$;
    $current\_mem \leftarrow \min(current\_mem + Q_i, M_0)$;
  else
    `%` $T_i = pop$
    `% We do not count read operations`
    $current\_mem \leftarrow \min(current\_mem - Q_i, 0)$;
  $i \leftarrow i + 1$

Algorithm B.1: *I/O* volume computation of a sequence of memory accesses S with an available memory M_0.

However the continuity of $V^{I/O}$ with respect to M_0 does not appear obviously with this approach. That is why we first carry out a transformation independent from M_0 which will bring to light the true potential sources of *I/O*.

We revisit the examples of Figure 2.4 to illustrate Algorithm B.1, starting with the sequence (*push*,4); (*pop*,4) (see first picture of Figure B.2(a)). If $M_0 > 4$ (for example $M_0 = 4.5$), no *I/O* will be necessary. If $M_0 = 2$, applying Algorithm B.1 will lead to a volume of *I/O* equal to 2. If now $M_0 = 0.5$, we obtain a volume of *I/O* equal to 3.5. When the physical memory available M_0 decreases, we observe that the maximum volume of *I/O* that we can obtain is 4. We say that we have a *potential of* I/O equal to 4. Indeed on such a sequence the volume of *I/O* will be equal to $\max(4 - M_0, 0)$. If we now consider sequence (b) ((*push*,4); (*pop*,4); (*push*,4); (*pop*,4)) there are two peaks which constitute two potential sources of *I/O*. In that case the volume of *I/O* is equal to $2 \times \max(4 - M_0, 0)$. The potentials of *I/O* corresponding to the two peaks of memory are both equal to 4.

As shown in the two examples above, with each peak i in $\mathcal{P}$ we have associated a *potential of* I/O Pot_i, leading to an overall volume of *I/O* equal to $V^{I/O}(M_0) = \sum_{i \in \mathcal{P}} \max(Pot_i - M_0, 0)$.

Let us now take a slightly more complex example: sequence (*push*,4); (*pop*,2); (*push*,1); (*pop*,3) from Figure B.2(c). In that case, we again start doing *I/O* when the physical memory available M_0 becomes smaller than 4. If $M_0 = 2$, then the first peak $M = 4$ will force us to write 2 MB from the bottom of the stack. Then the memory M decreases until $M = 2$. When M increases again until reaching the second peak $M = 3$, the bottom of the stack is still on disk and no supplementary *I/O* is necessary. Finally M decreases to 0 and the bottom of the stack (2 MB) that was written will be read from disk and consumed by the application. For this value of M_0 (2), the volume of (written) *I/O* is only equal to 2 MB. In fact if $M_0 > 1$ the second peak has no impact on the volume of *I/O*. In this example, even if there are two peaks of sizes 4 MB and 3 MB, we can indeed notice that 2 MB are shared by these two peaks. This common amount of data can only be processed out-of-core once. By trying other values of M_0, we would see that the volume of *I/O* $V^{I/O}(M_0)$ is in fact equal to $\max(4 - M_0, 0) + \max(1 - M_0, 0)$. Therefore we associate a potential of *I/O* of 4 with the first peak but a potential of *I/O* of only 1 to the second. Indeed the potential of *I/O* for the second peak is obtained by subtracting 2 (data common to the two peaks, for which *I/O* is only performed once) to 3 (value of the second peak).

We now describe more precisely the process consisting in replacing peaks by potentials of *I/O*. Each potential of *I/O* is equal to the maximum volume of *I/O* due to each peak. The key point is that each data accessed is attributed to one peak and only one as follows. The first potential source of *I/O*, corresponding to the highest peak, is selected first and receives a potential of *I/O* equal to the memory of this peak. Data corresponding to this peak will be written to disk at most once. But part of these data is shared with other peaks. That is why we carry out a transformation consisting in *subtracting* data shared with other peaks from these other peaks.

