Memory and performance issues in parallel multifrontal factorizations and triangular solutions with sparse right-hand sides

François-Henry Rouet

Université de Toulouse

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

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

Présentée et soutenue par :

François-Henry Rouet

Le 17 Octobre 2012

Titre :

Memory and performance issues in parallel multifrontal factorizations and triangular solutions with sparse right-hand sides

---

Problèmes de mémoire et de performance de la factorisation multifrontale parallèle et de la résolution triangulaire à seconds membres creux

---

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

Unité de recherche :
IRIT - UMR 5505

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

Rapporteurs :
Esmond Ng (Lawrence Berkeley National Laboratory)
Sivan Toledo (Tel-Aviv University)

Autres membres du jury :
Iain Duff (Rutherford Appleton Laboratory)
François Pellegrini (Laboratoire Bordelais de Recherche en Informatique)

Résumé

Nous nous intéressons à la résolution de systèmes linéaires creux de très grande taille sur des machines parallèles. Dans ce contexte, la mémoire est un facteur qui limite voire empêche souvent l'utilisation de solveurs directs, notamment ceux basés sur la méthode multifrontale. Cette étude se concentre sur les problèmes de mémoire et de performance des deux phases des méthodes directes les plus coûteuses en mémoire et en temps : la factorisation numérique et la résolution triangulaire. Dans une première partie nous nous intéressons à la phase de résolution à seconds membres creux, puis, dans une seconde partie, nous nous intéressons à la scalabilité mémoire de la factorisation multifrontale.

La première partie de cette étude se concentre sur la résolution triangulaire à seconds membres creux, qui apparaissent dans de nombreuses applications. En particulier, nous nous intéressons au calcul d'entrées de l'inverse d'une matrice creuse, où les seconds membres et les vecteurs solutions sont tous deux creux. Nous présentons d'abord plusieurs schémas de stockage qui permettent de réduire significativement l'espace mémoire utilisé lors de la résolution, dans le cadre d'exécutions séquentielles et parallèles. Nous montrons ensuite que la façon dont les seconds membres sont regroupés peut fortement influencer la performance et nous considérons deux cadres différents : le cas hors-mémoire (*out-of-core*) où le but est de réduire le nombre d'accès aux facteurs stockés sur disque, et le cas en mémoire (*in-core*) où le but est de réduire le nombre d'opérations. Finalement, nous montrons comment améliorer le parallélisme.

Dans la seconde partie, nous nous intéressons à la factorisation multifrontale parallèle. Nous montrons tout d'abord que contrôler la mémoire active spécifique à la méthode multifrontale est crucial, et que les techniques de répartition (*mapping*) classiques ne peuvent fournir une bonne scalabilité mémoire : le coût mémoire de la factorisation augmente fortement avec le nombre de processeurs. Nous proposons une classe d'algorithmes de répartition et d'ordonnancement conscients de la mémoire (*memory-aware*) qui cherchent à maximiser la performance tout en respectant une contrainte mémoire fournie par l'utilisateur. Ces techniques ont révélé des problèmes de performances dans certains des noyaux parallèles denses utilisés à chaque étape de la factorisation, et nous avons proposé plusieurs améliorations algorithmiques.

Les idées présentées tout au long de cette étude ont été implantées dans le solveur MUMPS (Solveur MUltifrontal Massivement Parallèle) et expérimentées sur des matrices de grande taille (plusieurs dizaines de millions d'inconnues) et sur des machines massivement parallèles (jusqu'à quelques milliers de coeurs). Elles ont permis d'améliorer les performances et la robustesse du code et seront disponibles dans une prochaine version. Certaines des idées présentées dans la première partie ont également été implantées dans le solveur PDSLin (solveur linéaire hybride basé sur une méthode de complément de Schur).

Mots-clés : matrices creuses, méthodes directes de résolution de systèmes linéaires, méthode multifrontale, graphes et hypergraphes, calcul haute performance, calcul parallèle, ordonnancement.

Abstract

We consider the solution of very large sparse systems of linear equations on parallel architectures. In this context, memory is often a bottleneck that prevents or limits the use of direct solvers, especially those based on the multifrontal method. This work focuses on memory and performance issues of the two memory and computationally intensive phases of direct methods, namely, the numerical factorization and the solution phase. In the first part we consider the solution phase with sparse right-hand sides, and in the second part we consider the memory scalability of the multifrontal factorization.

In the first part, we focus on the triangular solution phase with multiple sparse right-hand sides, that appear in numerous applications. We especially emphasize the computation of entries of the inverse, where both the right-hand sides and the solution are sparse. We first present several storage schemes that enable a significant compression of the solution space, both in a sequential and a parallel context. We then show that the way the right-hand sides are partitioned into blocks strongly influences the performance and we consider two different settings: the out-of-core case, where the aim is to reduce the number of accesses to the factors, that are stored on disk, and the in-core case, where the aim is to reduce the computational cost. Finally, we show how to enhance the parallel efficiency.

In the second part, we consider the parallel multifrontal factorization. We show that controlling the active memory specific to the multifrontal method is critical, and that commonly used mapping techniques usually fail to do so: they cannot achieve a high memory scalability, i.e., they dramatically increase the amount of memory needed by the factorization when the number of processors increases. We propose a class of memory-aware mapping and scheduling algorithms that aim at maximizing performance while enforcing a user-given memory constraint and provide robust memory estimates before the factorization. These techniques have raised performance issues in the parallel dense kernels used at each step of the factorization, and we have proposed some algorithmic improvements.

The ideas presented throughout this study have been implemented within the MUMPS (MUltifrontal Massively Parallel Solver) solver and experimented on large matrices (up to a few tens of millions unknowns) and massively parallel architectures (up to a few thousand cores). They have demonstrated to improve the performance and the robustness of the code, and will be available in a future release. Some of the ideas presented in the first part have also been implemented within the PDSLin (Parallel Domain decomposition Schur complement based Linear solver) package.

Keywords: sparse matrices, direct methods for linear systems, multifrontal method, graphs and hypergraphs, high-performance computing, parallel computing, scheduling.

Acknowledgements

First of all, I would like to express my sincere gratitude to my advisors, Patrick Amestoy and Alfredo Buttari, as well as Jean-Yves L'Excellent and Bora Uçar who followed my work very closely. Thank you for encouraging and guiding my research, and for the tremendous amount of work that you put into this thesis. Collaborating with people like you is priceless for a student, and it was a privilege for me to work and learn from you. Again, *merci*!

I am grateful to all the people with whom I have collaborated during these three years. Firstly, I want to thank Abdou Guermouche and Emmanuel Agullo who revealed some of the magic behind scheduling algorithms. I want to thank Iain Duff for his help on many aspects of my work and for perfecting my English. I am also grateful to Sherry Li and Esmond Ng for working with me and hosting me twice at the Berkeley Lab. I also thank Ichitaro Yamazaki, Laurent Bouchet, Kamer Kaya, Yves Robert, Erik Boman, Siva Rajamanickam, and Nicolas Renon.

I would like to thank the referees of this dissertation, Esmond Ng and Sivan Toledo, for reviewing and commenting the manuscript, and for travelling overseas to attend the defense. I also thank the other members of the committee, Iain Duff and François Pellegrini.

I certainly owe many thanks to the people who populated my work environment, starting with the young folks, especially my office mates Clément Climoune Weisbecker, Mohamed Zen Zenadi and Florent La Flop Lopez, as well as Gaëtan La Gat André. Thank you for your support, you never-failing good humor and the (countless. . .) games of Blobby Volley and foosball! Many thanks to Guillaume La Josse Joslin and Mohamed La Wiss Sid-Lakhdar as well. I am grateful to many people at the IRIT Lab, including Chiara Puglisi, Daniel Ruiz, Serge Gratton, and the invaluable Sylvie Armengaud and Sylvie Eichen. I also thank many of my friends in Toulouse and elsewhere for their support; in a random order: Geoffroy, Thibault, Andra, Vivian, Benoît, Robert, Fred, Paul, David, Pascal, Erwan, Jean-Christophe and many others. . .

Finally, I am deeply grateful to my parents, my sister and my brother for their support during all these years of studies, that may have seem long to them!

Contents

Chapter 1

General introduction

1.1 General context

1.1.1 Sparse linear systems and direct methods

We consider the solution of linear systems of the form

$$Ax = b$$

where A is a large square sparse matrix, b is the right-hand side (possibly with several columns) and x is the unknown. In Wilkinson's definition, a sparse matrix is any matrix with enough zeros that it pays to take advantage of them . Sparse matrices appear in numerous scientific applications (mechanics, fluid dynamics, quantum chemistry, data analysis, optimization…). In physical applications, sparsity is often due to the fact that the matrix represents loosely coupled data or a discretized physical domain where a limited-range interaction takes place; this implies that some unknowns of the linear system do not interact with each other, resulting in structural zeros in the matrix. Solving sparse linear systems is often a keystone in numerical simulations; nowadays, sparse linear systems from practical applications commonly have millions of unknowns, sometimes billions.

There are two main classes of methods for the solution of sparse linear systems: *direct methods*, that are based on a factorization of A (e.g., $L\,U$ or $Q\,R$), and *iterative methods*, that build a sequence of iterates that hopefully converges to the solution. Direct methods are acknowledged for their numerical stability but often have large memory and computational requirements, while iterative methods are less memory demanding and often faster but less robust in general; the choice of a method is usually complicated since it depends on many parameters such as the properties of the matrix and the application. *Hybrid methods*, that aim at combining the strengths of both classes, are increasingly popular.

In this dissertation, we focus on direct methods that rely on Gaussian elimination, i.e., algorithms that compute a factorization of the matrix A under the form $L\,U$ (in the general case), $L\,D\,L^T$ (in the symmetric case) or $L\,L^T$ (in the symmetric positive definite case, *Cholesky factorization*). We provide in Algorithm 1.1 a simplistic sketch of $L\,U$ factorization in the dense case, i.e., the nonzero pattern of the matrix is not taken into account. We ignore pivoting and numerical issues as it is not our point in this section and we assume that any entry found on the diagonal is nonzero. Each step i in the factorization consists of *eliminating* a pivot (the diagonal entry a_{ii}), which yields a new column in L and a new row in U, and in modifying the trailing part of the matrix by performing a rank-one update. These two operations are denoted by **Factor** and **Update** in the algorithm. The approach presented in Algorithm 1.1 is a *right-looking* approach, which means that as soon as a column of the L factors is computed (**Factor**), an update of the trailing part (columns on the right of column i) is performed (**Update**). The computations can be reorganized to obtain a *left-looking* approach, in which, at every step, **Update** is performed before **Factor**; in this case, **Update** is performed using the columns that are already computed, that lie on the left of column i.

Algorithm 1.1 Dense LU factorization.

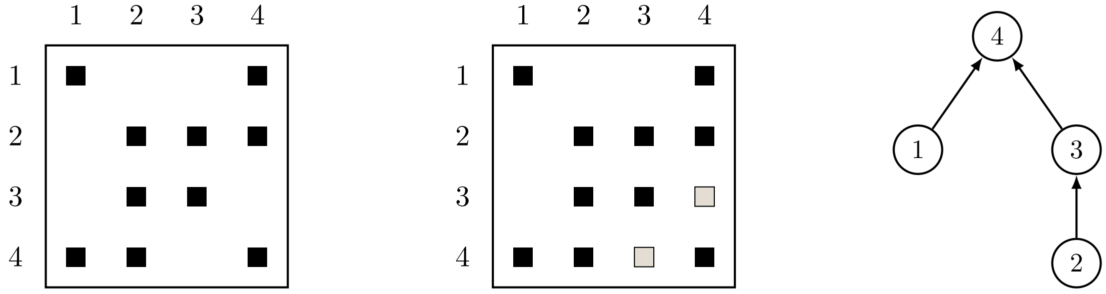
```
/* Input: a square matrix $A$ of size $n$; $A = [a_{ij}]_{i=1:n,j=1:n}$      */
/* Output: $A$ is replaced with its $LU$ factors                               */

for $k = 1$ to $n$ do
    Factor: $a_{k+1:n\ k}\quad \frac{a_{k+1:n,k}}{a_{kk}}$
    Update: $a_{k+1:n\ k+1:n}\quad a_{k+1:n\ k+1:n}\quad a_{k+1:n\ k} \times a_{k\ k+1:n}$
end for
```

In *sparse* direct methods, a key issue is the *fill-in* phenomenon; the **Update** operation can be written as

$$i\ \ j > k\ \ a_{ij}\quad a_{ij}\quad a_{ik}\ \ a_{kj}$$

If $a_{ij} = 0$ but $a_{ik} = 0$ and $a_{kj} = 0$, then a zero entry in the initial matrix becomes nonzero in the factors; in the end, the nonzero pattern of the factors is a superset of the nonzero pattern of the initial matrix. We illustrate this phenomenon in Figure 1.1, where we show the pattern of an initial matrix (a) and the pattern of its factors (b); elements (4 3) and (3 4) are created when eliminating the second pivot. Sparse direct methods exploit the nonzero pattern of the matrix to compute a graph that represents the nonzero structure of the factors and is used as a task graph during the factorization.

We give a formal presentation of this idea in the context of the multifrontal method in Section 1.2. Here, we provide a very simple example that we use to highlight the different variants of sparse Gaussian elimination. Consider the example in Figure 1.1. One can see that since $a_{12} = 0$ and $a_{13} = 0$, the **Update** operation performed when eliminating the first pivot (a_{11}) does not modify columns 2 and 3. Symmetrically, since $a_{21} = 0$ and $a_{31} = 0$, the elimination of a_{11} does not modify rows 2 and 3; therefore, one is free to eliminate pivots a_{22} and a_{33} before or after pivot a_{11}, without any incidence on the result. On the contrary, the second pivot must be eliminated before the third pivot (because $a_{23} = 0$ and $a_{32} = 0$). Similarly, the fourth pivot must be eliminated after all the other variables. This yields a dependency graph that we show in Figure 1.1(c).



(a) Initial matrix.

(b) Filled matrix.

(c) Task graph.

Figure 1.1: Nonzero pattern of a sparse matrix (a), nonzero pattern of its factors where the *fill-in entries* are shaded (b), and task graph of the factorization (c).

Two main classes of sparse direct methods exist. *Supernodal methods* are the natural extension of the left-looking and right-looking methods to sparse matrices; at each node in the graph, some rows and columns of the factors are computed and stored, and are used at some other nodes in the graph to perform updates. In a right-looking approach, once the rows and columns of the factors associated with a given node are computed, they are immediately used at some other nodes in the graph to perform some updates; in the left-looking approach, this is the opposite: the updates are delayed as much as possible. From this perspective, the *multifrontal method* can be considered as a variant of the right-looking approach where every node has a temporary zone where partial updates are computed; the updates associated with a node are carried throughout the task graph and accumulated with some partial updates before they reach their final destination. We illustrate this in the example in Figure 1.1. Consider the elimination of pivot 2. In the right-looking method, once pivot 2 is eliminated, updates are applied to the rows and columns associated with nodes 3 and 4. In particular, at node 4, the update a_{44} a_{44} a_{42} a_{24} is applied. In the multifrontal method, the partial update a_{42} a_{24} is passed from node 2 to node 3. Node 3 adds this partial update to its own partial update (a_{43} a_{34}) and passes the whole update to node 4. The drawback of the method is that some temporary memory has to be used to store and accumulate partial updates; however in practice the method delivers high performance because it allows for large matrix-matrix operations and limits the use of indirect addressing. The notion of *aggregates* was introduced in right-looking methods to capture this advantage of the multifrontal methods. In this dissertation, we focus on the multifrontal method (although many ideas can be applied to supernodal methods), that we describe in detail in the next section.

Once the factorization is completed, the solution x of the system is computed in two steps:

The forward elimination consists of solving $Ly = b$ for y, where b is the right-hand side of the system.

The backward substitution consists of solving $Ux = y$, where x is the solution of the initial system.

We describe the two steps of triangular solution in Algorithm 1.2 in the dense case and illustrate them in Figure 1.2. Note that we illustrate a right-looking approach for the forward elimination (i.e., at each step, the trailing part of the solution vector is updated), and a left-looking approach for the backward phase (i.e., each step begins with aggregating contributions from the components of the solution that are already computed); this corresponds to a natural storage scheme where the L and U factors are stored by columns and rows respectively. We describe in detail the multifrontal triangular solution phase in Chapter 2.

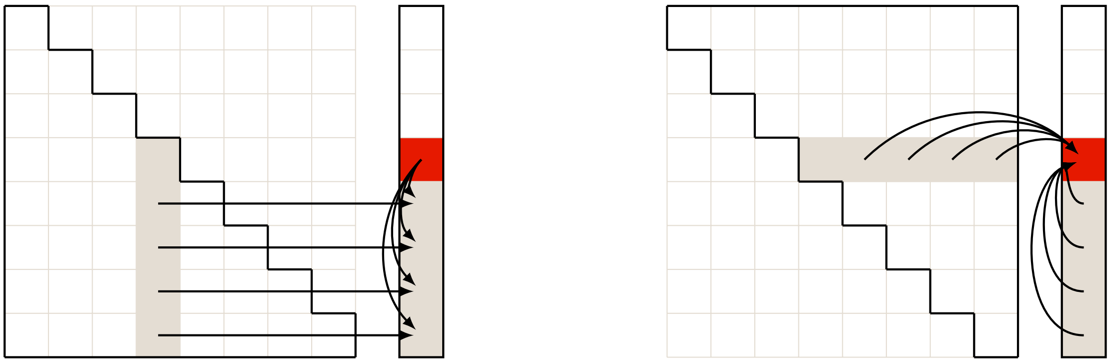
1.1.2 Distributed-memory sparse direct solvers

A large part of this thesis is dedicated to the study of sparse direct methods in a parallel context. We are particularly interested in distributed-memory architectures, in which each one of the *processes* that execute a given code has its own memory address space. Processes need to explicitly communicate using messages in order to exchange data; MPI is by far the most widespread library for doing so. Note than since *processors* now have several cores and can run several *processes* at the same time, we try to use these two terms accurately; in particular, when describing parallel algorithms, the relevant metric will most of the time be the number of *processes*.

Algorithm 1.2 Dense triangular solution algorithms.

```
Solution of $Ly = b$ for $y$
(right-looking approach)
$L$ is a unit lower triangular matrix

1: $y$   $b$
2: for $j = 1$ to $n$ do
3:     for $i = j + 1$ to $n$ do
4:         $y_i$   $y_i$   $l_{ij}$   $y_j$
5:     end for
6: end for
```

```
Solution of $Ux = y$
(left-looking approach)
$U$ is an upper triangular matrix

1: $x$   $y$
2: for $i = n$ to $1$ by   $1$ do
3:     for $j = i + 1$ to $n$ do
4:         $x_i$   $x_i$   $u_{ij}$   $x_j$
5:     end for
6:     $x_i$   $x_i / u_{ii}$
7: end for
```



(a) Forward elimination (right-looking).

(b) Backward substitution (left-looking).

Figure 1.2: Dense triangular solution algorithms.

Parallel, distributed-memory sparse direct methods consist of distributing the abovementioned task graph over a set of processes. The second part of this thesis is dedicated to these aspects. Among publicly available and active distributed-memory sparse direct solvers, one can cite PasTiX [52] and SuperLU_DIST [63], that implement different variants of supernodal methods, and MUMPS [9, 11], that implements a multifrontal method. We describe MUMPS in detail in Section 1.3.1.

1.2 Background on multifrontal methods

The *multifrontal method* by Duff & Reid [32, 33] heavily relies on the notion of *elimination tree*. The elimination tree was introduced by Schreiber [82] for symmetric matrices. The unsymmetric case is much more complex; different structures that represent the elimination process and various strategies appear in the literature. The definition of elimination trees for the unsymmetric case is recently introduced by Eisenstat & Liu [34, 36]. Firstly, we describe the symmetric case in detail, then we briefly review the unsymmetric case.

1.2.1 The symmetric case

We start with the symmetric positive definite case; let A be a square symmetric positive definite matrix of order n, and L the Cholesky factor of A ($A = L\,L^T$). Firstly, we define the *adjacency graph* of A:

Definition 1.1 - Adjacency graph.
The adjacency graph of A is a graph $G(A) = (V\ E)$ with n vertices such that:

There is a vertex $v_j \quad V$ for each row (or column) j of A.

There is an edge $\ v_i\ v_j \quad E$ if and only if $a_{ij} = 0$.

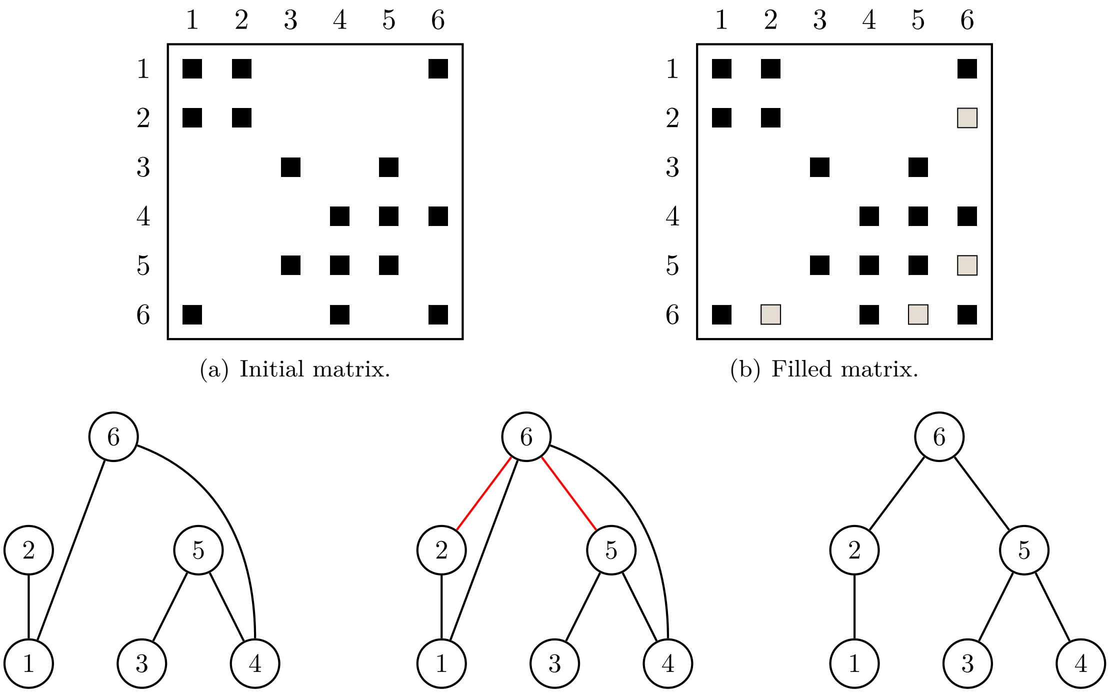
Similarly, the *filled graph* of A is the adjacency graph $G(F)$ of $F = L + L^T$, the *filled matrix* of A. Note that, because of the fill-in entries, the graph of A is a subgraph of the filled graph (with the same vertices).

Many equivalent definitions exist for the (symmetric) elimination tree; we recommend the survey by Liu [70] where many definitions, results and construction algorithms are provided. The simplest definition is probably the following:

Definition 1.2 - Elimination tree (symmetric case); from [70].
The elimination tree of A is a graph with n vertices such that p is the parent of a node j if and only if

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

If A is irreducible, this structure is a tree; otherwise it is a forest. Throughout this study, we will always assume that A is irreducible, unless stated otherwise. Another definition is that the elimination tree is the transitive reduction of the filled graph[1]. We illustrate the notions of adjacency graph, filled graph and elimination tree in Figure 1.3.



(a) Initial matrix.

(b) Filled matrix.

(c) Adjacency graph.

(d) Filled graph.

(e) Elimination tree.

Figure 1.3: An initial matrix (a) and its factors (b), its adjacency graph (c), its filled graph (d), and its elimination tree (e).

[1] More precisely, it is the transitive reduction of the *directed filled graph*, which is the same graph as the filled graph, except that every edge $\{v_i, v_j\}$ with $i < j$ is directed from i to j.

The multifrontal method is based on the elimination tree. We strongly recommend the survey by Liu [71] that provides a comprehensive and instructive presentation of the multifrontal method. Here we recall the main ingredients that will be useful in the following (we follow the proof and the notation from the above-mentioned survey). We begin with two fundamental results:

Theorem 1.1 - [71, Theorem 3.1].
If node k is a descendant of j in the elimination tree, then the structure of the vector $(l_{jk} \qquad l_{nk})^T$ is contained in the structure of $(l_{jj} \qquad l_{nj})^T$.

Theorem 1.2 - [71, Theorem 3.2].
If $l_{jk} = 0$ and $k < j$, then the node k is a descendant of j in the elimination tree.

Consider the j-th column of the L factor; let $j \; i_1 \qquad i_r$ be the row indices of the nonzeros in $L_{j:n\,j}$ and $\mathcal{T}(j)$ the subtree rooted at j. In the multifrontal method, every node j in the elimination tree is associated with a dense *frontal matrix* F_j, defined as

$$F_j = \begin{matrix} a_{jj} & a_{ji_1} & & a_{ji_r} \\ a_{i_1 j} & & & \\ \vdots & & 0 & \\ a_{i_r j} & & & \end{matrix} \quad + \overline{U}_j \text{ with } \overline{U}_j = \sum_{k \in \mathcal{T}(j) \quad j} \begin{matrix} l_{jk} \\ l_{i1\,k} \\ \vdots \\ l_{i_r k} \end{matrix} \quad \begin{matrix} l_{jk} & l_{i1\,k} & & l_{i_r k} \end{matrix}$$

The first term in the two-term sum that defines F_j is called the *arrowhead matrix* and consists of nonzero elements of the initial matrix. Note that the entries that correspond to fill-in entries are zeros. The second term is the *subtree update matrix* and is the sum of all the rank-one updates from the columns that are descendant of j in the tree. It follows from Theorem 1.1 and Theorem 1.2 that when F_j is computed, the first column (and the first row by symmetry) is fully updated; the variable j is said to be *fully-summed*. Therefore, one can apply a step of Gaussian elimination to F_j to eliminate the j-th pivot, yielding the factor column $L_{j:n\,j}$:

$$F_j = \begin{matrix} l_{jj} & 0 \\ l_{i_1 j} & \\ \vdots & I \\ l_{i_r j} & \end{matrix} \quad \begin{matrix} 1 & 0 \\ 0 & U_j \end{matrix} \quad \begin{matrix} l_{jj} & l_{i_1 j} & & l_{i_r j} \\ 0 & & I & \end{matrix}$$

U_j, the $r \times r$ Schur complement that results from the step of elimination, is also called the *contribution block* associated with F_j. The following result holds:

Theorem 1.3 - [71, Theorem 3.3].

$$U_j = \sum_{k \in \mathcal{T}(j)} \begin{matrix} l_{i1\,k} \\ \vdots \\ l_{i_r k} \end{matrix} \quad \begin{matrix} l_{i1\,k} & & l_{i_r k} \end{matrix}$$

This shows that once a step of elimination is applied to F_j, U_j, the (2,2) block of F_j contains the sum of all the rank-one updates of the subtree rooted at j; those are the updates that are needed to update the subsequent columns. Therefore, the frontal matrix F_j contains all the information needed for the subsequent steps of the factorization, and the information from the frontal matrices corresponding to descendants of j is no longer needed. This is what the following result states by showing how to compute a frontal matrix F_j using only the information coming from the children of j in the tree; denote s_{j1} $s_{j\,nc_j}$ the children of j:

Theorem 1.4 - [71, Theorem 4.1].

$$F_j = \begin{array}{cc} a_{jj} & a_{ji_1} \quad\quad a_{ji_r} \\ a_{i_1 j} & \\ \vdots & 0 \\ a_{i_r j} & \end{array} \qquad U_{s_{j1}} \qquad\qquad U_{s_{j\,nc_j}}$$

The frontal matrix at a given node j is *assembled* (computed) by summing the arrowhead matrix with the contribution blocks from the children of j; note that this is not a regular sum, the contribution blocks have to be extended by introducing zero rows and columns to conform with the indices of F_j. The operator is referred to as the *extend-add* operator. Therefore, the multifrontal method consists of, at every node j in tree:

1. *Assembling* the nonzeros from the original matrix (the arrowhead matrix) together with the contribution blocks from the children nodes of j into the frontal matrix.

2. *Eliminating* the fully-summed variable j; this produces the j-th column in the factors and a Schur complement (the contribution block) that will be assembled into the parent node at a subsequent step in the factorization.

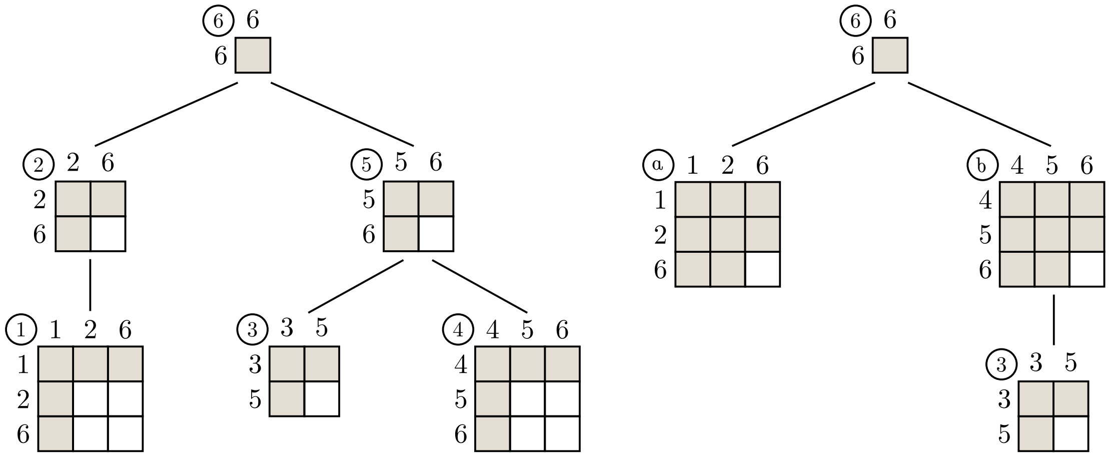
This process implies a *topological traversal* of the tree, i.e., a parent node must be processed after its children. In practice, a *postorder traversal* is used; a postordering of the tree is such that all the nodes in a given subtree are number consecutively. This allows the use of a stack mechanism to store the contribution blocks. We discuss in detail the choice of a postorder versus a topological traversal in Section 7.2.1.

The elimination tree and the associated frontal matrices are often referred to as the *assembly tree* (although we will often use the generic term *elimination tree* in this thesis). We illustrate in Figure 1.4(a) the assembly tree of the matrix from Figure 1.3(a). Note that, when assembling node 5, the contribution block from 3 (a 1×1 matrix) needs to be extended to a 2×2 matrix (in practice, other techniques are used).

The use of *supernodes* is a major practical improvement to the multifrontal method; a supernode is a range of columns of the factors with the same lower diagonal nonzero structure. They correspond to nodes that form a clique in the filled graph and are connected to the same nodes. For example, in Figure 1.3(d), nodes 3 and 4, as well as nodes 1 and 2, form supernodes. The corresponding frontal matrices can be merged without introducing any fill-in in the factors; this is referred to as *no-fill amalgamation*. This yields the tree in Figure 1.4(b). The interest is that this enables the use of matrix-matrix dense kernels; each front has a 2×2 block structure where several variables are eliminated at the same time, enabling the use of dense matrix-matrix kernels. The criterion for forming supernodes can be relaxed: columns whose structures are not exactly the same can be grouped together. This generates some fill-in but also provides more efficiency.

1.2.2 The general case

Generalizing the elimination tree and the multifrontal method to the unsymmetric case is not straightforward. The difficulty is that the structure of the factors cannot be repre-



(a) Elimination tree.

(b) Tree with some amalgamation.

Figure 1.4: Elimination tree (a) and tree with some amalgamation (b) associated with the matrix in Figure 1.3(a), with the frontal matrices shown at each node. The shaded entries correspond to the factors and the empty entries correspond to contribution blocks.

sented using a tree. Indeed, the transitive reductions of the directed graphs of L and U are directed acyclic graphs (dags for short), that Gilbert & Liu call *elimination dags* [45]. These structures are for example used in the supernodal code SuperLU_DIST.

In their initial approach, Duff & Reid choose to rely on the elimination tree of the symmetrized matrix $A + A^T$ [33], and many codes, such as MUMPS, follow this idea. This works well for problems that are structurally symmetric or nearly so, but this can dramatically increase the memory and computational requirements for very unsymmetric problems (in the structural sense). Other structures and approaches exist. We recommend the survey by Pothen & Toledo [76, Section 5] and a discussion in one of Eisenstat & Liu's papers [35, Section 3]. Eisenstat & Liu define elimination trees for unsymmetric matrices as follows:

Definition 1.3 - Elimination tree (general case); from [34].
The elimination tree of A is a graph with n vertices such that p is the parent of a node j if and only if

$$p = \min \quad i > j : i \ \ ^{L} \ j \ \ ^{U} \ i$$

where $i \ \ ^{L} \ j$ (respectively $i \ \ ^{U} \ j$) if and only there is a path from i to j in the directed graph of L (respectively U).

Then they show that the two following properties are incompatible [35, Section 3]:

(a) The rows (respectively columns) of every frontal matrix F_k correspond to the rows i such that $l_{ik} = 0$ (respectively, columns such that $u_{kj} = 0$). Note that this means that frontal matrices are possibly rectangular.

(b) Every update (contribution block) can be sent and assembled into a single frontal matrix.

In the usual approach where the elimination tree of the symmetrized matrix $A + A^T$ is used, (b) is satisfied: the contribution block of a front is sent to its parent only. However

(a) is violated because some fronts have nonzero entries in rows (respectively columns) whose entries in the factor part are zero. Amestoy & Puglisi suggested an approach where they remove on the fly, when a front is assembled, rows and columns that are structurally zero [12][2]; property (a) is still not ensured but this allows to reduce the computational requirements and the effects are very interesting in practice.

The incompatibility between (a) and (b) shows that the unsymmetric elimination tree cannot be used as a dataflow graph since contribution blocks might have to be sent to multiple nodes, that can lie on different branches. Eisenstat & Liu propose an approach where the tree is used as a task graph and a dag, which is for instance a supergraph of the elimination tree, is used to enforce data dependencies. This idea appears in Gupta's work on the WSMP code [50] where he uses a variant of elimination dags as a task graph and others dags as dataflow graphs; similarly, the UMFPACK code by Davis [29] relies on the *column elimination tree* (elimination tree of the symmetric matrix $A^T A$) as a task graph and some structures relying on biclique covers as dataflow graphs .

In this study, we always consider the symmetric elimination tree, i.e., the elimination tree of $A + A^T$. In the first part, where we address problems related to sparse right-hand sides, we heavily rely on the elimination tree as a representation of the structure of the factors; in the second part where we tackle memory scalability problems, we view the elimination more as a task graph. We believe that generalizing the ideas we present in this thesis to unsymmetric elimination trees is not straightforward (although probably feasible).

1.2.3 The three phases

Multifrontal codes often follow a *three phase approach*: *analysis*, *numerical factorization*, *triangular solution*.

The *analysis* phase applies numerical and structural pretreatments to the matrix, in order to optimize the subsequent phases. One of the main preprocessings, called *reordering*, aims at reducing the fill-in; it consists of permuting the rows and columns of the initial matrix so that less fill-in will occur in the factorization of the permuted matrix. Minimizing the fill-in is NP-complete [92]. Numerous heuristics have been studied to obtain efficient techniques that significantly decrease the fill-in. They fall into two main classes: *local methods* choose the pivot order by traversing the adjacency graph following a local heuristic (e.g., nodes with lowest degree first). The Approximate Minimum Degree algorithm (AMD) is one of the most popular techniques [6]. *Global methods* try to partition the adjacency graph and usually provide solvers with a tree of separators (sets of nodes or edges whose removal disconnect the graph). Scotch [74] and MeTiS [55] are probably the two most popular graph partitioning software packages. Once the ordering is computed, the *symbolic factorization* computes the structure of the factors (in practice, part of the work can be done during the ordering phase). The analysis phase also involves numerical pretreatments that aim at avoiding problems during the factorization. *Scaling* is a typical example of such a preprocessing: it computes two diagonal matrices D_r and D_c such that $D_r\, A\, D_c$ has better numerical properties, that is, less pivoting and numerical fill-in will occur during the factorization.

The numerical factorization computes the factors, relying on the pivot order and the elimination tree computed during the analysis. A keystone issue is the *numerical pivoting*, that aims at ensuring a good numerical accuracy by avoiding divisions by small pivots (diagonal elements). In case a bad pivot is found, the associated row is swapped with

[2] This implies that fronts can be rectangular.

another unfactored row so that the new diagonal element has better properties. In this dissertation we do not focus at all on numerical issues, therefore we do not describe in detail the different existing pivoting strategies. We simply highlight a specific property of the multifrontal method. Within a frontal matrix (corresponding to a supernode), pivots can only be chosen inside the (1,1) block, because the rows and columns that touch the contribution block are not fully summed; when a pivot cannot be eliminated without jeopardizing the numerical accuracy, the corresponding rows and columns are swapped with other columns and pushed to the border of the (1,1) block. They remain unfactored and are delayed to the frontal matrix of the parent, where they appear as fully-summed variables and are placed at the border of the (1,1) block; this is referred to as *delayed pivoting*. We illustrate this in Figure 1.5. The frontal matrix of the parent becomes larger and some fill-in occurs because of the variables that appear in the parent node but not in the child. A variable might be delayed several times if needed. Therefore, the elimination tree is a dynamic structure that can change during the factorization; in a parallel context, this implies that some dynamic scheduling should be used in order to compensate for the imbalance in workloads and memory loads that might arise because of delayed pivots and that cannot be forecast. We highlight these aspects in the second part of this thesis.

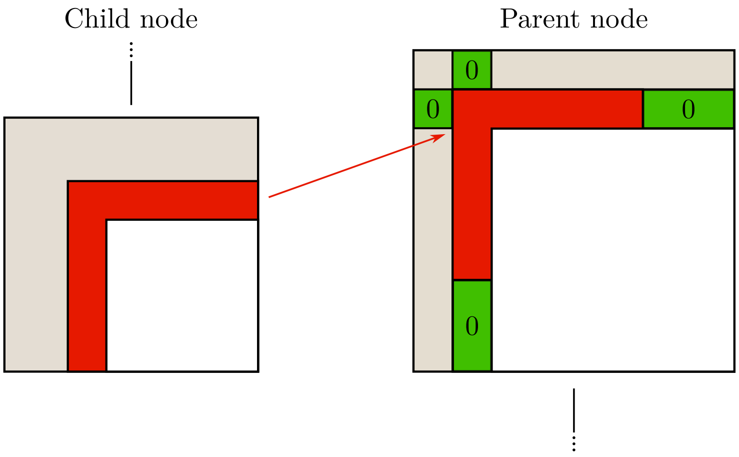


Figure 1.5: Delayed pivoting: some pivots that cannot be eliminated (dark rows and columns in the figure) are delayed to the parent node. Note that this introduces some numerical fill-in (explicit zero entries in the parent node).

An interesting feature of the multifrontal method (as well as right-looking methods) is that it can efficiently run *out-of-core*; this consists of using storage disks as an extension of the main memory, which is useful when the memory requirements for the factorization are larger than the main memory. In the multifrontal method, once the contribution block of a frontal matrix is computed, the rows and columns of factors for this frontal matrix are no longer needed (see Theorem 1.3 and 1.4); therefore, factors can be written to disk as soon as they are computed (and removed from the main memory) without being read in the rest of the factorization. This limits the volume of accesses to storage disks, that are usually much slower than the main memory. In Chapter 4, we address a problem related to the triangular solution phase in an out-of-core context.

The last step is the solution phase that computes the solution of the system in two steps: forward elimination and backward substitution, as in the dense case. They consist of a bottom-up and a top-down traversal of the tree, respectively. We describe these two algorithms in detail in Section 2.4.

Finally, we provide some notation that will be useful in the following; for a given node i (sometimes $\mathcal{N}_i$):

$nfront_i$ is the order of the frontal matrix associated with i.

$sfront_i$ is the surface of the front.

$npiv_i$ is the number of fully-summed variables in the front.

ncb_i is the size of the contribution block.

scb_i is the surface of the contribution block.

nc_i is the number of children of i.

$sib(i)$ is the set of siblings of i.

1.3 Experimental environment

1.3.1 The MUMPS solver

The main motivation for this work was to study and experiment algorithmic ideas in order to improve the MUMPS solver. The MUMPS solver [9, 11] is a parallel direct solver that implements the multifrontal method. MUMPS started in 1996 with the European project PARASOL; it was inspired by the shared-memory code MA41 by Amestoy and Duff [7, 3], which itself relies on the earlier HSL code MA37 by Duff and Reid. MUMPS primarily targets at distributed-memory architectures; at the moment, multithreading is achieved only using a multithreaded BLAS, but work is in progress to exploit shared-memory parallelism using OpenMP [5].

MUMPS relies heavily on the elimination tree of the matrix A to be factored; if A is unsymmetric, MUMPS uses the sparsity pattern of $A + A^T$. Both tree parallelism and node parallelism are exploited. We refer to sequential nodes and parallel nodes as *Type 1* nodes and *Type 2* nodes respectively. Type 2 nodes are distributed following a 1D row-wise partitioning: the so-called *master process* holds the (1,1) block and the (1,2) block (rows of the U factor) and is in charge of organizing the computations; *slave processes* are in charge of the (2,1) block (columns of L) and the (2,2) block (the contribution block). The root node can optionally be processed using a two-dimensional partitioning relying on ScaLAPACK, in which case we call it a *Type 3 root*. We illustrate this in Figure 1.6. Contrary to most codes that rely on static approaches where tasks are assigned to processes in advance, MUMPS relies on a dynamic scheduling of the tasks, using a completely asynchronous approach; this increases the complexity of the code but makes it able to perform dynamic pivoting, which guarantees the numerical stability. We will describe all these aspects in detail in Chapter 7.

MUMPS provides a large range of numerical features that make it a very robust code, and it also provides many functionalities:

Input: MUMPS provides Fortran, C and MATLAB interfaces. The input matrix can be unsymmetric or symmetric; it can be centralized or distributed, assembled or in elemental format (i.e., represented as an expended sum of dense matrices). Real and complex, single and double precision arithmetic are supported.