Formally, this subtraction process is described by the operation $S' \leftarrow Subtract(S, P_i)$ from Algorithm B.2. For any value of M_0, it is such that $V^{I/O}(S, M_0) = \max(Pot_i - M_0, 0) + V^{I/O}(S', M_0)$, where $Pot_i = \sum_{j=1}^{i} Q_{push_j} - \sum_{j=1}^{i-1} Q_{pop_j}$ is the *potential* associated with P_i. Recall that for this relation to hold, we have to choose P_i as the one that corresponds to the largest volume of memory (or potential), *i.e.* the one first responsible of *I/O* when M_0 decreases. For instance, in example (d) from Figure B.2, applying this subtraction to the peak associated with a memory of 3 MB (instead of the one associated with a memory of 4 MB) would give an incorrect volume of *I/O* equal to $\max(3 - M_0, 0) + \max(2 - M_0, 0)$ (instead of $\max(4 - M_0, 0) + \max(1 - M_0, 0)$), whereas *I/O* clearly starts occurring as soon as M_0 is smaller than 4 MB. Algorithm B.2 is further explained in the caption of Figure B.1 where we unroll it on a more general example. Algorithm B.3 now applies recursively the transformation to the new sequence (after the suitable subtractions). At the end, we have got a series of potentials $(Pot_i)_{i \in \mathcal{P}}$ - that we keep in the same order as the peaks they are associated with for a better readability. We call the result of this recursive transformation the *potential transform*. By construction, and as we have seen on the examples, the volumes of *I/O* for each potential are cumulated, and the total volume of *I/O* is thus given by:

$$V^{I/O}(M_0) = \sum_{i \in \mathcal{P}} \max(Pot_i - M_0, 0). \tag{B.3}$$

To achieve the proof, let us notice that the transformation is independent from M_0 and so a potential Pot_i too. Thus the function $M_0 \mapsto \max(Pot_i - M_0, 0)$ is a piecewise affine with steepness -1 for $M_0 < Pot_i$ and 0 for larger values of M_0. Finally, $M_0 \mapsto V^{I/O}(M_0)$, as the sum of such functions is a piecewise affine function whose pieces have a steepness of decreasing (in absolute value) negative integer values. ■

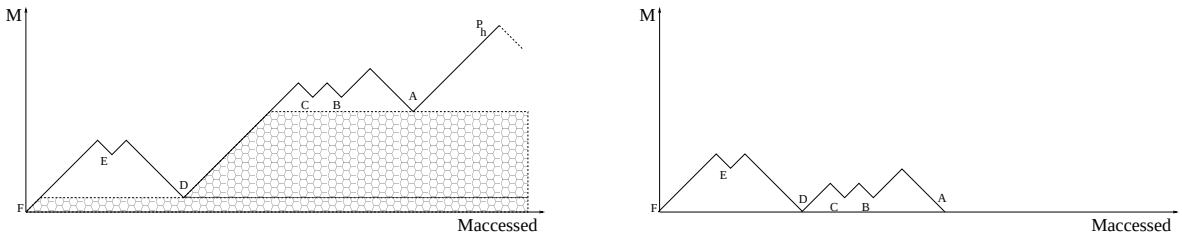
For each example from Figure B.2, we unroll the algorithm and successively replace the largest peak by a potential of *I/O* equal to the memory associated with that peak. We represent each potential of *I/O* obtained by a vertical bar. At the end of the transformation, all the peaks have been replaced by their respective potentials as shown in the third picture of the figure.

Finally (for each example), the subsequent volume of *I/O* is illustrated by the fourth series of pictures of Figure B.2. This result may be interpreted from a geometric point of view. The steepness of the graph of the function $V^{I/O}(M_0)$ for a given value M_0 is the number of potentials crossed by the horizontal line M_0. For instance, with sequences (c) and (d) (that have the same *potential transform*), if the amount M_0 of available memory is more than 4 (say equal to 4.5), the corresponding horizontal line (says 4.5) does not cross any potential: no *I/O* is required. If M_0 is between 1 and 4 (says 2), the horizontal line (say 2) crosses one potential: the steepness is one. In other words, locally, the volume of *I/O* grows as fast as the memory decreases. Finally, when M_0 is less than 1 (say 0.5), the horizontal line (say 0.5) crosses two potentials: the steepness is two. The volume of

I/O grows *twice* as fast as the physical memory available decreases.

Input: $S = (P_1, \ldots, P_n)$: The sequence of memory accesses as a list of *local peaks*
Input: $P_h = (push, Q_{push_h}), (pop, Q_{pop_h})$: A *local peak* to subtract from the sequence
Output: S': Sequence of memory accesses after subtraction of peak P_h
% Recompute potential of P_h
$Pot_h \leftarrow \sum_{i=1}^{h} Q_{push_i} - \sum_{i=1}^{h-1} Q_{pop_i}$ % Pop P_h from the sequence:
$S' \leftarrow S \setminus P_h$;
$pos\_current\_peak \leftarrow h$;
% Step (1) - Decrease peaks prior to P_h and sharing data with it:
$current\_hollow \leftarrow Pot_h - Q_{push_h}$;
$lower\_hollow \leftarrow current\_hollow$;
% While there are data shared with other peaks
while $lower\_hollow > 0$ **do**
    % Look for the previous peak
    $pos\_current\_peak \leftarrow pos\_current\_peak - 1$;
    % Evaluate its local hollow
    $current\_hollow = current\_hollow + Q_{pop_{pos\_current\_peak}} - Q_{push_{pos\_current\_peak}}$;
    % If there is shared data with h
    if $current\_hollow < lower\_hollow$ **then**
        % Subtract shared data from current peak
        $Q_{push_{pos\_current\_peak}} \leftarrow Q_{push_{pos\_current\_peak}} + lower\_hollow - current\_hollow$;
        % Update $lower\_hollow$ value
        $lower\_hollow \leftarrow current\_hollow$;
% Step (2) - Decrease peaks that are after P_h and that share data with P_h:
...;
% similar to step (1) except that we decrease Q_{pop} values
...;

Algorithm B.2: Subtraction of a peak from a sequence of memory accesses: $S' \leftarrow$ *Subtract(S,*P_h*)*. Only the treatment of the peaks before P_h are examined (step (1)); peaks after P_h would be processed similarly (step (2), not presented).



(a) Initial sequence (b) Sequence after subtract

Figure B.1: Illustration of Algorithm B.2 on a toy sequence of memory accesses and the subtraction of its highest peak, P_h. Initially the highest peak is subtracted and *current_hollow* and *lower_hollow* are equal to A. Next *current_hollow* $= B$ but *lower_hollow* does not change because $B > A$. When *current_hollow* $= D$, an amount of data equal to $A - D$ is subtracted from the *push* operation that follows and the value of *lower_hollow* is set to D. Then *current_hollow* $= E$ but *lower_hollow* does not change as $E > D$. Finally *current_hollow* is equal to F and this induces the subtraction of an amount of data equal to $D - F$ from the corresponding peak (and from the *push* operation that follows F). Note that we only illustrated the process for peaks that are before the subtracted peak (point (1) of the algorithm).

Input: $S = (P_1, \ldots, P_n)$: sequence of memory accesses as a list of *local peaks*
Output: T: *Potential transform* as a list of potentials
`% Initialization`
$T = \emptyset$;
`% Main loop`
while $S \neq \emptyset$ **do**
  `% Find the highest local peak` P_h`, of potential` Pot_h`:`
  $Pot_h = \max_{h=1..n} \sum_{i=1}^{h} Q_{push_i} - \sum_{i=1}^{h-1} Q_{pop_i}$;
  `% Add its potential to the list of potentials:`
  $T \leftarrow Pot_h :: T$;
  `% Subtract` P_h `from` S`:`
  $S \leftarrow$ Subtract(S,P_h);

Algorithm B.3: Computing the *potential transform* of a sequence of memory accesses: *Transform(S)*

We consider the multifrontal method with a *classical* assembly scheme and a terminal allocation as presented in Chapter 2 and now prove Property 2.2 that we recall here and that can be considered as a corollary of Theorem B.1:

Property 2.2. *For a given postorder of the* etree*, 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*