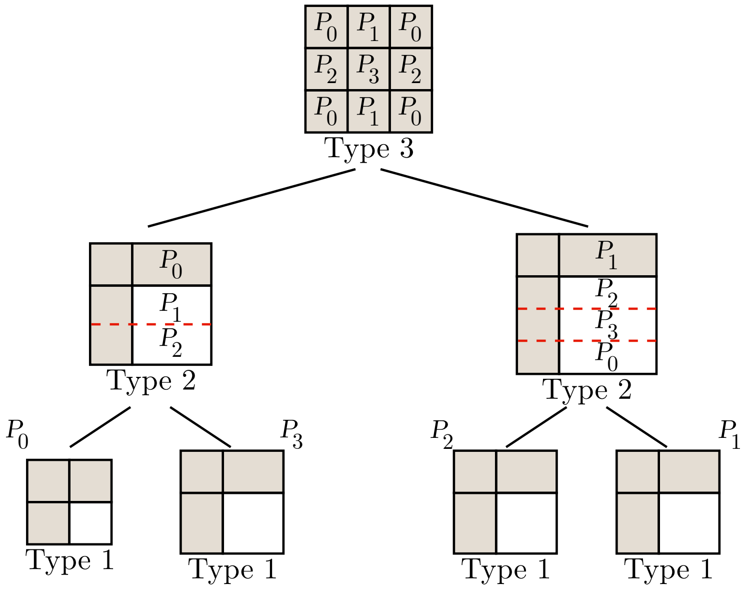


Figure 1.6: Node types used within MUMPS: the Type 3 root node is distributed following a 2D block cyclic partitioning, Type 2 nodes follow a 1D row-wise partitioning where the master process holds the rows of the U factor while slave processes share off-diagonal columns of the L factor and the contribution block. Type 1 nodes are processed by a single process.

Analysis: many structural and numerical pretreatments (scalings) can be applied and different ordering strategies and packages can be used, in particular AMD [6] and some other local heuristics, PORD [83], MeTiS [55], Scotch [74] and their parallel versions ParMeTiS [56] and PT-Scotch [28].

Factorization: LU, LDL^T and Cholesky factorizations can be performed, depending on the matrix. Many pivoting strategies (two-by-two pivots, delayed pivoting...) are implemented and guarantee the numerical robustness of the code. The factorization can be performed in an *out-of-core* mode, where partial factors are saved on disks as soon as they are no longer needed; using asynchronous I/O operations, this has a relatively small impact on the performance but significantly reduces the memory usage [1, 84].

Solve step: dense and sparse right-hand sides are supported; the solution can be either centralized or distributed.

Postprocessing: iterative refinement can be applied to the solution vector, and a numerical analysis can be performed to return more information to the user (backward error, etc.).

Miscellaneous: many features are available beyond the classical analysis-factorization-solution scheme: partial factorization and computation of the Schur complement (see the next section), computation of the inertia and the determinant of the matrix, computation of a nullspace basis for rank-deficient matrices...

The aim of this study is to tackle two difficulties that arise in MUMPS as well as in many other sparse direct solvers:

1. The triangular solve step of direct solvers is known to be less efficient than the factorization: while many sparse factorization codes are able to reach reasonable flop rates and speed-ups, the triangular solution often lags behind, which is critical especially in applications where several solve steps (often thousands) are performed after a factorization. This was confirmed in the last MUMPS Users' Group Meeting in 2010, where many users put a more efficient solution phase on their wish list. In this study, we have been interested in improving the solution phase especially in the case where sparse right-hand sides and/or sparse solution vectors are involved: this is described in the first part of this dissertation, where we suggest different ideas to improve the performance and the memory usage of the solution phase.

2. Memory is known to be a limitation to the use of direct solvers. In MUMPS, the memory estimates computed prior to the factorization are often not very reliable because of the dynamic nature of the code, where tasks are not assigned in advance. Furthermore, MUMPS relies on the multifrontal factorization, which makes use of a (parallel) stack to store contribution blocks: the behavior of this stack is difficult to predict in a parallel environment. Thus users quite often face the error code -9 , which means that the workspace allocated before the factorization is too small, because memory estimates were too optimistic. In this study we are interested in developing mapping and scheduling algorithms that enforce some memory constraints and provide reliable memory estimates prior to the factorization: this is described in the second part of this dissertation.

1.3.2 The PDSLin solver

Hybrid methods attempt to combine the strengths of direct and iterative methods in order to obtain solvers that are able to tackle larger problems. Among these, the Schur complement methods [85] have gained popularity in recent years and have demonstrated to be scalable on large numbers of processors. In this method, the original linear system $Ax = b$ is first reordered into a system with the following block structure:

$$\begin{array}{cccc|c} D_1 & & & & E_1 \\ & D_2 & & & E_2 \\ & & \ddots & & \vdots \\ & & & D_k & E_k \\ \hline F_1 & F_2 & & F_k & C \end{array} \quad \begin{array}{c} u_1 \\ u_2 \\ \vdots \\ u_k \\ y \end{array} \quad = \quad \begin{array}{c} f_1 \\ f_2 \\ \vdots \\ f_k \\ g \end{array} \tag{1.1}$$

where D is referred to as the -th *interior subdomain*, C consists of *separators*, and E and F are the *interfaces* between D and C. To compute the solution of the linear system (1.1), we first compute the solution vector y on the interface by solving the Schur complement system,

$$Sy = g \tag{1.2}$$

where the Schur complement S is defined as

$$S = C \quad \sum_{=1}^{k} F D^{-1} E$$

and $g = g \quad \sum_{=1}^{k} F D^{-1} f$. Then, to compute the solution vector u on the -th subdomain, we solve the -th subdomain system

$$D u = f \quad E y \tag{1.3}$$

Many software packages implement this method: one can cite HIPS [38], MaPHyS [47], ShyLU [78] and PDSLin [64]. We focus on the last, as some of the ideas presented in Chapter 5 have been implemented and experimented within this solver. PDSLin uses the parallel direct solver SuperLU_DIST [63] on each one of the interior domains D (1.3). PDSLin does not explicitly form the Schur complement S: its solves the Schur complement system (1.2) using a preconditioned iterative method, the preconditioner being the LU factorization of an approximate Schur complement S obtained with SuperLU_DIST.

The approximate Schur complement S is obtained through the following process: first, a *local matrix* A is associated with each subdomain D :

$$A = \begin{matrix} D & E \\ F & 0 \end{matrix}$$

where E and F are the nonzero columns of E and nonzero rows of F respectively. Then D is factorized using SuperLU_DIST, yielding $P\,D\,\overline{P} = L\,U$ where P and $\overline{P}$ are row and column permutations respectively. Then the update matrix T is computed as

$$\begin{aligned} T &= F\,D^{-1}E \\ &= \quad F\,\overline{P}\,U^{-1} \quad L^{-1}P\,E \\ &= W\,G \end{aligned}$$

where $W = F\,\overline{P}\,U^{-1}$ and $G = L^{-1}P\,E$ are computed using the parallel triangular solver of SuperLU_DIST. A large amount of fill might occur in W and G . Thus, in order to reduce the memory and computational cost, they are approximated by discarding entries with magnitude less than a prescribed threshold, yielding their approximations W and G and the approximate update matrix $T = W\,G$. Therefore, using the interpolation matrices R_{E_ℓ} and R_{F_ℓ} that map the columns and rows of E and F respectively, the approximate Schur complement is formed as

$$S = C \quad \sum_{=1}^{k} R_{F_\ell} T\, R_{E_\ell}^{T}$$

Finally, in order to further reduce the cost of the preconditioner, small nonzero elements are discarded from S, yielding the final approximate Schur complement S.

Two combinatorial problems are of interest in the framework of the PDSLin solver:

1. Computing the doubly-bordered form (1.1) so that multiple constraints are satisfied: balancing the workload between the different subdomains (i.e., the cost of the factorizations), balancing the workload between the different interfaces (i.e., the cost of triangular solves) and reducing the size of the Schur complement. We have used different graph and hypergraph partitioning formulations, and have developed a recursive hypergraph bisection method. We have tackled this problem but it is out of the scope of this dissertation, and we refer the reader to [57] and [91].

2. Partitioning the right-hand sides columns F^T and E in order to speed up the computation of $W = F\,\overline{P}\,U^{-1}$ and $G = L^{-1}P\,E$. This is described in Chapter 5, where we have examined different heuristics and a hypergraph model.

1.3.3 Test problems

We describe in Table 1.1 the matrices that were used in the experiments carried out during this study. They arise from various industrial and academic applications from our partners; they all correspond to physical problems taking place in three-dimensional domains with the exception of matrix211. This kind of matrices is often challenging for direct solvers as they exhibit large amount of fill. The matrices we have selected exhibit different properties in terms of symmetry, fill-in, etc., but they are not very demanding numerically, as this was not the point of this study. We indicate in the table the size of the factors obtained by ordering the matrices with MeTiS [55].

Matrix name	Order N	Entries (millions)	Factors (GB)	Sym	Arithmetic	Description; origin
NICE20MC$^{( )}$	715923	28.1	8.3	sym	double real	Seismic processing; BRGM lab
matrix211	801378	55.8	3.9	uns	double real	Fusion, M3D-C^1 code; Center for Extended MHD Modeling
AUDI$^{( )}$	943695	39.3	9.9	sym	double real	Automotive crankshaft model; Parasol collection
bone010$^{( )}$	986703	36.3	8.6	sym	double real	3D trabecular bone; Mathematical Research Institute of Oberwolfach
pancake2_3	1004060	49.1	39.8	uns	single cmplx	3D electromagnetism; Padova University
tdr190k	1110242	43.3	5.7	sym	double real	Accelerator; Omega3P code, SLAC National Accelerator Laboratory
CONESHL$^{( )}$	1262212	43.0	5.5	sym	double real	3D finite element, SAMCEF code; SAMTECH
Hook_1498$^{( )}$	1498023	31.2	12.3	sym	double real	3D model of a steel hook; Padova University
FLUX-2M$^{( )}$	2001728	212.9	34.8	uns	single cmplx	3D finite element, FLUX code; CEDRAT
CAS4R_LR15	2423135	19.6	4.5	sym	single cmplx	3D electromagnetism; EADS Innovation Works
meca_raff6	3269763	130.1	63.5	sym	double real	Thermo-mechanical coupling, Code_Aster; EDF

Table 1.1: Set of matrices used for the experiments. Those marked with $^{( )}$ are publicly available on `gridtlse.org`. We indicate the symmetry either using sym (symmetric) or uns (unsymmetric).

We have also used two generators that are able to create matrices with prescribed size, corresponding to an $nx \times ny \times nz$ physical domain: the d11 generator produces symmetric real matrices corresponding to an eleven-point stencil, and the Geoazur generator, provided by Stéphane Operto and Jean Virieux (Geoazur consortium), produces unsymmetric complex matrices corresponding to a 27-point stencil. This generator is used in Geophysics (modeling of acoustic wave propagation in the frequency domain), and the matrices it generates are challenging for direct solvers as they are very memory demanding. For example, with $nx = ny = nz = 192$, the size of the matrix is $N = 7077888$, the number of entries in the matrix is 189 1 millions, and the size of the LU factors is 144 GB (in single precision arithmetic).

1.3.4 Computational systems

We describe the different systems on which we have carried out our experiments:

`Pret`, desktop machine at ENSEEIHT: one quad-core Intel Xeon W3350 processor

@3 GHz, 4 GB of memory and a 7200 RPM hard drive. Used for the experiments in Chapter 4.

`Franklin`, Cray XT4 machine at the National Energy Research Scientific Computing Center (NERSC): 9572 nodes with one quad-core AMD Opteron 1356 processor @2.3 GHz and 8 GB memory per node. Used for the experiments in Section 5.1.

`Hyperion`, Altix ICE 8200 machine at the Calcul en Midi-Pyrénées resource center (CALMIP): 352 nodes with two quad-core Intel Xeon 5560 processors @2.8 GHz and 32 GB memory per node. Used for the experiments in Section 5.2, Chapter 10 and Chapter 11.

`Hopper`, Cray XE6 machine at the National Energy Research Scientific Computing Center (NERSC): 6384 nodes with two twelve-core AMD Opteron 6172 processors @2.1 GHz and 32 GB memory per node. Used for some of the experiments in Chapter 10.

Part I

Solution phase with sparse right-hand sides: memory and performance issues

Chapter 2

Introduction

The first part of this thesis focuses on memory and performance aspects of the multifrontal solution phase with sparse right-hand sides and/or sparse solution vectors. While much effort has been dedicated to exploiting the sparsity of the matrix in the factorization phase, few studies have considered exploiting sparsity of the right-hand side(s) in the solution phase. However, the main result was established in the 80's by Gilbert [44][1], who showed that the structure of the solution of $Ax = b$ (with A non singular) is the *closure* of the structure of b (see Section 2.2 for details). Interestingly, Gilbert and Liu used this theorem to compute the structure of the LU factors by formulating the sparse LU factorization as a sequence of sparse triangular solves [45], but paradoxically, few studies or codes have used this result in the solution phase. Furthermore, little has been done about multiple right-hand sides. In this work, we are interested in computations that involve multiple sparse right-hand sides and sparse solution vectors; we focus on compressing the memory space used by the solution phase, reducing the computational cost in different contexts (in-core and out-of-core settings), and enhancing parallelism.

In this introductory chapter, we first mention in Section 2.1 various applications that involve sparse right-hand sides and sparse solutions, and we especially highlight applications that involve computation of a set of entries of the inverse of a sparse matrix (that we refer to as *inverse entries*); as we will see, this amounts to solving linear systems where both the right-hand sides and the solution vectors are sparse. We also give a brief survey of the existing methods for computing inverse entries. In Section 2.2, we describe how to exploit sparsity of the right-hand sides and/or the solution vectors; we show how this can be applied to the computation of inverse entries. We then describe the problems we have focused on and our contributions in Section 2.3. Finally, we describe in Section 2.4 the general parallel multifrontal solution algorithm, as it will be useful for the subsequent chapters.

2.1 Applications and motivations

2.1.1 Applications with sparse right-hand sides and/or sparse solution

Many physical applications involve solving linear systems with sparse right-hand sides that correspond to data that is nonzero at only a few points of the physical domain. Similarly, it is also often required to compute the solution of a problem at only a few points of the domain, which corresponds to computing only a few entries of the solution vectors (we

[1] The publication is dated 1994 but the associated technical report is actually from 1986.

refer this to as *selected solution* or *selected entries*). These situations are for example described in [16] in the context of Poisson's equations.

Solving linear programming problems with the simplex algorithm often exhibits highly-reducible matrices, sparse right-hand sides and sparse solution vectors. This situation is referred to as hyper-sparsity [51].

Several methods for solving sparse linear systems involve intermediate steps that rely on sparse right-hand sides, even if the initial right-hand side is not necessarily sparse. This is the case in the Schur complement method where the update of the Schur complement requires the solution of triangular systems with sparse right-hand sides (see Section 1.3.2). This is also the case in the block-Cimmino method [13]; in this iterative method, several augmented systems of the form

$$\begin{matrix} G & {A^i}^T \\ A^i & 0 \end{matrix} \quad \begin{matrix} u^i \\ v^i \end{matrix} = \begin{matrix} 0 \\ r^i \end{matrix}$$

need to be solved at each iteration (the A^i's correspond to a row-wise partition of the initial matrix A). In these augmented systems, the right-hand side is sparse and only the upper part, u^i, of the solution is needed in the iterative process.

Nullspace computations in the context of rank revealing LU factorization for general unsymmetric matrices also require the solution of linear systems with sparse right-hand sides. In order to compute a basis of the nullspace of a rank-deficient matrix, one has to solve $Ax = 0$. First, a rank-revealing factorization is performed, which provides $A = LU$ with $det(L) = 0$ and U containing zero rows , due to the detection of null pivots. Therefore, one has to solve $Ux = 0$. Assume for example that pivot j is a null pivot. We write $Ux = 0$ in the following form:

$$\begin{matrix} U_{1:j-1\;1:j-1} & U_{1:j-1\;j} & U_{1:j-1\;j+1:n} \\ & 0 & 0 \\ & & U_{j+1:n\;j+1:n} \end{matrix} \quad \begin{matrix} x_{1:j-1} \\ x_j \\ x_{j+1:n} \end{matrix} = \begin{matrix} 0 \\ 0 \\ 0 \end{matrix}$$

Assuming $U_{j+1:n\;j+1:n}$ is not rank deficient, we have $x_{j+1:n} = 0$, and x_j can be set to 1. We thus have to solve:

$$U_{1:j-1\;1:j-1}\,x_{1:j-1} = \quad U_{1:j-1\;j}$$

and the final nullspace vector is $x = (x_{1:j-1}; 1; 0)$. Note that the same set of equations can be obtained by setting $u_{j\,j}$ to 1 during the factorization and the right-hand side to e_j. Therefore, the nullspace vector can be computed by solving $Ux = e_j$. In the general case, U has several zero rows, and one has to solve $UX = E$, where each column of E is a vector e_j associated with a null pivot j, and the columns of E are sorted in the order in which the null pivots have been detected. This scheme is explained in detail in [84].

2.1.2 Applications involving the computation of inverse entries

The inverse of an irreducible sparse matrix is structurally full [31], thus it is impractical to compute or store all its entries. However, many applications require the computation of a set of inverse entries. Linear least-squares problems provide a typical example: the inverse entries of the *covariance matrix* associated to the problem are of interest; in particular, the diagonal entries of the inverse of the covariance matrix provide a measure of the quality of the fit [19]. We have been particularly motivated by a data analysis application in astrophysics, where these diagonal entries were requested: the aim was to analyze images coming from a high-resolution spectrometer aboard the INTEGRAL (INTERnational Gamma-Ray Astrophysics Laboratory) satellite [22, 20, 21]. This involved solving a overdetermined system with about 1,000,000 equations and 100,000 unknowns.

It seems that historically, one of the first applications that required the numerical computation of inverse entries, and that drove the development of specific methods to do so, was the study of short-circuit currents, that involves the calculation of the diagonal elements of the inverse of a so-called *impedance matrix* [86]. Other applications that require the computation of inverse entries include quantum-scale device simulation, such as the atomistic simulation of nanowires [25, 72], electronic structure calculations [65], approximations of condition numbers [19], and uncertainty quantification in risk analysis [18]. In all these applications, not all the inverse entries are requested, but only a subset, quite often the diagonal elements or some subblocks along the diagonal.

Tang and Saad [87] propose an iterative method to compute the diagonal of the inverse; they focus on matrices whose inverses have a decay property. Other studies mostly rely on direct methods, and more specifically on the *Takahashi equations* [86]. These equations assume an LDU factorization of the initial matrix A (of size $N \times N$), and relate the factors L, D and U to $Z = A^{-1}$, using the equation $Z = U^{-1}D^{-1}L^{-1}$:

$$
\begin{aligned}
Z &= D^{-1}L^{-1} + (I \quad U)Z \\
Z &= U^{-1}D^{-1} + Z(I \quad L)
\end{aligned}
$$

Using these equations, one can compute the entries in the upper part of Z and the entries in the lower part of Z by using the U and L factors respectively:

$$
\begin{aligned}
i \leqslant j \;\; z_{ij} &= d_{ij}^{-1} \quad \sum_{k>i}^{N} u_{ik} z_{kj} \\
i \geqslant j \;\; z_{ij} &= d_{ij}^{-1} \quad \sum_{k>j}^{N} z_{ik} l_{kj}
\end{aligned}
$$

Using these equations, one can compute the whole inverse matrix (starting with z_{NN}). However, as said before, it is often not advisable nor necessary to compute the whole inverse. Erisman and Tinney [37] propose a method to compute a subset of elements. They first give an algorithm that computes the parts of the inverse of A that correspond to the nonzero structure of $(L + U)^T$, starting from entry $(N \; N)$ and proceeding in a reverse Crout order. At every step, an entry of the inverse is computed using the factors L and U and the already computed entries of the inverse. This approach is later extended to the computation of any set of entries of the inverse, rather than the whole set in the pattern of $(L + U)^T$ [73]. Finally, Campbell and Davis [23] proposed a multifrontal-like approach (that they call the *inverse multifrontal* approach) to compute a set of elements in the case of a numerically symmetric matrix using Takahashi's equations. This method uses the elimination tree of the initial matrix (processed from the root to the leaves) and takes advantage of Level 3 BLAS routines (matrix-matrix computations). They propose a parallel implementation in [24]. Lin et al. propose similar ideas in the symmetric case with the SelInv algorithm [67] and have implemented an efficient parallel version [66].

Another approach is described in [61]. It uses the observation that if many LU factorizations are performed, each of them with an ordering that puts a given diagonal entry last, then the corresponding entry of the inverse is equal to $1/u_{NN}$ (for each U). The efficiency of the method relies on the fact that local factorizations are reused as much as possible to limit the complexity and a nested dissection tree based on the physical finite element mesh is used. The approach is shown to be very efficient on 2D meshes; it can be generalized to compute any arbitrary set of entries and is parallelizable. The algorithm is later improved in [62] to exploit sparsity and symmetry.

2.2 Exploiting sparsity in the solution phase

2.2.1 General case

In the general case, a forward triangular solve with a dense right-hand side consists of a complete bottom-up traversal of the elimination tree (from the leaves up to the root node, following a topological order). The sparsity of the right-hand side can be exploited to reduce this traversal to following a simple path; the key result is given by Gilbert & Liu [45]):

Theorem 2.1 - Structure of the solution of a triangular system (Theorem 2.1 in [45]). For any lower triangular matrix L, the structure of the solution vector x of $Lx = b$ is given by the set of nodes reachable from nodes associated with the right-hand side entries by paths in the directed graph $G(L^T)$ of the matrix L^T.

We can reword this result in terms of elimination tree since we consider the LU factorization of a matrix with a symmetric or symmetrized pattern (pattern of $A + A^T$). We denote $\mathcal{P}(i)$ the path from node i to the root node in the tree.

Lemma 2.1
The indices of the nonzero entries of the solution vector y of $Ly = b$ are equal to the indices of the nodes of the elimination tree that are in $_{i\in struct(b)}\mathcal{P}(i)$, that is in the union of the paths from the nodes corresponding to nonzero entries of b to the root.

We illustrate this result in Figure 2.1: solving $Ly = e_3$ consists of visiting all the nodes on the path from node 3 up to the root node, 6, i.e., nodes 3, 5 and 6. One can note that the branch 1 4 is not traversed.

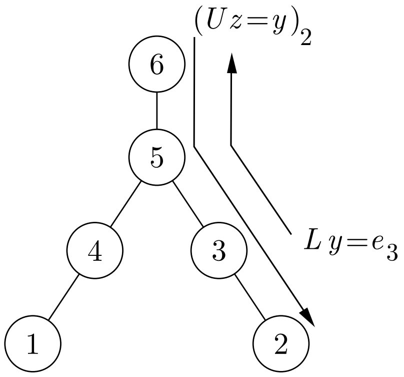


Figure 2.1: Traversal of the elimination tree when solving with a sparse right-hand side and/or a sparse solution. $Ly = e_3$ is solved by visiting nodes 3, 5 and 6. $U^{-1}y$ $_2$ is found by traversing 6, 5, 3 and finally 2, assuming $y_6 = 0$.

We can address the case of selected entries in the solution using a similar reasoning: a backward solution phase normally consists of visiting the whole tree following a top-down traversal, but the set of nodes to be visited can be reduced if only a few components of the solution are needed. We first provide a lemma that shows which part of the elimination tree needs to be visited when only the i-th entry of the solution of $Uz = y$ is requested; the extension to the computation of several entries is then straightforward. Although very similar to the previous result, the proof does not seem to appear anywhere in the literature.

Lemma 2.2
In order to obtain the i-th component of the solution to $Uz = y$, that is $z_i = (U^{-1}y)_i$, one has to solve the equations corresponding to the nodes that are in the unique path from the highest node in $struct(y) \setminus \mathcal{P}(i)$ to i.

Proof. We first show (1) that the set of components of z involved in the computation of z_i is the set of ancestors of i in the elimination tree. We then (2) reduce this set using the structure of y.

(1) We prove, by top-down induction on the tree, that the only components involved in the computation of any component z_l of z are the ancestors of l in the elimination tree:

Root node: z_n is computed as $z_n = y_n/u_{nn}$ (thus requires no other component of z) and has no ancestor in the tree.

For any node l: following a left-looking scheme, z_l is computed as:

$$z_l = \quad y_l \quad \sum_{k=l+1}^{n} u_{lk} z_k \quad /u_{ll} = \quad y_l \quad \sum_{k;u_{lk}=0}^{n} u_{lk} z_k \quad /u_{ll}$$

All the nodes in the set $\mathcal{K}_l = \quad k : u_{lk} = 0 \quad$ are ancestors of l by definition of the elimination tree (since we assume $struct(U^T) = struct(L)$). Thus, by applying the induction hypothesis to all the nodes in $\mathcal{K}_l$, all the required nodes are ancestors of l.

(2) The pattern of y can be exploited to show that some components of z which are involved in the computation of z_i are zero. Noting k_i the highest node in $struct(y) \setminus \mathcal{P}(i)$, that is the highest ancestor of i such that $y_{k_i} = 0$, we have:

$$\begin{aligned} z_k &= 0 \quad \text{if } k \quad \mathcal{P}(k_i) \quad k_i \\ z_k &= 0 \quad \text{if } k \quad (\mathcal{P}(i) \;\; \mathcal{P}(k_i)) \qquad k_i \end{aligned}$$

Both statements are proved by induction using the same left-looking scheme.

Therefore, the only required components of z lie in the path between i and k_i, the highest node in $struct(y) \setminus \mathcal{P}(i)$. □

Note that if the initial matrix A is irreducible and if y results from the solution of $Ly = b$, Lemma 2.1 implies that the last component of y, that corresponds to the root of the elimination tree, is nonzero. Therefore, in Lemma 2.2, $struct(y) \setminus \mathcal{P}(i)$ is non-empty (it contains the root node at least), and thus $(U^{-1}y)_i$ is nonzero. Conversely, if the initial matrix A is reducible, $struct(y) \setminus \mathcal{P}(i)$ could be empty: this can happen if no nonzero of y lies in the connected component where i is. In that case, $(U^{-1}y)_i = 0$.

We illustrate the previous lemma using Figure 2.1: the second component of the solution of $Ux = y$ has to be computed. Assuming that $y_6 = 0$, this consists of visiting all the nodes on the path from the root node, 6, to node 2, that is, nodes 6, 5, 3, and 2. The branch 1 4 is not visited.

The two lemmas (that can clearly be combined if the right-hand side is sparse and selected entries of the solution are to be computed) reduce the parts of the tree that have to be visited to a few paths; this enables significant savings in terms of computations (since, at each node, floating-point operations have to be performed), in terms of accesses to the factors and in terms of storage for the solution space; we emphasize these properties in the next chapters. When processing a block of right-hand sides instead of a single one, the set of nodes to be visited is the union of the paths given by the previous lemmas. We refer this set to as the *pruned tree*, and call *tree pruning* the process of computing the pruned tree. We are particularly interested in computations with multiple sparse right-hand sides and describe our contributions (and the content of the following chapters) in Section 2.3.

We give in Algorithm 2.1 a sketch of tree pruning procedure. We assume that we have a list of *targets*, that either correspond to the nonzero pattern of a block of sparse right-hand sides (if we want to compute the pruned tree for the forward phase), or correspond to the pattern of entries of the solution that are requested (if we want to compute the pruned tree for the backward phase). The aim is to mark nodes that belong to the pruned tree (array *to_process*) and to obtain the list of leaves of the pruned tree (which will be used to initialize the pool of tasks in the forward elimination). For any entry in the list of targets, the algorithm follows the path from the node corresponding to that entry up to the root node; the algorithm stops if a node that has been previously visited is detected (in order to avoid visiting branches more than once).

2.2.2 Application to the computation of inverse entries

The approach we use in this study relies on a traditional solution phase and makes use of the equation $AA^{-1} = I$. More specifically, we compute a particular entry $a_{ij}^{-1} = \; A^{-1} \; {}_{ij}$ as $\; A^{-1}e_j \; {}_i$; here e_j is the j-th column of the canonical basis (the identity matrix). We assume that the matrix A, whose inverse entries will be computed, has been factorized using a multifrontal or supernodal approach into $A = L\,U$ (or $A = L\,D\,L^T$ in the indefinite symmetric case, or $A = L\,L^T$ in the symmetric positive-definite case). Thus, a_{ij}^{-1} can be obtained by solving successively two triangular systems:

$$
\begin{aligned}
y &= L^{-1}e_j \\
a_{ij}^{-1} &= (U^{-1}y)_i
\end{aligned}
$$

If all entries in the pattern of $(L+U)$ are requested, then the method that implements the algorithm in [37] might be advantageous, whereas a method based on the traditional solution of linear systems, such as the one we present here, has to solve at least n linear systems and requires considerably more memory. On the other hand, if a set of entries in the inverse is requested, any implementation based on the equations by Takahashi et al. should set up necessary data structures and determine the computational order to compute all the entries that are necessary to compute those requested. This seems to be a rather time consuming operation.

We see from the above equations that, in the forward elimination phase, the right-hand side vector, e_j, contains only one nonzero entry and that, in the backward step, only one entry of the solution vector is required. For efficient computation, we have to take advantage of both these observations along with the sparsity of L and U. In our case, the sparse vector b is a column of the identity, say e_j. Lemma 2.1 implies that the only nodes in the tree that have to be visited are the nodes in the unique path from node j to the root. As the matrix is assumed irreducible, the last entry, y_n, corresponding to the root

Algorithm 2.1 Tree pruning procedure.

```
/* Input: $targets$, list of targets                                   */
/* Output: $to\_process$, nodes that belong to the pruned tree          */
/*          $leaves$, leaf nodes of the pruned tree                     */

$to\_process$      false
$is\_leaf$      true
for $i$ in $targets$ do
    $\mathcal{N}$      the node variable $i$ belongs to.
    /* Follow the path from $\mathcal{N}$ to the root node */
    $go\_up$      true
    while $go\_up$ do
        $to\_process(\mathcal{N})$      true
        if $\mathcal{N}$ is not the root node then
            $\mathcal{N}$      parent of $\mathcal{N}$
            $is\_leaf(\mathcal{N})$      false
            if $to\_process(\mathcal{N})$ then /* $\mathcal{N}$ has been met before */
                $go\_up$      false
            end if
        else /* Nothing more to do */
            $go\_up$      false
        end if
    end while
end for
$leaves$      $\varnothing$
for nodes $\mathcal{N}$ in the tree do
    if $is\_leaf(\mathcal{N})$ then
        Add $\mathcal{N}$ to $leaves$
    end if
end for
```

of the tree, is nonzero. Therefore, Lemma 2.2 states that we need to solve the equations that correspond to nodes that lie in the unique path from the root node to node i. By combining the two previous lemmas, one can see that to compute a particular entry a_{ij}^{-1} in A^{-1}, the only entries of the factors which have to be used are the L factors on the path from node j up to the root node, and the U factors on the path going back from the root to node i (see also [84, Chapter 8] which follows a different approach to reach this conclusion).

Figure 2.1 (page 22) illustrates the computation of a_{23}^{-1}. In the first step (forward phase with sparse right-hand side), nodes 3, 5 and 6 are visited; in the second step (backward phase with sparse solution), nodes 6, 5, 3 and 2 are visited. One can notice that the branch 1 4 of the tree is not traversed at all; it is pruned, and the subset 2 3 5 6 to be traversed is the pruned tree.

Note that the method that we have just discussed corresponds to the parenthesization

$$U^{-1} \quad L^{-1}e_j \quad {}_i \tag{2.1}$$

of the general formula:

$$\begin{aligned} a_{ij}^{-1} &= e_i^T A^{-1} e_j \\ &= e_i^T U^{-1} L^{-1} e_j \end{aligned}$$

There are, however, two other parenthesizations:

$$a_{ij}^{-1} = \quad e_i^T U^{-1} \quad L^{-1} \quad {}_j \tag{2.2}$$

$$= \quad e_i^T U^{-1} \quad L^{-1} e_j \tag{2.3}$$

We have the following theorem regarding the equivalence of these three parenthesizations, when L and U are computed and stored in such a way that their sparsity patterns are the transposes of each other. In a more general case, the three parenthesization schemes may differ when L and U^T have different patterns.

Theorem 2.2
The three parenthesizations for computing a_{ij}^{-1} given in equations (2.1), (2.2) and (2.3) access the same columns of the factor L and the same rows of the factor U.

Proof. Consider the following three computations

$$v^T = e_i^T U^{-1} \tag{2.4}$$

$$y = L^{-1} e_j \tag{2.5}$$

$$z_j = (v^T L^{-1})_j \tag{2.6}$$

The parenthesization (2.2) is computed using (2.4) and (2.6); the parenthesization (2.3) is computed using (2.4) and (2.5). As we showed before, our parenthesization (2.1) requires accessing the columns of L associated with the nodes in $\mathcal{P}(j)$, the unique path from the root to node j, and the rows of U associated with the nodes in $\mathcal{P}(i)$, the unique path from the root to node i. We now show that the other parenthesizations require accessing the same parts of the factors.

As $v = U^{-T} e_i$ and U^T is lower triangular, by Lemma 2.1, computing v requires accessing the rows of U associated with the nodes in $\mathcal{P}(i)$. Consider $z = L^{-T} v$. As L^T is upper triangular, by Lemma 2.2, z_j requires accessing columns of the L factor associated with the nodes in $\mathcal{P}(j)$, since $v_N = (U^{-T} e_i)_N = 0$ for an irreducible A. Hence the parenthesization (2.2) accesses the same parts of the factors as the parenthesization (2.1). Computing $y = L^{-1} e_j$ requires accessing the columns of L associated with the nodes in $\mathcal{P}(j)$. Here we again use the fact that A is irreducible, and hence the parenthesization (2.3) accesses the same parts of the factors. □

Although we use equation (2.1) in our algorithms and implementation, we note that the parenthesization (2.3) can be advantageous while computing only the diagonal entries with the factorizations $L L^T$ or $L D L^T$ (with a diagonal D matrix), because in this case we need to compute a single vector and compute the square of its norm. This formulation can also be useful in a parallel setting where the solves with L and U can be computed in parallel whereas, in the other two formulations, the solves using one of the factors have to wait for the solves using the other to complete. We also note that if the triangular solution procedures for $U^{-T} e_i$ and $L^{-1} e_j$ are available, then one can benefit from this parenthesization if the number of row and column indices involved in the requested entries is smaller than the number of these requested entries. In this case, many of the calculations can be reused if one computes a set of vectors of the form $U^{-T}e_i$ and $L^{-1}e_j$ for different i and j and obtains the requested entries of the inverse by computing the inner products of these vectors.

Note that computing inverse entries has been our main motivation during this study, but, as mentioned above, this is essentially the same thing as computing selected entries of the solution of a system with sparse right-hand sides, and the ideas we present throughout this study apply equally to both situations.

Finally, to indicate the importance of exploiting the sparsity of the right-hand sides and solution vectors, we show in Table 2.1 results for a large matrix from our experimental set (see Table 1.1). In this table, we show the size of the factors, the total amount of factors accessed and the execution time of the MUMPS solver for the computation of a set of inverse entries. One can notice that exploiting sparsity reduces the solution time by a factor of almost 25 on this example. The volume of factors to be accessed, which is representative of the number of times nodes of the elimination tree are traversed, is reduced by a factor 160.

Matrix	Factor size	Factors loaded (in GB.)		Time (in secs.)	
	(in GB.)	Dense	Sparse	Dense	Sparse
FLUX-2M	34.8	103955	638	1041555	43947

Table 2.1: Random 10% diagonal entries of the inverse are computed. The columns Dense correspond to solving the linear systems as if the right-hand side vectors were dense. The columns Sparse correspond to solving the linear system while exploiting the sparsity of the right-hand side vectors. The computations were performed on the `Pret` system defined in Section 1.3.4.

2.3 Contributions

Exploiting sparsity in the right-hand sides and solution vectors has been studied in the context of Slavova's PhD thesis [84]; Slavova examined computations of inverse entries and nullspace bases in an out-of-core context, where exploiting sparsity enables a significant reduction in access to the factors. This work was the baseline for our study; we extend this previous contribution by focusing on the following points:

Reducing the amount of memory used in the solution phase by exploiting sparsity was not considered in [84]. This turns out to be critical when running large experiments and we have addressed this point. In Chapter 3, we describe different storage schemes for the solution when sparse right-hand sides and/or sparse solution are used; they significantly reduce the amount of memory used during the solution phase, thus enabling us to process much larger blocks of right-hand sides at once. We first present some improvements for the storage scheme used with dense right-hand sides. We then present a storage scheme based on the path theorem mentioned in the previous section and that can be used in any case that involves sparse right-hand sides and/or sparse solution. We finally present a storage scheme based on the height the tree, that can be used when computing diagonal entries of the inverse. We especially emphasize implementation issues in a parallel context, as this is not trivial.

Processing large sets of multiple sparse right-hand side was considered in [84]. Since it is usually not possible to process large sets of right-hand sides at once because of memory constraints (even when using a very space-saving storage scheme), the set of right-hand sides has to be partitioned into blocks. In her PhD thesis, Slavova showed that the way right-hand sides are partitioned determines which parts of the tree are pruned for each block, and thus permuting and partitioning the right-hand sides strongly influences the volume of factors to be loaded during the solution phase. Slavova studied this partitioning problem in the case where only diagonal entries are computed and suggested a few heuristics. We have extended this study: we showed that this partitioning problem is strongly NP-complete, and developed heuristics and models even for the most general case (i.e., not only computing diagonal entries). The compression of the solution space described in Chapter 3 also enabled us to run much larger experiments. All these points are described in Chapter 4.

Reducing the computation cost in an in-core context was not addressed in [84]. This problem is tackled in Chapter 5 in two different ways. First, we show that when sparsity is not exploited within blocks (but only *between* blocks), the way the right-hand sides are partitioned strongly influences the number of operations. Interestingly, this problem is quite different from the above-mentioned partitioning problem (reducing the volume of factors in an out-of-core context). We have developed different models and heuristics that have been experimented within the PDSLin solver, where exploiting sparsity in the right-hand sides is interesting when computing the Schur complement. Secondly, we have shown that exploiting sparsity *within* each block of right-hand sides can strongly reduce the number of operations: we have implemented and experimented this capability in the MUMPS solver.

Slavova gave some hints on how to combine tree pruning and tree parallelism. We extended these ideas and were able to push further the parallel performance. We show that even though processing multiple blocks of right-hand sides might seem embarrassingly parallel, there is no choice but to process them one after another, at least in a distributed-memory context. We then show that reducing the size of the pruned tree (which decreases the computational cost) and enabling tree parallelism are contradictory objectives if sparsity is not exploited *within* blocks. Finally, we show that combining an *interleaving* strategy with exploiting sparsity within blocks enables good parallel performance. This is presented in Chapter 6, the last chapter of this part of the thesis dedicated to the solution phase.

2.4 The multifrontal solution phase

We briefly recall how the parallel multifrontal solution phase works. Assuming that we have *NBRHS* right-hand sides and a block size *B* (the number of right-hand sides held and processed as a single block), the solution phase consists of the following loop:

```
for $i = 1$ to $NBRHS$ by $B$ do
    $ibeg \quad i$
    $iend \quad \min(ibeg + B \quad 1 \; NBRHS)$
    Forward elimination on columns $ibeg : iend$
    Backward substitution on columns $ibeg : iend$
end for
```

We describe the forward elimination and the backward substitution algorithms. We depict a simplified version of what was done in MUMPS before the beginning of this study. The algorithms correspond to an asynchronous implementation in a distributed memory environment; they take advantage of tree parallelism and node parallelism as in the factorization phase by inheriting the mapping and the use of Type 1, Type 2 and Type 3 nodes. Our point here is to highlight the memory structures used during the solution phase; we omit technical details like delayed pivots, using ScaLAPACK on the root node, etc. We only present algorithms that deal with a single right-hand side, but they can naturally be extended to process a block of B right-hand sides at the same time: every array has B columns instead of one, and level 3 BLAS (instead of level 2 BLAS) operations are performed (`_TRSM` and `_GEMM` instead of `_TRSV` and `_GEMV`).

As we will see, both the forward elimination and the backward substitution use similar data structures: an array that scales with the number of processes (i.e., its total size is N and is distributed across the processes) called `WRHS` and an array of size N per process (Wb in the forward phase, $Wsol$ in the backward phase). This was acceptable in applications with dense right-hand sides (the whole set of right-hand sides of size $N \times NBRHS$ is already allocated in the user space) and on moderate numbers of processes, but it became a bottleneck in applications involving sparse right-hand sides, where the number of right-hand sides is usually large (e.g., $NBRHS = N$ when computing all the diagonal entries of the inverse) and where large block sizes are desirable (see the subsequent chapters). Therefore, it became crucial to compress the solution space; this is described in the next chapter.

2.4.1 The forward phase

The forward elimination process is described in Algorithm 2.2; it is the generalization of the sequential algorithm described in Section 1.1. We highlight the two important data structures that are used in the algorithm: Wb and `WRHS`. Wb is an array (of real or complex entries) of size N used to store temporary data (contributions of the form $l_{ij}y_j$), avoiding the use of a stack as is done in the factorization. First, it is initialized with the values of the right-hand sides that correspond to fully-summed variables mapped onto $Myid$ (process considered):

$$\begin{array}{ll} Wb(i) = b(i) & \text{if variable } i \text{ is part of the pivot block of a front mapped onto } Myid \\ Wb(i) = 0 & \text{otherwise} \end{array}$$

Then it is used to accumulate the contributions from one node to another. An important property has to be highlighted: every time a contribution is computed (in array $Wtmp2$) and sent to the parent of the current node, *the corresponding entries in Wb are reset to 0.* This is mandatory since a process uses the same Wb array for all nodes mapped onto it; thus nodes mapped onto the same process can use the same slots in Wb to store temporary contributions. We illustrate in Figure 2.2 that if entries of Wb are not reset to zero, a contribution might be counted more than once. In this example, we denote the array Wb mapped onto process P_i as Wb_i. We follow what happens to variable 5, i.e., which contributions are stored in $Wb(5)$ on every process and how x_5 is finally computed.

First, nodes 1 and 2 are processed in parallel, on P_0 and P_1 respectively. At node 1, x_1 is computed and the contribution $l_{51}\ x_1$ is stored in $Wb_0(5)$; it is then sent to P_2 (because node 4, the parent of node 1, is mapped onto P_2). Similarly, at node 2, x_2 is computed and the contribution $l_{52}\ x_2$ is stored in $Wb_1(5)$ and sent to P_2. Assume that P_2 receives the messages from P_0 and P_1 before activating node 3 (this is one of the many potential schedulings); x_3 is computed and the contribution $l_{53}\ x_3$ is added in $Wb_2(5)$, that already contains $l_{51}\ x_1\ \ l_{52}\ x_2$ (contributions from P_0 and P_1). Thus $Wb_2(5) =\ l_{53}\ x_3\ \ l_{51}\ x_1\ \ l_{52}\ x_2$ is sent to P_3 that holds the parent of node 3. Here we assume that $Wb_2(5)$ is *not reset to zero* and show that this leads to a wrong result. Node

Algorithm 2.2 Forward elimination algorithm.

```
Main algorithm (forward elimination)
/* Input: the right-hand side                                                  */
/*        POSinWRHS, position of a node in WRHS                                */
/* Output: WRHS, components of the solutions corresponding to variables        */
/*                in the pivot block of nodes mapped onto $Myid$               */
/* Work arrays: $Wb$ (size $N$), $Wtmp1$ and $Wtmp2$ (size maximum front size) */

Initialize the pool with the leaf nodes mapped onto $Myid$
Expand into $Wb$ the entries of the right-hand side corresponding to variables in the
  pivot block of nodes mapped onto $Myid$
while termination not detected do
    if a message is available then
        Process the message
    else if the pool is not empty then
        Extract a node $\mathcal{N}$ from the pool
        Fwd_Process_Node($\mathcal{N}$)
    end if
end while

Fwd_Process_Node($\mathcal{N}$)
/* $L_{11}$ and $L_{21}$ are the $L$ factors of $\mathcal{N}$ */
/* $Pparent$ is the master process of the parent of $\mathcal{N}$ */
$Wtmp1$      entries of $Wb$ corresponding to fully summed variables of $\mathcal{N}$
$Wtmp1$      $L_{11}^{-1}$   $Wtmp1$ (_TRS_)
Copy $Wtmp1$ into WRHS (contiguous locations given by POSinWRHS($\mathcal{N}$))
$Wtmp2$      entries of $Wb$ corresponding to row indices of $L_{21}$
Reset the corresponding entries of $Wb$ to zero
if $\mathcal{N}$ is of Type 1 then
    $Wtmp2 = Wtmp2$   $L_{21}$   $Wtmp1$ (_GEM_)
    Send the resulting contribution ($Wtmp2$) to $Pparent$
else if $\mathcal{N}$ is of Type 2 then
    for all $Islave$ slave of $\mathcal{N}$ do
        Send to $Islave$ $Wtmp1$ and rows of $Wtmp2$ matching rows owned by $Islave$
    end for
end if

On reception of $Wtmp1$ + rows of $Wtmp2$ by a slave process
Multiply rows of $L_{21}$ owned by the slave by $Wtmp1$ and subtract from $Wtmp2$ (_GEM_)
Send the resulting contribution to $Pparent$

On reception of a contribution corresponding to $\mathcal{N}$ by $Pparent$
Assemble the contribution into $Wb$ (scatter and add)
if all contributions corresponding to the parent of $\mathcal{N}$ have been received then
    Insert parent of $\mathcal{N}$ into the pool
end if
```

4 is activated: x_4 is computed and the contribution l_{54} x_4 is added in $Wb_2(5)$ which already contains l_{53} x_3 l_{51} x_1 l_{52} x_2 because it was not previously reset to 0. Therefore,

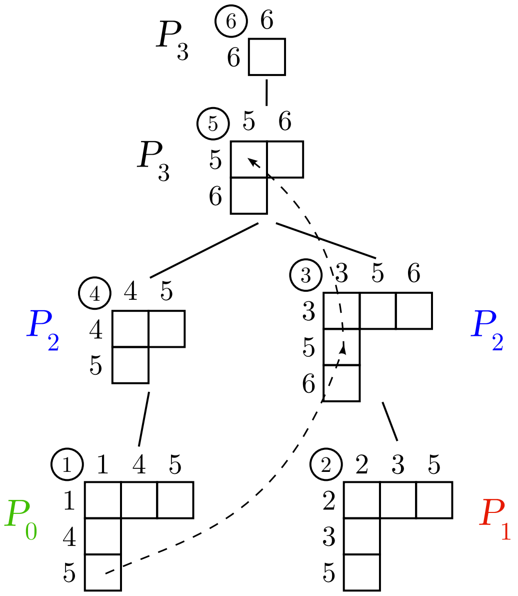


Figure 2.2: Forward solution phase: the structure (indices) of the L and U factor is indicated at each node. If node 3 is activated before node 4, then l_{51} x_1, the contribution from node 1 to component 5 of the solution, passes through node 3 and not through node 4.

P_3 receives l_{53} x_3 l_{51} x_1 l_{52} x_2 from node 3 and l_{41} x_4 l_{53} x_3 l_{51} x_1 l_{52} x_2 from node 4: this is clearly wrong, as some contributions are present twice, and thus x_5 will be wrong.

If $Wb_2(5)$ is reset to zero after P_2 sends it to P_3, then, when node 4 is processed, the update l_{54} x_4 is added in $Wb_2(5)$ which contains zero. Thus, P_3 receives l_{53} x_3 l_{51} x_1 l_{52} x_2 from node 3 and l_{54} x_4 from node 4; this time the contributions are present only once and the result is correct. It is worth noticing that contributions to unfactored variables do not necessarily follow regular paths in the tree: in our example, the contribution from node 1 to variable 5 is sent and accumulated into node 3 (that does not lie in the same branch), while node 4, the child of node 5, does not receive this contribution. This behavior is due to the asynchronous nature of the algorithm and to the fact that no property can be assumed for the node to process mapping.

The other important data structure is `WRHS`: it scales with the number of processes and contains, for each process, only the entries corresponding to fully summed variables mapped onto this process. At the end of the forward elimination process, `WRHS` contains the (distributed) solution of $Lx = b$. Each process holds an array of indirect addressing (that we call an indirection array in the following) `POSinWRHS` which is precomputed to obtain, for each node of the tree, its position in `WRHS`. Note that in the case of multiple right-hand-sides (processed by blocks), `POSinWRHS` needs to be computed only once for all blocks.

At the end of the forward phase, Wb can be freed and `WRHS` is a perfectly compact structure holding the solution of the forward elimination (its total size, over the processes, is N). Having such a compact structure is very useful in applications where the forward and backward phases are separated, for example when using the Schur complement method (see Section 1.3.2). In these applications, the forward phase is performed for all the blocks of right-hand sides, and then the backward phase is performed: the partial solutions from the forward phase for all the right-hand sides have to be stored, thus the need to have a compressed structure.

If the right-hand side is sparse, the tree pruning procedure is performed prior to the forward elimination. Then, the only difference in the forward phase is that the pool of tasks is initialized with the list of leaves of the pruned tree (which might not be leaves in the complete tree).

2.4.2 The backward phase

The backward substitution process is described in Algorithm 2.3. We consider the unsymmetric case and thus rely on the U factors: U_{11} and U_{12} are the U factors at a given node $\mathcal{N}$. In the symmetric case, U_{11} would be L_{11}^T and U_{12} would be L_{21}^T. Throughout the algorithm, the array $Wsol$ (one array of size N per process) is used to save parts of the solution, with the property that the solution for variable i will at least be available in $Wsol(i)$ on the process in charge of the pivot block containing i. Note that, for Type 2 nodes, each slave process performs a matrix-vector product and sends the result back to the sender, thus implying more communications than in the forward elimination algorithm, where slave processes do not have to send anything back to their corresponding master process (they communicate directly with the parent node). The algorithm relies on the fact that the solution entries that a node requires in order to compute the entries associated with its fully-summed variables are the entries corresponding to its contribution block, as the following left-looking equation shows: $x_i = \quad y_i \quad \sum_{j=i+1}^{n} u_{ij} \quad x_j \quad /u_{ii}$. These indices also belong to the structure of the parent of that node (either as fully-summed variables or as contribution block indices), therefore, it is enough that a node passes to its children the partial solution corresponding to its indices.

If the solution is sparse (i.e., if selected entries are requested), then the tree pruning procedure is performed before the backward substitution. Then, during the backward phase, the only difference is that a given node does not send pieces of solution to all its children, but only to those that belong to the pruned tree. The solution phase ends when the leaves of the pruned tree are reached. Finally, when extracting the solution, only the requested entries are kept.

Algorithm 2.3 Backward substitution algorithm.

```
Main Algorithm (backward substitution)
/* Input: WRHS, output of the forward elimination                    */
/*        POSinWRHS, position of a node in WRHS                       */
/* Output: the solution                                                */
/* Work arrays: $Wsol$ (size $N$), $x_1$, $x_2$, and $y_1$            */

Initialize the pool with the root node(s) mapped onto $Myid$
while termination not detected do
    if a message is available then
        Process the message
    else if the pool is not empty then
        Extract a node $\mathcal{N}$ from the pool
        Bwd_Process_node($\mathcal{N}$)
    end if
end while
Extract solution from $Wsol$ arrays

Bwd_Process_node($\mathcal{N}$)
$x_2$    entries of $Wsol$ corresponding to columns of $U_{12}$
if $\mathcal{N}$ is of Type 1 then
    $y_1$    entries of WRHS corresponding to fully summed variables of $U_{11}$
    Solve $U_{11}x_1 = y_1 \quad U_{12}x_2$ for $x_1$ (_GEM_ and _TRS_)
    Expand $x_1$ into $Wsol$
    Send partial solution $x_1 \ x_2$ to masters of children nodes
else if $\mathcal{N}$ is of Type 2 then
    Send (distribute) entries of $x_2$ to the slaves, according to their structure
end if

On reception of $x_1 \ x_2$ from $\mathcal{N}$
Update my view of the solution (scatter into $Wsol$)
Insert children of $\mathcal{N}$ mapped onto $Myid$ into the local pool

On reception of parts of $x_2$ by a slave of $\mathcal{N}$
Multiply the part of $U_{12}$ mapped onto $Myid$ by the piece of $x_2$ just received (_GEM_)
Send the negative of the result back to the master process of $\mathcal{N}$

On reception of a portion of   $U_{12}x_2$ from a slave process by a master process
Add the contribution to WRHS
if this is the last update (all slaves sent their part) then
    $y_1$    entries of WRHS corresponding to fully summed variables of $U_{11}$
    Solve $U_{11}x_1 = y_1$ for $x_1$ (_TRS_)
    Expand $x_1$ into $Wsol$ using POSinWRHS
    Send partial solution $x_1 \ x_2$ to the master processes of children nodes
end if
```

Chapter 3

Compressing the solution space

We mentioned in the previous chapter that our baseline algorithms for the forward elimination and the backward substitution require a work array of size N, the size of the matrix, on *every* process. This turned out to be critical in applications involving large numbers of right-hand sides. In these applications, one would like to use large blocks of right-hand sides at the same time as this is often very beneficial for the performance of the BLAS and to reduce the number of accesses to the factors (we discuss these points in the following chapters). For example, setting a block size of 1000 on a problem of size 10,000,000 would imply the allocation 10 000 000 × 1 000 × 8 bytes per entry $= 80$ GB per process which is of course infeasible. Furthermore, in the case where the right-hand sides and/or the solution vectors are sparse, one could expect the memory consumption to be reduced along the lines of the path theorem by Gilbert & Liu presented in the previous chapter. Since only the variables that lie in a path or a few paths in the tree are nonzero in the solution vectors, a workspace of size (N) is not needed. In this chapter, we describe storage strategies that significantly compress the solution space: the scheme we propose for the case of dense right-hand sides scales almost perfectly with the number of processes (i.e., the total memory consumption is of order (N)); in the sparse case, we propose two schemes: one fits the most general case (it works for any set of sparse right-hand sides and any set of selected entries in the solution) and works in a parallel context, and the other is reserved for the sequential case where the right-hand sides and the solution vectors have only one nonzero component. This is typically useful when computing diagonal entries of the inverse.

These storage schemes turn out to exhibit significant compression rates, especially in the sparse case, and they also provide interesting performance gains as they enforce much more locality than the baseline strategy.

3.1 Dense storage scheme

3.1.1 Idea

In this section we consider the case where both the right-hand sides and the solution vectors are dense (i.e., all the components of the solution are requested). We can significantly compress the solution space using the idea that a given process does not need to access *all* the N components of the solution vectors. We adapt the `WRHS` array used in Algorithms 2.2 and 2.3 so that all the work can be done in that array (Wb and $Wsol$ are thus no longer needed) and so that, on each process, `WRHS` is compressed as much as possible. Each process also has an indirection array to access `WRHS`. We list the properties from which we derive the construction of `WRHS` and the indirection arrays:

Anywhere in the tree, both in the forward elimination and the backward substitution, only master processes read from and write to WRHS: they *write* the components of the solution that they have computed, the contributions that they receive from their children (in the forward elimination), and the partial solutions that they receive from their parent node (in the backward substitution); they also *read* pieces of WRHS to be sent to their slaves, or children nodes, or parent nodes. On the other hand, slave processes of Type 2 nodes never manipulate WRHS. Therefore, for any process p, the indirection is defined only for variables that belong to a node whose master is p. For the other variables, the indirection is undefined and no space needs to be allocated in WRHS. Therefore, building the indirection consists of visiting, for each process p, the list of nodes for which p is the master process.

A given process might manipulate different variables in the forward phase and backward phase because of pivoting. We illustrate this in Figure 3.1 which represents an extreme case of what can happen structurally because of numerical issues: (a) is an assembly tree prior to factorization, and (b) is the same assembly tree after factorization; some pivoting has occurred: a pivot has been delayed at nodes 1 and 2, two pivots have been delayed at node 3 and a pivot has been delayed at node 4. This results in some asymmetry in the nodes; not only do the order of row indices and column indices differ (e.g., at node 1, row indices are 2 1 5 while column indices are 1 2 5), but also some variables might appear as row indices but not as column indices and vice versa (e.g., at node 3, variable 1 appears as a row index but is not in the list of column indices). Therefore, we need two indirection arrays: one for the forward elimination (row indices) and one for the backward substitution (column indices). We call them POSinWRHS_row and POSinWRHS_col respectively. They are of size N and are built at the same time.

For any process p, the length of WRHS is therefore the maximum between the number of variables that this process touches during the forward elimination (i.e., row indices) and the number of variables that it touches during the backward substitution (column indices). These two numbers might be different: for example, in Figure 3.1(c), P_1, the master of nodes 2 and 5, is in charge of seven row indices (4 8 9 5 3 6 7 , the union of the row indices of nodes 2 and 5) but only six column indices (3 8 9 7 4 6). In any case, the total number of indices that p touches is smaller than the sum of the size of the fronts mapped onto p, since some variables might appear several times (at most once as eliminated variables, but possibly many times as variables of contribution blocks). Summing over the number of processes, the total length of the solution space is larger than N: the difference comes from the contribution blocks, more specifically from variables that appear as contribution blocks on a process but as eliminated variables on another one. Consider for example row indices in Figure 3.1: 5 and 8 appear as eliminated variables on P_1 and as indices of contribution blocks on P_0; similarly, 3, 6 and 7 appear as row eliminated variables on P_0 and in contribution blocks on P_1. Hence, because of these five variables, the sum of the size of WRHS over P_0 and P_1 is $N + 5$. Therefore, this new WRHS does not scale perfectly: intuitively, its length will increase with the number of processes. In a sequential context, the length of WRHS is N since only one process is in charge of all the N variables.

We want to enforce some data locality *within* each node:

We want to improve data locality during both the forward phase and the backward phase: the eliminated variables of each node should correspond to consecutive slots in WRHS, therefore, when we build the two indirection arrays, we traverse the list of eliminated variables of each node mapped onto the considered process and assign them to consecutive slots in WRHS. In the next step, we also do this with variables of contribution blocks but here we cannot guarantee that all the variables will get consecutive slots in WRHS, since variables that also appear as eliminated variables are already assigned to a slot. For example, in Figure 3.1, the column indices corresponding to the contribution block of node 2 are not assigned to consecutive positions in WRHS: indeed, since 7 also appears as an eliminated variable (for node 5), it is grouped with the other eliminated variables of that node, namely 8 and 9. However, given the way variables of contribution blocks are ordered (variables corresponding to the block of eliminated variables of the parent, then variables corresponding to the block of eliminated variables of the grandparent, etc.), it is very likely that some locality will be achieved.

We try to enforce some locality *between* the forward and backward phases: since a component x_1 of the solution is computed as $U_{11}x_1 = y_1 \quad U_{12}x_2$ where y_1 is a component of the solution from the forward phase, it is probably beneficial to have x_1 and y_1 in the same slots in WRHS. We try to enforce this by assigning row eliminated indices and column eliminated indices of a node to the same slots in WRHS. In cases where there is no pivoting, these lists are the same and thus x_1 and y_1 are located in the same portion of WRHS. For example, in the assembly tree of Figure 3.1, there will be some locality between the two phases for variables 8 and 9 at node 5 since they are assigned to the same slots in WRHS for the forward and the backward phase; this is due to the fact that they have not been affected by pivoting and still have symmetric positions in the tree.

We want to enforce some locality at the tree level: the eliminated variables of nodes that are consecutive in the postorder should have consecutive slots in WRHS. This will improve locality from one node to another (at least in subtrees mapped onto the same process). Therefore, when constructing the indirections, we traverse the list of nodes following a postorder (more precisely: the postorder used in the sequential factorization).

3.1.2 Construction

Using the above-described properties, we derive in Algorithm 3.1 the construction of the two indirection arrays POSinWRHS_row and POSinWRHS_col. It consists of two passes of the list of nodes mapped onto a given process *Myid*, relying on the postorder:

1. The first pass of the algorithm assigns slots in WRHS, i.e., it assigns the two indirection arrays POSinWRHS_row and POSinWRHS_col, for eliminated variables.

2. The second pass assigns slots in WRHS for variables that correspond to contribution blocks and that have not been assigned during the first pass. Contrary to the first pass, a different number of row and column variables can be processed, thus the two indirection arrays are treated separately.

Note that we still have to allocate two arrays of size N, namely POSinWRHS_row and POSinWRHS_col, on every process; however, this reduces significantly the size of WRHS and is far better than the baseline storage scheme, that requires an array of size $N \times B$ to be allocated on every process (Wb in the forward phase, $Wsol$ in the backward). Overall, the size of the solution space is significantly reduced, even counting the size of the indirection arrays, especially for large block sizes B. We illustrate the gains in storage and in performance of this new storage scheme in Section 3.4.1.

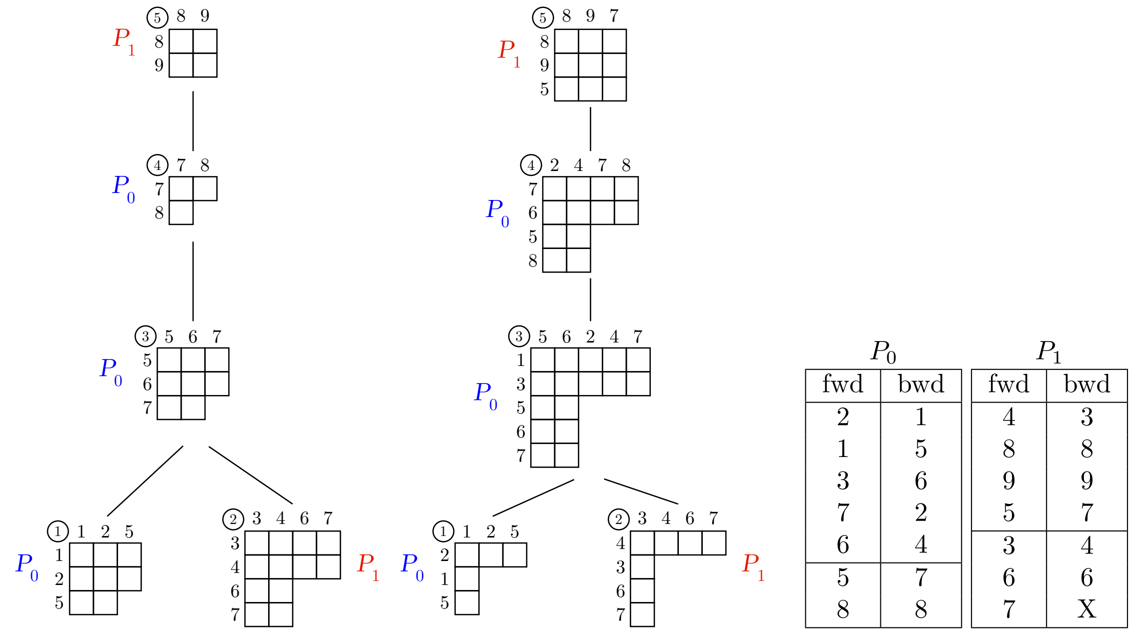
Algorithm 3.1 Construction of the indirection arrays for the compressed solution space (dense case).

```
$m$    0; POSinWRHS_row    0; POSinWRHS_col    0
**for** any node $\mathcal{N}$ mapped onto $Myid$, following the postorder **do**
    /* $npiv_{\mathcal{N}}$ is the number of eliminated pivots */
    /* $nfront_{\mathcal{N}}$ is the size of the front */
    /* $row\_list_{\mathcal{N}}$ is the list of row indices */
    /* $col\_list_{\mathcal{N}}$ is the list of column indices */
    **for** $k = 1$ **to** $npiv_{\mathcal{N}}$ **do**
        $i$    $row\_list_{\mathcal{N}}(k)$; $j$    $col\_list_{\mathcal{N}}(k)$
        $m$    $m + 1$
        POSinWRHS_row($i$) $= m$
        POSinWRHS_col($j$) $= m$
    **end for**
**end for**
$m_{row}$    $m$; $m_{col}$    $m$
**for** any node $\mathcal{N}$ mapped onto $Myid$, following the postorder **do**
    **for** $k = npiv_{\mathcal{N}} + 1$ **to** $nfront_{\mathcal{N}}$ **do**
        $i$    $row\_list_{\mathcal{N}}(k)$; $j$    $col\_list_{\mathcal{N}}(k)$
        **if** POSinWRHS_row($i$) $= 0$ **then**
            $m_{row}$    $m_{row} + 1$
            POSinWRHS_row($i$)    $m_{row}$
        **end if**
        **if** POSinWRHS_col($j$) $= 0$ **then**
            $m_{col}$    $m_{col} + 1$
            POSinWRHS_col($j$)    $m_{col}$
        **end if**
    **end for**
**end for**
```

3.2 A storage scheme based on a union of paths

3.2.1 Idea

We describe in this section a storage scheme dedicated to the case where both the right-hand sides and the solution vectors are sparse (e.g., computation of inverse entries). If the right-hand side or the solution is dense, there is not much to do in terms of memory compression since one of the two triangular solution phases will require a workspace of size (N), and we use the storage scheme described in the previous section. If both the right-hand sides and the solution vectors are sparse then, both in the forward elimination and the backward substitution, only the nodes that lie in the pruned tree need to be traversed, and only the corresponding variables need to be stored in the solution vector WRHS. The storage scheme we present here is therefore a simple combination of the tree



(a) Assembly tree before factorization.

(b) Assembly tree after factorization.

P_0 fwd	P_0 bwd	P_1 fwd	P_1 bwd
2	1	4	3
1	5	8	8
3	6	9	9
7	2	5	7
6	4	3	4
5	7	6	6
8	8	7	X

(c) Structure of `WRHS`.

Figure 3.1: Example of construction of `WRHS`. (a) is the initial assembly tree, before factorization; during the factorization, some pivoting occurs, which modifies the structure of the tree and generates some asymmetry in the front indices (b). (c) is the structure of `WRHS` (for each position in `WRHS`, the index of the row/column variable that occupies this position is shown); fwd refers to the forward phase and bwd refers to the backward phase. An X indicates that the position is not used.

pruning procedure and the dense storage scheme previously described. Since it strongly relies on the pruned tree, which is the union of the paths from a set of targets (pattern of the right-hand and requested components of the solution) to the root node, we refer to this storage scheme as the *union sparse scheme.*

The construction of the two indirection arrays is the combination of Algorithm 3.1 (dense storage scheme) and Algorithm 2.1 (tree pruning): instead of traversing the whole set of nodes of the tree, the algorithm traverses the pruned tree by following each path from a target node up to the root node, and tries to enforce locality:

within nodes: eliminated variables of a given node are assigned to contiguous positions in `WRHS`. When possible (cf. the reasons mentioned in Section 3.1), contribution block variables are assigned to consecutive slots in `WRHS` as well.

between nodes: the algorithm assigns eliminated variables of nodes that belong to the same branch to consecutive sections in `WRHS`. Since the solution phase will follow these paths, this provides some locality.

between the forward phase and the backward phase: eliminated variables of nodes that are traversed both during the forward elimination and during the backward substitution (note that some nodes might be traversed during one phase but not the other) are assigned to the same slots in `WRHS`.

`WRHS` is built such that its structure is the following:

1. Fully-summed variables that are reached during the forward elimination.
2. Fully-summed variables that are reached during the backward substitution and that do not appear in the forward elimination.
3. Variables that only appear in contribution blocks.

The construction follows three main steps:

1. `POSinWRHS_row` is set for variables that appear as eliminated variables in the forward elimination phase, by traversing the pruned tree, i.e., by following paths from nodes in the nonzero pattern of the right-hand sides up to the root, as in Algorithm 2.1. `POSinWRHS_col` is also set for nodes that belong to the pruned tree corresponding to the forward elimination, with negative values m (m being the current position in `WRHS`); this will be useful for telling which nodes are encountered in both phases.
2. `POSinWRHS_col` is set for variables that appear as eliminated variables in the backward substitution phase. If a negative value m is found, then it is set to m: this way, row eliminated variables and column eliminated variables of a node visited both in the forward phase and the backward phase are assigned to the same slots in `WRHS`.
3. `POSinWRHS_col/row` are set for contribution block variables.

3.2.2 Construction

We describe in Algorithm 3.2 the construction of the indirection arrays. Figure 3.2 illustrates the structure of `WRHS` for the computation of $A^{-1}e_2$ $_3$ with the assembly tree of Figure 3.1(b). Since the right-hand side is e_2, the pruned tree for the forward elimination is the path from node 1 (node where 2 is a row eliminated variable) to the root node. Since only component 3 of the solution is requested, the pruned tree for the backward substitution is the path from node 2 (node where 3 is a column eliminated variable) to the root node. Consider process P_1: in the forward phase, the only node of the pruned tree mapped onto P_1 is node 5; node 5 has three variables thus three slots are required in `WRHS` to store the solution. In the dense case, seven slots were needed (for the four variables in node 2 and the three variables in node 5). Note that, on P_0, the row eliminated variables and the column eliminated variables of nodes 3 and 4 are assigned to the same positions in `WRHS` (slots 2 to 5); however, the first slot in `WRHS` is used for the row variable 2, which belongs to node 1; node 1 is not visited in the backward phase, therefore the first slot in `WRHS` is not used in the backward phase (it is designated by an X in the figure).

P_0		P_1	
fwd	bwd	fwd	bwd
2	X	8	8
1	5	9	9
3	6	5	7
7	2	X	3
6	4	X	4
5	7	X	6
8	8		

Figure 3.2: Structure of `WRHS` for the computation of $A^{-1}e_2$ $_3$ with the assembly tree of Figure 3.1(b). fwd refers to the forward phase and bwd refers to the backward phase.

Algorithm 3.2 Construction of the indirection arrays for the compressed solution space (union sparse scheme).

```
/* Input: targets_rhs, nonzero pattern of the block of right-hand sides */
/*        targets_sol, requested components of the solution */

/* Step 1: following the pruned tree for the forward phase, assign slots to row eliminated
variables and mark column eliminated variables */
to_process    false; $m$    0
for $l$ in targets_rhs do
    $\mathcal{N}$    the node variable $l$ belongs to.
    /* Follow the path from $\mathcal{N}$ to the root node */
    go_up    true
    while go_up do
        to_process($\mathcal{N}$)    true
        /* $npiv_{\mathcal{N}}$ is the number of eliminated pivots */
        /* $nfront_{\mathcal{N}}$ is the size of the front */
        /* $row\_list_{\mathcal{N}}$ is the list of row indices */
        /* $col\_list_{\mathcal{N}}$ is the list of column indices */
        if $\mathcal{N}$ is mapped onto $Myid$ then
            for $k = 1$ to $npiv_{\mathcal{N}}$ do
                $i$    $row\_list_{\mathcal{N}}(k)$; $j$    $col\_list_{\mathcal{N}}(k)$
                $m$    $m + 1$
                POSinWRHS_row($i$) = $m$
                POSinWRHS_col($j$) =    $m$
            end for
        end if
        if $\mathcal{N}$ is not the root node then
            $\mathcal{N}$    parent of $\mathcal{N}$
            if to_process($\mathcal{N}$) then /* $\mathcal{N}$ has been met before */
                go_up    false
            end if
        else /* Nothing more to do */
            go_up    false
        end if
    end while
end for

/* Step 2: following the pruned tree for the backward phase, assign slots to column eliminated
variables */
m_row    $m$; m_col    $m$
to_process    false
for $l$ in targets_sol do
    $\mathcal{N}$    the node variable $l$ belongs to.
    /* Follow the path from $\mathcal{N}$ to the root node */
    go_up    true
    while go_up do
        to_process($\mathcal{N}$)    true
```

Continued on next page

Algorithm 3.2: continued

```
        if $\mathcal{N}$ is mapped onto $Myid$ then
            for $k = 1$ to $npiv_{\mathcal{N}}$ do
                $j$    $col\_list_{\mathcal{N}}(k)$
                if POSinWRHS_col(j) = 0 then
                    $m\_col$    $m\_col + 1$
                    POSinWRHS_col($j$) = $m\_col$
                else /* A negative value   $m$ is met; it is set to $+m$ */
                    POSinWRHS_col($j$) =   POSinWRHS_col($j$)
                end if
            end for
        end if
        if $\mathcal{N}$ is not the root node then
            $\mathcal{N}$    parent of $\mathcal{N}$
            if $to\_process(\mathcal{N})$ then /* $\mathcal{N}$ has been met before */
                $go\_up$    false
            end if
        else /* Nothing more to do */
            $go\_up$    false
        end if
    end while
end for

/* Step 3a: assign slots to row variables corresponding to contribution blocks */
$to\_process$    false
for $l$ in $targets\_rhs$ do
    $\mathcal{N}$    the node variable $l$ belongs to.
    /* Follow the path from $\mathcal{N}$ to the root node */
    $go\_up$    true
    while $go\_up$ do
        $to\_process(\mathcal{N})$    true
        if $\mathcal{N}$ is mapped onto $Myid$ then
            for $k = npiv_{\mathcal{N}} + 1$ to $nfront_{\mathcal{N}}$ do
                $j$    $col\_list_{\mathcal{N}}(k)$
                if POSinWRHS_row(i)=0 then
                    $m\_row$    $m\_row + 1$
                    POSinWRHS_row($i$) = $m$
                end if
            end for
        end if
        if $\mathcal{N}$ is not the root node then
            /* etc. same as before */
        end if
    end while
end for

/* Step 3b: assign slots to column variables corresponding to contribution blocks */
/*                 [Same as above with $targets\_sol$ and $col\_list$]          */
```

3.3 A storage scheme based on the height of the elimination tree

3.3.1 Assumptions

In the context of the computation of inverse entries, we have been interested in going even further in the compression of the solution space, in order to be able to process even larger blocks of right-hand sides at once (several thousands of columns per block on large problems). We will show that, when computing diagonal entries of the inverse, it possible to hold the solution space in an array whose size is proportional to the height of the pruned elimination tree. This last storage scheme is less general than the two schemes previously described. It assumes the following context:

A sequential execution.

Computation of *diagonal* entries of A^{-1}, or, slightly more generally, a computation where both the right-hand sides and the solution vectors have only one nonzero entry.

It also requires a very particular feature of the computational scheme: sparsity must be exploited *within* blocks of right-hand sides. This is described in detail in Chapter 5, and here we simply sketch the idea. Let us take a simple example: consider the assembly tree of Figure 3.1(b) and assume that inverse entries a_{22}^{-1} and a_{44}^{-1} are computed at the same time, that is, there is a single block of right-hand sides $[e_2\ e_4]$. Consider the forward elimination phase: the pruned tree associated with this block of right-hand sides is the union of the path from node 1 (which contains variable 2) to the root node and the path from node 2 (which contains variable 4) to the root node. Since node 1 is not on the path from node 2 to the root node, $L^{-1}e_4 {}_2 = 0$ (variable 2 being an eliminated variable of node 1). Therefore, when solving with the right-hand side block $[e_2\ e_4]$, it is necessary to store the partial solution $L^{-1}e_2 {}_2$ but not the component $L^{-1}e_4 {}_2$. Similarly, since node 2 is not on the path from node 1 to the root node, $L^{-1}e_2 {}_4$ is zero and does not have to be stored. Only $L^{-1}e_4 {}_4$ is nonzero and has to be stored. The storage scheme presented in the previous section would not exploit this sparsity: a row of WRHS would be used to store $L^{-1}[e_2\ e_4] {}_2$ and another row would be used to store $L^{-1}[e_2\ e_4] {}_4$, ignoring the fact that some elements of these rows are zeros. We can go further in memory compression: since these two rows have complementary (non overlapping) patterns, a single row in WRHS is enough to store their nonzero entries. In the first column of that row we store $L^{-1}e_2 {}_2$, and in the other column we store $L^{-1}e_4 {}_4$.

3.3.2 Construction

Assuming that we have the ability to exploit sparsity *within* blocks, i.e., at each node, to perform computations only on nonzero columns of the block of right-hand sides and to store only the nonzero part of each partial solution, we can derive a storage scheme whose memory requirement is proportional to the height of the tree: the location of a variable in WRHS is its height in the tree. This works because every column of the block of right-hand sides corresponds to a single path in the tree. Exploiting sparsity within the block is roughly equivalent to saying that columns are processed independently and thus can be stored in an array of size the height of the pruned tree, which is the maximum length of the paths defined by each column. We refer to this scheme as the *tree height storage scheme*. Once again, the storage scheme is designed to enforce locality within nodes (the eliminated variables of a node are stored in contiguous locations in WRHS) and at the tree level (in WRHS, the variables of the children of a node are located next to the variables of that node). We describe in Algorithm 3.3 how the indirection arrays are constructed.

Algorithm 3.3 Construction of the indirection arrays for the compressed solution space (tree height scheme).

```
/* Input: $targets$, indices of the diagonal elements of the inverse to be computed */

Compute the pruned tree; $to\_process(\mathcal{N})$ is true if $\mathcal{N}$ belongs to the pruned tree
$height(:) = 0$
$pool =$  the root node
while $pool$ is not empty do
    Extract a node $\mathcal{N}$ from the pool
    if $\mathcal{N}$ is mapped onto $Myid$ then
        /* $npiv_{\mathcal{N}}$ is the number of eliminated pivots */
        /* $row\_list_{\mathcal{N}}$ is the list of row indices */
        /* $col\_list_{\mathcal{N}}$ is the list of column indices */
        /* $nc_{\mathcal{N}}$ is the number of children of $\mathcal{N}$ */
        /* $children(\mathcal{N})$ is the list of children of $\mathcal{N}$ */
        $m$     $height(\mathcal{N})$
        for $k = 1$ to $npiv_{\mathcal{N}}$ do
            $i$     $row\_list_{\mathcal{N}}(k)$; $j$     $col\_list_{\mathcal{N}}(k)$
            $m$     $m + 1$
            POSinWRHS_row($i$) $= m$
            POSinWRHS_col($j$) $= m$
        end for
    end if
    for $k = 1$ to $nc_{\mathcal{N}}$ do
        $\mathcal{N}_k = children(\mathcal{N}_k)$
        if $to\_process(\mathcal{N}_k)$ then /* $\mathcal{N}_k$ belongs to the pruned tree */
            $height(\mathcal{N}_k)$     $height(\mathcal{N}_k) + npiv_{\mathcal{N}}$
            Add $\mathcal{N}_k$ to the pool of nodes
        end if
    end for
end while
```

We illustrate in Figure 3.3 the structure of WRHS when computing the inverse entries a_{22}^{-1} and a_{44}^{-1} on the assembly tree of Figure 3.1(b). The tree height scheme improves over the union sparse scheme by storing the partial solutions associated to row eliminated variables 2 and 4 on the same line of WRHS.

It is not possible to extend the tree height scheme to the most general case where the right-hand sides or the solution vectors have more than one nonzero component (since in that case a column of the right-hand side might require traversing more than one path in the tree). In practice, we have limited our implementation to the sequential case (even though a parallel implementation should be feasible). The ability to exploit sparsity within blocks is described in Chapter 5; it allows us to significantly decrease the computational cost and is particularly interesting in a parallel context for exploiting tree parallelism, as explained in Chapter 6.

fwd	bwd
2	2
1	4
3	8
7	9
6	7
8	X
9	X
5	X
4	X

(a) Union sparse scheme.

fwd	bwd
8	8
9	9
5	7
7	2
6	4
1	X
3	X
2&4	X

(b) Tree height scheme.

Figure 3.3: Structure of WRHS for the computation of a_{22}^{-1} and a_{44}^{-1} with the assembly tree of Figure 3.1(b), in a sequential context. fwd refers to the forward phase and bwd refers to the backward phase. The tree height scheme improves on the union sparse scheme by storing the components of the solution corresponding to row variables 2 and 4 in the same locations in WRHS.

3.4 Experiments

We report in this section experimental results that highlight the benefits of the storage schemes previously described, both in terms of memory and performance.

3.4.1 Dense right-hand sides: compression and locality effects

We report in Table 3.1 the size of the solution space for matrix AUDI for different number of processes. We compare the baseline algorithm and our new dense storage scheme, corresponding to Algorithm 3.1. We recall that the baseline algorithm uses an array of size N on every process (Wb in the forward elimination, $Wsol$ in the backward substitution), and an array, WRHS, that scales perfectly with the number of processes; therefore the total size of the solution space is N times the number of processes plus one, times the block size. In the new algorithm, there is only one array, WRHS, that scales almost perfectly with the number of processes. The block size is set to 128 for the purpose of the illustration, even though it does not influence the comparison since, in both cases, the total size of the solution space is proportional to the block size. Results show that, while the size of the solution space grows linearly with the number of processes in the baseline algorithm, it is fairly close to being constant in the new algorithm: it slowly increases, but there is only a 30% increasing when going from 1 process to 32. Therefore, the compression rate is almost equal to the number of processes (e.g., 25 on 32 processes).

Figure 3.4 illustrates the performance gains provided by the new storage scheme. We measure the time for the solution phase in MUMPS on matrix AUDI, in a sequential execution with a single right-hand side block with 128 columns. We compare the baseline algorithm, the new storage scheme described in Algorithm 3.1 without enforcing locality at the tree level (i.e., without traversing nodes in postorder), and the new storage scheme where locality is enforced at both the node level and the tree level. Moreover, although this is not related to memory compression, we also experimented with a row-major storage scheme for WRHS, which is expected to deliver a better locality. This is also illustrated in Figure 3.4. Results show that enforcing locality both at the node level and the tree level improves the performance; the row-major storage also provides some speed-up. Altogether,

Processes	Size of the of the solution space (GB)	
	Baseline algorithm	New algorithm
1	1.8	0.89
2	2.7	0.90
4	4.5	0.95
8	8.1	0.97
16	15.3	1.03
32	29.7	1.20

Table 3.1: Total size of the solution space for matrix AUDI with a block size equal to 128 and different numbers of processes.

these optimizations decrease the solution time by around 30% on this example. Matrix AUDI corresponds to a 3D problem, but we experienced even larger improvements on 2D problems.

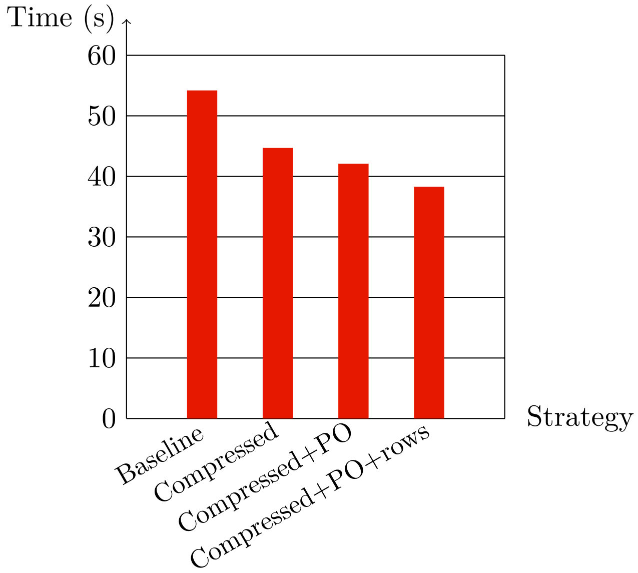


Figure 3.4: Time in seconds for the solution phase on matrix AUDI, in a sequential execution and a single right-hand side block with 128 columns. Baseline refers to the baseline algorithm (no compression); Compressed refers to the compressed scheme where locality is enforced within nodes but not at the tree level (the list of nodes is not traversed following the postorder); Compressed+PO refers to the compressed scheme with locality enforced at the tree level; Compressed+PO+rows refers to the compressed scheme with a row-major storage for `WRHS`.

3.4.2 Sparse right-hand sides

We illustrate the gains in memory requirements for the two above-mentioned storage schemes. Table 3.2 corresponds to the computation of the diagonal entries of the inverse of matrix AUDI, in a sequential context, for different block sizes B. We compare the baseline algorithm (which requires a solution space of size $2 \times N \times B$ in a sequential execution), the union sparse scheme, and the tree height scheme (which is applicable here since we compute diagonal entries). We assume that the N right-hand sides (columns of the identity matrix) are ordered according to their only nonzero entry, following the postorder of the elimination tree used during the factorization. We will show in the next chapters that this is often a good strategy to decrease both the memory requirements and the computational cost. Results show that the storage schemes that exploit sparsity are able to significantly decrease the memory consumption. On this example, there is a factor of nearly a hundred in memory consumption. The tree height storage scheme is slightly better than the union sparse scheme. Gains tend to grow when the block size increases. This is intuitively expected, since the union sparse scheme yields a solution space proportional to the *size* of the pruned tree, while the tree height scheme yields a solution space proportional to the *height* of the pruned tree.

Size of the solution space (MB)					
Baseline algorithm		Union sparse storage		Tree height storage	
$B = 128$	$B = 1024$	$B = 128$	$B = 1024$	$B = 128$	$B = 1024$
1843	14745	20	162	19	154

Table 3.2: Size of the solution space for matrix AUDI when computing all the diagonal entries of the inverse. Right-hand sides are ordered according to their only nonzero entry, following the postorder used during the factorization.