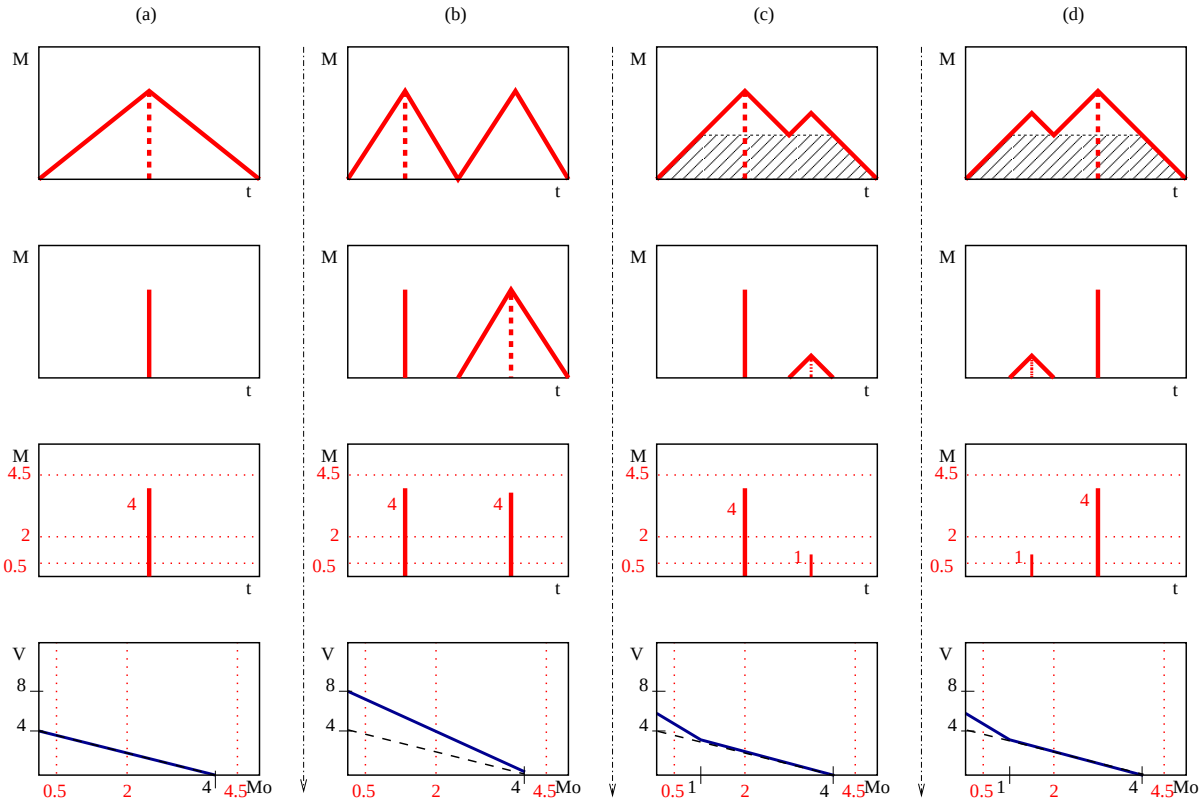


Figure B.2: Computing the *potential transform* and deducing the I/O volume: four instructive examples. On each column, corresponding to a given sequence of memory accesses, the transformation is unrolled on the first three pictures (the potentials are the vertical bars) and the deduced I/O volume V (as a function of the memory available M_0) is given by the fourth one (the lower bound function "peak - M_0" is there represented with dashed line).

absolute value decreases when the value of M_0 increases.

Proof. In this proof, we show that the active storage of the multifrontal factorization can match the *memory* as defined in Theorem B.1.

In practice, we do not have a pure stack mechanism in the multifrontal factorization: a frontal matrix is first allocated; the contribution blocks of the children are then consumed; the frontal matrix is factored and its factors are stored to disk; and finally the contribution block of the active frontal matrix is moved to the top of the stack.