Table 3.3 illustrates the computation of random off-diagonal entries of the inverse of matrix AUDI. We assume again that the right-hand sides are ordered according to their only nonzero entry, following the postorder of the elimination tree used during the factorization. This time, the tree height scheme is not applicable. The union sparse scheme offers significant gains upon the baseline scheme, but less than in Table 3.2. This comes from the fact that we work with the union of the pruned tree for the forward phase and the pruned tree for the backward phase. Since we have ordered the blocks of right-hand sides according to the pattern of the right-hand sides, ignoring the pattern of the solution, we have no control over the pruned tree for the backward phase (union of the paths from the root node to the nodes corresponding to the requested entries). This problem is highlighted in the next chapter. The table shows that gains tend to decrease when the block size increases, but we still have a factor of almost five for $B = 1024$.

Size of the solution space (MB)			
Baseline algorithm		Union sparse storage	
$B = 128$	$B = 1024$	$B = 128$	$B = 1024$
1843	14745	216	3264

Table 3.3: Size of the solution space for matrix AUDI when computing random general (off-diagonal) inverse entries.

Chapter 4

Minimizing accesses to the factors in an out-of-core execution

In this chapter, we consider computations involving large number of sparse right-hand sides in an out-of-core context. We focus on the computation of inverse entries as this was our motivating application, but the results presented in this chapter apply to any computation involving sparse right-hand sides and/or sparse solution vectors. When the number of right-hand sides is large (e.g., $NBRHS = N$), one generally cannot process them all at once because of the excessing memory usage, even using a sophisticated storage scheme such as those introduced in the previous chapter. Therefore, right-hand sides are processed by blocks. We show that the way the set of right-hand sides is divided into blocks strongly influences the volume of factors to be loaded. We formulate this as a partitioning problem and show that it is NP-complete. We also show that we cannot get a close approximation to the optimal solution in polynomial time. We thus need to develop heuristic algorithms, and we propose: (i) a lower bound on the cost of an optimum solution; (ii) an exact algorithm for a particular case; (iii) two other heuristics for a more general case; and (iv) hypergraph partitioning models for the most general setting. We compare the proposed algorithms and illustrate the performance of our algorithms in practice using MUMPS on large matrices.

4.1 A tree-partitioning problem

4.1.1 Formulation

We first focus on the case were only diagonal entries of the inverse are computed. As seen in Section 2.2, in order to compute a_{ii}^{-1} using the formulation

$$\begin{aligned} y &= L^{-1}e_i \\ a_{ii}^{-1} &= (U^{-1}y)_i \end{aligned}$$

we have to access the columns of L that correspond to the nodes in the unique path from node i to the root, and then access the rows of U that correspond to the nodes in the same path. As discussed above, these are the necessary and sufficient parts of L and U that are needed. In other words, we know how to solve efficiently for a single requested diagonal entry of the inverse. Now suppose that we need to compute a set R of diagonal entries of the inverse. If R is small, then we could identify all the columns of L and rows of U that must be loaded for at least one requested entry in R and then solve for all R at once, accessing the necessary and sufficient parts of L and U only once. However, R is very often large. In the application areas mentioned in Section 2.1, one often needs to compute a large set of entries, such as the whole diagonal of the inverse (in that case, $R = N$). The computations proceed in blocks of size B, where at each block a limited number of diagonal entries are computed. This necessitates accessing (loading from disk) some parts of L and U multiple times (in different blocks) according to the entries computed in the corresponding blocks. In our computational scheme, after some factors are loaded from disk to main memory and used at a given node of the tree, they must be discarded from memory to have room for the next block of factors to be loaded; when the same part of an L or U factor is needed for two successive blocks, we therefore cannot assume that they are still in memory. The main combinatorial problem then becomes that of partitioning the requested entries into blocks in such a way that the overall cost of disk-to-memory traffic for the factors is minimized. In the rest of this chapter, we will often use the terminology factors loaded from disk , which is the same as factors accessed , and also corresponds to the traffic from disk to main memory.

Let us first discuss the influence of the block size B: in terms of volume of accesses to the factors, the optimal block size is $B = NBRHS$. Indeed, if there is only one block of right-hand sides, every node in the tree is traversed at most once (once if it belongs to the pruned tree corresponding to the single block of right-hand sides, and not otherwise). On the contrary, a block size $B = 1$ maximizes the volume of factors: the set of nodes to be traversed during the whole solution phase is the same as before, but some nodes are accessed multiple times because they appear in the pruned tree of different blocks; for example, the root node is accessed for every block, i.e., *NBRHS* times. In this chapter, the block size B is fixed (the choice of B is discussed later), and we consider the problem of partitioning the set of right-hand sides into blocks of size B, so that the volume of factors to be loaded is minimized.

Figure 4.1 illustrates the influence of the partitioning on the volume of factors to be loaded. Diagonal entries $a^{-1}_{1\,1}$, $a^{-1}_{2\,2}$, $a^{-1}_{3\,3}$ and $a^{-1}_{4\,4}$ are requested and the block size B is set to 2. The partitioning 1 2 3 4 leads to a volume of loaded factors of 10 (assuming unit weights): indeed, the first block requires to access the nodes that are on the path from node 1 to the root node and the nodes that are on the path from node 2 to the root node, i.e., the six nodes 1 2 3 4 5 6 . Similarly, the second block requires to access the nodes that are on the path from node 3 to the root node and the nodes that are on the path from node 4 to the root node, i.e., the four nodes 3 4 5 6 . However, the partitioning 1 4 2 3 leads to a volume of 8: indeed, nodes 3 and 4 are accessed only once while they are accessed twice with the previous partitioning.

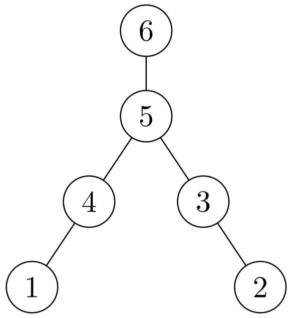


Figure 4.1: An example of the influence of the partitioning: diagonal terms 1, 2, 3 and 4 are requested and the block size is 2. The partitioning 1 2 3 4 leads to a volume of loaded factors of 10 (assuming unit weights), whereas the partitioning 1 4 2 3 leads to a volume of 8.

We now formally introduce the combinatorial problem. Let T be the elimination tree on N nodes, where the factors associated with each node are stored on disks (*out-of-core*). Let $w(i)$ denote the cost of loading the parts of the factors, L or U, associated with node i of the elimination tree. Similarly let $w(i\ \ j)$ denote the sum of the costs of the nodes in the path from node i to node j. The cost of computing a_{ii}^{-1} is

$$cost(i) = \sum_{k\in\mathcal{P}(i)} 2 \times w(k) = 2 \times w(i\ \ r)$$

If we solve for a set R of diagonal entries at once, then the overall cost is

$$cost(R) = \sum_{i\in\mathcal{P}(R)} 2 \times w(i) \text{ where } \mathcal{P}(R) = \quad\underset{i\in R}{\quad}\mathcal{P}(i)$$

Let B denote the maximum number of diagonal entries that can be computed at the same time, i.e., the maximum size of a block. The TreePartitioning problem is formally defined as follows: given a tree T with N nodes, a set $R = \ \ i_1 \quad\quad i_m\ \ $ of nodes in the tree, and an integer $B \leqslant m$, partition R into a number of subsets $R_1\ \ R_2 \quad\quad R_K$ so that $\ R_k\ \leqslant B$ for all k, and the total cost

$$cost(R) = \sum_{k=1}^{K} cost(R_k) \tag{4.1}$$

is minimum.

The number of subsets K is not specified, but obviously $K \geqslant \lceil \frac{m}{B} \rceil$. Without loss of generality, we can assume that there is a one-to-one correspondence between R and leaf nodes in T. Indeed, if there is a leaf node i where $i\ /\ R$, then we can delete node i from T since it does not appear in the pruned tree of any of the blocks. Similarly, if there is an internal node i where $i \quad R$, then we can create a leaf node i of zero weight, make it an additional child of i, and set $R = R \quad i \quad\quad i\ \ $. For ease of discussion and formulation, for each requested node (leaf or not) of the elimination tree we add a leaf node with zero weight. To clarify the execution scheme, we now specify the algorithm that computes the diagonal entries of the inverse specified by a given R_k. We first find $\mathcal{P}(R_k)$, the union of the paths from the nodes corresponding to the entries in R_k to the root node; we then start loading the associated L factors from the disk following a postorder and perform the forward solves with L. When we reach the root node, we have $\ R_k\ $ vectors and we start loading the associated U factors from the disk and perform backward substitutions along the paths that we traversed (in reverse order) during the forward substitutions.

4.1.2 A lower bound

We present a lower bound for the cost of an optimal partition. Let $nl(i)$ denote the number of leaves of the subtree $T(i)$, which can be computed as follows:

$$nl(i) = \quad \begin{array}{ll} 1 & i \text{ is a leaf node} \\ \sum_{j\in children(i)} nl(j) & \text{otherwise} \end{array}$$

We note that as all the leaf nodes correspond to the requested diagonal entries of the inverse, $nl(i)$ corresponds to the number of forward and backward solves that have to be performed at node i.

Given the number of forward and backward solves that pass through a node i, it is easy to define the following lower bound on the amount of factors loaded.

Theorem 4.1 - Lower bound on the amount of factors to load.
Let T be a node weighted tree, $w(i)$ be the weight of node i, B be the maximum allowed size of a partition, and $nl(i)$ be the number of leaf nodes in the subtree rooted at i. Then we have the following lower bound, denoted by η, on the optimal solution $c^\star$ of the TreePartitioning problem:

$$\eta = 2 \times \sum_{i \in T} w(i) \times \left\lceil \frac{nl(i)}{B} \right\rceil \leqslant c^\star$$

Proof. Follows easily by noting that each node i has to be loaded at least $\left\lceil \frac{nl(i)}{B} \right\rceil$ times both in the forward and backward substitution phases. □

Each internal node is on a path from (at least) one leaf node, therefore $\lceil nl(i)/B \rceil$ is at least 1, and we have $2 \times \sum_{i \in T} w(i) \leqslant c^\star$. We also note that by removing $w(i)$ from the formula, one obtains a lower bound on the total number of times the factors are loaded.

Figure 4.2 illustrates the computation of the lower bound. Entries a_{11}^{-1}, a_{33}^{-1}, and a_{44}^{-1} are requested, and the elimination tree of Figure 4.1 is modified accordingly to have leaves (with zero weights) corresponding to these entries. The numbers $nl(i)$ are shown next to the nodes. Suppose that each internal node has unit weight and that the block size B is 2. Then, the lower bound is:

$$\begin{aligned} \eta &= 2 \times \left(\left\lceil \frac{1}{2} \right\rceil + \left\lceil \frac{1}{2} \right\rceil + \left\lceil \frac{2}{2} \right\rceil + \left\lceil \frac{3}{2} \right\rceil + \left\lceil \frac{3}{2} \right\rceil \right) \\ &= 14 \end{aligned}$$

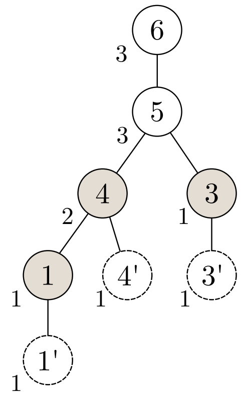


Figure 4.2: Number of leaves of the subtrees rooted at each node of a transformed elimination tree. The nodes corresponding to the requested diagonal entries of the inverse are shaded, and a leaf node is added for each such entry. Each node is annotated with the number of leaves in the corresponding subtree, resulting in a lower bound of $\eta = 14$ with $B = 2$.

4.1.3 NP-completeness

We have the following computational complexity result, that shows that our partitioning problem is difficult (it is unlikely that a solution can be found in polynomial time):

Theorem 4.2
The TreePartitioning problem is NP-complete.

Proof. We consider the associated decision problem: given a tree T with m leaves, a value of B, and a cost bound c, does a partitioning S of the m leaves into subsets whose size does not exceed B, and such that $cost(S) \leqslant c$ exist? It is clear that this problem belongs to NP since if we are given the partition S, it is easy to check in polynomial time that it is valid and that its cost meets the bound c. We now have to prove that the problem is in the NP-complete subset.

To establish the completeness, we use a reduction from 3-Partition [39], which is NP-complete in the strong sense. Consider an instance $\mathcal{I}_1$ of 3-Partition: given a set a_1 a_{3p} of $3p$ integers, and an integer Z such that $\sum_{1 \leqslant j \leqslant 3p} a_j = pZ$, does a partition of 1 $3p$ into p disjoint subsets K_1 K_p, each with three members, such that for all $1 \leqslant i \leqslant p$, $\sum_{j \in K_i} a_j = Z$ exist?

We build the following instance $\mathcal{I}_2$ of our problem: the tree is a three-level tree composed of $N = 1 + 3p + pZ$ nodes: the root v_r, of cost w_r, has $3p$ children v_i, of the same cost w_v, for $1 \leqslant i \leqslant 3p$. In turn, each v_i has a_i children, each being a leaf node of zero cost. This instance $\mathcal{I}_2$ of the TreePartitioning problem is shown in Figure 4.3. We let $B = Z$ and ask whether there exists a partition of leaf nodes of cost $c = pw_r + 3pw_v$. Here w_r and w_v are arbitrary values (we can take $w_r = w_v = 1$). We note that the cost c corresponds to the lower bound shown in Theorem 4.1; in this lower bound, each internal node v_i is loaded only once, and the root is loaded p times, since it has $pZ = pB$ leaves below it. Note that the size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$. Indeed, because 3-Partition is NP-complete in the strong sense, we can encode $\mathcal{I}_1$ in unary, and the size of the instance is $O(pZ)$.

Now we show that $\mathcal{I}_1$ has a solution if and only if $\mathcal{I}_2$ has a solution. Suppose first that $\mathcal{I}_1$ has a solution K_1 K_p. The partition of leaf nodes corresponds exactly to the subsets K_i: we build p subsets S_i whose leaves are the children of vertices v_j with j K_i. Suppose now that $\mathcal{I}_2$ has a solution. To meet the cost bound, each internal node has to be loaded only once, and the root at most p times. This means that the partition involves at most p subsets to cover all leaves. Because there are pZ leaves, each subset is of size exactly Z. Because each internal node is loaded only once, all its leaves belong to the same subset. Altogether, we have found a solution to $\mathcal{I}_1$, which concludes the proof. □

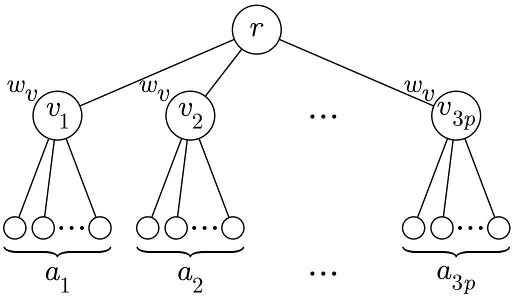


Figure 4.3: The instance of the TreePartitioning problem corresponding to a given 3-Partition problem. The weight of each node is shown next to the node. The minimum cost of a solution for $B = Z$ to the TreePartitioning problem is $p \times w_r + 3p \times w_v$ which is only possible when the children of each v_i are all in the same part, and when the children of three different internal nodes, say v_i v_j v_k, are put in the same part. This corresponds to putting the numbers a_i a_j a_k into a set for the 3-Partition problem which sums up to Z.

We can further show that we cannot get a close approximation to the optimal solution in polynomial time.

Theorem 4.3
Unless P=NP, there is no $1 + o(\frac{1}{N})$ polynomial approximation for trees with N nodes in the TreePartitioning problem.

Proof. Assume that there exists a polynomial $1 + \frac{(N)}{N}$ approximation algorithm for trees with N nodes, where $\lim_{N} \quad (N) = 0$. Let $\ (N) < 1$ for $N \geqslant N_0$. Consider an arbitrary instance $\mathcal{I}_0$ of 3-Partition with a set $\ a_1 \qquad a_{3p}\ $ of $3p$ integers, and an integer Z such that $\sum_{1 \leqslant j \leqslant 3p} a_j = pZ$. Without loss of generality, assume that $a_i \geqslant 2$ for all i (hence $Z \geqslant 6$). We ask if we can partition the $3p$ integers of $\mathcal{I}_0$ into p triples of the same sum Z. Now we build an instance $\mathcal{I}_1$ of 3-Partition by adding X times the integer $Z \quad 2$ and $2X$ times the integer 1 to $\mathcal{I}_0$, where $X = \max \quad \left\lceil \frac{N_0-1}{Z+3} \right\rceil \quad p \ \ 1 \ \ .$ Hence $\mathcal{I}_1$ has $3p + 3X$ integers and we ask whether these can be partitioned into $p + X$ triples of the same sum Z. Clearly, $\mathcal{I}_0$ has a solution if and only if $\mathcal{I}_1$ does (the integer $Z \quad 2$ can only be in a set with two 1s).

We build an instance $\mathcal{I}_2$ of TreePartitioning from $\mathcal{I}_1$ exactly as we did in the proof of Theorem 4.2, with $w_r = w_v = 1$, and $B = Z$. The only difference is that the value p in the proof has here been replaced by $p + X$, therefore the three-level tree now has $N = 1 + 3(p+X) + (p+X)Z$ nodes. Note that X has been chosen so that $N \geqslant N_0$. Just as in the proof of Theorem 4.2, $\mathcal{I}_1$ has a solution if and only if the optimal cost for the tree is $c^\star = 4(p+X)$, and otherwise the optimal cost is at least $4(p+X)+1$.

If $\mathcal{I}_1$ has a solution, and because $N \geqslant N_0$, the approximation algorithm will return a cost of at most

$$\left(1 + \frac{(N)}{N}\right) c^\star \leqslant \left(1 + \frac{1}{N}\right) 4(p+X) = 4(p+X) + \frac{4(p+X)}{N}$$

But $\frac{4(p+X)}{N} = \frac{4(N-1)}{(Z+3)N} \leqslant \frac{4}{9} < 1$, so that the approximation algorithm can be used to determine whether $\mathcal{I}_1$, and hence $\mathcal{I}_0$, has a solution. This is a contradiction unless P=NP. □

4.1.4 The off-diagonal case

As discussed before, the formulation for the diagonal case carries over to the off-diagonal case as well. An added difficulty in this case is related to the actual implementation of the solver. Assume that we have to solve for a^{-1}_{ij} and a^{-1}_{kj}, that is, two entries in the same column of A^{-1}. As seen in Section 2.2,

$$\begin{aligned} y &= L^{-1} e_j \\ a^{-1}_{ij} &= (U^{-1} y)_i \end{aligned}$$

so only one y vector suffices. Similarly one can solve for the common nonzero entries in $U^{-1}y$ only once for i and k. This means that, for the forward solves with L, we can perform only one solve, and for the backward solves, we can solve only once for the variables in the path from the root to the least common ancestor of i and j, $lca(i \ \ k)$. Clearly, this will reduce the operation count. However, in an out-of-core context, this does not affect the number of factors that we have to load. Avoiding the unnecessary repeated solves only affects the operation count.

If we were to exclude the algorithms that take advantage of the common indices among the solves for two requested entries, then we can immediately generalize our results and models developed for the case of diagonal entries to the off-diagonal case. Of course, the partitioning problem will remain NP-complete (it contains the instances with diagonal entries as a particular case). The lower bound can also be generalized to yield a lower bound for the case with arbitrary entries. Indeed, we only have to apply the same reasoning twice: once for the column indices of the requested entries, and once for the row indices of the requested entries. We can extend the model to cover the case of multiple entries in the same column; when indices are repeated (say a_{ij}^{-1} and a_{kj}^{-1} are requested), we can distinguish them by assigning each occurrence to a different leaf node (we add two zero-weighted leaf nodes to the node j of the elimination tree). Then adding these two lower bounds yields a lower bound for the general case. However, in our experience, we have found this lower bound to be loose. Note that applying this lower bound to the case where only diagonal entries are requested yields the lower bound given in Theorem 4.1.

4.2 Heuristics

In this section, we suggest heuristics that address the case where only diagonal entries are requested. We first suggest a very intuitive heuristic that relies on a postorder of the tree and then describe a more sophisticated approach based on recursive bisection and a matching algorithm.

4.2.1 A partitioning based on a postorder of the tree

A postordering of the tree is such that all the nodes in a given subtree are numbered consecutively. Therefore, one can expect that partitioning the right-hand sides following a postorder of the tree will provide a good locality for factor reuse between consecutive columns of the right-hand sides: intuitively, consecutive right-hand sides for a topological ordering will correspond to nodes which are likely to be close in the tree and thus to share a long common path to the root node. We thus derive the PoPart heuristic: the PoPart heuristic first orders the leaf nodes according to their rank in the postorder. It then puts the first B leaves in the first part, the next B leaves in the second part, and so on. This simple partitioning approach results in $\lceil F/B \rceil$ parts, for a tree with F leaf nodes, and puts B nodes in each part, except maybe in the last one. We have the following theorem which states that this simple heuristic obtains results that are at most twice the cost of an optimum solution.

Theorem 4.4 - Approximation guarantee of the PoPart heuristic.
Let $\ _{PO}$ be the partition obtained by the algorithm PoPart and $c^\star$ be the cost of an optimum solution, then

$$cost(\ _{PO}) \leqslant 2 \times c^\star$$

Proof. Consider node i. Because the leaves of the subtree rooted at i are sorted consecutively, the factors of node i will be loaded at most $\left\lceil \frac{nl(i)}{B} \right\rceil + 1$ times. This is illustrated in Figure 4.4; at a given node, the number of leaf nodes in the subtree rooted at i can be written $nl(i) = r_1 + q \quad B + r_2$ with $0 \leqslant r_1 \quad r_2 < B$. If $r_1 > 0$, $r_2 > 0$ and $r_1 + r_2 < B$, the factors of node i are loaded $q + 2 = \left\lceil \frac{nl(i)}{B} \right\rceil + 1$ times; otherwise, they are loaded $\left\lceil \frac{nl(i)}{B} \right\rceil$

times. Therefore the overall cost is at most

$$\begin{aligned} cost(\ _{PO}) &\leqslant 2 \times \sum_i w(i) \times \left(\left\lceil \frac{nl(i)}{B} \right\rceil + 1\right) \\ &\leqslant \eta + 2 \times \sum_i w(i) \\ &\leqslant 2 \times c^\star \end{aligned}$$

□

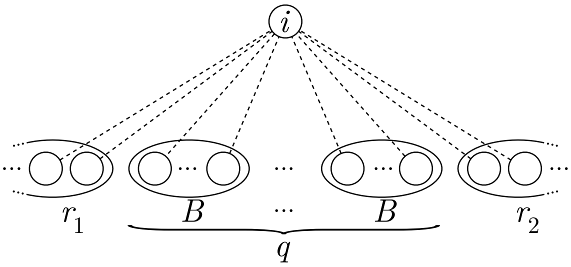


Figure 4.4: Approximation guarantee of the PoPart heuristic. At node i, $nl(i) = r_1 + q\ \ B + r_2$; at most, i is accessed $\left\lceil \frac{nl(i)}{B} \right\rceil$ times.

We note that the factor two in the approximation guarantee is rather loose in practical settings, as $2 \times \sum_i w(i)$ will be much smaller than the lower bound η with a practical B and a large number of nodes.

We have experimented with some heuristics aimed at improving the postorder by performing vertex move refinements (à la Kernighan-Lin [58]). The results were rather mixed and so we will not discuss them here, but we refer the reader to [79].

4.2.2 A matching algorithm

In this section, we first propose an algorithm that solves the partitioning problem exactly when $B = 2$. It will serve as a building block for an algorithm for partitioning into blocks of size $B = 2^k$, for a positive integer k in the next subsection.

Since the blocks are of size 2, a matching can be used to define the blocks. Consider the complete graph $G = (V\ V \times V)$ of the leaves of a given tree, and assume that the edge $(i\ j)$ represents the decision to put the leaf nodes i and j together in a part. Given this definition of the vertices and edges, we associate the value $m(i\ j) = cost(\ \ i\ j\ \ )$ to the edge $(i\ j)$ if $i = j$, and $m(i\ i) = \sum_{n \in V} w(n)$ (or any sufficiently large number). Then a minimum weighted matching in G defines a partitioning of the vertices in V with the minimum cost (as defined in (4.1)). Although this is a short and immediate formulation, it has a high run time complexity of $O(\ V\ ^3)$ and $O(\ V\ ^2)$ memory requirements. Therefore, we propose another exact algorithm for $B = 2$.

The proposed algorithm Match proceeds from the parents of the leaf nodes to the root. At each internal node n, those leaf nodes that are in the subtree rooted at n and which are not put in a part yet are matched two by two (arbitrarily) and each pair is put in a part; if there is an odd number of leaf nodes remaining to be partitioned at node n, one of them (arbitrarily) is passed to $parent(n)$. This algorithm attains the lower bound shown in Theorem 4.1, and hence it finds an optimal partition for $B = 2$. The key idea can be seen using Figure 4.4 and the corresponding notation. Assume for simplicity that in Match algorithm, leaf nodes are matched following a left-right order (instead of being matched arbitrarily). We can never have, at a given node i, $nl(i) = r_1 + q \quad B + r_2$ with $r_1 = 1$ and $r_2 = 1$; indeed, we necessarily have $r_1 = 0$ and $r_2 = 0$ (if the number of leaf nodes in the subtree rooted at i is even) or $r_2 = 1$ (if the number of leaf nodes is odd). Thus node i is loaded $\left\lceil \frac{nl(i)}{B} \right\rceil$ times, which corresponds to the lower bound.

The memory and the run time requirements are $O(\ V \ )$. We note that two leaf nodes can be matched only at their least common ancestor.

Algorithm 4.1 Match: An exact algorithm for $B = 2$.

```
/* Input: $T = (V \ E)$ with $F$ leaves; each requested entry corresponds to a leaf node. The
root node is denoted by $r$.                                                              */
/* Output: $\ _2 = \ R_1 \quad R_K$ where $K = \lceil F/2 \rceil$ */
for each leaf node   do
    Add   to $inds(parent( \ ))$,
end for
Compute a postorder of the nodes of $T$
$k$    1
for Each non-leaf node $n$ in postorder do
    if $n = r$ and $inds(n)$ contains an odd number of vertices then
              the node with the least weight in $inds(n)$
        Move   to $inds(parent(n))$, add $w(n)$ to the weight of    /* relay to the parent */
    else if $n = r$ and $inds(r)$ contains an odd number of vertices then
              a node with the least weight in $inds(n)$
        Make   a singleton
    end if
    for $i = 1$ to  $inds(n)$  by 2 do
        Put the $i$th and $i + 1$st vertices in $inds(n)$ into $R_k$, increment $k$ /* match the
    $i$th and $i + 1$st items in the list $inds$ */
    end for
end for
```

We have slightly modified the basic algorithm and show this modified version in Algorithm 4.1. The modifications keep the run time and memory complexities the same and are included to enable the use of Match as a building block for a more general heuristic. In this algorithm, $parent(n)$ gives the parent of node n, and $inds(n)$ is a list of indices associated with node n. The sum of the sizes of the $inds(\ )$ lists is $\ V \ $. The modification is that when there are an odd number of leaf nodes to partition at node n, the leaf node with the least cumulative weight is passed to the parent. The cumulative weight of a leaf node i when Match processes node n is defined as $w(i \ \ n) \quad w(n)$: the sum of the weights of the nodes in the unique path between nodes i and n, including i, but excluding n. This is easy to compute: each time a leaf node is relayed to the parent of the current node, the weight of the current node is added to the cumulative weight of the relayed leaf node. By doing this, the leaf nodes which traverse longer paths before being partitioned are chosen before those with smaller weights.

We propose a heuristic algorithm that extends the previous approach when $B = 2^k$ for some $k \geqslant 2$: the BiseMatch algorithm is shown in Algorithm 4.2. It is based on a bisection approach. At each bisection, a matching among the leaf nodes is found by a call to Match. Then, one of the leaf nodes of each pair is removed from the tree; the remaining one becomes a representative of both. Since the remaining node at each bisection step is a representative of two representative nodes of the previous bisection step, after $\log B = k$ steps, BiseMatch obtains nodes that represent at most $B \quad 1$ other nodes. At the end, the nodes, if not a representative node, are included in the same part as their representative node.

Algorithm 4.2 BiseMatch: A heuristic algorithm for $B = 2^k$.

```
/* Input: $T = (V \ E)$ with $F$ leaves; each requested entry corresponds to a leaf node */
/*        $B = 2^k$: the maximum allowable size of a part */
/* Output: $_{2^k} = \ R_1 \qquad R_K$ where $R_i \ \leqslant B$ */
for level = 1 to $k$ do
    $M \quad$ Match($T$)
    For each pair $(i \ j) \quad M$ remove the leaf node $j$ from $T$, and mark the leaf node $i$ as representative
    Clean up the tree $T$ so that all leaf nodes correspond to some requested entry
end for
Each remaining leaf node $i$ corresponds to a part $R_i$ where the nodes that are represented by $i$ are put in $R_i$
```

As seen in the algorithm, at each stage a matching among the remaining leaves is found by using the Match algorithm. When leaf nodes i and j are matched at their least common ancestor $lca(i \ j)$, if $w(i \ lca(i \ j)) \geqslant w(j \ lca(i \ j))$ we designate i to be the representative of the two by adding $(i \ j)$ to M, otherwise we designate j to be the representative by adding $(j \ i)$ to M. With this choice, the Match algorithm is guided to make decisions at nodes close to the leaves. The run time of BiseMatch is $O(\ V \ \log B)$, with an $O(\ V \ )$ memory requirement.

4.3 Hypergraph models

We show how the problem of finding an optimal partition of the requested entries can be transformed into a hypergraph partitioning problem. Our aim is to develop a general model that can address both the diagonal and the off-diagonal cases. We first provide a few definitions; we then give the model for diagonal entries and finally its generalization to the off-diagonal case; we refer to this heuristic as the HP partitioning.

4.3.1 The hypergraph partitioning problem

A hypergraph $\mathcal{H} = (\mathcal{V} \ \mathcal{N})$ is defined as a set of vertices $\mathcal{V}$ and a set of nets $\mathcal{N}$. Every net is a subset of vertices. The cardinality of a net h_i is denoted as $\ h_i \ $. The size of a hypergraph is $\sum_{h_i \in N} \ h_i \ $. Weights can be associated with vertices. We use $w(j)$ to denote the weight of the vertex v_j. Costs can be associated with nets. We use $c(h_i)$ to denote the cost associated with the net h_i.

$\quad = \ \mathcal{V}_1 \qquad \mathcal{V}_K$ is a K-way vertex partition of $\mathcal{H} = (\mathcal{V} \ \mathcal{N})$ if each part is nonempty, parts are pairwise disjoint, and the union of the parts gives $\mathcal{V}$. In $\quad$, a net is said to connect a part if it has at least one vertex in that part. The *connectivity set* $\quad (i)$ of a net h_i is the set of parts connected by h_i. The *connectivity* $\quad (i) = \quad (i)$ of a net h_i is the number of parts connected by h_i. In $\quad$, the weight of a part is the sum of the weights of vertices in that part.

In the hypergraph partitioning problem, the objective is to minimize

$$cutsize(\quad) = \sum_{h_i \in \mathcal{N}} (\quad (i) \quad 1) \times c(h_i) \qquad (4.2)$$

This objective function is widely used in the VLSI community [59] and in the scientific computing community [15, 26, 88]; it is referred to as the *connectivity*-1 cutsize metric. The partitioning constraint is to satisfy a balancing constraint on part weights:

$$\frac{W_{max} \quad W_{avg}}{W_{avg}} \leqslant$$

Here W_{max} is the largest part weight, W_{avg} is the average part weight, and   is a predetermined imbalance ratio. This problem is NP-hard [59].

4.3.2 Hypergraph model   diagonal case

We build a hypergraph whose partition according to the cutsize (4.2) corresponds to the total size of the factors loaded. The requested entries (which correspond to the leaf nodes) are going to be the vertices of the hypergraph, so that a vertex partition will define a partition on the requested entries. The more intricate part of the model is the definition of the nets. The nets correspond to edge disjoint paths in the tree, starting from a given node (not necessarily a leaf) and going up to one of its ancestors (not necessarily the root); each net is associated with a cost corresponding to the total size of the nodes in the corresponding path. We use $path(h)$ to denote the path (or the set of nodes of the tree) corresponding to a net h. A vertex i (corresponding to the leaf node i in the tree) will be in a net h if the solve for a_{ii}^{-1} passes through $path(h)$. In other words, if $path(h) \quad \mathcal{P}(i)$, then $v_i \quad h$. Therefore, if the vertices of a net h_n are partitioned among $\ (n)$ parts, then the factors corresponding to the nodes in $path(h_n)$ will have to be loaded $\ (n)$ times. As we load a factor at least once, the extra cost incurred by a partitioning is $\ (n) \quad 1$ for the net h_n. Given this observation, one case see the equivalence between the total size of the loaded factors and the cutsize of a partition plus the total weight of the tree.

We now define the hypergraph $\mathcal{H}_D = (\mathcal{V}_D \ \mathcal{N}_D)$ for the diagonal case. Let $T = (V \ E)$ be the tree corresponding to the modified elimination tree so that the requested entries correspond to the leaf nodes. Then the vertex set $\mathcal{V}_D$ corresponds to the leaf nodes in T. As we are interested in putting at most B solves together, we assign a unit weight to each vertex of $\mathcal{H}_D$. The nets are best described informally. There is a net in $\mathcal{N}_D$ for each internal node of T. The net h_n corresponding to the node n contains the set of vertices which correspond to the leaf nodes of subtree $T(n)$. The cost of h_n is equal to the weight of node n, i.e., $c(h_n) = w(n)$. This model can be simplified as follows: if a net h_n contains the same vertices as the net h_j where $j = parent(n)$, that is if the subtree rooted at node n and the subtree rooted at its parent j have the same set of leaf nodes, then the net h_j can be removed, and its cost can be added to the cost of the net h_n. This way the net h_n represents the node n and its parent j. This process can be applied repeatedly so that the nets associated with the nodes in a chain, except the first (the one closest to a leaf) and the last (the one which is closest to the root), can be removed, and the cost of those removed nets can be added to that of the first one. After this transformation, we can also remove the nets with single vertices (these correspond to the parent of a leaf node with a single child) as these nets cannot contribute to the cutsize. We note that the remaining nets will correspond to disjoint paths in the tree T.

Figure 4.5 shows an example of such a hypergraph: the requested entries are a_{11}^{-1}, a_{22}^{-1}, and a_{55}^{-1}. Therefore, $\mathcal{V} = \quad 1 \ 2 \ 5$ and $\mathcal{N} = \quad h_1 \ h_2 \ h_4 \ h_5$ (net h_3 is removed according to the rule described above and the cost of h_2 includes the weight of nodes 2 and 3). Each net contains the leaf vertices which belong to the subtree rooted at its associated node, therefore $h_1 = \quad 1 \quad h_2 = \quad 2 \quad h_4 = \quad 1 \ 2 \quad h_5 = \quad 1 \ 2 \ 5$. For example, for the partition

$V_1 = \quad 2 \quad$ and $V_2 = \quad 1 \; 5 \quad$ shown on the right of the figure, the cutsize is:

$$\begin{aligned} cutsize(V_1 \quad V_2) &= c(h_1) \times (\quad (h_1) \quad 1) + c(h_2) \times (\quad (h_2) \quad 1) \\ &\quad + c(h_4) \times (\quad (h_4) \quad 1) + c(h_5) \times (\quad (h_5) \quad 1) \\ &= c(h_4) \times (2 \quad 1) + c(h_5) \times (2 \quad 1) \\ &= c(h_4) + c(h_5) \end{aligned}$$

Consider the first part $V_1 = \quad 2 \quad$. We have to load the factors associated with the nodes 2 3 4 5. Consider now the second part $V_2 = \quad 1 \; 5 \quad$. For this part, we have to load the factors associated with the nodes 1 4 5. Hence, the factors associated with the nodes 4 and 5 are loaded twice, while the factors associated with all other (internal) nodes are loaded only once. Since we have to access each node at least once, the extra cost due to the given partition is $w(4) + w(5)$ which is equal to the cutsize $c(h_4) + c(h_5)$.

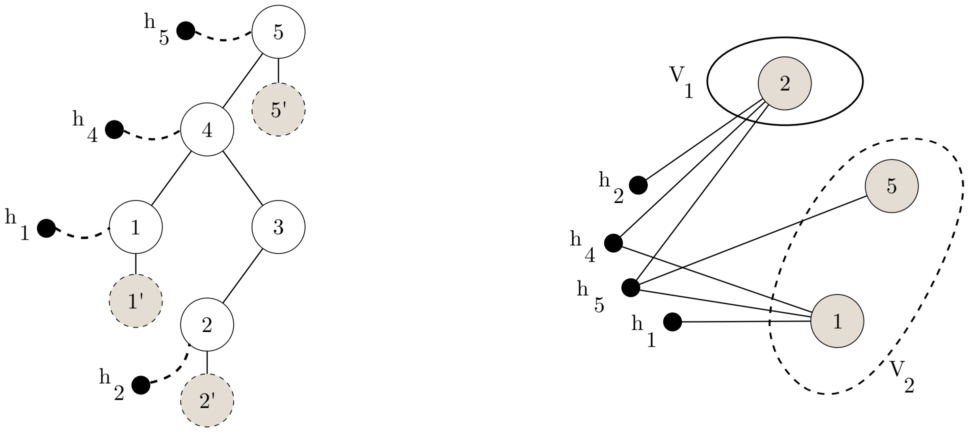


Figure 4.5: The entries a_{11}^{-1}, a_{22}^{-1}, and a_{55}^{-1} are requested. For each requested entry, a leaf node is added to the elimination tree as shown on the left. The hypergraph model for the requested entries is built as shown on the right.

Consider a tree of height H with F leaf nodes, then the size of the hypergraph model can be at most $(H \quad 1) \times F$. Assuming a nested dissection-like ordering on a matrix of size $N \times N$, the size of the hypergraph corresponding to the case where all of the diagonal entries of the inverse are requested will be $\quad (N \log N)$.

4.3.3 Hypergraph model general case

For the general case where the set of requested entries is not restricted to diagonal entries, the idea is to model the forward and backward solves with two different hypergraphs, and then to partition these two hypergraphs simultaneously. It has been shown how to partition two hypergraphs simultaneously in [88]. The essential idea, which is refined in [89], is to build a hypergraph by amalgamating the relevant vertices of the two hypergraphs, while keeping the nets intact; this technique is referred to as *vertex amalgamation*. In our case, the two hypergraphs would be the model for the diagonal entries associated with the column subscripts (forward phase), and the model for the diagonal entries associated with the row subscripts (backward phase), assuming that the same indices are distinguished by associating them with different leaf nodes. We have then to amalgamate any two vertices i and j where the entry a_{ij}^{-1} is requested.

Figure 4.6 shows an example where the requested entries are a_{71}^{-1}, a_{62}^{-1} and a_{95}^{-1}. The transformed elimination tree and the nets of the hypergraphs associated with the forward

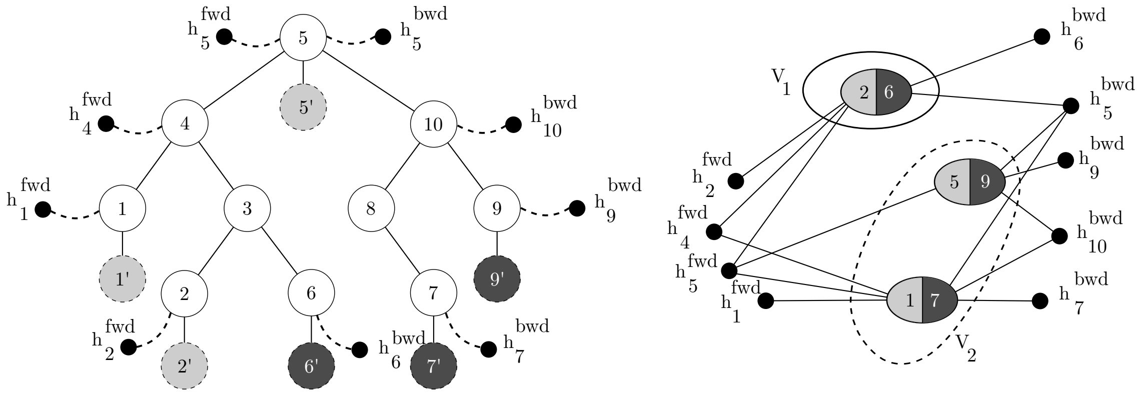


Figure 4.6: Example of hypergraph model for the general case: a_{71}^{-1}, a_{62}^{-1} and a_{95}^{-1} are requested.

(h^{fwd}) and backward (h^{bwd}) solves are shown. Note that the nets h_3^{fwd} as well as h_3^{bwd}, h_4^{bwd}, and h_8^{bwd} are removed. The nodes of the tree which correspond to the vertices of the hypergraph for the forward solves are shaded with light grey; those nodes which correspond to the vertices of the hypergraph for the backward solves are shaded with dark grey. The composite hypergraph is shown in the right-hand figure. The amalgamation of light and dark grey vertices is done according to the requested entries (vertex i and vertex j are amalgamated for a requested entry a_{ij}^{-1}). A partition is given in the right-hand figure: $\quad = \quad a_{62}^{-1} \quad a_{71}^{-1}\ a_{95}^{-1}$. The cut size is $c(h_5^{\text{bwd}}) + c(h_4^{\text{fwd}}) + c(h_5^{\text{fwd}})$. Consider the computation of a_{62}^{-1}. We need to load the L factors associated with the nodes 2 3 4 and 5 and the U factors associated with 5 4 3 and 6. Now consider the computation of a_{71}^{-1} and a_{95}^{-1}; the L factors associated with 1 4 and 5, and the U factors associated with 5 10 8 7 and 9 are loaded. In the forward solution, the L factors associated with 4 and 5 are loaded twice (instead of once if we were able to solve for all of them in a single pass), and in the backward solution the U factor associated with 5 is loaded twice (instead of once). The cutsize again corresponds to these extra loads.

We note that building such a hypergraph for the case where only diagonal entries are requested yields the hypergraph of the previous section, where each net is repeated twice.