However, the key point is that we may assess that the whole active storage is accessed as a stack without modifying the volume of *I/O*. Indeed, when a frontal matrix is just factored, we may consider that we pop this complete frontal matrix as well as all the contribution blocks of its children and that we finally push its own contribution block. Because we have the assumption that a frontal matrix holds in-core, this involves the same amount of *I/O* as the real mechanism of the method.

Therefore, considering that (i) we may assess that the active storage is accessed as a stack, that (ii) the active storage is empty both initially and eventually (any contribution block or frontal matrix will be reused during the factorization step and popped), that (iii) the sequence of accesses does not depend on M_0 (the postorder is given), and that (iv) the volume of *I/O* performed is minimum (use of Property 2.1), we can apply Theorem B.1. ■

Note that the *potential transform* also easily gives the volume of accesses to the active storage: it is the sum of the potentials and it is also equal to the volume of *I/O* when M_0 tends to 0. However this model can only be applied to our application if M_0 remains larger than the largest frontal matrix. When a frontal matrix cannot fit in-core (because its size is larger than M_0) we have no more guarantee that we may respect a *read-once / write-once* scheme. For such values of M_0, the volume of *I/O* computed with this model becomes a lower bound of the actual volume of *I/O*. Subsequently, the sum of the potentials becomes a lower bound on the amount of data accessed.

Appendix C

Complements to Chapter 4

In this appendix, we prove two results related to Chapter 4. In Section C.1, we prove that the decision problem associated with a knapsack that can spill out is NP-complete. In Section C.2, we show the use of a multiple allocation does not improve the I/O volume compared to a flexible allocation scheme and that it does not reduce the complexity of the I/O minimization problem either.

C.1 The decision problem associated with a knapsack that can spill out is NP-complete

In this section, we prove Lemma 4.2 which is used in Chapter 4 to show that the decision problem associated with the minimization of the I/O volume of the multifrontal method with a flexible allocation scheme is NP-complete. The proof consists in a reduction from Partition to Knapsack-Spill-Dec. We recall those two decision problems and propose a proof of our lemma.

Problem 4.2 (Knapsack-Spill-Dec). We have n items $(1, \ldots, n)$. Each item j has a value p_j and a cost c_j such that: $0 \leq c_j \leq p_j$. We moreover assume that the capacity V of the bag is limited: $0 \leq V \leq \sum_{i=1}^{n} p_i - \max_{i=1}^{n} p_i$. Can we achieve an algebraic benefit K? Or, formally, is the following assertion true: $(\exists \mathcal{S} \subset \{1; \ldots; n\}) \left(\min(\sum_{j \in \mathcal{S}} p_j, V) - \sum_{j \in \mathcal{S}} c_j \geq K \right)$?

Problem 4.3 (Partition). Given n positive integer numbers $(x_1, \ldots, x_n)$ of half-sum $X = \frac{\sum_{j=1}^{n} x_j}{2}$, is there a subset $\mathcal{S}$ such that $\sum_{j \in \mathcal{S}} x_j = X$?

Lemma 4.2. *Problem Knapsack-Spill-Dec is NP-complete.*

Proof. First, Knapsack-Spill-Dec belongs to NP. If we are given a subset $\mathcal{S}$ whose algebraic benefit is greater than or equal to K, we can evaluate in polynomial time in the size of the instance that amount $\min \left(\sum_{j \in \mathcal{S}} p_j, V \right) - \sum_{j \in \mathcal{S}} c_j$ and check that it is indeed greater than or equal to K.

To prove the NP-completeness of Knapsack-Spill-Dec, we show that Partition can be polynomially reduced to Knapsack-Spill-Dec. We consider an arbitrary instance I_1 of Partition with integer numbers $(x_1, \ldots, x_n)$ of half-sum $X = \frac{\sum_{j=1}^{n} x_j}{2}$. We assume $\max_{i=1}^{n} x_i \leq X$ (otherwise we know there is no solution). We build an instance I_2 of Knapsack-Spill-Dec as follows. We consider n items of respective value p_i and respective cost c_i such that: $p_i = 2x_i, c_i = x_i, 1 \leq i \leq n$. We also let $V = 2X$ and $K = X$. The construction of I_2 is polynomial (and even linear) in the size of I_1. I_2 is effectively an instance of Knapsack-Spill-Dec since $p_i \geq c_i$ and $0 \leq V \leq \sum_{i=1}^{n} p_i - \max_{i=1}^{n} p_i$. Indeed, the first inequality stands because $2x_i \geq x_i$ and the second one can be established as follows:

$$\max_{i=1}^{n} p_i = 2 \max_{i=1}^{n} x_i \leq 2X = 4X - 2X = \sum_{j=1}^{n} p_j - V.$$