4.4 Experiments

We conduct three sets of experiments. In the first set, we compare the quality of the results obtained by the PoPart, BiseMatch and hypergraph-based heuristics using MATLAB implementations. For these experiments, we created a large set of TreePartitioning problems, each of which is associated with computing some diagonal entries in the inverse of a sparse matrix. In the two other sets, we perform experiments with MUMPS on large matrices coming from various industrial applications. Using the out-of-core option of MUMPS, we investigate the performance of the natural ordering, the PoPart heuristic and the hypergraph model (as we can use off-the-shelf hypergraph partitioning software). In the second set of experiments, we investigate the computation of a set of diagonal entries, and in the third set we investigate the computation of off-diagonal entries.

4.4.1 Assessing the heuristics

Our first set of experiments focuses on the computation of diagonal entries and compares the heuristics PoPart, BiseMatch, and the hypergraph partitioning which were discussed in Sections 4.2 and 4.3. We have implemented the first two heuristics in MATLAB and used PaToH [27] for hypergraph partitioning. We use a set of matrices from the University of Florida (UFL) sparse matrix collection[1]. The matrices we choose satisfy the following characteristics: $10000 \leqslant N \leqslant 100000$, the average number of nonzeros per row is greater than or equal to 2.5, and in the UFL index the `posdef` field is set to 1. At the time of writing, there were a total of 60 matrices satisfying these properties. We have ordered the matrices using the `metisnd` routine of the Mesh Partitioning Toolbox [46] and have built the elimination tree associated with the ordered matrices using the `etree` function of MATLAB. We assume that the size of the solution space is proportional to the height of the elimination tree (as in Section 3.3), and we consider several settings with memory sizes equal to $k \times N$ for k 2 8 32 128 512 1024 . Since the solution space is of size the height of the tree h multiplied by the block size, and since we want B to be a power of 2 for BiseMatch, the corresponding block sizes are of the form $B = k \times 2^{\log \frac{N}{h}}$.

We have assigned random weights in the range 1 200 to tree nodes. Then, for each P 0 05 0 10 0 20 0 40 0 60 0 80 1 00 , we have created 10 instances (except for $P =$ 1 00) by randomly selecting $P \times N$ integers between 1 and N and designating them as the requested entries in the diagonal of the inverse. Notice that for a given triplet of a matrix, B, and P, we have 10 different trees to partition, resulting in a total of $10 \times 6 \times 6 \times 60 + 6 \times 60 = 21960$ TreePartitioning problems.

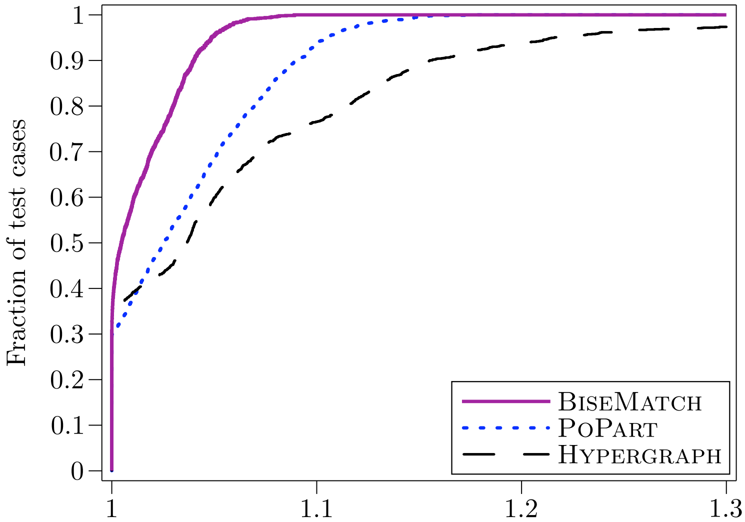


Figure 4.7: The performance profile of the proposed PoPart, BiseMatch, and hypergraph partitioning heuristics with respect to the lower bound η on the cost of the partition with respect to the amount of factors loaded. The cost obtained by each method for each TreePartitioning problem instance is divided by η for that instance, and the performance profiles are drawn with respect to these numbers. The closer the plots are to 1.0, the better the method. The graphs measure the fraction of the test cases for which the method is within x of the lower bound where x is the value on the x axis. Thus in about 90% of the test cases PoPart obtains results that are smaller than 1 1 × η.

We summarize the results with the performance profiles (proposed in [30]) shown in

[1] http://www.cise.ufl.edu/research/sparse/matrices/

Figure 4.7. The performance profile for a specific heuristic shows the fraction of the TreePartitioning problems for which the heuristic gives results that are within some value of the lower bound η. As seen in the figure, the three heuristics obtain results that are close to the optimum in almost all cases, where the ranking of the heuristics is clearly BiseMatch, PoPart, and the hypergraph partitioning-based model. The plot shows that in 90% of the cases, PoPart obtains results that are less than $1\,1 \times \eta$. Among these three heuristics, the run time complexity of the hypergraph partitioning problem is the highest. In addition to this, the memory requirements are also the highest. We think therefore that either BiseMatch or PoPart should be used in real applications when computing all the diagonal entries of the inverse. Upon a closer look at the data, we see that BiseMatch is better than PoPart in almost 13000 instances whereas the reverse is true in almost 2000 instances, in both cases the difference was very small (see also the performance profile). As these results show, we do not lose much by not implementing the BiseMatch heuristic in MUMPS. Furthermore, PoPart is the fastest of the discussed heuristics. Therefore, we identify PoPart as our preferred partitioner when computing diagonal entries of the inverse.

4.4.2 Large-scale experiments diagonal entries

In this section, we give results obtained by using MUMPS with the out-of-core option and a nested dissection ordering provided by MeTiS [55]. We use a subset of the matrices from industrial applications described in Table 1.1. All experiments have been performed with direct I/O access to files so that we can guarantee effective disk access independently of both the size of the factors and the size of the main memory. Because of the direct I/O access to the files, we are sure that the operating system will not use a system cache of uncontrolled size. I/O buffers of controlled size (see [8]) are then introduced to enable efficient prefetching. The direct I/O mechanism is also helpful in the standard solution phase [8]. All results are obtained on the `Pret` machine described in Section 1.3.4.

As indicated earlier, since we consider the computation of diagonal entries in a sequential context, we can use the tree height storage scheme for the solution space. We first calibrate the size of the solution space `WRHS` and then define the block size B by dividing the size of `WRHS` by the height of the elimination tree. Let us take a typical large matrix (e.g., AUDI) to explain how we dimension the size of the workspace `WRHS`. On this matrix, an I/O buffer area of size 700 MB is recommended for the efficient prefetching of the factors (see [8]). Furthermore, to enable Level 3 BLAS operations on dense contiguous data, a workspace of size the maximum front size times the block size is needed to hold temporary data from `WRHS`. For a typical block size of a few thousand as used in practice for this matrix, it is appropriate to have a workspace of maximum size 600 MB. For all integer and internal data 300 MB are used, and 500 MB are needed to store the initial matrix. Finally, we assume that we wish to save 400 MB for the operating system and 500 MB for the application data. This adds up to 3 GB, and since 4 GB of memory are available in our experimental system, 1 GB can be used for `WRHS`. Therefore, we will assume in the following experiments that, for a given matrix, the block size, B, is obtained from dividing of the size of the `WRHS` work array (1 GB) by the height of the associated elimination tree. The block size will thus depend on the matrix and its size will be indicated.

Table 4.1 shows the total size of the factors loaded and the execution time of the solution phase of MUMPS with different settings and partitions. A random selection of $N/10$ diagonal entries (10%) of the inverse of the given matrices is computed with the block size B defined as above. The values in the column Lower bound are computed according to Theorem 4.1. The columns Nat and PoP correspond respectively to the natural partitioning (the indices are partitioned in the natural order into blocks of size B) and to the PoPart heuristic. When reordering the right-hand sides following a postorder (PoPart) instead of following the natural order (Nat), the total number of loaded factors is reduced significantly, resulting in a noticeable impact on the execution time. The hypergraph model also provides very good volumes of loaded factors. However, compared to the PoPart heuristic which is almost free, it is rather expensive. We have tried to modify PaToH's `quality` parameter setting, which resulted in a much smaller partitioning time but did not produce good enough partitions; we set PaToH's parameter `bisec_fixednetsizetrsh` to the number of vertices in a given hypergraph.

	Lower	Volume of loaded			Running time of the			Block	Hypergraph information			
	bound	factors (GB)			solution phase (s.)			size	Time		Cells	Nets
Matrix	(GB)	Nat	PoP	HP	Nat	PoP	HP		(s.)	Size	|V|	|N|
NICE20MC	29	66	32	32	5534	1635	1619	6633	204	860840	71592	13028
AUDI	34	175	40	39	10494	2044	1988	6349	588	1351869	94369	20792
CONESHL	21	55	23	24	3622	1702	1695	8856	571	1628724	126221	34892
FLUX-2M	142	638	152	161	43947	12848	13178	4486	9925	11954507	200172	46361
CAS4R_LR15	12	123	13	12	44228	1487	1481	11316	2838	3489756	242313	127255

Table 4.1: Total size of the loaded factors and execution times using MUMPS with three different partitionings. A random 10% of the diagonal entries are requested. The memory constraint for the solution space is 1GB; the corresponding block size B is shown. The out-of-core executions use direct I/O access to the files. Columns Nat, PoP and HP refer to exploiting the sparsity of the right-hand side vectors under a natural partitioning, the PoPart heuristic, and a hypergraph partitioning, respectively. The time for partitioning the hypergraph and its size (sum of the number of vertices in each net) are shown in the last four columns.

4.4.3 Large-scale experiments off-diagonal entries

We perform two sets of experiments for the off-diagonal case to assess the proposed hypergraph partitioning-based model. In the first set, we conduct experiments with MUMPS (with the out-of-core factorization option and standard settings including an ordering based on nested dissection). As we are in the off-diagonal case, we use the union sparse storage scheme for the solution space (as mentioned in the previous chapter, the tree height scheme is restricted to diagonal entries). We set a fixed block size $B = 384$, which leads to a solution space of size close to 1 GB for the first five matrices in Table 1.1. As this leads to a smaller scale comparison in terms of block size, we also present simulation results with varying and larger block sizes as in the diagonal case. In both sets of experiments, we compute a random selection of $N/20$ off-diagonal entries (no two in the same column).

The results are shown in Table 4.2. The table displays the lower bound and the total size of the factors loaded with a natural ordering, a PoPart partition on the column indices, and a partition of the hypergraph using PaToH with default options, except that we have requested a tighter balance for the sizes of the parts.

As expected, the formulation based on hypergraph partitioning obtains a better result than PoPart in most cases. Compared to what happens in the diagonal case, the cost of partitioning the hypergraph is negligible compared to the cost of the solution phase. The storage for the model is also modest compared to the requirements of the solution phase; since the sum of the sizes of the nets of the model amounts to a few million (almost 12 million for the largest case), partitioning the hypergraph requires a few tens to a few

	Lower	Volume of loaded			Running time of the			Hypergraph information			
	bound	factors (GB)			solution phase (s.)			Time		Cells	Nets
Matrix	(GB)	Nat	PoP	HP	Nat	PoP	HP	(s.)	Size	|V|	|N|
NICE20MC	161	860	714	545	9665	7372	5651	48	858927	35796	20168
AUDI	189	1350	1069	831	13653	11036	8686	55	1347103	47184	31520
CONESHL	159	712	662	668	9115	8529	8561	101	1621030	63110	52759
FLUX-2M	712	4543	3646	3053	60318	48607	41246	311	11747048	100086	71649
CAS4R_LR15	69	1271	761	824	24588	15173	16121	178	3453464	121156	160666

Table 4.2: Total size of the loaded factors and execution times using MUMPS with three different partitionings. A random selection of $N/20$ off-diagonal entries are requested; entries are partitioned into blocks of size 384. The out-of-core executions use direct I/O access to the files. Columns Nat, PoP and HP refer to exploiting the sparsity of the right-hand side vectors under a natural partitioning, the PoPart heuristic and a hypergraph partitioning, respectively. The time for partitioning the hypergraph and its size (sum of the number of vertices in each net) are shown in the last four columns.

hundred Megabytes. There is, however, a large difference between the lower bounds and the performance of the heuristics.

Similar to the diagonal case, it is possible to modify the PaToH options to improve the quality of the partitioning, and to limit the amount of loaded factors with HP. With the same block size of 384, the volume of loaded factors with HP are for example reduced from 824 GB to 640 GB for CAS4R_LR15 and from 668 GB to 314 GB for CONESHL.

Finally, we present simulation results obtained in the off-diagonal case if an algorithm similar to the one used for the diagonal case were implemented (so that a workspace proportional to B multiplied by the height of the tree would be used, and hence the block sizes would be identical to those in Table 4.1). We show in Table 4.3 the resulting volume of loaded factors with PoPart and HP, together with the lower bound obtained. The characteristics of the hypergraph are the same as those reported in Table 4.2, because the same set of off-diagonal entries is used. We observe that the volume of factors loaded is much smaller, indicating that such an algorithm would also significantly decrease the time for solution in such an out-of-core context. They are also much closer to the lower bound than with the smaller blocks of Table 4.2.

	Lower	Volume of loaded		
	bound	factors (GB)		Block
Matrix	(GB)	PoP	HP	size
NICE20MC	22	61	50	6633
AUDI	24	91	66	6349
CONESHL	15	49	35	8856
FLUX-2M	84	403	254	4486
CAS4R_LR15	10	49	38	11316

Table 4.3: Total size of the loaded factors with PoPart and HP. The same $N/20$ off-diagonal entries as in Table 4.2 are requested but a large block size is used.

Chapter 5

Minimizing computational cost in an in-core execution

In this chapter, we consider reducing the computational cost (more specifically the number of operations) in computations involving multiple sparse right-hand sides. We have seen in Section 2.2.1 how to reduce the traversal of the tree to the *pruned tree* by exploiting sparsity in the right-hand sides and/or solution vectors. However, there are two ways of performing operations at each node of the pruned tree: either operations are performed on the whole block of right-hand side columns, or sparsity is exploited within each block, i.e., at a given node, operations are performed only on the columns of the right-hand side that this node has to update, i.e., that are nonzero at the rows corresponding to that node. Let us take an example. Consider the elimination tree in Figure 5.1 and assume that we want to solve $Lx = [e_1\ e_2]$ at the same time, i.e processing the two right-hand side columns at once. The pruned tree is 1 2 4 5 6 7 . If, at each node, operations are performed on the whole block of columns, then some unnecessary operations are performed; indeed, at node 1, the partial right-hand side block, i.e., the set of rows of the right-hand side that is passed to node 1, is [1 0], therefore unnecessary operations are performed on the second column. Similarly, since nodes 4 and 5 are not on the path from node 1 to the root node, the first component in the right-hand side block at these nodes is zero, leading to some useless operations. These extra operations are performed on the padded zeros that have been introduced so that the multiple columns of the right-hand sides have the same pattern.

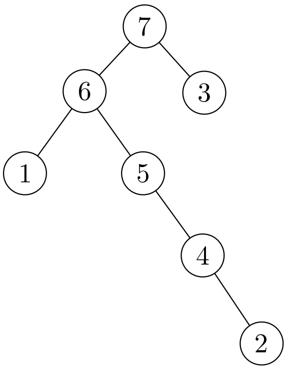


Figure 5.1: A simple elimination tree.

On the contrary, one could perform a symbolic analysis to determine which columns of the block of right-hand sides a given node needs to process (i.e., on which columns `_TRS_` and `_GEM_` need to be performed); this is what we refer to as exploiting sparsity *within* blocks.

In this chapter, we examine the two possibilities of exploiting or not exploiting the sparsity within blocks. In Section 5.1, we examine the case where sparsity is not exploited within blocks. In that context, the way the right-hand sides are partitioned into blocks strongly influences the number of operations. The problem sounds similar to what we have considered in the previous chapter, but it is actually quite different. We tackle this problem in the context of the triangular solution of the PDSLin solver described in Section 1.3.2. In Section 5.2, we examine how sparsity can be exploited within blocks. We have studied and implemented this feature in MUMPS; to our knowledge, this property is not exploited in any other solver.

5.1 Performing operations on a union of paths

We consider computations with multiple sparse right-hand sides where at each node of the pruned tree, operations are performed on the whole block (of size B) of right-hand sides (this is what we refer to as performing operations on a union of paths). In this context, the ordering of the right-hand sides influences the number of operations. Let us consider the example in Figure 5.1 and consider the solution of $Lx = [e_1\ e_2\ e_3]$, with a block size $B = 2$; one block contains two right-hand sides, and the other contains only one. Assume for simplicity that the number of operations performed at each node is equal to the number of columns in the processed block. Consider the partitioning $e_1\ e_2 \quad e_3$: the number of operations for the first block is 12 (there are six nodes in the pruned tree, and two operations are performed at each node since there are two columns in the block). The number of operations for the second block is 2 (there are two nodes in the pruned tree, 3 and 7, and one column to be processed), thus the total number of operations is 14. Now consider the partitioning $e_1\ e_3 \quad e_2$: the number of operations is 13 (4×2 for the first block plus 5×1 for the second block). The extra number of operations for the first partitioning comes from the fact that at nodes 4 and 5 two operations are performed while this does not happen with the second partitioning. Two padded zeros have been introduced to the structure of the first column (that corresponds the solution of $Lx = e_1$), at indices 4 and 5, so that the two columns of the first block (solution of $Lx = [e_1\ e_2]$) have the same structure; this is illustrated in Figure 5.2. Therefore, the partitioning of the columns of the right-hand sides influences the number of operations to be performed, and hence the performance of the triangular solution. If one uses the *union sparse* storage scheme described in Section 3.2, then the padded zeros also represent an extra memory cost since, in that case, the solution space follows the union of the paths that form the pruned tree. The problem is thus to find, for a fixed block size B, a partitioning that minimizes the number of padded zeros.

This problem sounds like the TreePartitioning discussed in the previous chapter but it is actually quite different. Let us first discuss the influence of the block size B: in the out-of-core problem, the optimal block size was $B = NBRHS$ since the nodes of the pruned tree were loaded once only; on the contrary, $B = 1$ was the worst choice. In this in-core problem, the opposite is true: $B = 1$ is the best choice as it introduces no extra operations; on the contrary $B = NBRHS$ maximizes the number of operations as, at every node in the union of paths from the structures of the *NBRHS* right-hand sides to the root node, operations will be performed on *NBRHS* columns, thus introducing many padded zeros.

$$\begin{array}{r|cc|}
\cline{2-3}
1: & X & 0 \\
2: & X & X \\
4: & 0 & X \\
5: & 0 & X \\
6: & X & X \\
7: & X & X \\
\cline{2-3}
\end{array}$$

Figure 5.2: Structure of the solution of $Lx = [e_1\ e_2]$ with the elimination tree of Figure 5.1. Components 4 and 5 of the first column are structural zeros but are stored and involved in computations; they are padded zeros .

For a given block size B, the two problems are also different: let us again consider the example in Figure 5.1 and the solution of $Lx = [e_1\ e_2\ e_3]$. We showed that in the in-core problem, the partitioning $e_1\ e_3$ e_2 was better than $e_1\ e_2$ e_3 . In the out-of-core problem, the opposite is true: on the one hand, $e_1\ e_3$ e_2 leads to accesses to 9 nodes (four for the first block, 1, 3, 6 and 7, plus five for the second block: 2, 4, 5, 6 and 7). On the other hand, $e_1\ e_2$ e_3 leads to 8 accesses (six for the first block plus two for the second block) and is thus better. The difference between the two partitionings comes from node 6, which is loaded only once in the first partitioning. The two problems are thus different; we have tried similar approaches, i.e., a postordering and hypergraph model, but the hypergraph model, that we describe in Section 5.1.2, is actually completely different from the one described in Section 4.3.

This in-core problem was motivated by the computation of the Schur complement in the PDSLin solver, where sparsity is not exploited within blocks; we reuse the notation from Section 1.3.2, although the results are applicable to any sparse triangular solution. In this context, the objective is to reorder the columns of E in order to maximize the structural similarity among adjacent columns and hence minimize the number of padded zeros in order to reduce the cost of the computation of $G = L^{-1}P E$. For the rest of this section, we drop the subscript in D , G , E , and use to denote the -th part of the m-way partition of E.

5.1.1 A partitioning based on a postorder of the tree

Similar to what was presented in Chapter 4 in the context of the computation of inverse entries, using a partitioning based on a postorder of the tree is a reasonable idea. Contrary to the computation of inverse entries, the right-hand sides here have more than one nonzero entry. We choose to reorder the columns of E according to their first nonzero entry, following the postorder of the elimination tree of the corresponding subdomain D. The reason this ordering may reduce the number of padded zeros is the same as in the previous chapter: let i and j be the first nonzero indices in two adjacent columns. Since the RHS columns are sorted by the first nonzero row indices, the two nodes i and j are likely to be close together in the postordered elimination tree, and the two paths from the i-th node and the j-th node to the root node are likely to have large degree of overlap in the elimination tree. As a result, this reordering technique is likely to increase the structural similarity among adjacent columns.

This simple heuristic is easy to implement, and is effective in practice. However, it only considers the first nonzeros in the columns, and ignores the fill-ins generated by other nonzeros.

5.1.2 Hypergraph model

Our second reordering technique is based on a hypergraph model. We reuse some of the definitions and notation introduced in Section 4.3. Our goal is to partition the columns of E into m parts, where the similarity of the row structure among the corresponding columns of G in the same part is maximized. To partition the RHS columns E into m parts, we use the *row-net hypergraph model* of the solution vectors G, whose nonzero structure is obtained by a symbolic triangular solution (using the path theorem presented in Section 2.2.1). The row-net hypergraph model of G is such that:

There is a vertex v_j for each column j of G.

There is a net n_i for each row i of G, such that $v_j \quad n_i \qquad g_{ij} = 0$.

The weight of a vertex v_j is the number of nonzeros in column j of G: $\quad i : g_{ij} = 0 \quad$.

Consider a partition $\quad_m = \quad \mathcal{V}_1 \; \mathcal{V}_2 \qquad \mathcal{V}_m \quad$ of the columns of G into m parts, with B the number of columns in each part. To simplify our discussion, we assume that the number of columns is divisible by B. Let r_i denote the set of columns of G whose i-th row have nonzeros, i.e., $r_i \qquad j : g_{ij} = 0 \quad$. Then, for a given part $\mathcal{V} \;$, the number of zeros to be padded in the i-th row is given by the formula

$$cost(r_i \;\; \mathcal{V} \;) = \begin{matrix} \mathcal{V} \qquad r_i \setminus \mathcal{V} & \text{if } r_i \setminus \mathcal{V} \; = \varnothing \\ 0 & \text{otherwise} \end{matrix}$$

If the i-th row does not have any nonzero in any columns of $\mathcal{V} \;$, then clearly no zeros are padded in the i-th row of $\mathcal{V} \;$. On the other hand, if $\mathcal{V} \;$ has a nonzero in the i-th row, then for each column in $\mathcal{V} \;$ for which $g_{ij} = 0$, there will be a padded zero. Hence, this cost function counts the number of padded zeros in the i-th row of $\mathcal{V} \;$. The total cost of $\quad_m$ is the total number of padded zeros and is given by

$$cost(\quad_m) = \sum_{i=1}^{n_G} \sum_{\mathcal{V}_\ell \in \quad_i} (\; \mathcal{V} \qquad r_i \setminus \mathcal{V} \;\;)$$

where n_G is the number of rows in G.

In our numerical experiments, we used PaToH to partition the first $m \times B$ columns of G enforcing each part to have B columns by setting the imbalance parameter to be zero. The remaining columns of G are gathered into one part at the end. Since each part has B columns, we obtain

$$cost(\quad_m) = \sum_{i=1}^{n_G} (\quad_i B \qquad r_i \;\;) \tag{5.1}$$

We can further manipulate the formula (5.1) and obtain

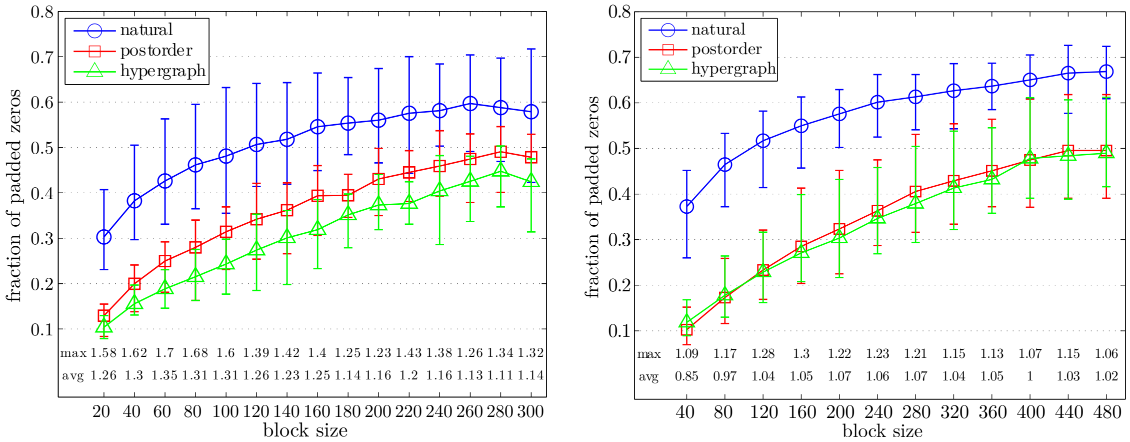
$$\begin{aligned} \sum_{i=1}^{n_G} (\quad_i B \qquad r_i \;\;) &= \sum_{i=1}^{n_G} \quad_i B \quad nnz(G) \\ &= \sum_{i=1}^{n_G} (\quad_i \quad 1)B + \sum_{i=1}^{n_G} B \quad nnz(G) \\ &= \sum_{i=1}^{n_G} (\quad_i \quad 1)B + n_G B \quad nnz(G) \end{aligned}$$

Hence, for a given G, the cost function (5.1) and the connectivity-1 metric (4.2) with each net having the constant cost of B differ only by the constant value ($n_G B \quad nnz(G)$). Therefore, one can minimize (5.1) by minimizing (4.2).

5.1.3 Experiments

We examine the performance of the two reordering techniques, namely the postordering of the elimination tree and the hypergraph partitioning-based ordering, in terms of fraction of padded zeros and in terms of triangular solution time. All results are obtained on the `Franklin` machine described in Section 1.3.4. We report results on the last two matrices of Table 1.1; a few more results can be seen in [91]. For both matrices, we extract eight subdomains using PT-Scotch [28] (parallel nested dissection), and a minimum degree ordering is applied to each subdomain.

Figure 5.3 shows the fraction of padded zeros using the postordering of the elimination tree, the hypergraph-based ordering and the ordering coming from the nested dissection (PT-Scotch) of the global matrix, that we refer to as the natural ordering . For each block size B and each of the three orderings, the bar and the marker represent the range and average over the eight data points corresponding to the eight triangular solution phases (one for each subdomain) respectively. One can see that the number of padded zeros increases as B increases. Using a postordering or the hypergraph-based ordering significantly reduces the number of padded zeros over the natural ordering. On matrix tdr190k, there is a clear difference between the postordering and the hypergraph-based ordering, while this is not the case with matrix matrix211. We believe that this comes from the fact that the filled interfaces G are much sparser on matrix211 than on tdr190k: the effective density of the eight G is between 1.1% and 3.7% for matrix211 and between 2.2% and 4.7% for tdr190k. A larger effective density provides more chance for the reordered columns to have similar row structures, and the hypergraph model seems to exploit this property better than the postordering, since the postordering takes only the first nonzero position into account. The numbers shown at the bottom of the plots are the maximum and average ratios of the number of padded zeros from the postordering over that from the hypergraph ordering.

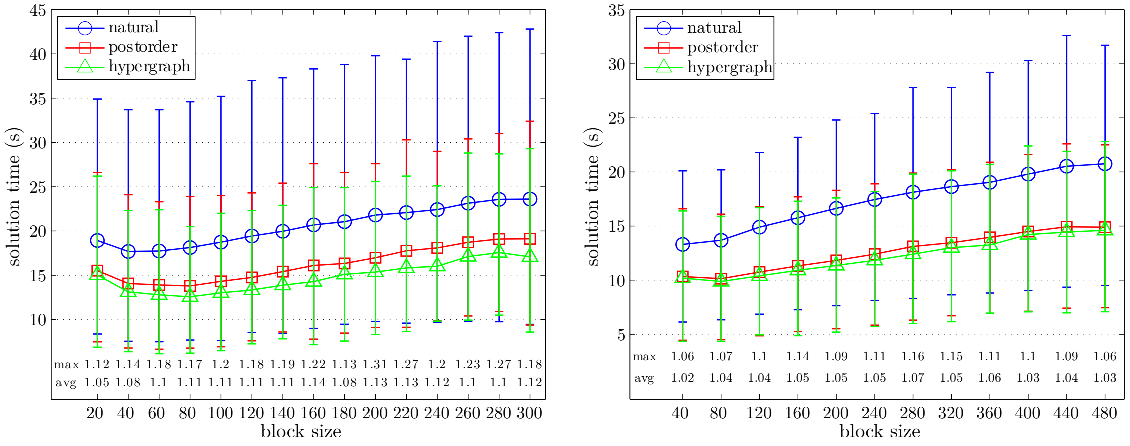


(a) tdr190k.

(b) matrix211.

Figure 5.3: Fraction of the padded zeros using different ordering schemes with varying partition block size B.

Figure 5.4 shows the total time spent in the triangular solves $L^{-1}E$ using the three above-mentioned orderings. On the two medium-size problems tdr190k and matrix211, the optimal block size seems to be around 80 when using the postordering or the hypergraph-based partitioning. Using one of these two orderings instead of the natural ordering provides gains in completion time between 15% and 30%. The speed-ups gained by the hypergraph ordering over the postordering are shown in the figures at the bottom of the plots.

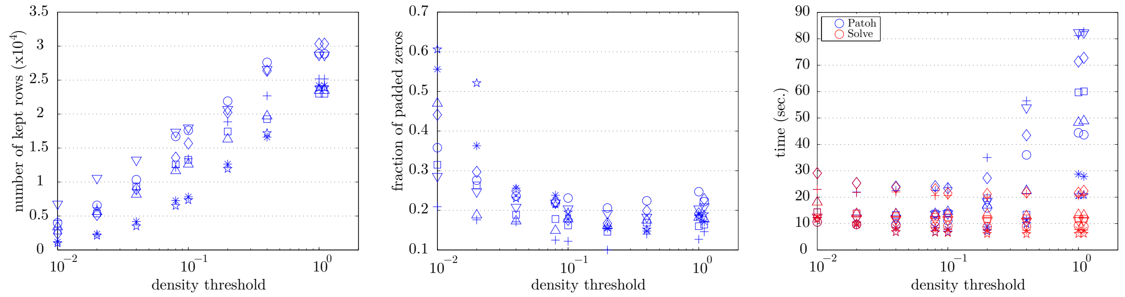


(a) tdr190k.

(b) matrix211.

Figure 5.4: Sparse triangular solution time using different ordering schemes with varying partition block size B.

We are interested in reducing the cost of the hypergraph partitioning, which was prohibitive in some cases. We examined the effect of removing quasi-dense rows of the solution vectors from the model: indeed, since these rows are nonzero for almost all the columns of the solution vectors, they do not give much information and unnecessarily increase the size of the model. We report on some experiments in Figure 5.5. The block size B is fixed at 60, and we remove from the model the rows whose density is greater than or equal to a density threshold . Figure 5.5(a) shows the number of rows that are kept in the model as a function of the threshold ; Figure 5.5(b) shows the fraction of padded zeros (using the hypergraph-based partitioning) as a function of ; finally, Figure 5.5(c) shows the triangular solution time and the time for partitioning the hypergraph as a function of .



(a) Number of kept rows.

(b) Number of padded zeros.

(c) Run time.

Figure 5.5: Effects of applying a density threshold before partitioning the sparse right-hand sides, for matrix tdr190k. Each marker (e.g., circle or square) represents one of the eight subdomains generated.

One can see that when using a threshold   around 0 1, about 50% of rows are removed; the time for computing the hypergraph is significantly decreased, but the quality of the partitioning (number of padded zeros) remains the same, thus the time for the triangular solution is not affected. When becomes too small (10^{-2}), almost all rows are removed; the time for hypergraph partitioning becomes negligible, but the fraction of padded zeros significantly increases (which means that too much information was removed from the model), and the solution time increases as well.

5.2 Exploiting sparsity within blocks

We now examine the exploitation of sparsity within blocks of right-hand side columns. This idea came out when working on the parallelization of the A^{-1} feature in MUMPS: it turned out that exploiting sparsity within blocks was the only way to enable tree parallelism without increasing the number of operations. This is described in the next chapter. In this chapter, we show how to exploit sparsity within blocks of right-hand sides and present a few experimental results in a sequential context. For the sake of simplicity, we first consider the computation of a set of diagonal inverse entries, and we then provide a few remarks about the general case (off-diagonal entries or general sparse right-hand sides).

5.2.1 Core idea and construction

We have illustrated the exploitation of sparsity within blocks in Chapter 3 in order to introduce the tree height storage scheme. Let us recall the same example. Consider the elimination tree in Figure 5.6(a). Assume that we want to compute a_{22}^{-1} and a_{44}^{-1} at the same time. This implies solving $Lx = [e_2\ e_4]$ with the two right-hand side columns processed at the same time, i.e., $B = 2$. Here the pruned tree is the whole elimination tree. Variable 2 belongs to node 1 and variable 4 belongs to node 2; since node 1 is not on the path from node 2 to the root node $L^{-1}e_4\ {}_2$, is a structural zero. Therefore, at node 1 where the computations associated with the eliminated variable 2 are performed, it is not necessary to process the whole block (the two columns) of right-hand sides; processing only the first column suffices. A similar idea applies to the backward phase: at the end of the solution of $Ax = [e_2\ e_4]$ with $B = 2$, a_{22}^{-1} and a_{24}^{-1} can be found at the second row of the block of solution vectors. However, a_{24}^{-1} is not requested: at node 1 (that holds the eliminated variable 2) it it thus not necessary to process the second column of the block of right-hand sides; processing only the first column suffices.

The core idea is to work at each node only on the columns from the B right-hand sides currently being processed that are *active* at that node. We denote $[e_{j_1}\ e_{j_2}\quad e_{j_B}]$ the right-hand side block. By saying that a column e_{j_k} is active at a node $\mathcal{N}$, we mean that this node belongs to $\mathcal{P}(j_k)$, the pruned tree corresponding to the single column e_{j_k} (which is included in the pruned tree corresponding to the whole B-sized block). This is equivalent to saying that there is an eliminated variable i belonging to node $\mathcal{N}$ and such that $(L^{-1}y_i)_{j_k}$ is structurally nonzero. Therefore, at node $\mathcal{N}$ where row i is processed, it is not necessary to operate on the B columns, but only on the subset of columns that are active at that node; this is what we call exploiting sparsity within the B-sized block. Thus the set on which we do our computations is as small as it can be and so we are as efficient in terms of operation count as possible if the sparsity of an individual right-hand side is not exploited (i.e., for $B = 1$). Thus, at the leaf nodes, the set of right-hand sides on which computations are performed will normally be quite small, corresponding only to entries present at that node. However, note that because of tree pruning, there will be at least one right-hand side being operated on at every leaf node. As we progress up the tree, the computational set will increase, with the set at any node being the union of the set at the children with any new entries appearing at the node. At the root node (assuming irreducibility) the set will be of size B, i.e., the whole right-hand side block. A very important feature is that, if we postorder the B block of right-hand sides, the set at any node will always be a contiguous subset of entries from the B block so that only the position of the first and last entries need be passed and the merging process at a node is trivial. This implies that we do not need a costly indirection to indicate which columns need to be processed by a given node: we simply need to provide every node with an *interval* of columns to be processed. No copy into a temporary buffer is needed.

There is no constraint on the size of the subblock to be processed at each node. For example, at a node where only one variable is active, the size of the subblock to be processed is 1. This is likely to happen at a leaf node of the elimination tree. On an earlier version of our algorithm, we requested a minimum size of computational block (that we called B_{sparse}) so that the block size at any node was a multiple of B_{sparse} although the contiguity property was still maintained. However, although this was attractive because of specifying minimum computational units for the BLAS, it could mean that we did totally unnecessary operations. With our present approach, as we progress up the tree, we will have bigger blocks when it is more efficient to use the BLAS particularly if we wish to exploit parallel BLAS.

These blocks can be computed in two steps, as presented in Algorithm 5.1:

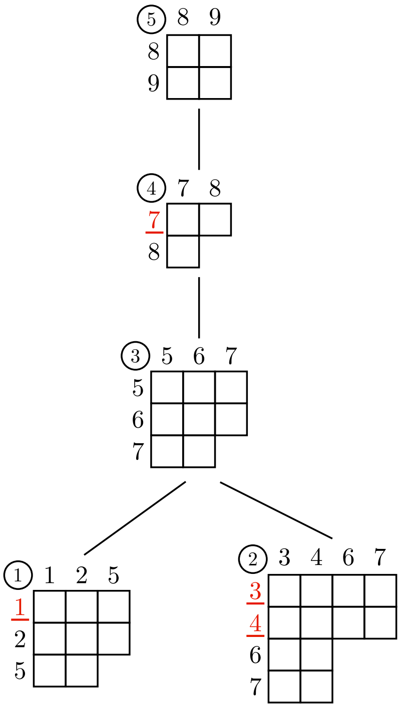
1. *Initialization:* we work on requested nodes only, that is nodes of the pruned elimination tree containing a variable which is a row or column subscript of a requested entry (for the forward and backward step respectively). The block of right-hand sides that we will deal with at this node is initialized for each entry of the inverse corresponding to a column index of the frontal matrix. Since each B block of right-hand sides is postordered, the variables of each requested node that are a column index (or row in the backward case) of a requested entry correspond to consecutive columns of the right-hand side; therefore, each set is an interval.

2. *Propagation:* following a bottom-up traversal, the block for each node is computed as the union of its own (initialized) block and the blocks of its children. Since we are working on the pruned tree, every leaf corresponds to at least one requested entry and has an initialized set, thus ensuring that all sets are well-defined. By recursion (starting at the leaves), since each block of right-hand sides is postordered, the sets of the children at every node of the pruned tree are consecutive intervals, and thus the union is an interval. Therefore, the block of right-hand sides associated with every node of the pruned tree is an interval of columns from the B block.

Algorithm 5.1 Exploiting sparsity using subblocks of a B-sized block.

```
1: Initialization phase: a subblock of columns of the right-hand sides is associated
   with each node of the assembly tree corresponding to (a)requested entry(ies).
2: Propagation phase: the columns to be processed by a node are defined as the union
   of its subblock and those of its children.
```

We provide in Figure 5.6 an example of computations of the intervals along the lines of the algorithm presented above. B is fixed at 4 and diagonal entries a_{11}^{-1}, a_{33}^{-1}, a_{44}^{-1} and a_{77}^{-1} are requested, therefore, during the forward elimination step, the system $Lx = [e_1\ e_3\ e_4\ e_7]$ is solved. During the initialization phase, nodes 1, 2 and 4 are concerned as they contain the variables corresponding to the requested entries. For example, node 4 contains variable 7, which corresponds to the fourth column in the right-hand side block; thus its interval is initialized with [4 4]. Then, during the propagation phase, it is computed as [4 4] union the interval associated with node 3 (its single child). The interval associated with node 3 is empty after the initialization (because no entry of the right-hand side corresponds to node 3), and then is the union of the interval corresponding to 2 with the interval corresponding to 3; thus it is ∅ [1 1] [2 3] = [1 3]. Therefore, the interval associated with 4 is [1 3] [4 4], yielding the interval [1 4].



(a) Elimination tree.

Node	Intervals after:	
	Initialization	Propagation
1	[1 1]	[1 1]
2	[2 3]	[2 3]
3	∅	[1 3]
4	[4 4]	[1 4]
5	∅	[1 4]

(b) Intervals.

Figure 5.6: Example of computation of the interval to be processed at each node. The requested entries are a^{-1}_{11}, a^{-1}_{33}, a^{-1}_{44} and a^{-1}_{77}; they correspond to the underlined indices in the tree (a). The intervals to be processed at each node are indicated in (b).

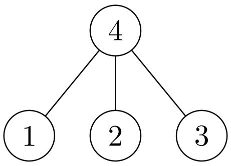
We conclude this section with some remarks:

In the general case where the right-hand sides or solution vectors do not have a single nonzero entry, we cannot necessarily permute the blocks in order to guarantee that the sets of columns to be processed at each node will be contiguous, i.e., intervals. In this case, in order to keep an efficient and simple implementation, the set of columns to be processed at each node is defined as the interval that borders that set. This introduces some padded zeros (as in the previous section) and increases the number of operations (although it is still reduced compared to the case where sparsity is not exploited within each block), but it avoids the use of indirections and lists. Similar to what is presented in the previous section, the way the columns are ordered influences the number of padded zeros; we have not investigated this problem, but once again using a postorder-based permutation seems to be reasonable strategy.

We provide an example: assume that one has to solve

$$Lx = \begin{matrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{matrix}$$

using the elimination tree in Figure 5.7, with $B = 3$. Whatever the permutation of the right-hand sides is, one of the nodes has to process a discontiguous set of columns of the right-hand sides: for example, processing the right-hand sides in their original order, node 1 has to process columns 1 and 3 but not column 2.



(a) Elimination tree.

$$\begin{array}{r|ccc|}
\hline
1: & X & 0 & X \\
2: & X & X & 0 \\
3: & 0 & X & X \\
4: & X & X & X \\
\hline
\end{array}$$

(b) Structure of the solution.

Node	Interval
1	1 3 [1 3]
2	[1 2]
3	[2 3]
4	[1 3]

(c) Intervals.

Figure 5.7: Example of elimination tree (a) for which, if one has to solve $Lx = [e_1 + e_2\ e_2 + e_3\ e_1 + e_3]$ with $B = 3$, there is no way to permute the right-hand side columns so that the set of columns to be processed at each node is contiguous, unless some padded zeros are introduced. The structure of the solution (b) and the intervals associated with each node (c) are shown in the case where the permutation is the natural order 1 2 3 .

When exploiting sparsity within each block, the order of the columns *within* each block has an influence on the number of operations (in the same manner as in Section 5.1), while this is not the case when sparsity is not exploited within blocks (since operations are performed on the whole block at every node of the pruned tree). Therefore, this provides some leeway for the backward phase, which is interesting when computing a set of off-diagonal inverse entries. We showed in Chapter 4 and in Section 5.1 that, when computing off-diagonal entries (or, more generally, when dealing with sparse right-hand sides and sparse solution vectors with different structures) it is difficult to find a permutation of the right-hand side columns that can minimize the number of accesses to the factors or the computational cost. This is because it is hard to find a permutation than can be beneficial for both the forward and backward phases. When exploiting sparsity within blocks, one can for example choose to use a global permutation that has a good effect for the forward phase and then reorder each block between the two phases following a permutation that will locally have a good effect on the backward solution.