Finally, I_1 has a solution if and only if I_2 has a solution. First, let $\mathcal{S}$ be a solution of I_1: $\sum_{j \in \mathcal{S}} x_j = X$. We deduce that $\mathcal{S}$ is also a solution of I_2 as follows:

$$\min\left(\sum_{j \in \mathcal{S}} p_j, V\right) - \sum_{j \in \mathcal{S}} c_j = \min(2X, 2X) - X = X = K \geq K.$$

Let now $\mathcal{S}$ be a solution of I_2: $\min\left(\sum_{j \in \mathcal{S}} p_j, V\right) - \sum_{j \in \mathcal{S}} c_j \geq K$. We reword this statement with equivalent forms in the following formulas:

$$\sum_{j \in \mathcal{S}} p_j \geq \sum_{j \in \mathcal{S}} c_j + K \quad \& \quad V \geq \sum_{j \in \mathcal{S}} c_j + K \tag{C.1}$$

$$2\sum_{j \in \mathcal{S}} x_j \geq \sum_{j \in \mathcal{S}} x_j + X \quad \& \quad 2X \geq \sum_{j \in \mathcal{S}} x_j + X \tag{C.2}$$

$$\sum_{j \in \mathcal{S}} x_j \geq X \quad \& \quad \sum_{j \in \mathcal{S}} x_j \leq X \tag{C.3}$$

Hence, $\sum_{j \in \mathcal{S}} x_j = X$; which exactly means that $\mathcal{S}$ is a solution to I_1. ■

C.2 Multiple allocation

The flexible allocation was initially designed in [44] to decrease the storage requirement by exploiting the freedom that we have to decide of the moment of the parent allocation. The purpose of this chapter is to extend this freedom to the out-of-core case in order to limit the *I/O* volume. We have assumed that, as in the in-core case, once the frontal matrix of the parent has been allocated (after child p has been processed), each supplementary child ($j > p$) is assembled into the parent on the fly. As explained in Section 4.3, if such a child cannot be processed in-core together with the frontal matrix of the parent, then part of that frontal matrix (or that whole frontal matrix) has to be written to disk in order to make room and process the child with a maximum of

available memory. If we want to perform the assembly on the fly, we need to read back the frontal matrix from disk. However, in an out-of-core context, we might prefer not to read it back directly but waiting to have accumulated more contribution blocks before performing their assembly. For instance, if we have a family that contains 20 children, we might imagine to process them 5 by 5, or, said differently, consuming the accumulated contribution blocks each time 5 children have been processed. We name such a mechanism a multiple allocation scheme. This mechanism may appear as a natural extension to the flexible allocation scheme. Let us consider a family composed of n children. We assume that the frontal matrix of the parent is allocated q times on this family. We note $\mathcal{S}^k$ the subset of the children which are processed between the $(i-1)^{th}$ and the i^{th} allocation; their contribution blocks are thus all assembled into the parent at the moment of the i^{th} allocation. The subsets $\mathcal{S}^1$,..., $\mathcal{S}^q$ form a disjoint union of the set $\{1,\ldots,n\}$ of children: $\{1,\ldots,n\} = \coprod_{k=1}^{q} \mathcal{S}^k$. A flexible allocation is exactly a multiple allocation for which $\mathcal{S}^2$,...,$\mathcal{S}^q$ are singletons. In this sense, the multiple allocation generalizes the flexible allocation. However, the use of a multiple allocation does not improve the *I/O* volume compared to a flexible allocation scheme as we now show.