When exploiting sparsity within blocks, the block size B does not have an influence on the operation count (for a fixed ordering of the columns, and assuming that no padded zeros are introduced). Therefore, it is advantageous to use blocks as large as possible (the block size being perhaps limited by memory constraints), as this will be beneficial for the BLAS and will not increase the number of operations.

5.2.2 Experiments

We report on experiments with some of the matrices of our experimental set (Table 1.1) in Table 5.1. 10% of the diagonal entries of the inverse are computed using the sparse inverse functionality in MUMPS; the block size is $B = 1024$. Experiments were performed on one node (i.e., 8 cores) of the `Hyperion` system defined in Section 1.3.4, using an 8-way multithreaded BLAS and one MPI process. We report the time and the number of operations for the solution phase, with and without exploiting sparsity within each block.

	Operations ($\times 10^{12}$)		Time (s)	
Matrix	w/o ES within blocks	w/ ES within blocks	w/o ES within blocks	w/ ES within blocks
AUDI	42.2	36.9	1385	985
NICE20MC	35.4	31.1	1137	817
bone010	33.3	28.7	1213	719
CONESHL	34.1	31.3	1285	808
Hook_1498	111.2	104.9	2807	2141
CAS4R_LR15	15.0	12.9	1144	582

Table 5.1: Influence of exploiting sparsity (ES) within each block of right-hand sides. 10% of the diagonal inverse entries are computed using the sparse inverse functionality in MUMPS, with $B = 1024$, on the `Hyperion` system defined in Section 1.3.4, using an 8-way multithreaded BLAS.

As expected, exploiting sparsity within each block of right-hand sides decreases the number of operations to be performed; this results in some improvement in computation time. We see that the improvement is a superlinear function of the number of operations: we believe this comes from the fact that working on a reduced set of columns at each node improves data locality, and that it also comes from the fact that the operations that are suppressed (compared to a strategy where sparsity is not exploited within blocks) are likely to correspond to nodes at the bottom of the tree, where the flop rates are often low, since those nodes are usually small. Note that when exploiting sparsity within the blocks, we reach a speed of 49 GFlops/s (for matrix Hook_1498), which is almost 60% of the peak of `DGEMM` on this system. This is a very good speed for a sparse triangular solution, especially since we exploit sparsity at many different levels (in the factors, in the right-hand sides and between columns of the right-hand sides). We provide further results in the next chapter, where the ability to exploit sparsity within blocks of right-hand sides is used in order to obtain an efficient parallelization of the computation of inverse entries.

Chapter 6

Enhancing parallelism

In this chapter, we concentrate on the parallel performance of the solution phase with multiple sparse right-hand sides. Once again, our leading application is the computation of inverse entries but all the ideas presented here equally apply to general sparse right-hand sides. We first show that although performing triangular solution on multiple blocks seems embarrassingly parallel, since one would like to process several blocks simultaneously, we have no choice but to process blocks of right-hand sides one after each other. We then show that in the case of multiple sparse right-hand sides, enabling tree parallelism and reducing the size of the pruned tree (in order to reduce the computational cost) are contrasting objectives. We finally suggest that in order to enable tree parallelism without increasing the computational cost, one has to exploit sparsity *within* blocks of right-hand sides, as described in the previous section.

6.1 Processing multiple right-hand sides in a parallel context

In the context of computing many entries of the inverse, we solve for several right-hand sides at the same time and, at first glance, this seems to exhibit the classical phenomenon of being embarrassingly parallel. However, when we combine this with the fact that we wish to exploit a parallel matrix factorization, the situation is not that straightforward and we find that we lack the mechanism to fully exploit the independence of the right-hand sides.

In a distributed memory environment, we would like to run parallel instances of the linear solver in parallel, each instance using the whole set of processes to solve for a block of right-hand sides (because all distributed factors might have to be accessed), which is not possible using MPI. A potential workaround would be to replicate the factors on all processes, or to write the factors shared on disk(s), and to simulate a shared memory paradigm by launching sequential instances in parallel, each of them accessing the distributed factors. Unfortunately, this is not feasible for any distributed-memory sparse solver; furthermore, such a solution would then really lose the benefit of our parallel factorization and the cost of accessing the distributed factors (which might be stored on local disks) would be prohibitive.

Therefore, the blocks of entries to be computed are processed one at a time. Note that, in a shared memory environment where each thread has access to the single copy of the factors held in the main memory, blocked solves could be performed in parallel (perhaps using threaded BLAS). We could expect a good speed-up (up to the number of cores/processors), but the approach will be limited because of constraints in the shared memory system: parallel accesses to the single instance of the factors might induce much bus contention between the threads.

6.2 Combining tree parallelism and tree pruning

6.2.1 An interleaving strategy

We first assume that sparsity is not exploited within blocks, as in most sparse solvers; as presented in Section 5.1, using a good permutation of the right-hand sides is crucial in order to the reduce the number of operations. In the case of the computation of inverse entries, using a postorder is a reasonable strategy (especially when computing diagonal entries) as it puts together nodes which are close in the tree and thus reduces the size of the pruned tree. Commonly used mapping techniques are usually close to a subtree to subcube mapping [40]; nodes in the lower part of the tree are likely to be mapped onto a single process, whereas nodes in the upper part of the tree are mapped onto several processes (we provide more detail on this in Chapter 7). As a consequence, few processes (probably only one) will be active in the lower part of tree when processing a block of right-hand sides. An *interleaving* strategy was suggested in [84] to remedy this situation: it consists of interleaving the different right-hand sides so that every process will be active when a block of entries is computed. This algorithm is described in Algorithm 6.1. We propose some modifications and some alternative strategies later, in particular for the management of Type 2 nodes.

Algorithm 6.1 Interleaving algorithm.

```
/* Input: old_rhs: preordered list of requested entries        */
/*        node-to-master process mapping                       */
/* Output: new_rhs: the interleaved list of entries            */

Current process    $P_0$
while old_rhs has not been completely traversed do
    Look for an entry corresponding to a node mapped onto the current process
    if an entry has been found then
        Add the entry to new_rhs
    end if
    Change current process (cyclicly)
end while
```

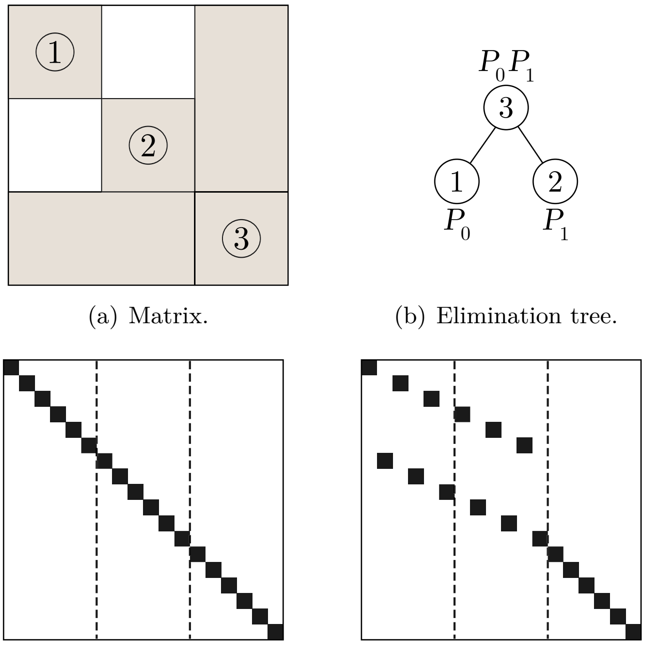
We illustrate the problems raised by this approach on the following (archetypal) example, illustrated in Figure 6.1. We assume that all the diagonal entries of A^{-1} are requested and that the block size B is $N/3$. The right-hand sides are processed following a postordering (which is straightforward here). Let us examine different situations:

1 proces: on this example, for a given B, the postorder is optimal with respect to the number of operations. As mentioned in the previous chapter, the optimal block size in terms of operation count is in general 1: indeed, as the assembly tree is pruned for each block, and as each node of the pruned assembly tree processes B right-hand sides (as we do not exploit sparsity within the blocks), increasing the block size necessarily increases the number of operations.

2 processes: nodes 1 and 2 are mapped onto processes P_0, and P_1 respectively, and node 3 is a Type 2 node (mapped onto P_0 *and* P_1).

Without interleaving: when processing the first block (columns 1 to $N/3$ of the right-hand sides shown in Figure 6.1(c)), only nodes 1 and 3 are traversed. Thus, at the bottom of the tree (node 1), only one process, P_0, is active. Similarly, when processing the second block, only P_1 is active at the bottom of the tree.

With interleaving: entries are computed by blocks such that each block has $N/6$ entries on P_0 and $N/6$ entries on P_1 (see Figure 6.1(d)). However, all the $N/3$ columns of the block must be operated on by each node (we do not take advantage of sparsity in the right-hand sides), so the operation count is multiplied by 2 and so is the speed (2 processes active at the same time). Thus, there is no speed-up.



(a) Matrix.

(b) Elimination tree.

(c) Right-hand sides.

(d) Interleaved RHS.

Figure 6.1: Archetypal example: computation of the diagonal entries of the matrix in (a), corresponding to the elimination tree in (b). The original right-hand side is (c) and is permuted using the interleaving procedure shown in Algorithm 6.1 in order to enable tree parallelism (d): every block has components that belong to node 1 and 2 thus processes P_0 and P_1 are active at the bottom of the tree when processing the first two blocks, but the operation count increases.

This example illustrates the fact that interleaving is a good way to put all processes to work, but tends to destroy the benefits of the postordering (or any other permutation aimed at reducing the number of operations).

6.2.2 Combining interleaving and sparsity within blocks

We showed that we need a strategy able to find a trade-off between the number of operations and parallelism (i.e., performing activities that involve as many processes as possible): this can be achieved by exploiting sparsity within each block. Let us take the same example: first, a postorder is applied within each block of right-hand sides, yielding the set of columns in Figure 6.2. Then, each node in the tree is provided with a subset of columns of the right-hand sides that this node has to process: in the first and the second blocks, node 1 only has to process the first $N/6$ columns of the block (instead of the whole block of $N/3$ columns), since only these columns contain variables that belong to that node. Node 2 only has to process the last $N/6$ columns of the block. At node 3, the whole block is processed (the set of columns to be processed is the union of the intervals coming from its children, for instance the union of the first half and the second half of the $N/3$ columns). Therefore, the number of operations is the same as it is without using interleaving; however, tree parallelism is still exploited (processes P_0 and P_1 are active at the same time). Thus, combining the interleaving strategy with the ability to exploit sparsity within blocks of right-hand sides enables tree parallelism without affecting the operation count, and is thus likely to provide some parallel efficiency.

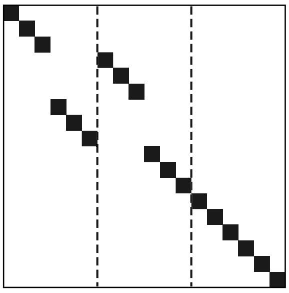


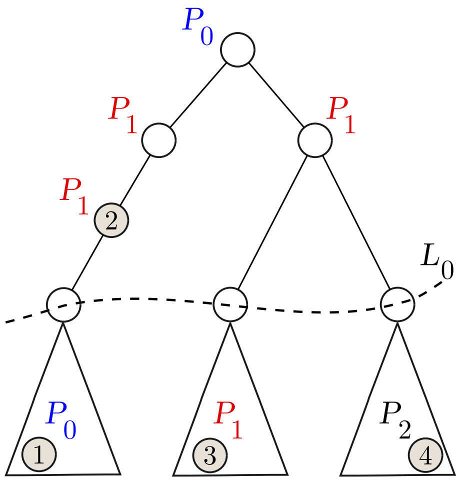
Figure 6.2: Set of right-hand side columns after postordering each block.

We also address two issues related to the interleaving process described in Algorithm 6.1. The first issue is related to the way the node-to-process mapping is done. Mapping algorithms usually identify a set of sequential tasks (subtrees that will be processed by a single process), relying for example on the Geist-Ng algorithm [40]. We call the set of roots of the sequential subtrees *layer* L_0. Then the upper part of the tree is mapped, and this mapping can strongly influence the behaviour of the interleaving algorithm. We illustrate this in Figure 6.3, where four entries are requested. With a block size of 2, Algorithm 6.1 yields the partitioning {{1,2},{3,4}}. This gives poor parallelism in the first block since nodes corresponding to entries 1 and 2 are processed one after another (the computation of the block involves two processes but they are not active at the same time). We suggest the following strategy: perform an interleaving pass first on the lower part of the tree (sequential subtrees at layer L_0), then on the upper part. On the example of Figure 6.3, the partitioning becomes {{1,3},{4,2}}, which provides a better parallelism since two processes are active *at the same time* within each block.

We now address the issue of managing the parallel nodes (so-called Type 2 nodes) when interleaving the requested entries over the processes. We note that we did not consider Type 2 nodes in Algorithm 6.1. The strategy used in [84] is, whenever a Type 2 node is found, to keep selecting entries that belong to nodes mapped onto the current process. The assumption is that the requested entries are ordered following the postorder, and thus this strategy implies that all entries in the current node will be ordered consecutively. This idea works well if all processes are involved in the Type 2 node, but might give a poor interleaving if this is not the case, that is if Type 2 nodes are rather small and involve only a small number of processes.

Another strategy is to count the load on each process (that is the number of entries selected on the process), and, when a Type 2 node is encountered:

1. The load on all the processes concerned with this node is updated;



(a) Elimination tree.

Entries in postorder:
| 1 2 3 4 |
Interleaving with original management of L_0:
| 1 2 3 4 |
Interleaving with new management of L_0:
| 1 3 4 2 |

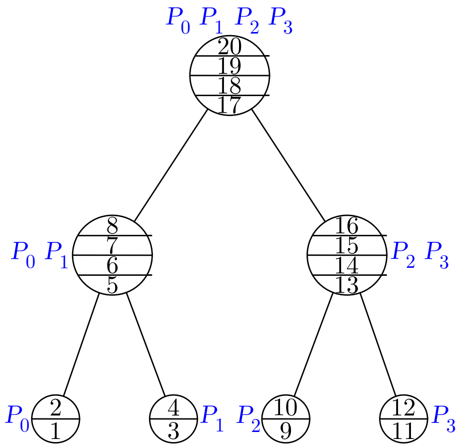
(b) Entries.

Figure 6.3: Example of interleavings obtained with different management of layer L_0.

2. The least loaded process (among all processes) becomes the current process.

This strategy is expected to give better parallelism (more processes involved in the computation of a block) and better load balancing. Note that, in this case, a complete mapping has to be provided, that is, for each node, the master and the list of processes that take part in the processing of the node.

We illustrate this idea in Figure 6.4. On this example, all the diagonal entries are requested. If $B = 4$, we see that, with the first strategy, some blocks involve only a few processes. For example, the third block is [5 6 7 8] with the first strategy, and involves only two processes; with the alternative strategy described above, the third block becomes [5 13 6 14] and involves all the processes.



(a) Elimination tree.

Postorder:
|1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20|

Interleaving with original management of Type 2 nodes:
|1 3 9 11 2 4 10 12 5 6 7 8 17 18 19 20 13 14 15 16|

Interleaving with new management of Type 2 nodes:
|1 3 9 11 2 4 10 12 5 13 6 14 7 15 8 16 17 18 19 20|

(b) Entries.

Figure 6.4: Example of interleavings with different managements of Type 2 nodes.

We have implemented these different variants but we have not pushed our experiments very far. We have not observed clear differences between these variants, thus we do not report on results. In the experimental results reported in the next section, we use the baseline interleaving strategy.

6.3 Experiments

6.3.1 Influence of the different strategies

We first illustrate in Table 6.1 the influence of the different strategies (interleaving and exploiting sparsity within blocks) on a medium size problem (11-pt stencil discretization of a $200 \times 200 \times 20$ domain), using one node (eight cores) of the `Hyperion` system defined in Section 1.3.4, with single-threaded BLAS and $B = 512$. We show the time for the solution phase, the operation count and the average number of processes active at the same time during the computation, for one, four and eight MPI processes. Firstly, we see that using the baseline strategy (i.e., neither interleaving nor exploiting sparsity within blocks), the parallel efficiency is low (e.g., the speed-up is 1 3 on eight processes): this is because the average number of processes that are active at the same time is close to one. If the interleaving procedure is activated but sparsity is not exploited within blocks, the operation count significantly increases (e.g., it is multiplied by 4 on eight processes); this prevents good speed-up, even though the average number of active processes increases. However, when sparsity is exploited within blocks, the operation count is no longer increased (it actually decreases), and thus speed-ups are significantly better: on eight processes, the speed-up is almost 4.

Procs	Strategy		Time (seconds)	Operation ($\times 10^{12}$)	Active procs
1	-		1667	16.2	1
4	IL off	ES off	1366	16.2	1.20
	IL on	ES off	2028	45.4	3.92
		ES on	659	15.2	
8	IL off	ES off	1241	13.3	1.10
	IL on	ES off	1508	61.0	7.76
		ES on	418	12.4	

Table 6.1: Computation of a random 10% of the diagonal entries of the inverse of a matrix corresponding to an 11-pt stencil discretization of a $200 \times 200 \times 20$ domain, using one node of the `Hyperion` system defined in Section 1.3.4. The different strategies are compared: IL stands for the interleaving strategy and ES for exploiting sparsity within blocks of right-hand sides. We indicate the time for the solution phase, the operation count and the average number of processes that are active at the same time during the computation.

We report in Table 6.2 on experiments on some problems from our experimental set (Table 1.1), using four nodes of the `Hyperion` system. Here we simply compare the baseline strategy with the new one where both interleaving and exploiting sparsity within blocks are enabled. The new strategy is significantly better than the baseline algorithm: on matrix NICE20MC, the speed-up increases from 3 8 to 22 (on 32 processes). This is a satisfying performance for a triangular solution phase, especially considering that sparsity is exploited at many different levels.

Matrix	MPI processes		
	1	32	
		Baseline	IL+ES
AUDI	1143	380	75
NICE20MC	945	245	43
bone010	922	327	70
CONESHL	803	293	48
Hook_1498	1860	720	173
CAS4R_LR15	882	482	116

Table 6.2: Time in seconds for the computation of a random 1% of the diagonal entries of the inverse. The block size is $B = 1024$. We compare the sequential performance with the parallel performance on 32 MPI processes (4 nodes of the `Hyperion` system), using the baseline algorithm and the combination of interleaving (IL) and sparsity within blocks (ES).

6.3.2 Influence of the block size

We show in Table 6.3 the influence of the block size B for the 11-point problem described above. In the sequential case, increasing the block size B from 64 to 128 decreases the time for the solution phase, which is probably due to the efficiency of the BLAS on larger blocks. However, when increasing B from 128 to 512 and then 1024, the efficiency of the BLAS cannot compensate for the fact that the operation count increases as a function of B; therefore, the solution time increases. However, when exploiting sparsity within blocks, the operation count does not depend on B; therefore, on eight processes, when right-hand sides are interleaved and sparsity is exploited within blocks, the time for the solution phase decreases as a function of B. With $B = 1024$, the speed-up is 5 3. If we compare the best sequential time with the best parallel time, the speed-up is almost 4.

Procs	Strategy		Block size			
			64	128	512	1024
1	-		1518	1432	1667	2002
8	IL on	ES on	555	466	418	379

Table 6.3: Influence of the block size: a random 10% of the diagonal entries of the inverse of a matrix corresponding to an 11-pt stencil discretization of a $200 \times 200 \times 20$ domain are computed. The influence of the block size B on the time for the solution phase (indicated in seconds) is illustrated both for sequential and parallel executions, with interleaving (IL) and exploiting sparsity within blocks (ES), on one node of the `Hyperion` system.

Part II

Controlling active memory in the parallel multifrontal factorization

Chapter 7

Introduction

The second part of this dissertation addresses memory issues in the parallel sparse factorization. Much work has been carried out to improve the performance of sparse factorizations, but it is well known that memory is often a bottleneck that prevents the use of direct solvers. We address the problem of reducing the amount of temporary data used during the factorization. The multifrontal factorization requires to store some temporary data (the contribution blocks) that adds up to the memory reserved for the factors. The *active memory*, which consists of a frontal matrix being processed and a stack of contribution blocks, can vary much during the factorization and it can be significantly large. Similarly, efficient supernodal methods relying on the fan-in scheme also require the use of temporary memory spaces called *aggregated update blocks* [14]. We focus exclusively on the multifrontal factorization (contrary to the first part of the study that applies both to supernodal and multifrontal methods), although many ideas could be adapted to the supernodal case.

Some studies, that we review in Section 7.2, have tackled the problem of minimizing the active memory of the multifrontal factorization in a sequential context; however, the parallel case has not been addressed much. Controlling and estimating the active memory in a parallel context are difficult, especially in an asynchronous solver such as MUMPS. We have assessed this problem both theoretically and with practical experiments, and we showed that commonly used node-to-process mapping strategies do not lead to a good memory scalability of the active memory. The main goal of this study has thus been to develop mapping and scheduling algorithms that are able to enforce some memory constraints (provided by the user or computed automatically), and that lead to a predictable behavior in terms of memory usage. Being able to control the active memory is crucial since, as we demonstrate in this chapter, the amount of active memory can dangerously grow with the number of processes. It is also important to have accurate memory estimates for the robustness of direct solvers, especially for those, such as MUMPS, that rely on estimates computed during the analysis phase to perform a static allocation of the memory at the beginning of the factorization. We have implemented these strategies within MUMPS and have carried out experiments on large matrices. A property of our approach is that, compared to the mapping technique previously used in MUMPS, it tends to assign more processes to nodes at the top of the tree; this raised performance issues in the parallel dense kernels used within MUMPS, and we have suggested and implemented some enhancements.

The plan of this second part is as follows: in this introductory chapter, we describe the parallel multifrontal factorization, and we particularly emphasize the asynchronous scheme used within MUMPS. We describe commonly used static mapping, dynamic scheduling and memory estimates strategies, once again emphasizing what is done in MUMPS. We review previous studies that address the problem of minimizing active memory in a sequential context, and we formulate the parallel counterpart of this problem. In Chapter 8, we show why commonly used mapping techniques fail at achieving a good memory scalability: we start with a very simple example, and we then derive a theoretical proof on regular grids with nested dissection. We also provide some experimental results that illustrate the need for a better control of the active memory. In Chapter 9, we describe a mapping technique that we refer to as the *memory-aware mapping*. It aims at maximizing performance while enforcing a given memory constraint. The starting point for this part of the study is Agullo's PhD thesis [1, Chapter 10], where the main idea of the *memory-aware mapping* is introduced and some preliminary experimental results are provided; we have suggested a few improvements and have pushed the implementation and the experiments further. In Chapter 10, we highlight some implementation and performance issues; we especially emphasize some communication pattern issues that arise in the parallel dense kernels used within MUMPS; although this is not directly related to our main problem (controlling the active memory), these issues are raised by the nature of our mapping algorithm. We have tackled these problems (in particular, we have studied and implemented an asynchronous broadcast algorithm), and this enabled to leverage the performance of MUMPS. Finally, we present in Chapter 11 experimental results on large matrices and large number of cores.

7.1 The parallel multifrontal factorization

7.1.1 An asynchronous scheme

We describe a simple asynchronous scheme that outlines what is done in MUMPS or other asynchronous solvers. The idea of an asynchronous approach is to maintain a pool of ready tasks on every process during the factorization. An initial *static mapping* initializes the pool of tasks of every process with tasks that can be processed immediately (typically the leaves of the tree). Then, whenever the dependencies of a task are met, this task is inserted in one of the local pools. Large tasks can be processed using several processes. The *dynamic scheduling* is in charge of two decisions:

On a given process, the selection from its local pool of the next ready task to be processed.

The subdivision of large tasks and their distribution on a set of processes. The process in charge of distributing and organizing a task is called the *master process*; the other processes are called the *slave processes*. Slave processes are selected dynamically according to a scheduling policy that relies on an estimated global view of the state (memory usage, workload. . .) of all the processes. This implies that processes have to exchange information with what we call *state information messages*.

We provide in Algorithm 7.1 a generic asynchronous approach that can be used in many parallel computations. The main advantage of such an approach is that it is dynamic; the scheduling can compensate the load imbalances or delays that arise during the computation. This is clearly interesting because the complexity of modern hardware architectures makes it almost impossible to predict the performance. It is also especially interesting in the context of numerical pivoting, since it tends to create load imbalance; furthermore, delayed pivots require to modify the graph of tasks (the tree in our case) on the fly, as described in Section 1.2. This tends to increase the size of the nodes near the top of the tree, thus these nodes might need to be processed using more processes than predicted.

Algorithm 7.1 A generic asynchronous algorithm (local view of a given process).

```
/* Input: a set of tasks to be processed */

pool    share of initial ready tasks of the process /* static mapping */
while the global termination is not detected do
    if a state information message is ready to be received then
        Receive the message
        Process the message, i.e., update the estimated state information of other processes
    else if an application-related message is ready to be received then
        Receive the message
        Process the message, i.e., typically, perform some computations, insert a new ready task in the pool, update scheduling dependencies, send a new message...
    else if the local pool is not empty then
        Extract a task $T$ from the pool /* dynamic scheduling */
        if $T$ is large then
            Select some slaves /* dynamic scheduling */
            Subdivide $T$, i.e., send work to the slaves (application-related messages)
            Process $T$ in cooperation with the slaves (using asynchronous application-related messages)
        end if
        Update local state information and send it to other processes
    end if
end while
```

7.1.2 Common mapping techniques

In this section we review common static mapping techniques for sparse factorizations (particularly the multifrontal factorization), and we emphasize the mapping and scheduling strategies used in MUMPS. One of the earliest efforts is the *sparse-wrap* scheme [42]; its first step consists of mapping the leaves of the tree on the processes in a wrap fashion (similar to a round-robin). Then, assuming the leaves are removed from the tree, the new set of leaves is mapped in the same manner, and this process is repeated until the root of the tree is mapped. The main drawback of this technique is that it generates large amounts of communication, because it ignores the topology of the tree. The subtree-to-subcube mapping [43] addresses this issue. It assumes a balanced tree (e.g. the nested dissection of a regular problem) and a number of processes equal to a power of two. The central motivation is to obtain a mapping that fits multiprocessor systems with a hypercube topology. The mapping process consists of a top-down traversal of the tree. Chains of nodes (that correspond to separators of the domain) are mapped in a wrap fashion. Whenever a node with two children subtrees[1] is met, the list of processes is split into two lists (that correspond to two disjoint hypercubes of lower dimension) and each subtree is mapped onto one of the lists. This scheme is shown to minimize communication requirements on regular problems and is also shown to significantly enhance the performance of the triangular solution. However, it is not suitable for irregular problems.

Geist and Ng generalized the subtree-to-cube mapping and suggested the *bin-pack mapping* [40], that can be used on irregular trees. It consists of computing a layer in

[1] Or four children subtrees, depending on the models; cf. the study on the nested dissection in Chapter 8.

the tree (that we denote L_0) with minimum size such that the subtrees rooted at that layer can be mapped onto the processes ensuring a load balance criterion. Firstly, the root node is taken as the potential layer L_0; then, until the balance condition is met, the potential layer is modified by replacing the node with the heaviest load by its children. We illustrate this process in Figure 7.1. When L_0 is found, every subtree rooted at a node of that layer is processed on a single processor; subtrees are mapped following a *first-fit-decreasing* heuristic (i.e., trees with the heaviest load first). Above this layer, nodes are mapped in a wrap fashion. Pothen and Sun adapted this scheme to clique trees (i.e., assembly trees instead of elimination trees in the strict sense) and suggested a different way of mapping the nodes above L_0 (which they call the *dominating cliques*) [75]. In a bottom-up traversal starting from L_0, every node is mapped onto the union of the lists of processors its children are mapped onto.

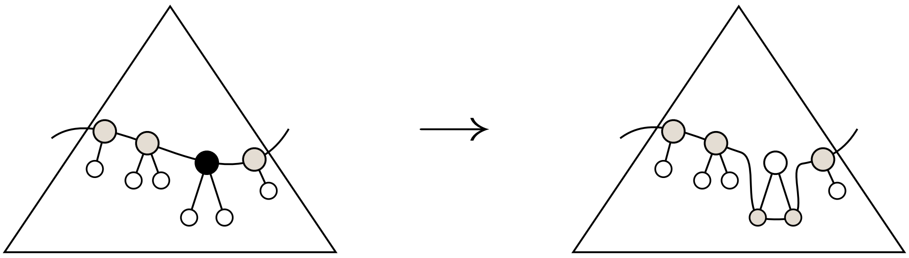


Figure 7.1: A step of construction of the layer L_0 in Geist and Ng bin-pack mapping. The potential layer (shaded nodes) is increased by replacing the node with the heaviest load (solid node) by its children.

The proportional mapping by Pothen and Sun generalizes the subtree-to-cube mapping [75]; the aim is to better take the shape of the tree and the distribution of the workloads into account. The mapping process consists of a top-down traversal of the tree. Firstly, all the processes are assigned to work on the root node. Then, at every node in the tree, the list of processes working at that node is split among its children, proportionally to the weights (workloads) of the subtrees rooted at these children. Consider a node in the tree with nc_f children. Denote p_f the number of processes working at that node and W_i the load of the subtree rooted at a child i. The number of processes given to node i is

$$p_i = \frac{W_i}{\sum_{j=1}^{nc_f} W_j} \quad p_f$$

Here the metric used at each step of the mapping is the workload of each subtree; this yields what we refer to as a *workload-based proportional mapping*. Clearly, this criterion can be replaced by another depending on the objectives. If one aims at enforcing a memory balance rather than a load balance, one can use a *memory-based* variant; we report on experimental results using this strategy in the next chapters. Prasanna and Musicus proposed an optimal (in terms of run time) scheduling strategy for tree-shaped graphs of tasks, where the time for computing a parallel task (a *malleable task*) using p processes is $\frac{L}{p^{\alpha}}$ (with $0 < \quad \leqslant 1$), where L is the length of the task [77]. Beaumont and Guermouche assessed this behaviour in MUMPS [17]; they found that parallel nodes often exhibit slightly superlinear speed-ups (i.e., $\quad > 1$) because of the partitioning used in MUMPS, where the master process is always in charge of the same part of the node, regardless of the number of processes. For example, when going from 2 to 4 processes, the number of slave processes goes from 1 to 3, yielding a superlinear speed-up (assuming the master part is small enough). Beaumont and Guermouche however proposed to use the scheduling suggested by Prasanna and Musicus even if > 1. This amounts to a proportional mapping where the number of processes given to a node i

$$p_i = \frac{L_i^{\frac{1}{\alpha}}}{\sum_{j=1}^{nc_f} L_j^{\frac{1}{\alpha}}} \quad p_f$$

Note that in general, a step of proportional mapping generally attributes decimal number of processes to each subtree. One can choose to arrange these numbers so that they perfectly add up to the number of processes given to the parent node, yielding a perfect partitioning of the set of processes on the children subtrees. We present a more robust scheme in Chapter 9.

An interesting property of the proportional mapping is that the traversal of every process (i.e., the set of tasks that this process performs and the order in which it processes them) is fully known in advance. Indeed, every process is in charge of a sequential subtree and takes part in the computation of the parallel nodes that lie on the path between the root node of this sequential subtree and the root node of the elimination tree; this defines a unique possible traversal. We illustrate this in Figure 7.2. Denoting i the root node of the sequential subtree mapped onto a given process, the traversal followed by that process is $\mathcal{T}(i) \quad \mathcal{P}(i)$. If no dynamic scheduling strategy is enabled, every process will follow such a traversal.

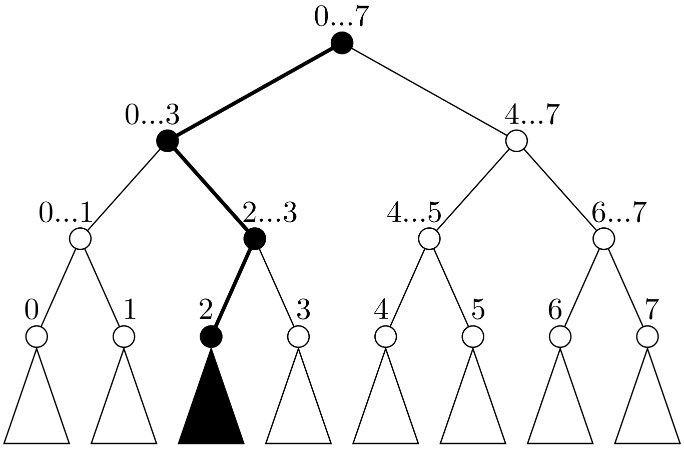


Figure 7.2: When using a proportional mapping, every process follows a partial postorder traversal of the tree. In the figure, a perfectly balanced complete binary tree is mapped onto 8 processes. The traversal for process 2 is highlighted.

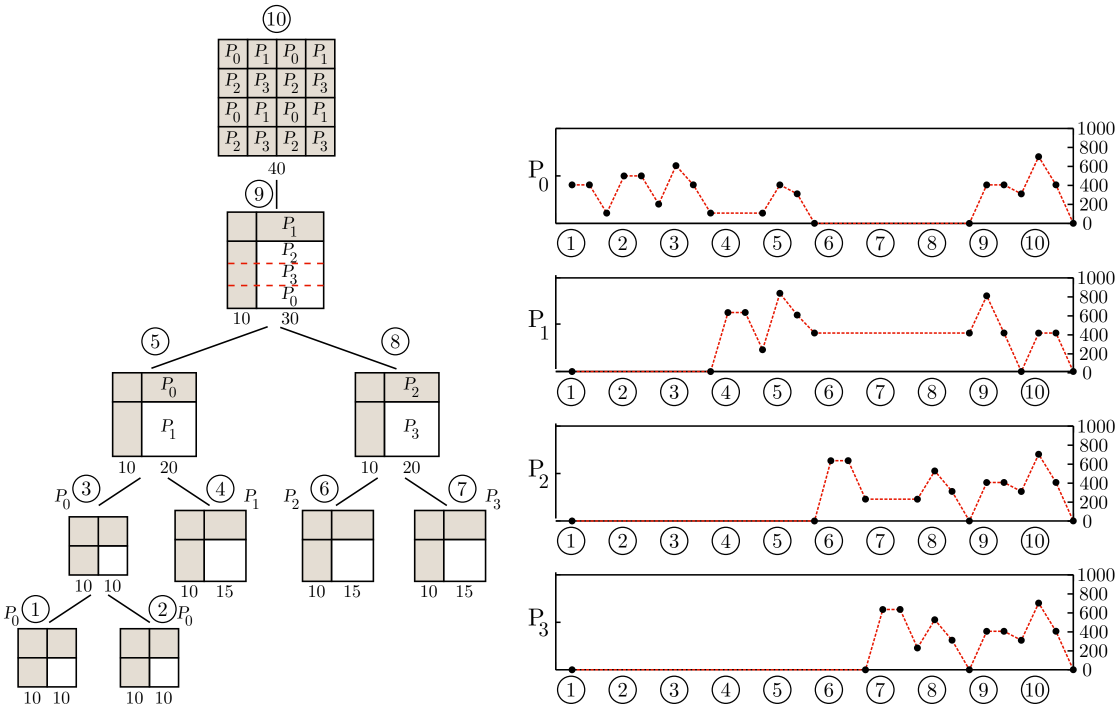
The main advantage of the above-mentioned property is that it is very easy to estimate the memory usage of a process (both the factors part and the active memory, but we focus on the active memory); indeed, one can simply simulate the traversal followed by the process and simulate the variations in its memory usage. We illustrate this in the unsymmetric case and consider that no in-place assembly is performed (cf. Section 7.2). Consider a given process P; when a node i is activated:

The frontal matrix associated with i is allocated; if P works on i, the memory usage of P increases accordingly.

The contribution blocks of the children nodes of i are freed from the memory; since we consider a proportional mapping, P works on only one of the children nodes of i. The memory usage of P decreases.

The factor part of the frontal matrix associated with i is deducted from the active memory (we recall that we consider the active memory and not the total memory).

We illustrate this behavior in Figure 7.3, where we show, for a given tree mapped onto 4 processes using a proportional mapping, the behavior of the memory consumption following the global sequential postorder (remember that we know that every process follows a partial postorder). Consider for example process P_0. P_0 works on node 1, that has 10 fully-summed variables and a contribution block of size 10. Firstly, P_0 allocates the whole frontal matrix corresponding to 1, i.e., 400 reals. Then, since node 1 has no children, nothing is freed from memory. In the third step, the factor part of 1 is deducted from the active memory, therefore the memory usage becomes 400 300 = 100 reals (this is the contribution block of 1). We traverse the whole tree repeating this process and we have in the end an history of the memory usage of every process.



(a) Example of tree.

(b) Memory consumption (reals).

Figure 7.3: Memory consumption for a tree mapped onto 4 processes using a proportional mapping: the tree and its mapping with *npiv* and *ncb* shown below each node (a) and the memory consumption for each one of the processes as a function of the global postorder (node 1 to node 10) (b). Every node in the tree corresponds to three potential changes in the memory usage of a process: the node is allocated, the contribution blocks of its children are freed, the factor part of the frontal matrix is deducted from the active memory.

We provide in Algorithm 7.2 an algorithm that estimates the peak of active memory of a given process by following a postorder traversal of the tree; every node corresponds to three possible updates in the peak, as in the above-mentioned process. However, there is no assumption on the mapping; in the case of a proportional mapping, the algorithm computes the exact peak of active memory thanks to the above-mentioned property. If a mapping where the traversal for each process cannot be predicted is used, then the algorithm provides only an estimate, without any guarantee if the mapping does not exhibit any good property. For simplicity, we assume that we are in the unsymmetric case and that slaves of Type 2 nodes share equals parts of the node.

Algorithm 7.2 Estimation of the peak of active memory of a given process.

```
/* Input: the elimination tree $T$, its mapping and its sequential postorder $order$   */
/*        $P$ a given process                                                          */
/* Output: $S$, the peak of active memory of the process we consider                   */
/* Local: $M$, the current active memory of the process we consider                    */

$S \quad 0 \; M \quad 0$
for all nodes $i$ in $T$, following $order$ do
    /* 1/ The front is allocated */
    if $P$ works on $i$ then
        if $i$ is of Type 1 then
            $M \quad M + sfront_i$ /* Allocate the whole front */
        else if $i$ is of Type 2 then
            if $P$ is the master process of $i$ then
                $M \quad M + npiv_i \quad nfront_i$ /* Allocate the (1,1) and (2,2) block */
            else
                $M \quad M + \frac{ncb_i}{p_i - 1} \quad nfront_i$ /* Allocate a stripe of the (2,1) and (2,2) block */
            end if
        else if $i$ is of Type 3 then
            $M \quad M + \frac{sfront_i}{p_i}$ /* Processes equally share the surface of the front */
        end if
    end if
    $S = \max(S \; M)$ /* Potential update of $S$ */
    /* 2/ The contribution blocks of the children of $i$ are freed */
    for all children nodes $j$ of $i$ do
        if $P$ works as a slave process on $j$ then
            $M \quad M \quad \frac{ncb_j}{p_j - 1} \quad nfront_j$ /* Free a stripe of contribution block */
        end if
    end for
    /* 3/ The factor part is deducted for the active memory */
    if $P$ works on $i$ then
        if $i$ is of Type 1 then
            $M \quad M \quad {npiv_i}^2 \quad 2 \quad npiv_i \quad ncb_i$ /* Deduct the whole factors part */
        else if $i$ is of Type 2 then
            if $P$ is the master process of $i$ then
                $M \quad M \quad npiv_i \quad nfront_i$ /* Deduct the (1,1) and the (1,2) block */
            else
                $M \quad M \quad \frac{ncb_i}{p_i - 1} \quad npiv_i$ /* Deduct a stripe of (1,1) block */
            end if
        end if
    end if
end for
```

Until now, we have described *strict* proportional mapping strategies in which the set of processes associated with a node is split into disjoint sets that are distributed to its children. As we will demonstrate in the next chapter, this is not a memory friendly strategy. One can choose to use a *relaxed* proportional mapping in which the sets of processes given to different siblings can overlap. The main drawback is that the traversal followed by a process is no longer predictable; indeed, a process is allowed to work on two parallel branches and will follow a traversal that interleaves the nodes of these branches in some way. In consequence, one cannot accurately estimate the memory usage of a given process.

7.1.3 Mapping and scheduling techniques in MUMPS

We briefly trace the history of the mapping and scheduling techniques used in MUMPS. In the earliest versions, MUMPS was fully dynamic in the sense that every process could be selected to work on any Type 2 node [9]. MUMPS 4.0 works as follows. Any process needs to allocate enough memory to be able to work on every Type 2 node, and, in order to enable an efficient dynamic scheduling during the factorization, parts of the initial matrix are duplicated onto all the processes. Furthermore, there is an upper bound on the size of the slave tasks due to a limit on the size of the communication buffers. Thus, there is a minimum number of slave processes per Type 2 node. Memory estimates are computed following a worst-case scenario where, at every node, only this minimum number of slaves is selected. For example, if the minimum number of processes per Type 2 node is 3, estimates are computed assuming that every process has to store, for every node i in the tree, $\frac{scb_i}{3}$. This leads to severe overestimates. The mapping process is the following:

1. Using the above-mentioned Geist-Ng algorithm, the layer L_0 is built.

2. Every node above L_0 is given one of the types described in Section 1.3.1 (1, 2, or 3).

3. A master process is assigned to every Type 2 node.

During the factorization, the scheduler gives the priority to message reception and processing. The idea is that since a message is a potential source of work and parallelism, giving the priority to message reception can avoid that a sender with a full buffer is blocked. We will come back to this aspect in Chapter 10.