Lemma C.1. *Let us consider a family processed with a given multiple allocation configuration (we are provided the decomposition of the children $(\mathcal{S}^k)_{k=1,q}$) and one subset $\mathcal{S}^k$ composed of at least two children. If the last child processed in $\mathcal{S}^k$ cannot fit in core with the parent, then we can move the first child to subset $\mathcal{S}^1$ without increasing the total volume of* I/O*.*

Proof. We name k_1 the first child of subset $\mathcal{S}^k$. If we remove this child from the subset, it is immediate to see that the peak of storage related to the treatment of the children of $\mathcal{S}^k$ decreases of at least cb_{k_1}. Since the last child of $\mathcal{S}^k$ cannot fit in core with the parent, the decrease of the peak of storage exactly corresponds to a decrease of the *I/O* volume of the same amount. In other words, the contribution of k_1 to the *I/O* volume of $\mathcal{S}^k$ is as least equal to cb_{k_1}. On the other hand, according to Lemma 4.1, the maximum contribution to the *I/O* volume of this child is equal to cb_{k_1} if it is processed before the first parent allocation, *i.e.*, in $\mathcal{S}^1$. Therefore, moving the child from $\mathcal{S}^k$ to $\mathcal{S}^1$ does not increase the *I/O* volume. ■

Theorem C.1. *The use of a multiple allocation does not improve the* I/O *volume compared to a flexible allocation scheme.*

Proof. We exhibit an algorithm which takes in entry a family and a multiple allocation configuration (given by the decomposition of the children $(\mathcal{S}^k)_{k=1,q}$) inducing an *I/O* volume $V^{multiple}$ and which computes a flexible allocation configuration (expressed as the subset $\mathcal{S}$ of the children that have to be processed after the parent allocation) inducing an *I/O* volume $V^{flexible}$ lower than or equal to $V^{multiple}$. To do so, we consider a subset $\mathcal{S}^k$ $(k > 1)$ and we reduce it to a singleton without increasing the *I/O* volume, the output$\mathcal{S}$ being the union of these singletons. First, let us notice that Property 4.1 can be expressed as follows in a multiple allocation context: any child j that would fit in core with the parent (it verifies $S_j^{multiple} + m \leq M_0$ if we note $S_j^{multiple}$ the storage required to process

the subtree whose root is j) should be processed in a singleton (since it does not induce I/O this way). Therefore, even if it requires a first step which discards such children, we can assume that all the children verify: $S_j^{multiple} + m > M_0$. In particular, we suppose that $\mathcal{S}^k$ is composed of n_k children processed in the order $k_1, k_2, \ldots, k_{n_k}$ and we thus have $S_{k_{n_k}}^{multiple} + m > M_0$. According to Lemma C.1, we can move children $k_1, k_2, \ldots, k_{n_k} - 1$ to $\mathcal{S}^1$ without increasing the I/O volume. The subset $\mathcal{S}^k$ has been reduced to a singleton without increasing the I/O volume, which ends up the proof. ■

Even if the multiple allocation does not allow to improve the I/O volume compared to a flexible allocation, this scheme could remain interesting if it allowed to find an optimum solution for the I/O minimization problem more easily than in the flexible case. However, the related decision problem, Multiple-MinIO-Dec, remains NP-complete as we now prove.

Theorem C.2. *Problem Multiple-MinIO-Dec is NP-complete.*

Proof. We let the reader check that Multiple-MinIO-Dec belongs to NP and we show that it is NP-hard. To do so, we assume to the contrary that there exists an algorithm $\mathcal{A}_1$ that can solve Problem Multiple-MinIO-Dec in polynomial time in the size of the instance given in entry. Let I_1 be an instance of Flex-MinIO-Dec from which we canonically build an instance I_2 of Multiple-MinIO-Dec. We apply $\mathcal{A}_1$ to I_1 followed by the algorithm presented in the proof of Theorem C.1 (that we name $\mathcal{A}_2$). Algorithm $\mathcal{A}_2$ also has a complexity polynomial (even linear) in the size of the instance and thus $\mathcal{A}_2 \circ \mathcal{A}_1$ too. But, according to the proof of Theorem C.1, $\mathcal{A}_2 \circ \mathcal{A}_1$ solves Flex-MinIO-Dec. This is a contradiction. ■

In-place assembly schemes

Theorems C.1 and C.2 apply to both *last-in-place* and *max-in-place* schemes: the proofs presented in this section are independent of the assembly scheme before the parent allocation.

Lastly, these theorems also apply to the *in-place-realloc* scheme as we now explain. When the algorithm presented in the proof of Theorem C.1 is applied, the children that are processed after the parent allocation of the final flexible family correspond to the ones that are processed at the moment of a parent allocation or reallocation of the initial family. Therefore, the extra-I/O volume due to the parent allocations will be reduced of the same amount in both configurations.

Résumé :

La factorisation d'une matrice creuse est une approche robuste pour la résolution de systèmes linéaires creux de grande taille. Néanmoins, une telle factorisation est connue pour être coûteuse aussi bien en temps de calcul qu'en occupation mémoire. Quand l'espace mémoire nécessaire au traitement d'une matrice est plus grand que la quantité de mémoire disponible sur la plate-forme utilisée, des approches dites hors-mémoire (out-of-core) doivent être employées : les disques étendent la mémoire centrale pour fournir une capacité de stockage suffisante. Dans cette thèse, nous nous intéressons à la fois aux aspects théoriques et pratiques de telles factorisations hors-mémoire. Les environnements logiciel `MUMPS` et `SuperLU` sont utilisés pour illustrer nos discussions sur des matrices issues du monde industriel et académique. Tout d'abord, nous proposons et étudions dans un cadre séquentiel différents modèles hors-mémoire qui ont pour but de limiter le surcoût dû aux transferts de données entre la mémoire et les disques. Pour ce faire, nous revisitons les algorithmes qui ordonnancent les opérations de la factorisation et proposons de nouveaux schémas de gestion mémoire s'accommodant aux contraintes hors-mémoire. Ensuite, nous nous focalisons sur une méthode de factorisation particulière, la méthode multifrontale, que nous poussons aussi loin que possible dans un contexte parallèle hors-mémoire. Suivant une démarche pragmatique, nous montrons que les techniques hors-mémoire permettent de résoudre efficacement des systèmes linéaires creux de grande taille. Quand seuls les facteurs sont stockés sur disque, une attention particulière doit être portée aux données temporaires, qui restent en mémoire centrale. Pour faire décroître efficacement l'occupation mémoire associée à ces données temporaires avec le nombre de processeurs, nous repensons l'ordonnancement de la factorisation parallèle hors-mémoire dans son ensemble.

Mots-clés :

Matrices creuses, factorisation hors-mémoire, méthodes multifrontales, méthodes supernodales, scalabilité mémoire, ordonnancement.

Abstract:

Factorizing a sparse matrix is a robust way to solve large sparse systems of linear equations. However such an approach is known to be costly both in terms of computation and storage. When the storage required to process a matrix is greater than the amount of memory available on the platform, so-called out-of-core approaches have to be employed: disks extend the main memory to provide enough storage capacity. In this thesis, we investigate both theoretical and practical aspects of such out-of-core factorizations. The `MUMPS` and `SuperLU` software packages are used to illustrate our discussions on real-life matrices. First, we propose and study various out-of-core models that aim at limiting the overhead due to data transfers between memory and disks on uniprocessor machines. To do so, we revisit the algorithms to schedule the operations of the factorization and propose new memory management schemes to fit out-of-core constraints. Then we focus on a particular factorization method, the multifrontal method, that we push as far as possible in a parallel out-of-core context with a pragmatic approach. We show that out-of-core techniques allow to solve large sparse linear systems efficiently. When only the factors are stored on disks, a particular attention must be paid to temporary data, which remain in core memory. To achieve a high scalability of core memory usage, we rethink the whole schedule of the out-of-core parallel factorization.

Keywords:

Sparse matrices, out-of-core factorization, multifrontal methods, supernodal methods, memory scalability, scheduling.