The concept of *candidate processes* is introduced in MUMPS 4.2 [10]. Each Type 2 node has a limited set of potential slave processes. The interest is that all non-candidates of a node do not take the memory or the workload associated with this node into account in their estimates, which leads to a more accurate prediction. Furthermore, this notion can be used to enforce a subtree-to-subcube principle; nodes that are close in the tree can be mapped onto neighboring processors. This enhanced mapping works as follows:

1. The layer L_0 is built as before.

2. A relaxed workload-based proportional mapping is applied and determines a set of *preferential processes* for each node. For each layer in the tree,

    a) The tree is modified (nodes can be amalgamated or split cf. Chapter 10).

    b) The type of every node in the layer is determined. Firstly, the number of candidates of every Type 2 node is equal to its number of preferential processes. Then, optionally, candidates of the layer can be redistributed according to the relative weight of the nodes. The motivation is that since the proportional mapping relies on the weight of the subtrees, a large node rooting a small subtree might have a small number of preferentials.

c) Tasks are mapped using a *layer-wise task mapping*, which is a variant of list scheduling [48]. Each task has a list of potential processes (its preferentials, then its non-preferentials, sorted by increasing load). Tasks are mapped one after another, from the most to the least expensive, on the first process (of their list) that satisfies some constraints (e.g. memory or work limit).

3. The tree is postprocessed following a top-down traversal to improve memory balance. For every Type 2 node, the master process and one of the candidates may be exchanged if this improves memory balance.

At run time, the slave processes of a Type 2 node are selected among its candidates, the least loaded ones first. The number of slaves of a Type 2 node, n_{slaves}, must satisfy a minimum granularity condition: $n_{slaves} \quad \max \quad \frac{ncb}{\texttt{k\_max}} \quad 1$, where `k_max` is a parameter; when `k_max` increases, the amount of communications decrease (because Type 2 nodes are mapped onto fewer processes) and the performance tends to increase, but the gap between memory estimates and the effective memory usage increases.

Finally, in MUMPS 4.6, the zone above L_0 is replaced with 3 zones (L_0 is Zone 4) [11]:

Zone 1: a relaxed proportional mapping is applied at the top of the tree, and is aimed at providing some flexibility during the factorization, which enables to correct imbalances that appear in the tree.

Zone 2: below Zone 1, a strict(er) proportional mapping is applied.

Zone 3: between Zone 2 and L_0 (Zone 4), each node inherits the preferentials of its parent. This means that this zone locally behaves like the fully dynamic code mentioned above. The size of Zone 3 is based on a maximum number of processes `proc_max`; during the mapping, if the number of preferentials given to a node is less than `proc_max`, then this node and its descendants (down to L_0) belong to Zone 3 and have the same set of candidates. This choice is motivated by the following ideas:

The fully dynamic code is competitive on small number of processes, and the number of preferentials of a node of this zone is likely to be small.

Increasing the number of candidates near L_0 makes the memory management easier (large contribution blocks are handled without problem), and the leeway in the choice of candidates is likely to provide a better balance.

This can be used to take memory locality on shared-memory systems or systems with shared-memory nodes into account.

The main drawback is that severe overestimates occur in this zone, as in the fully dynamic code.

The choice of the candidates of Type 2 nodes is similar to what is done in the above-mentioned strategy, except that it takes more constraints into account (the maximum size of a buffer, the remaining memory and the maximum size of the factors). Memory estimates are computed thanks to a bottom-up traversal that simulates the factorization; for each process, the memory estimate increases when an assembly or a factorization occurs, and decreases when a contribution block is discarded, as described in the previous section. The main difference with the previous versions of the code is that memory estimates no longer follow a worst-case scenario where, at every node, a minimum number of processes is used. Instead, they rely on a best-case scenario where, at every node, work can be assigned to all of its candidates. This leads to smaller estimates, that are more accurate but can sometimes be too optimistic. Memory estimates are relaxed (by a multiplicative factor), in order to give more flexibility and to anticipate the effects of delayed pivoting.

During the numerical factorization, the scheduler uses the following mechanisms:

- Memory monitoring: the supplementary memory of each process is kept up-to-date; it is the difference between the factor size predicted according to the optimistic scenario and the effective factor size assigned to the process. This monitoring allows the scheduler to deviate from the optimistic scenario by using this flexibility.
- Anticipation of the tasks: the arrival of costly tasks is anticipated slightly before they are inserted in the pool of tasks, and taken into account when choosing candidates. When selecting slave processes, if a potential slave process has a small current workload but will be heavily loaded soon, the master of a node will likely not select it as a slave.
- Dynamic selection of the slaves: a maximum number of slaves is determined; it is the minimum between the number of candidates and a bound on the ratio between the computational costs of the slaves and master tasks. Then, the scheduler tries to find a set of slaves so that:
  - All constraints are respected: maximum buffer size, remaining memory (taking the deviation from the static mapping into account) and maximum size of the factors.
  - The maximum workload (over the slaves) is minimized.

This last strategy is the one used in the current version of MUMPS at the time of writing (version 4.10.0). It is worth noticing that memory issues have often been the motivation for improving the mapping and scheduling strategies. The dynamic scheduling strategy used in the current version relies on very sophisticated mechanisms that compensate the load and memory imbalances that arise during the factorization. However, we believe that the static part can be improved. We demonstrate in Chapter 8 that proportional mapping-based strategies fail at achieving a good scalability of the active memory, and we suggest a new mapping strategy in Chapter 9.

7.2 Controlling the active memory

In this section, we provide some details about the mechanism of stack memory used in the multifrontal method. We review the studies that address the problem of minimizing the active memory in a sequential context, which amounts to finding an optimal traversal of the tree. We then describe the parallel counterpart of this problem; in particular, we define the notion of *memory efficiency* that will be our reference metric in the rest of the study. Finally, we show that our problem can be formulated as a *tree pebble game*. We recall the work by Liu that addresses the sequential case [69], and we show how to derive the parallel case. Although we do not use this formulation in the rest of the study, we believe it draws an interesting combinatorial parallel and could be used to derive some heuristics.

7.2.1 The sequential case

In the rest of the study, we denote S_i the sequential peak of active memory associated with a node i, i.e., the peak of active memory yielded by a sequential traversal of the subtree rooted at i. Note that this is not the same thing as the peak of active memory when reaching i during the traversal of the whole tree. We introduce the notion of *stacked set*[2]. We assume that the tree is traversed following a postorder (we discuss this later).

Definition 7.1 - Stacked set.
The stacked set $\mathcal{S}(i)$ of a node i is the set of nodes whose siblings have not been visited when the traversal reaches node i:

$$\mathcal{S}(i) = \ j : j \text{ is numbered before } i \text{ in the postorder and } sib(j) \setminus \mathcal{P}(i) = \varnothing$$

The notion of stacked set is illustrated in Figure 7.4. When reaching i, the maximum active memory usage is at least S_i plus the memory for the contribution blocks of the nodes in $\mathcal{S}(i)$, the stacked set of i, that are stored at least until i is reached. We insist that S_i is the peak of active memory for traversing the subtree rooted at i, forgetting about the rest of the tree. When comparing sequential and parallel executions, we will denote S_{seq} the peak of active memory for a sequential traversal of the tree; $S_{seq} = S_r$ with r the root node of the tree.

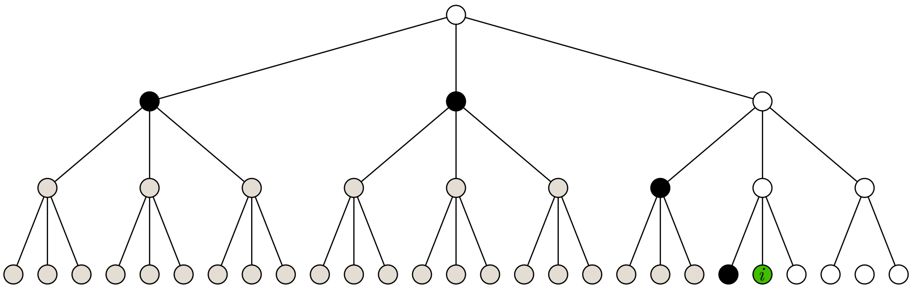


Figure 7.4: Stacked set $\mathcal{S}(i)$ of a node i. Shaded nodes are nodes that have been visited, and solid nodes are the nodes in the stacked set.

The way the peak is computed depends on the *assembly scheme*, that defines:

- The moment when the frontal matrix of a parent node is allocated.
- The choice of an overlap in memory between the frontal matrix of a parent node and the first contribution block that is assembled to it.

The first leeway is the moment when a frontal matrix is allocated (with respect to the frontal matrices of its children nodes). The possibilities are the following:

- The *terminal allocation scheme*: the memory for a frontal matrix is allocated after all its children have been processed. This is the most common scheme.
- The *early allocation scheme*: the memory for a frontal matrix is allocated right after its first child has been processed. This allows to consume the following contribution blocks on the fly.
- The *flexible allocation scheme:* the memory of the frontal matrix is allocated at an appropriately chosen time in order to consume some of the contribution blocks of its children on the fly. The way the tree is traversed and the position at which each frontal matrix is allocated can significantly improve the memory usage [49].

[2] The notion is derived from [90] where it is called the visited set and defined for binary trees.

In the following, we consider only the terminal allocation scheme in order to provide a simple global picture of the management of the active memory in a sequential context. We refer the reader to [60] for a thorough description.

The second margin is the possible overlap between a frontal matrix and contribution block of the first child assembled into it. The possibilities are the following:

The *classical assembly scheme*: at a given time, the frontal matrix associated with a node is not allowed to overlap with any of the contribution blocks in the stack memory. This is for example used in the MA41 code [3]. In this context (and assuming a terminal allocation), the sequential peak of active memory at a node i with children s_{ij} $j = 1 \quad nc_i$ is computed as

$$S_i = \max \quad \max_{j=1 \quad nc_i} \quad S_{s_{ij}} + \sum_{k=1}^{j-1} scb_{s_{ik}} \qquad sfront_i + \sum_{j=1}^{nc_i} scb_{s_{ij}}$$

The first term in the expression (the first term in the max) is the maximum, over the children of i, of the peaks of these children plus what is stacked in memory before each child. The second term is the allocation of node i; in the classical allocation scheme, the front associated with i is not allowed to overlap in memory with the contribution blocks of its children, therefore the memory needed to allocate i is the size of the associated frontal matrix plus the size of all the contribution blocks coming from its children.

The *in-place assembly scheme*[3]: the memory for the frontal matrix of a parent node is allowed to overlap with the contribution block at the top of the stack. Assuming that the tree is traversed following a postorder (we discuss this later in this section), this contribution block is the last one to be computed before that node and the first one to be assembled into it. It can be assembled in-place to form the frontal matrix. This requires the order of the variables of a node and its child to be compatible; we refer the reader to [60] for more details. This scheme is available in MUMPS (among other solvers) for sequential executions and sequential parts of the tree in parallel executions. In this context, the sequential peak of active memory for a node i is computed as

$$S_i = \max \quad \max_{j=1 \quad nc_i} \quad S_{s_{ij}} + \sum_{k=1}^{j-1} scb_{s_{ik}} \qquad sfront_i + \sum_{j=1}^{nc_i - 1} scb_{s_{ij}}$$

Note that the only difference compared to the classical scheme is in the term that represents the allocation of the frontal matrix associated with i: the contribution block of the last child is removed from the expression since it is included in the memory for the frontal matrix of i.

The *max-in-place assembly scheme*: this a natural extension of the in-place scheme where the a frontal matrix overlaps in memory with the largest contribution block [2]. This requires a different memory management (that we do not describe here) since the corresponding child is not necessarily the last one to be processed before its parent. Using this scheme, the sequential peak of active memory for a node is

$$S_i = \max \quad \max_{j=1 \quad nc_i} \quad S_{s_{ij}} + \sum_{k=1}^{j-1} scb_{s_{ik}} \qquad sfront_i + \sum_{j=1 \; j=j_{max}}^{nc_i} scb_{s_{ij}}$$

[3] Also known as the *last-in-place* scheme.

with j_{max} the index of the child node of i with the largest contribution block.

For a given assembly scheme, the way the tree is traversed strongly influences the peak of active memory. We recall that in the multifrontal method the only constraint is to traverse the tree following a topological ordering (i.e., a parent node is processed after its children). Among topological orderings, postorders are often preferred because they allow for a simple and efficient management of the memory using a stack mechanism that provides data locality. Furthermore, postorders also allow the use of the above-mentioned in-place assembly scheme, that provides significant gains in active memory in practice (often about 30% [60]). Liu [68, 69] and Jacquelin et al. [54] showed how to compute the best postorder and the best topological order; furthermore, Jacquelin et al. experimentally assess how these two possibilities compare to one another. We review these works. Firstly, Jacquelin et al. show that it is possible to build trees for which postorders perform arbitrarily bad compared to the best traversal. They consider general task trees where every node is a task associated with three types of files (input, output, temporary files); they consider top-down traversals of the tree but the transformation to bottom-up traversals is easy. They assume that the system has a 2-level memory hierarchy (typically a fast but small main memory, and a large but fast secondary memory) and tackle two problems:

> MinMemory: find the minimum amount of main memory needed to traverse the tree without accessing the secondary memory; provide the associated traversal.
>
> MinIO: given a main memory size, find the minimum volume of I/O (between the main memory and the secondary memory) needed to traverse the tree; provide the associated traversal and an I/O scheduling (i.e., the communications between the two levels of memory).

MinMemory is the problem we are interested in. The authors propose an algorithm, MinMem, that finds the best topological order. It relies on an algorithm called Explore that checks if the tree can be processed with a given amount of memory. If this is not the case, Explore returns a *cut* in the tree reachable with the given amount of memory, the associated traversal, and the extra amount of memory needed to add at least another node to the traversal. Then, by running Explore again with this extra amount of memory, the cut will be improved. MinMem simply consists of running Explore until the leaves of the tree are reached (remember that the authors consider top-down traversals) and thus finds the minimum amount of memory needed to traverse the tree.

Concerning the minimization of the volume of I/O, Jacquelin et al. show that the three following problems, that look increasingly difficult, are NP-complete[4]:

1. Given a postorder of the tree, find the I/O scheduling that minimizes the volume of I/O. Note that in the particular case of the multifrontal method, this problem is polynomial [2].

2. Find the minimum volume of I/O for any postorder.

3. Find the minimum volume of I/O for any topological order (this is MinIO)

They propose some heuristics that define the scheduling, relying on the order given by solving MinMemory. Their experimental findings are the following:

[4] They use a reduction from 2-Partition.

> On trees corresponding to large sparse matrices coming from real applications, the optimal traversal is a postorder 95% of the time. In the worst case, the overhead in memory induced by using the best postorder instead of the best order is 18%.
>
> On random trees, the optimal traversal is not a postorder 60% of the time. The average and maximum overheads induced by using a postorder are 18% and 122%.

The experimental results from [54] along with the advantages cited above reinforce the intuition that following a postorder in (sequential) multifrontal codes is probably the best choice. In the following, we always consider that a postorder traversal in followed in the sequential case.

Before Jacquelin et al., Liu tackled the MinMemory problem in the context of the multifrontal method. Here we describe how he finds the best postorder. We show how he finds the best topological ordering in the next section, as this is related to *tree pebble games*. Liu firstly assumes that an in-place assembly scheme is used, and, at any node i in the tree, writes the sequential peak of active memory as

$$S_i = \max \quad \max_{j=1 \quad nc_i} \quad S_{s_{ij}} + \sum_{k=1}^{j-1} scb_{s_{ik}} \qquad sfront_i + \sum_{j=1}^{nc_i-1} scb_{s_{ij}}$$

$$= \max_{j=1 \quad nc_i} \quad \max \; S_{s_{ij}} \;\; sfront_i \quad + \sum_{j=1}^{nc_i-1} scb_{s_{ij}} \tag{7.1}$$

Clearly, by minimizing the peak at each node following a bottom-up traversal, the overall peak, i.e., the peak at the root node will be minimized. At a given node, minimizing the peak amounts to finding an ordering of its children that minimizes the above expression. The key result is the following:

Theorem 7.1 - Liu [68, Theorem 3.2].
Given a set of values $(x_i \;\; y_i)_{i=1 \quad n}$, the minimal value of

$$\max_{i=1 \quad n} \quad x_i + \sum_{j=1}^{i-1} y_j$$

is obtained by sorting the sequence $(x_i \;\; y_i)$ in decreasing order of $x_i \quad y_i$.

By applying this result to Equation (7.1), Liu finds the postorder that minimizes the peak of active memory. At every node i in the tree, the subtrees rooted at that node $(s_{ij} \quad j = 1 \qquad nc_i)$ have to be traversed by decreasing order of $\max(S_{s_{ij}} \;\; sfront_i) \quad scb_j$. This yields a reordering algorithm that we report in Algorithm 7.3.

With different assembly schemes (e.g. the classical scheme or the max in-place scheme), the technique used to find the best postordering is the same. We show in Table 7.1 the terms that play the role of x_i and y_i for every assembly scheme. We refer the reader to Table 3.1 and Table 3.2 in [60, Chapter 3] for a complete analysis of the minimization of the active memory and the total storage for all the possible assembly and allocation schemes; our table is an excerpt of Table 3.1.

7.2.2 The parallel case

The problem of minimizing the active memory in a parallel context has been little addressed. Firstly, we emphasize that the memory usage for a parallel execution might be

Algorithm 7.3 Reordering of the tree applying Theorem 7.1. At every node i, child subtrees are reordered by decreasing order of max($S_{s_{ij}}$ $sfront_i$) scb_j.

```
/* Input: a tree $T$                                              */
/*        $in\_order$, an arbitrary postordering of $T$           */
/* Output: $out\_order$, the new postordering of $T$              */

for all nodes $i$ in $T$, following $in\_order$ do
    if $i$ is a leaf node then
        $S_i = sfront_i$
    else
        Sort the children $s_{ij}$ by decreasing order of max($S_{s_{ij}}$ $sfront_i$) $scb_j$
        Compute $S_i$ using the new ordering of the children and 7.1
    end if
end for
Traverse the tree following a top-down dept-first search using the new ordering of the
children in order to get the final postorder
```

Assembly scheme	x_i	y_i
classical	S_i	scb_i
in-place	max(S_i $sfront_i$)	scb_i
max-in-place	S_i	scb_i

Table 7.1: Ordering of the children that minimizes the peak of active memory, assuming a terminal allocation, depending on the assembly scheme. The optimal ordering is found by applying Theorem 7.1 with the adequate x_i and y_i.

completely different from that of a sequential execution; therefore, the problem is not only about nicely distributing the amount of memory needed for a sequential execution among the different processes. A bad mapping and/or scheduling might yield a total amount of memory significantly higher than that of a sequential execution. We give a simplistic example in Figure 7.5; the tree is mapped onto two processes. We assume that the two leaf nodes are significantly larger than the two others (C) and that the size of their contribution block is negligible. The total amount of active memory for a sequential traversal is (approximately) $S_{seq} = C$. However, using a proportional mapping, the total amount of memory is $2C$ (assuming that the two leaf nodes are active at the same time, which is natural here). This demonstrates that the mapping and the scheduling strategies influence not only the balance of the memory usage but also the total consumption.

In a parallel setting, the objective depends on the type of system we consider:

In a shared-memory context, the objective is to minimize the total memory footprint. Unless one considers memory locality issues, balancing the memory usages of the different processes is less important.

In a distributed-memory context, minimizing the total memory consumption is desirable but it is also crucial to maintain a balanced memory usage between the processes, to avoid that a process runs out of memory (assuming that all the processes can access the same amount of memory, which is the most frequent setting).

Minimizing the total memory consumption in a parallel setting is complicated as it

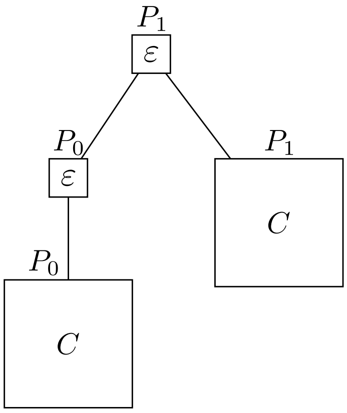


Figure 7.5: The tree is mapped onto two processes P_0 and P_1. We assume C and *nfront* C for the leaf nodes. The peak of active storage for a sequential traversal is C. However, the total amount of active memory in a parallel execution is $2C$.

is difficult to compute this total memory usage. Unless a scheduling with very specific properties is used, this would require to track at run time the memory usage of every process (so that, at any given time, we could sum the memory usages), which is almost infeasible, even in a shared-memory context. What is feasible though is to compute the maximum peak of active memory of every process. The sum of the peaks of the different processes provides an upper bound on the total consumption, that might be very loose depending on what processes do. For a parallel execution on p processes, we denote S_{max} and S_{avg} the maximum and average peaks of active memory (among the p processes) respectively. S_{avg} is computed as the sum of the p peaks divided by p; note that, with the above observation $p \; S_{avg}$ is an upper bound on the total consumption.

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

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

$e_{max}(p) = \dfrac{S_{seq}}{p \; S_{max}}$; this metric, combined with the other one, can detect the fact that one (or more) process uses too much memory, which is relevant in a distributed-memory context. For example, $e_{max}(p) = 0 \; 2$ means that at least one of the processes uses 5 times more memory than $\frac{S_{seq}}{p}$ which represents the ideal situation; indeed, the ideal situation is to have $S_{max} = \frac{S_{seq}}{p}$ (thus $e_{avg}(p) = e_{max}(p) = 1$) which means that the memory perfectly scales, i.e., the sequential peak is perfectly distributed among the processes.

We will show in the next chapter that the proportional mapping as well as the mapping used in MUMPS do not provide a good memory scalability. However, we mention a mapping technique that perfectly scales in memory. The idea is to map *all* the nodes of the tree on *all* the processes and to traverse the tree following the postorder used in the sequential case; this is what we call the *all-to-all mapping.* One easily sees that this technique does not exploit tree parallelism (all the branches are serialized), that it leads to dramatic amounts of communication (both within nodes when performing dense partial factorizations and between nodes when communicating contribution blocks), and that it does not deliver adequate granularities within nodes. Therefore, this is not feasible at all; we assess this in Chapter 11 where we show that using this technique, prohibitive amounts of communications (particularly in terms of numbers of messages) arise. However, ignoring practical implementation issues, this mapping technique clearly provides a perfect memory scalability ($e_{avg}(p) = e_{max}(p) = 1$). We will see in Chapter 9 that reusing the idea of an all-to-all mapping in some parts of the tree is a component of the mapping algorithm we suggest. This observation also implies that, in a parallel context, minimizing the active memory is not the right problem, since the solutions to this problem are likely to yield poor performance. We do not really want to minimize the active memory; we would rather *control* the active memory, i.e., to enforce a given memory constraint that would be provided by the user or defined by hardware specifications. Then, for this memory constraint, we would like to maximize the performance; this is exactly the aim of the mapping technique we describe in Chapter 9.

7.2.3 Relation to the tree pebble game

We conclude this chapter with an interesting formulation of our problem. A pebble game is a game played on an acyclic graph (here we consider only trees); the rules of the most simple variant are the following:

> Initially no vertices carry pebbles.
>
> A pebble may be placed on a input vertex (for us, a leaf) at any time.
>
> If all the predecessors (for us, the children) of an internal vertex carry pebbles, a new pebble may be placed on that vertex, or a pebble may be moved from one of its predecessors to that vertex.
>
> A pebble can be removed from the graph at any time.

The goal is to place pebbles on output vertices (for us, the root node). The objective can be to minimize the number of pebbles that are used (the *pebble cost*) or the number of time steps (turns). This game has applications in the VLSI community and in semantics [81]. Liu mentions in [68] that Gilbert pointed out the connection of the minimization of the sequential peak of active memory with the *tree pebble game*. Assuming an elimination tree where all the nodes have unitary weight (memory cost), i.e., $sfront = 1$, the pebble game works exactly as the traversal of an elimination tree. The pebbles represent units of memory. The rule that allows a pebble to move from a node to is parent is equivalent to an in-place assembly. Minimizing the number of pebbles amounts to minimizing memory requirements. Note that the rules induce a topological traversal of the tree, but not necessarily a postorder. Liu mentions that the traversal of the tree he suggests is actually a known optimal solution to the pebble game.

In our problem, traversing a node requires to use several units of memory. This suggests to use a generalized tree pebble game where every node i has a weight (i). Only one rule is modified: a pebble may be removed from a node i if there are (i) pebbles on it. In [69][5] Liu shows how minimizing the active memory in a left-looking sparse factorization can be formulated as a generalized tree pebble game. The elimination tree is transformed

[5] Note that is not the same reference as before; this publication is dedicated to the generalized tree pebble game.

using the following process: every node i is transformed into two nodes i^- and i^+ where i^+ is the only child of i^-. (i^+) is the number of nonzeros in columns of L associated with $\mathcal{T}(i)$ that are needed to compute column i; (i^-) is the number of nonzeros in columns of L associated with $\mathcal{T}(i)$ that are still needed after i is processed. Going from i^+ to i^- represents the fact that a row is written on disk and can be removed from the main memory.

Liu then describes an algorithm that finds the optimal solution to the generalized tree pebble game. The idea is to work recursively at each subtree; for a given node, the best ordering is obtained by merging the best orderings of its children subtrees. This merging process is the most difficult part of the algorithm. Firstly, Liu shows that the pebble cost is not a relevant metric; some orderings of a given subtree with the same pebble cost might lead to different results when they are merged with the ordering of another subtree. Liu introduces a new order relation between traversals and deduces the merging. Coarsely, in order to merge the orderings for two subtrees rooted at two siblings, the idea is to find a way to interleave these two orderings by detecting the nodes that yield local maxima and minima in the pebbling sequences (Liu calls them hills and valleys respectively). The idea is to take advantage of the valleys of a subtree to traverse the hills of the other one; we provide a very simple example in Figure 7.6, that also shows in which situations a postorder may be the best traversal. In the figure, one can see that any of the two postorderings needs at least 8 pebbles (since a node with $= 5$ and a node with $= 3$ need to be pebbled at some point), while the traversal 1-2-4-5-3-6-7 requires only 6 pebbles. The key idea is to stop one of the branch at a minimum in the pebble sequence (1 here) in order to start and traverse completely the other branch. By stopping the traversal of the left-hand branch at node 2 before traversing the right-hand branch, only one pebble is stacked instead of three for a postorder traversal.

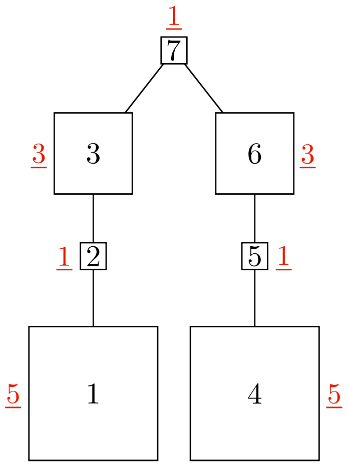


Figure 7.6: An example of generalized tree pebble game; the weight of each node is shown next to the node (and underlined). Any postorder traversal requires to use 8 pebbles, while the ordering 1-2-4-5-3-6-7 has a pebble cost of 6.

The algorithm proposed by Liu provides the optimal solution. Jacquelin et al. showed that their algorithm MinMem finds the same traversal and has the same worst-case complexity; however their algorithm is about three times faster in practice (they mention that Liu's algorithm is slow because it relies on a slow list merging process).

Finally we briefly describe how the parallel problem could be formulated as a generalized tree pebble game. We suggest to use pebbles with different colors, with color for each process. A pebble would thus represent a unit of memory *for a given process.* Using the same rules as in the game described above, we immediately have a game that represents the problem of the minimization of the active memory in a parallel context. As mentioned above, this is not exactly our problem: we are interested in finding a traversal of the tree that maximizes the performance under a given memory constraint. A memory constraint can be represented simply by limiting the number of pebbles of each color. We then need to represent the performance. Remember that one of the variants of the pebble game consists of finding the minimum number of turns (pebbles moves or placements) in the game, instead of finding the minimum number of pebbles. The number of turns can serve as a representation of the time for traversing the tree. Rules need to be added in order to have a good model of parallel performance; we give a few hints:

Several pebbles can be placed on a node at the same time; however, several pebbles of the same color cannot be placed on different nodes at the same time. This represents the fact that a process cannot work on different tasks at the same time; however tasks are divisible and a process can interleave fractions of different tasks.

If a node is pebbled with too many colors (the threshold needs to be defined), there is a penalty in the number of turns; this penalizes intra-node communication.

Similarly, a rule could be used to favour traversals that enforce a subtree-to-subcube principle (e.g. relying on the number of colors used for a given subtree).

We have not pushed this formulation further but we believe that it can be interesting. We also believe that some of the ideas from the study by Liu (more specifically the idea of hills and valleys) could be interesting for our parallel problem, but we have not explored this option.

Chapter 8

Memory scalability issues

In this chapter we show that commonly used mapping techniques do not achieve a good memory scalability. We start with a very simple but realistic example where we analyze step-by-step, on a given tree, the behavior of the proportional mapping with respect to the active memory. In this example, after two steps of proportional mapping, i.e., once the grandchildren of the root node are mapped, the memory efficiency is bounded by 0 16, which is unacceptably low. We then present a theoretical analysis of the memory scalability of the proportional mapping on 2D regular grids with nested dissection. We show that the memory efficiency tends rapidly to zero when the number of processes increases. Finally, we report on some experimental results with the MUMPS solver, using the default mapping strategy. We show that the memory efficiency typically lies between 0 10 and 0 40.

These theoretical and practical results show that the proportional mapping and the variant used in MUMPS tend to let the memory usage grow dangerously with the number of processes. This demonstrates that there is a strong need for mapping strategies aiming at controlling the active memory.

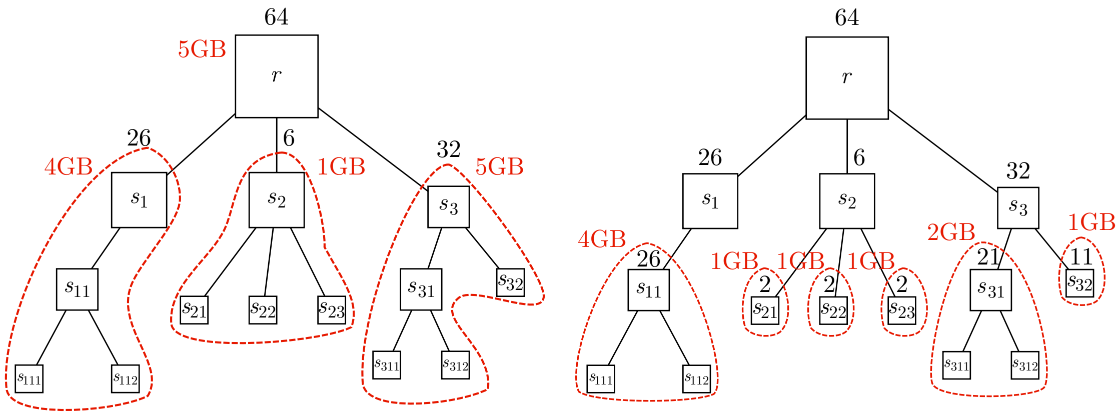
8.1 A simple example

In this simple example, we illustrate the behavior of the proportional mapping on a simple yet realistic elimination tree. We consider a memory-based proportional mapping strategy, but we could make the same observations about a workload-based strategy. We consider the tree in Figure 8.1 and assume that it will be mapped onto 64 processes. Firstly, these 64 processes are assigned to the root node r. Then, a first step of memory-based proportional mapping is used to distribute these 64 processes among the three children of r: s_1, s_2 and s_3, as illustrated in Figure 8.1(a). The sequential peaks of active memory of the subtrees rooted at s_1, s_2 and s_3 are 4 GB, 1 GB and 5 GB respectively. Therefore, s_1 gets $\frac{4}{4+1+5}$ 64   26 processes; s_2 gets $\frac{1}{4+1+5}$ 64   6 processes and s_3 gets $\frac{5}{4+1+5}$ 64   32 processes. Note that we have chosen to round the numbers of processes so that they add to 64 and are distributed without any overlap; a relaxed technique could be used but this would not change the result. At this stage, we can compute an upper bound of the peak of active memory of each process. Consider the 26 processes working on the subtree rooted at s_1. The sequential peak of active memory of this subtree is 4 GB, therefore, at best, that is, assuming a perfect memory scalability can be attained for this subtree, the maximum peak of active memory among these 26 processes will be $\frac{4\text{ GB}}{26} = 0\ 16$ GB. Similarly, the maximum peaks of active memory for the 6 processes working on the subtree rooted at s_2 and the 32 processes working on the subtree rooted at s_3 are lower bounded by $\frac{1\text{ GB}}{6}$

and $\frac{5\text{ GB}}{32}$ respectively; ignoring rounding errors, all these numbers are the same (0.16 GB) since we have applied a memory-based proportional mapping. We can now derive a first lower bound on the memory efficiency for this problem; since the sequential peak of active memory for the whole tree is 5 GB, the memory efficiency is bounded as follows:

$$e(p) = \frac{S_{seq}}{p \quad S_{max}(p)} \leqslant \frac{5\text{ GB}}{64 \quad 0\ 16} \leqslant 0\ 5$$

It is indeed fairly easy to see why the efficiency is bounded by $\frac{1}{2}$. The sequential peaks of the whole tree and the subtree rooted at s_3 are the same[1]; however, only half the processes work on the latter subtree. Therefore, even if a perfect memory scalability is attained on the subtree rooted at s_3, the memory usage for the 32 processes working on that subtree is twice what we are targeting ($\frac{5\text{ GB}}{32}$ instead of $\frac{5\text{ GB}}{64}$).



(a) First step of proportional mapping.

(b) Second step of proportional mapping.

Figure 8.1: A two-step example of memory-based proportional mapping. At each node, we indicate the sequential peak of active memory (in GB) and the number of processes assigned by the proportional mapping. 64 processes are assigned to the root node r; after one step of proportional mapping, they are distributed among the children of the root node, s_1 s_2 s_3 (a). Then another step of proportional mapping assigns processes to the grandchildren of the root node (b).

Now we consider a second step of proportional mapping, where the mapping of the grandchildren of r is computed. s_{11} is the only child of s_1 and thus inherits the 26 processes s_1 is mapped onto. s_2 has three children with the same sequential peak of active memory, thus the processes mapped onto s_2 are equally split, and every child inherits from $\frac{6}{3} = 2$ processes. s_3 is mapped onto 32 processes and has two children, s_{31} and s_{32}; their sequential peaks are 2 GB and 1 GB respectively. Therefore s_{31} is mapped onto $\frac{2}{3} \quad 32 \qquad 21$ processes, and s_{32} is mapped onto $\frac{1}{3} \quad 32 \qquad 11$ processes. Now we consider memory efficiency and follow the same reasoning as above. Assume we can obtain a perfect scalability on the six subtrees rooted at the grandchildren of r; the maximum peak, among the 64 processes, is at best max $\quad \frac{4\text{ GB}}{26} \ \frac{1\text{ GB}}{2} \ \frac{2\text{ GB}}{21} \ \frac{1\text{ GB}}{11} \quad = 0\ 5$ GB. In terms of efficiency, it yields

$$e(p) = \frac{S_{seq}}{p \quad S_{max}(p)} \leqslant \frac{5\text{ GB}}{64 \quad 0\ 5} \leqslant 0\ 16$$

[1] Actually they cannot be exactly the same since the peak at r is at least the peak at s_3 plus the contribution blocks of $s1$ and $s2$; here we assume for simplicity that these contribution blocks are negligible.

This time the bad scalability is due to the three subtrees rooted at the children of s_2. Their sequential peak is 1 GB, which represents one fifth of the sequential peak of the complete tree; however they receive a much smaller fraction of the 64 processes since they get only 2 processes. Therefore the memory efficiency is upper bounded by $5 \quad \frac{2}{64} \quad 0\ 16$.

This example is very simplistic but it illustrates what happens in reality. In our example, the subtree rooted at s_3 can be seen as a memory attractor ; its sequential peak is the same as the sequential peak of the whole tree, which means that the sequential peak of the whole tree actually takes place in this subtree. However, since the proportional mapping partitions the list of processes and distributes them among s_3 and its siblings, the memory efficiency necessarily decreases. At this stage of the mapping process, the only way to guarantee a perfect memory scalability is to assign the 64 processes to the subtree rooted at s_3. Of course, this prevents using a strategy à la proportional mapping since there is thus no way to assign disjoint sets of processes to siblings. This observation is one of the motivations for the strategies presented in the next chapter. Note that a workload-based proportional mapping (which is a more common setting) would exhibit the same behavior: indeed, unless the workload of the subtree rooted at s_3 is extremely large compared to the workloads of the two other subtrees (which is unlikely since the peaks of active memory are comparable), s_3 will not inherit from all or almost all the processes, yielding once again a low memory scalability.

8.2 Theoretical study on regular grids

8.2.1 Main result

We now extend the previous example to more general trees. A first step towards a more general result can be found in Agullo's PhD thesis [1, Chapter 10]. Agullo quantifies the sub-optimality of the proportional mapping on perfect binary trees with fronts of the same size, where the fully-summed blocks and contribution blocks have the same number of variables; he shows that in this particular case the following relation holds

$$e(p) = O\left(\frac{\log(p)}{p}\right)$$

We extend this result to elimination trees corresponding to a nested dissection of a regular 2D grid; we show the following result:

Theorem 8.1 - Sub-optimality of the proportional mapping on a 2D regular grid.
Let T be an elimination tree corresponding to a nested dissection of a regular 2D square grid with a nine-point stencil. For a given number of processes p, the memory efficiency of a strict memory-based proportional mapping of T is

$$e_{max}(p) = O(C \quad p \quad) \quad \text{(with } C \quad 0\ 18 \qquad 0\ 2)$$

This result is interesting since it quantifies what was surmised in the simple example presented above. However, we warn the reader that the proof is rather tedious and does not bring much information about our problem; it is presented in the next two subsections for the sake of completeness but can be skipped without loss of comprehension. At the end of the proof, we provide some simulation results that confirm our quantification. In the last section of the chapter, we report on experiments results using MUMPS that also confirm our theoretical result.

Note that we have investigated only the 2D case. The same theoretical study could be carried out for the 3D case. We believe that a similar conclusion would be reached.

8.2.2 Proof context

Our proof is based on the nested dissection model introduced by George in his original article [41]. The proof is as follows:

1. We recall the main ingredients of the nested dissection model described in George's paper, and we give a thorough description of the structure of the elimination tree associated with the sparse matrix that corresponds to the grid. Note that for the sake of simplicity we consider that an LU factorization is used (the computation of the memory usage is a bit easier than in the symmetric case); this does not change our reasoning nor the final result.

2. We compute the sequential peak of active memory associated with a multifrontal factorization of that matrix.

3. We describe how the elimination tree is mapped using a memory-based proportional mapping strategy with a given number of processes p.

4. We compute the maximum peak of active memory among the p processes and deduce the memory efficiency.

Firstly, we recall the nested dissection model and describe the structure of the elimination tree. The 2D regular grid is as follows. We consider $(n+1)^2$ nodes (square cells) on a square mesh. Nodes have coordinates $(i \quad h \ j \quad h)$, for $0 \leqslant i \ j \leqslant n$ where h is a geometrical length (not used in the following). We assume $n = 2^l$ for simplicity. Each node is connected to its eight closest neighbors (i.e., we take into account diagonal neighbors); the 5-point stencil can be derived easily. The main idea is to define geometrical separators that have a + shape, so that they recursively divide the domain into four parts. The elimination of a separator is as follows. Two branches or spokes (denoted (1) and (2) thereafter) of each + -shaped separator are eliminated independently and then the remaining part (denoted (3)) is eliminated, so that, in the end, the elimination process yields a binary tree where each + -shaped separator represents three nodes.

Separators of the domain are described via a function defined as[2]:

$$(0) = 0$$

$$(n) = 0$$

$$i \quad 1 \quad n \quad 1 \quad (i) = p + 1 \text{ with } p = \max \ k : mod(i \ 2^k) = 0 \ )$$

Then the level sets of separators are defined as:

$$Sep_k = \quad (i \ j) : \max(\ (i) \quad (j)) = k$$

We illustrate these level sets in Figure 8.2 for $n = 16$. One can notice that they indeed have + shapes; near the border the separators may have spokes with different lengths. We show in Figure 8.3 the tree of separators of the domain; note that this is not the same as the elimination tree, as shown in the following.

Since max $= l$ (remember that $n = 2^l$), there are $l+1$ levels of separators. Variables are eliminated from Sep_0 to Sep_l. Each level set Sep_k has $(n/2^k)^2$ + -shaped sets of unknowns (except Sep_0 that has only 4), which are independent from one another; thus they can be eliminated in any order. Within each set, the strategy chosen in [41] is to

[2] We use a slightly different definition than that from [41], but the tree is the same in the end.

0	1	2	1	3	1	2	1	4	1	2	1	3	1	2	1	0
1	1	2	1	3	1	2	1	4	1	2	1	3	1	2	1	1
2	2	2	2	3	2	2	2	4	2	2	2	3	2	2	2	2
1	1	2	1	3	1	2	1	4	1	2	1	3	1	2	1	1
3	3	3	3	3	3	3	3	4	3	3	3	3	3	3	3	3
1	1	2	1	3	1	2	1	4	1	2	1	3	1	2	1	1
2	2	2	2	3	2	2	2	4	2	2	2	3	2	2	2	2
1	1	2	1	3	1	2	1	4	1	2	1	3	1	2	1	1
4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
1	1	2	1	3	1	2	1	4	1	2	1	3	1	2	1	1
2	2	2	2	3	2	2	2	4	2	2	2	3	2	2	2	2
1	1	2	1	3	1	2	1	4	1	2	1	3	1	2	1	1
3	3	3	3	3	3	3	3	4	3	3	3	3	3	3	3	3
1	1	2	1	3	1	2	1	4	1	2	1	3	1	2	1	1
2	2	2	2	3	2	2	2	4	2	2	2	3	2	2	2	2
1	1	2	1	3	1	2	1	4	1	2	1	3	1	2	1	1
0	1	2	1	3	1	2	1	4	1	2	1	3	1	2	1	0

Figure 8.2: Level sets of separators for n =16. Nodes with the same number k belong to the same set of separators Sep_k. Each set of separators Sep_k consists of several + -shaped separators.

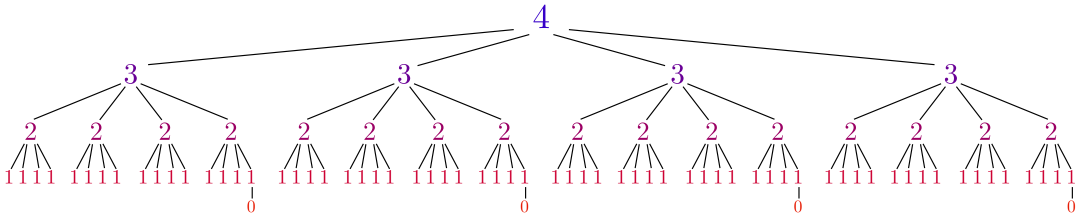


Figure 8.3: Tree of separators for n = 16. Every separator splits the domain into four disconnected subdomains.

follow a minimum-degree ordering, i.e., at each step the spoke that generates the least fill-in is chosen to be eliminated.

For each separator, we describe the structure (i.e., the number of fully summed variables and the size of the contribution block) of (1), (2), and (3) in the filled graph. For $0 < k < l$, each Sep_k has 3 type of sets:

There are $\left(\frac{n}{2^k} \quad 2\right)^2$ interior sets . Their structure is illustrated in Figure 8.4(a):

(1) has $2^{k-1} \quad 1$ unknowns, all connected to the same $6(2^{k-1} \quad 1) + 6 = 3 \quad 2^k$ unknowns.

(2) is similar to (1).

(3) has $2^k \quad 1$ unknowns, all connected to the same $8(2^{k-1} \quad 1)+8 = 2^{k+2}$ unknowns.

There are $4\left(\frac{n}{2^k} \quad 2\right)$ boundary sets . Their structure is illustrated in Figure 8.4(b):

(1) has 2^{k-1} unknowns, all connected to the same $4(2^{k-1} \quad 1) + 5 = 2^{k+1} + 1$ unknowns.

(2) has $2^{k-1} \quad 1$ unknowns, all connected to the same $6(2^{k-1} \quad 1) + 6 = 3 \quad 2^k$ unknowns.

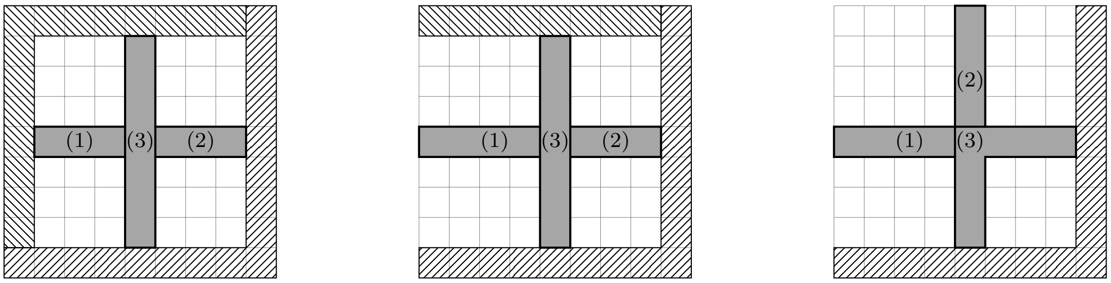
(3) has $2^k \quad 1$ unknowns, all connected to the same $6(2^{k-1} \quad 1) + 7 = 3 \quad 2^k + 1$ unknowns.

There are 4 corner sets . Their structure is illustrated in Figure 8.4(c):

(1) has 2^{k-1} unknowns, all connected to the same $3(2^{k-1} \quad 1) + 4 = 3 \quad 2^{k-1} + 1$ unknowns.

(2) has 2^{k-1} unknowns, all connected to the same $4(2^{k-1} \quad 1) + 5 = 2^{k+1} + 1$ unknowns.

(3) has $2^k \quad 1$ unknowns, all connected to the same $4(2^{k-1} \quad 1) + 5 = 2^{k+1} + 1$ unknowns.



(a) Interior sets. (b) Boundary sets. (c) Corner sets.

Figure 8.4: Structure of the three types of separators in the nested dissection model. Note that *every square in the figure is a node in the grid*. In each figure, every + -shaped separator has three parts: (1), (2) and (3) (in gray). Empty (white) squares correspond to nodes that descendants of the separator in the tree. Striped squares are spokes of separators that are ancestors of the separator we consider; in general, a + -shaped separator touches two other separators.

Consider for example the level set Sep_2 in Figure 8.2, i.e., the sets of nodes labeled with a 2: there are 4 interior sets, near the center of the grid, with a perfectly symmetric shape. There are 8 border sets; they touch one side of the square grid and the spoke that touches the border of the grid is longer than the 3 others. There are 4 corner sets; they touch two sides of the grid and the spokes that touch the border of the grid are longer than the 2 others.

Finally Sep_0 and Sep_l are two special cases:

Sep_0 has four sets consisting of a single variable connected to 3 variables.

Sep_l has a unique + -shaped set: its first two spokes have $n/2$ variables, connected to $n + 1$ variables, and the remaining part (root of the tree) has $n + 1$ variables.

We now describe the structure of the tree. The root node of the tree corresponds to part (3) of the unique separator of Sep_l:

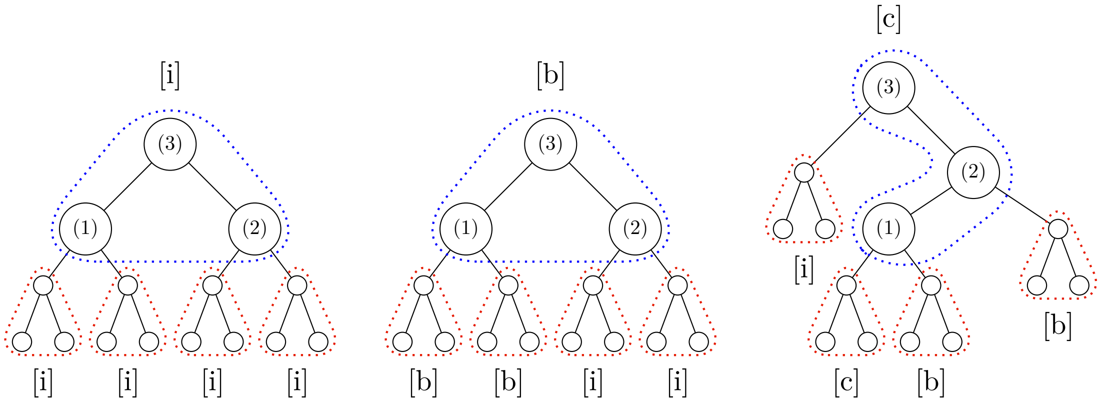
It has two children, corresponding to parts (1) and (2) of Sep_l.

Both children have two children, each of them corresponding to one of the 4 corner sets of Sep_{l-1}.

The rules that recursively define the structure of the tree are described in Figure 8.5; we denote interior sets with an [i], boundary sets with a [b] and corner sets with a [c]. We show, for each kind of separator:

How the nodes corresponding to (1), (2) and (3) are connected.

The kind of separators corresponding to the children nodes of (1), (2) and (3).



(a) Each interior set of Sep_k is the parent of 4 interior sets in Sep_{k-1}.

(b) Each boundary set of Sep_k is the parent of 2 boundary sets and 2 interior sets in Sep_{k-1}.

(c) Each corner set of Sep_k is the parent of 1 corner, 2 boundary sets and 1 interior set in Sep_{k-1}.

Figure 8.5: Structure of the subtrees associated with each kind of separator.

We provide a few more details for the sake of completeness:

In an interior subtree rooted at Sep_k, there are:

$2 \quad 4^{k-1} \quad 1$ nodes.

$(2^k \quad 1)^2$ eliminated variables.

In a boundary subtree rooted at Sep_k, there are:

$2 \quad 4^{k-1} \quad 1 + 2^{k-1}$ nodes.

$2^k \quad (2^k \quad 1)$ eliminated variables.

In a corner subtree rooted at Sep_k, there are:

$\frac{1}{2}4^k + 2^k$ nodes.

$(2^k)^2$ eliminated variables.

In the whole tree, there are:

$\frac{n^2}{2} + 2 \quad n + 3$ nodes.

$(n+1)^2$ eliminated variables.

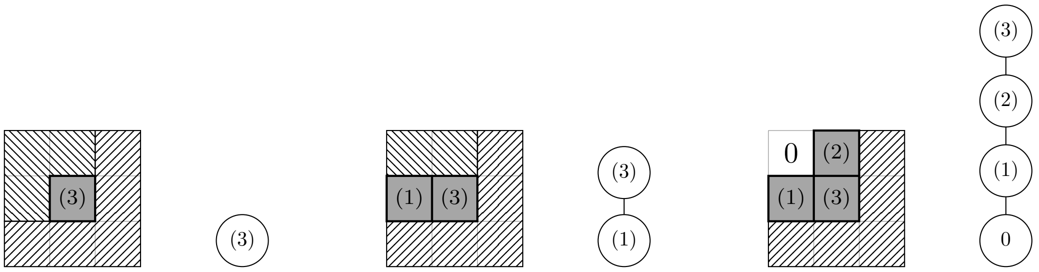
8.2.3 Proof – computation of the memory efficiency

Now that we have fully described the structure of the elimination tree, we can compute the memory efficiency. Firstly, we compute the sequential peak of active memory.

Lemma 8.1 - Sequential peak of active memory for a 2D regular grid.
The sequential peak of active memory is reached when assembling the root node of the last corner subtree (corresponding to level $l - 1$) and its value is $6.25 \cdot n^2 + 6 \cdot n + 3$.

Proof. We prove by induction on k (from 0 to $l - 1$, i.e., traversing the tree bottom-up) that for each one of the three types of subtree defined above, the peak of active memory is reached when assembling the root node. The final step consists of examining what happens at the top of the tree (Sep_l and Sep_{l-1}). We reuse the notation introduced in Section 7.2.

Basis: for $k = 1$, the proof is trivial and shown in Figure 8.6.



(a) Interior sets: the peak is obtained for (3) and represents $(1+8)^2 = 81$ reals.

(b) Boundary sets: the peak is obtained for the assembly of (3): $(1+7)^2 + 5^2 = 89$.

(c) Corner sets: the peak is obtained for the assembly of (3): $(1+5)^2 + 5^2 = 61$ (note the presence of Sep_0).

Figure 8.6: Computation of the sequential peak of active memory for $k = 1$.

Inductive step: we suppose that, at level Sep_k ($1 \leqslant k < l$), the peak of active memory of every subtree rooted at Sep_k is obtained when allocating the root node, and we prove that this is also the case for level Sep_{k+1}. At every node, the peak of active memory is expressed as a second order polynomial of 2^k; the comparisons needed to compute a "max" are done computing the roots of the polynomial, but we do not provide all the details. Sometimes we need to determine which ordering of the children nodes minimizes the peak; when the way the children are ordered has no influence on the peak, we indicate which ordering gives the lowest average memory consumption.

For each set of separators Sep_{k+1}, we consider the three kinds:

Interior sets:

1. Below nodes (1) and (2) are subtrees corresponding to interior sets of Sep_k. Their peak is reached when allocating their root node (induction hypothesis) and is (allocation of the root node + contribution blocks of its children):
$$\begin{aligned} &\left(2^k - 1 + 2^{k+2}\right)^2 + 2 \cdot \left(3 \cdot 2^k\right)^2 \\ = &\,43 \cdot 2^{2k} - 10 \cdot 2^k + 1 \end{aligned}$$
2. The peak of active memory for the subtree rooted at (1) (or (2) similarly) is (we do not take the ordering of the children into account since their subtrees

are the same):

$$\begin{aligned}
&\max \quad \mathit{sfront}_{(1)} + \mathit{scb}_{s_{(1),1}} + \mathit{scb}_{s_{(1),2}} \quad S_{s_{(1),2}} + \mathit{scb}_{s_{(1),1}} \\
=&\max \Big((2^k \quad 1 + 3 \quad 2^{k+1})^2 + 2 \quad (2^{k+2})^2 \quad 43 \quad 2^{2k} \quad 10 \quad 2^k + 1 + \quad 2^{k+2} \quad ^2 \Big) \\
=&\max \quad 81 \quad 2^{2k} \quad 14 \quad 2^k + 1 \quad 59 \quad 2^{2k} \quad 10 \quad 2^k + 1 \\
=&\,81 \quad 2^{2k} \quad 14 \quad 2^k + 1 \text{ (i.e., assembly of (1)/(2))}
\end{aligned}$$

3. Finally, at node (3), the peak is (we do not take the ordering of the children into account since their subtrees are the same):

$$\begin{aligned}
&\max \quad \mathit{sfront}_{(3)} + \mathit{scb}_{(1)} + \mathit{scb}_{(2)} \quad S_{(2)} + \mathit{scb}_{(1)} \\
=&\max \Big(\quad 2^{k+1} \quad 1 + 2^{k+3} \quad ^2 + 2 \quad 3 \quad 2^{k+1} \quad ^2 \quad 81 \quad 2^{2k} \quad 14 \quad 2^k + 1 + \quad 3 \quad 2^{k+1} \quad ^2 \Big) \\
=&\max \quad 172 \quad 2^{2k} \quad 20 \quad 2^k + 1 \quad 117 \quad 2^{2k} \quad 14 \quad 2^k + 1 \\
=&\,172 \quad 2^{2k} \quad 20 \quad 2^k + 1 \text{ (i.e., assembly of (3))}
\end{aligned}$$

Therefore, the peak of active memory is obtained when allocating the root node (i.e., (3)).

Boundary sets:

1. Below node (1) are two subtrees corresponding to boundary sets of Sep_k. Their peak is reached when allocating their root node (induction hypothesis) and is (allocation of the root node + contribution blocks of its children):

$$\begin{aligned}
&2^k \quad 1 + 3 \quad 2^k + 1 \quad ^2 + \quad 2^{k+1} + 1 \quad ^2 + \quad 3 \quad 2^k \quad ^2 \\
=&\,29 \quad 2^{2k} + 4 \quad 2^k + 1
\end{aligned}$$

2. The peak of active memory for the subtree rooted at (1) is (we do not take the ordering of the children into account since their subtrees are the same):

$$\begin{aligned}
&\max \quad \mathit{sfront}_{(1)} + \mathit{scb}_{s_{(1),1}} + \mathit{scb}_{s_{(1),2}} \quad S_{s_{(1),2}} + \mathit{scb}_{s_{(1),1}} \\
=&\max \Big(\quad 2^k + 2^{k+2} + 1 \quad ^2 + 2 \quad (3 \quad 2^k + 1)^2 \quad 29 \quad 2^{2k} + 4 \quad 2^k + 1 + \quad 3 \quad 2^k + 1 \quad ^2 \Big) \\
=&\max \quad 43 \quad 2^{2k} + 22 \quad 2^k + 3 \quad 38 \quad 2^{2k} + 10 \quad 2^k + 2 \\
=&\,43 \quad 2^{2k} + 22 \quad 2^k + 3 \text{ (i.e., assembly of (1))}
\end{aligned}$$

3. Below node (2) are two subtrees corresponding to interior sets of Sep_k. Their peak is reached when allocating their root node (induction hypothesis) and is (allocation of the root node + contribution blocks of its children):

$$\begin{aligned}
&2^k \quad 1 + 2^{k+2} \quad ^2 + 2 \quad 3 \quad 2^k \quad ^2 \\
=&\,43 \quad 2^{2k} \quad 10 \quad 2^k + 1
\end{aligned}$$

4. The peak of active memory for the subtree rooted at (2) is (we do not take the ordering of the children into account since their subtrees are the same):

$$
\begin{aligned}
&\max\;\; \mathit{sfront}_{(2)} + \mathit{scb}_{s_{(2),1}} + \mathit{scb}_{s_{(2),2}}\;\; S_{s_{(2),1}} + \mathit{scb}_{s_{(2),2}} \\
=&\max\Big(\;\; 2^k \quad 1+3\;\; 2^{k+1}\;\;{}^{2} + 2\;\; (2^{k+2})^2\;\; 43\;\; 2^{2k} \quad 10\;\; 2^k+1+\;\; 2^{k+2}\;\;{}^{2}\Big) \\
=&\max\;\; 81\;\; 2^{2k} \quad 14\;\; 2^k+1\;\; 59\;\; 2^{2k} \quad 10\;\; 2^k+2 \\
=&\,81\;\; 2^{2k} \quad 14\;\; 2^k+1 \text{ (i.e., assembly of (2))}
\end{aligned}
$$

5. Finally, at node (3), the peak is (this time the ordering of the children matters, so we consider both possibilities):

$$
\begin{aligned}
&\max\;\; \mathit{sfront}_{(3)} + \mathit{scb}_{(1)} + \mathit{scb}_{(2)}\;\; S_{(1)} + \mathit{scb}_{(2)}\;\; S_{(2)} + \mathit{scb}_{(1)} \\
& \qquad 2^{k+1} \quad 1+3\;\; 2^{k+1}+1\;\;{}^{2} + \;\; 2^{k+2}+1\;\;{}^{2} + \;\; 3\;\; 2^{k+1}\;\;{}^{2} \\
=&\max \qquad 43\;\; 2^{2k}+22\;\; 2^k+3+\;\; 3\;\; 2^{k+1}\;\;{}^{2} \\
& \qquad 81\;\; 2^{2k} \quad 14\;\; 2^k+1+\;\; 2^{k+2}+1\;\;{}^{2} \\
=&\max\;\; 116\;\; 2^{2k}+8\;\; 2^k+1\;\; 79\;\; 2^k+22\;\; 2^k+3\;\; 97\;\; 2^{2k} \quad 6\;\; 2^k+2 \\
=&\,116\;\; 2^{2k}+8\;\; 2^k+1 \text{ (i.e., assembly of (3))}
\end{aligned}
$$

Therefore, the peak of active memory is obtained when allocating the root node (i.e., (3)). An interesting point is that the peak is independent of the ordering of the children of (3). However, the average memory consumption can be decreased by processing (2) before (1) (because $S_{(1)} + \mathit{scb}_{(2)}$ is smaller than $S_{(2)} + \mathit{scb}_{(1)}$).

Corner sets:

1. Below node (1) are two subtrees corresponding to a corner set and a boundary set of Sep_k. Their peak is reached when allocating their root node (induction hypothesis) and is (allocation of the root node + contribution blocks of its children[3]):

$$
\begin{aligned}
\text{corner subtree:}\quad & 2^k \quad 1+2^{k+1}+1\;\;{}^{2} + \;\; 2^{k+1}\;\;{}^{2} + \;\; 2^{k+1}+1\;\;{}^{2} \\
=&\; 17\;\; 2^{2k}+4\;\; 2^k+1 \\
\text{boundary subtree:}\quad & 2^k \quad 1+3\;\; 2^k+1\;\;{}^{2} + \;\; 2^{k+1}+1\;\;{}^{2} + \;\; 3\;\; 2^k\;\;{}^{2} \\
=&\; 29\;\; 2^{2k}+4\;\; 2^k+1
\end{aligned}
$$

[3] Note that for the corner set, the interior child of the root node belongs to Sep_{k-1}; for $k=1$, it is null.

2. The peak of active memory for the subtree rooted at (1) is:

$$\begin{aligned}
&\max \quad \mathit{sfront}_{(1)} + \mathit{scb}_{s_{(1),1}} + \mathit{scb}_{s_{(1),2}} \;\; S_{s_{(1),1}} + \mathit{scb}_{s_{(1),2}} \;\; S_{s_{(1),2}} + \mathit{scb}_{s_{(1),1}} \\
&\qquad\quad 2^k + 3 \;\; 2^k + 1 \;\;^2 + \;\; 2^{k+1} + 1 \;\;^2 + \;\; 3 \;\; 2^k + 1 \;\;^2 \\
= &\max \quad 17 \;\; 2^{2k} + 4 \;\; 2^k + 1 + \;\; 3 \;\; 2^k + 1 \;\;^2 \\
&\qquad\quad 29 \;\; 2^{2k} + 4 \;\; 2^k + 1 + \;\; 2^{k+1} + 1 \;\;^2 \\
= &\max \quad 29 \;\; 2^{2k} + 18 \;\; 2^k + 3 \;\; 26 \;\; 2^k + 10 \;\; 2^k + 2 \;\; 33 \;\; 2^{2k} + 8 \;\; 2^k + 2 \\
= &\quad \begin{array}{l} \text{for } k = 1 \;\; 29 \;\; 2^{2k} + 18 \;\; 2^k + 3 \\ \text{for } k \geqslant 2 \;\; 33 \;\; 2^{2k} + \;\; 8 \;\; 2^k + 2 \end{array}
\end{aligned}$$

Here, the peak (not only the average consumption) is dependant of the ordering of the children. The best traversal processes the boundary subtree then the corner subtree; in that case, we get rid of the term $33 \;\; 2^{2k} + 8 \;\; 2^k + 2$ and the peak is reached when allocating (1):

$$29 \;\; 2^{2k} + 18 \;\; 2^k + 3$$

3. The peak of active memory for the subtree rooted at (2) is:

$$\begin{aligned}
&\max \quad \mathit{sfront}_{(2)} + \mathit{scb}_{(1)} + \mathit{scb}_{s_{(2),2}} \;\; S_{(1)} + \mathit{scb}_{s_{(2),2}} \;\; S_{s_{(2),2}} + \mathit{scb}_{(1)} \\
&\qquad\quad 2^k + 2^{k+2} + 1 \;\;^2 + \;\; 3 \;\; 2^k + 1 \;\;^2 + \;\; 3 \;\; 2^k + 1 \;\;^2 \\
= &\max \quad 29 \;\; 2^{2k} + 18 \;\; 2^k + 3 + \;\; 3 \;\; 2^k + 1 \;\;^2 \\
&\qquad\quad 29 \;\; 2^{2k} + 4 \;\; 2^k + 1 + \;\; 3 \;\; 2^k + 1 \;\;^2 \\
= &\max \quad 43 \;\; 2^{2k} + 22 \;\; 2^k + 3 \;\; 38 \;\; 2^k + 24 \;\; 2^k + 4 \;\; 38 \;\; 2^{2k} + 10 \;\; 2^k + 2 \\
= &\, 43 \;\; 2^{2k} + 22 \;\; 2^k + 3 \text{ (i.e., assembly of (2))}
\end{aligned}$$

Once again, the peak is reached when allocating the root node, independently of the ordering of the children. In order to minimize the average consumption, (1) can be processed before the other child. It is worth noticing that the ordering of the children of (1), that has an influence on the peak of the subtree rooted at (1), does not change anything here (with the other ordering, $38 \;\; 2^k + 24 \;\; 2^k + 4$ becomes $42 \;\; 2^k + 14 \;\; 2^k + 3$, which is still smaller than $43 \;\; 2^{2k} + 22 \;\; 2^k + 3$).

4. Finally, at node (3), the peak is:

$$\begin{aligned}
&\max \quad \mathit{sfront}_{(3)} + \mathit{scb}_{s_{(3),1}} + \mathit{scb}_{(2)} \;\; S_{s_{(3),1}} + \mathit{scb}_{(2)} \;\; S_{(2)} + \mathit{scb}_{s_{(3),1}} \\
&\qquad\quad 2^{k+1} \quad 1 + 2^{k+2} + 1 \;\;^2 + \;\; 2^{k+2} \;\;^2 + \;\; 2^{k+2} + 1 \;\;^2 \\
= &\max \quad 43 \;\; 2^{2k} \quad 10 \;\; 2^k + 1 + \;\; 2^{k+2} + 1 \;\;^2 \\
&\qquad\quad 43 \;\; 2^{2k} + 22 \;\; 2^k + 3 + \;\; 2^{k+2} \;\;^2 \\
= &\max \quad 68 \;\; 2^{2k} + 8 \;\; 2^k + 1 \;\; 59 \;\; 2^k \quad 2 \;\; 2^k + 2 \;\; 59 \;\; 2^{2k} + 22 \;\; 2^k + 3 \\
= &\, 68 \;\; 2^{2k} + 8 \;\; 2^k + 1 \text{ (i.e., assembly of (3))}
\end{aligned}$$

Once again, the peak is independent of the ordering of the nodes. However, the average consumption is minimized by processing (2) before the other subtree (interior set).

We showed that for all $1 \leqslant k < l$, the peak of a tree rooted at Sep_k is reached when assembling the root node. The last step is to examine $k = l$, that is the final separator:

1. Below (1) and (2) are subtrees corresponding to the four corner sets of Sep_{l-1}. Their peak is reached when allocating their root node and is ($k = l \quad 1$ in $17 \;\; 2^{2k} + 4 \;\; 2^k + 1$)

$$4\ 25 \quad n^2 + 2 \quad n + 1$$

2. At node (1) (or (2) similarly), the peak is (note that we do not matter about the ordering of the children since their subtrees are the same):

$$\begin{aligned} &\max \quad sfront_{(1)} + scb_{s_{(1),1}} + scb_{s_{(1),2}} \;\; S_{s_{(1),2}} + scb_{s_{(1),1}} \\ =&\max \quad (n/2 + n + 1)^2 + 2\,(n+1)^2 \;\; 4\ 25 \quad n^2 + 2 \quad n + 1 + (n+1)^2 \\ =&\max \quad 4\ 25 \quad n^2 + 7 \quad n + 3 \;\; 5\ 25 \quad n^2 + 4 \quad n + 2 \\ =&\, 5\ 25 \quad n^2 + 4 \quad n + 2 \text{ (for } n \geqslant 4 \ldots) \end{aligned}$$

Thus, the peak of the subtree rooted at (1) (or (2)) is reached when allocating the second child of (1) (or (2)).

3. Finally, at node (3), the peak is (we do not matter about the ordering of the children since their subtrees are the same):

$$\begin{aligned} &\max \quad sfront_{(3)} + scb_{(1)} + scb_{(2)} \;\; S_{(2)} + scb_{(1)} \\ =&\max \quad (n+1)^2 + 2\,(n+1)^2 \;\; 5\ 25 \quad n^2 + 4 \quad n + 2 + (n+1)^2 \\ =&\max \quad 3 \quad n^2 + 6 \quad n + 3 \;\; 6\ 25 \quad n^2 + 6 \quad n + 3 \\ =&\, 6\ 25 \quad n^2 + 6 \quad n + 3 \end{aligned}$$

Therefore, the peak is reached when allocating the second child of (2), i.e., the fourth corner separator with respect to the postorder.

□

We showed that the sequential peak of active memory is reached when allocating the root node of the last corner subtree. We illustrate this in Figure 8.7 that shows the shape of the top of the tree and the state of the active memory when the peak takes place. In Figure 8.8, we show the behavior of the active memory memory during a sequential traversal of a tree with $n = 8$. Each node of the tree is associated with three consecutive markers in the figure, that correspond to the allocation of that node, the removal of the contribution blocks of its children from the active memory, and finally the removal of its factored part (only the contribution block stays in the active memory). (c) corresponds to the allocation of the root node, (b) corresponds to the allocation of its right-most child (i.e., the one ordered last in the postorder), and (a) corresponds to the allocation of the last corner subtree, which corresponds to the peak of active memory.

The next step of the proof consists of computing a memory-based proportional mapping of the tree. We use two simplifications:

We work with decimal number of processes, i.e., we do not apply any rounding.

We assume that every task (node) can be perfectly distributed among the processes it is mapped to.

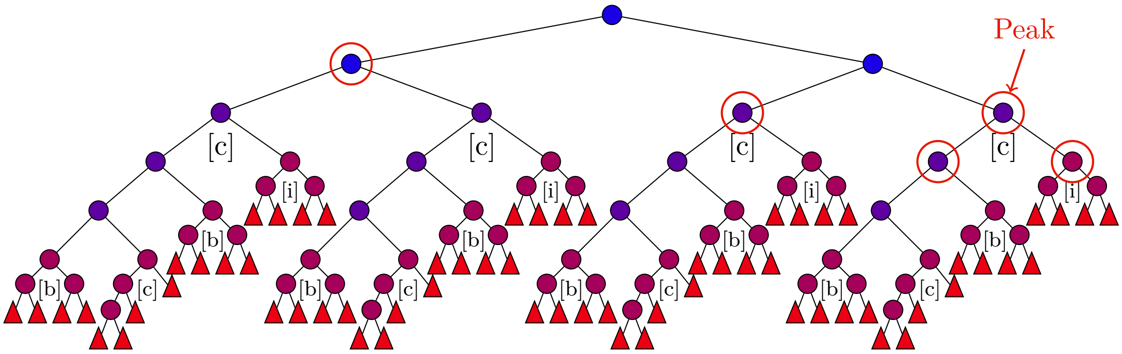


Figure 8.7: Top of the tree obtained from a nested dissection, with $n \geqslant 16$. The contents of the active memory when the peak is reached are shown with red circles.

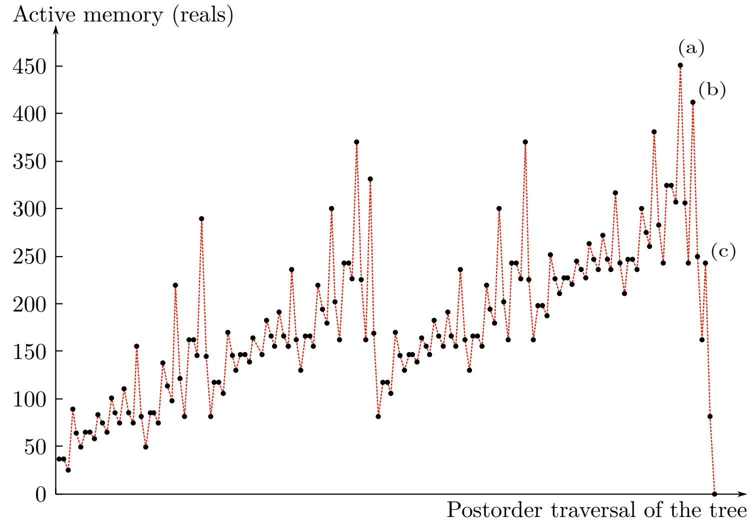


Figure 8.8: Behavior of the active memory during a sequential traversal of a tree with $n = 8$. Each node of the tree is associated with three consecutive markers in the figure, that correspond to the allocation of that node, the removal of the contribution blocks of its children from the active memory, and finally the removal of its factored part. (a) corresponds to the allocation of the last corner subtree, which corresponds to the peak of active memory, (b) corresponds to the allocation of its right-most child (i.e., the one ordered last in the postorder), and (c) corresponds to the allocation of the root node.

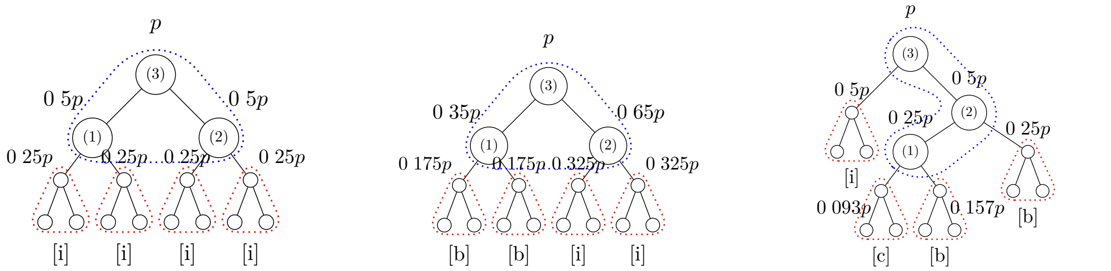
At each node, the way the processes are distributed to its children depends on the level k we consider; however, we can ignore low-order terms. For example, consider a boundary set at a given level k. Say node (3) has been given p processes; we want the compute the mapping for nodes (1) and (2). The sequential peak at (1) is $10\ 75 \quad 2^{2k} + 11 \quad 2^k + 3$ and the peak at (2) is $20\ 25 \quad 2^{2k} + 7 \quad 2^k + 1$. The sum of these two peaks is $31 \quad 2^{2k} + 18 \quad 2^k + 4$ thus the number of processes given to (1) is:

$$\frac{10\ 75 \quad 2^{2k} + 11 \quad 2^k + 3}{31 \quad 2^{2k} + 18 \quad 2^k + 4}$$

This ratio tends to $\frac{10\ 75}{31} = 0\ 346$ when p tends to infinity; at level 5, it is already $0\ 35$. Therefore, we can use a constant ratio by considering that these divisions occur only for $k \geqslant 5$. Indeed, we are interested in large problems, e.g. $l \qquad 7$ (i.e.,

N 16129). At level 5, the number of processes at each node has been divided by (roughly) 4^{l-5}. If it is 1, the subtrees below level 5 are processed sequentially, and we do not compute ratios. Otherwise, it probably means that the number of processes is unreasonably high compared to the size of the matrix. Therefore, when mapping the children, we will use the asymptotic ratios (as if $k = +$).

Using these two simplifications, we can derive the rules that describe how processes are distributed for each kind of separators; they are shown in Figure 8.9.



(a) Interior sets: processes are equally distributed among siblings.

(b) Boundary sets: (1) and (2) are given 35% and 65% of the processes respectively. At the level below, processes are equally distributed.

(c) Corner sets: processes are equally distributed below (3) and (2). The corner and boundary subtrees below (1) are given 37% and 63% of the processes respectively.

Figure 8.9: Distribution of the processes for the three kinds of separators using a memory-based proportional mapping.

The final step of the proof of the theorem consists of computing the memory efficiency. We compute a lower bound on the parallel peak. The reasoning is the same as the one we used in the simple example at the beginning of the chapter; at best, in every subtree $\mathcal{T}(i)$, the maximum peak of active memory is $\frac{S_i}{p_i}$, and, by computing the maximum over all the subtrees, we obtain a lower bound on the parallel peak on the whole tree. This bound might be rather loose but it is enough for our purpose and, in the end, we obtain an upper bound on memory efficiency that demonstrates that a proportional mapping is not suitable with respect to memory scalability. We consider the three types of subsets:

Interior sets: one can prove by induction that, in the best case where a perfect efficiency is attained in every subtree, the maximum peak among the processes is reached when allocating the root node. The key idea is the following (we consider only the terms in 2^{2k} for simplicity; sometimes the terms in 2^k need be checked to distinguish between two polynomials with the same term in 2^{2k}). Assume that there are p processes working at (3). The peak at this node is at least $\frac{172\ 2^{2k}}{p}$; this is larger than the term $\frac{81\ 2^{2k}}{0\ 5p}$ that comes from node (1) (or (2)) and the term $\frac{43\ 2^{2k}}{0\ 25p}$ that comes from the interior subtrees below.

Boundary sets: consider a boundary set of Sep_k mapped onto p processes. One can prove that the maximum peak is reached at the assembly of the root node of sequential boundary subtrees, or at the level above (it depends on roundings but does not change the asymptotic result in the end); we consider the last case in the following.

Corner sets: one can show that the peak comes from the boundary subtrees.

Finally, at level l, one can show that the assembly of (1), (2) and (3) do not change the peak. Therefore, the maximum peak is reached for sequential boundary subtrees. Among these trees, the maximum peak is reached for the highest sequential boundary trees (i.e., the ones with the highest root node among all the sequential boundary trees); there are four of them, and they are below the first boundary subtree of each corner set. These trees correspond to level k_s such that:

$$\begin{aligned} k_s = l &\quad 2 + \frac{\log p + \log \frac{17}{736}}{\log \frac{43}{248}} \\ l &\quad 0\;4 \log_2 p + 0\;15 \end{aligned}$$

Therefore, the maximum parallel peak $S_{max}(p)$ is lower bounded by :

$$29 \quad 2^{2k_s} + 4 \quad 2^{k_s} + 1$$

Finally, we can compute an asymptotic expression for $e_{max}(p)$ by keeping only the terms in 2^{2l}; for n and p large enough, the efficiency is bounded by:

$$\begin{aligned} e_{max}(p) &\leqslant \frac{S_{seq}(p)}{p \quad S_{max}(p)} \\ &\leqslant 0\;18 \quad p^{-0\;2} \end{aligned}$$

We have assessed this prediction for $n = 1024$, using a symbolic code that builds a tree that corresponds exactly to the nested dissection model presented above, and that computes the peak of each process by performing postorder traversals of the tree. We show the results in Table 8.1. We observe that the model fairly corresponds to the simulation results; remember that we neglected rounding problems and used asymptotic fractions in many places. Furthermore, what we have computed is an upper bound of the efficiency. We also report on experiments using MUMPS on the same problem. The ordering is computed with MeTiS. Note that MUMPS uses a mapping primarily based on a relaxed proportional mapping, while our model gives an upper bound on memory efficiency for a strict proportional mapping; this explains why, on 32 and 64 processes, the memory efficiency is higher than our bound. However, even with this difference, we observe that the results are very close to our prediction when the number of processes increases.

p	$e_{max}(p)$		
	Model	Simulation	MUMPS
32	0.090	0.107	0.189
64	0.078	0.091	0.113
128	0.068	0.073	0.062
256	0.059	0.053	0.053
512	0.051	0.044	0.040

Table 8.1: Memory efficiency computed using the model and a symbolic code, for a 2D nested grid with $n = 1024$.

This result shows that, using a strict memory-based proportional mapping, the memory efficiency $e_{max}(p)$ tends to significantly decrease when the number of processes increases. The proof is interesting since we have computed a lower bound on the parallel memory usage by considering that, in every subtree, a perfect memory scalability is attained. Even in this best case, the memory efficiency for the whole tree is low. This simply demonstrates that the proportional mapping does not assign enough processes to the nodes, at least to those that are in demanding-parts of the tree; when going down in the tree, the number of processes decreases too much compared to memory needs.

In the next section, we complement this result with practical experiments using the MUMPS solver; the results confirm the need for a better control of active memory.

8.3 Experimental results

In this section, we provide experimental results using the current version of MUMPS (4.10.0 at the time of writing). We assess the memory scalability of the code on different matrices from our experimental set. In Table 8.2, we report on experiments with the Geoazur generator described in Section 1.3.3; we generate a matrix corresponding to a $192 \times 192 \times 192$ grid. The size of the matrix is $N = 7077888$, the number of entries in the matrix is 189.1 millions, and the size of the LU factors is 144 GB (in single precision arithmetic). The sequential peak of active memory is 36.9 GB. The number of processes ranges from 32 to 512 and we measure e_{max} and e_{avg}.

p	$S_{max}(p)$ (GB)	$e_{max}(p)$	$S_{avg}(p)$ (GB)	$e_{avg}(p)$
32	5.24	0.22	3.83	0.31
64	3.02	0.19	1.88	0.30
128	1.66	0.18	0.99	0.29
256	1.09	0.13	0.49	0.29
512	0.70	0.10	0.26	0.28

Table 8.2: Memory scalability of MUMPS on a Geoazur problem corresponding to a 192^3 grid. We provide the maximum and average peaks of active memory and the corresponding memory efficiency.

We notice that the memory efficiency is low even with a small number of processes. e_{max} decreases significantly when the number of processes increases, while e_{avg} is rather stable. Therefore, there is a problem of balance of the memory usage of the different processes, but not only; we recall that $e_{avg} = 0\ 28$ means that, on average, processes use $1/0\ 28 = 3\ 5$ times more memory than they should if we wanted to perfectly divide the sequential peak of active memory. We observe that the behavior of e_{max} is close to what the model introduced in the previous section predicts, i.e., e_{max} strongly decreases when p increases. Thanks to the relaxed variant of proportional mapping used within MUMPS, the efficiency is better than what our model predicts, but is it still very low.

In Table 8.3, we report on experimental results with different matrices from Table 1.1, with a fixed number of processes ($p = 128$).

Matrix	S_{seq} (MB)	$S_{max}(p)$ (MB)	$e_{max}(p)$	$S_{avg}(p)$ (MB)	$e_{avg}(p)$
NICE20MC	1301	222	0.05	141	0.07
AUDI	1493	125	0.09	77	0.15
CONESHL	870	57	0.12	29	0.23
FLUX-2M	2967	340	0.07	124	0.19
CAS4R_LR15	131	67	0.02	18	0.06

Table 8.3: Memory scalability of MUMPS for different problems from Table 1.1 with a fixed number of processes $p = 128$.

As with the previous table, we notice that the memory scalability is very poor. Once again there is problem of balance; on matrix CAS4R_LR15, the maximum peak is almost 4 times the average peak. For the same matrix, $e_{avg} = 0\ 06$ which means that the average peak is more than 16 times what we are targeting, which of course might simply prevent a user to perform the factorization.

All these experimental results, along with the model introduced in the previous section, show that there is a need for a mapping technique able to enforce a much better memory scalability. The simple example introduced at the beginning of this chapter leads to surmise that the number of processes associated with each node should be increased, at least at the top of the tree and in the parts of the tree that are the most memory-demanding. However, in the parts of the tree that are not troublesome in terms of memory, using a proportional mapping is still a valid strategy. In the next chapter, we present different memory-aware mapping algorithms that aim at enforcing this idea: increasing the number of processes given to each node in the memory demanding parts of the tree, and using a proportional mapping whenever it is possible.

Chapter 9

A class of memory-aware algorithms

We demonstrated in Chapter 8 that the proportional mapping and its variants lead to a low scalability of the active memory; however, the proportional mapping is interesting in terms of performance since it maximizes tree parallelism and reduces the volume of communication within parallel nodes and between nodes of the tree. We also introduced in Chapter 7 an all-to-all mapping which consists of a constrained traversal of the tree where all the processes work on every node, following a postorder. However, this solution generates prohibitive amounts of communications, it does not exploit tree parallelism and yields too small granularities at nodes at the bottom of the tree. Therefore it is not feasible at all. In this chapter, we introduce a memory-aware mapping that hybridizes these two techniques and tries to enforce a given memory constraint (the maximum amount of active memory that a process can use). Basically, the idea is to use an all-to-all mapping in memory demanding parts of the tree, and a proportional mapping whenever we can be sure that it will not violate the memory constraint. This approach was introduced in Agullo's PhD thesis [1, Chapter 10]; we recall it in Section 9.1. We suggest in Section 9.2 some refinements that still enforce the memory constraints and leverage the performance of the factorization. In Section 9.3, we show how we convert a mapping with rational numbers of processes into a mapping with integer numbers following the technique proposed by Beaumont and Guermouche in [17]. In Section 9.3.3, we highlight some problems that require the use of some relaxation parameters in the memory-aware mapping.

9.1 A simple memory-aware mapping

9.1.1 Idea and algorithm

We assume that we are given a memory constraint M_0 that represents the maximum amount of active memory that a process is allowed to use. This constraint is defined by the user or can be set automatically. The memory-aware mapping works as follows. We assume that the tree has been reordered in order to reduce the sequential peak of active memory (following the algorithm described in Chapter 7). We also assume that a preliminary traversal of the tree has been performed in order to obtain the sequential peak S_i and the number of floating-point operations W_i for each subtree $\mathcal{T}(i)$. Then, a top-down traversal of the tree is performed which computes the mapping. Firstly, all the processes are assigned to the root node. Then, recursively, once a node is mapped onto p_f processes, its children are mapped. We first check if a proportional mapping of the children is feasible[1] by simulating a proportional mapping and checking the memory constraint for every child i. Denote p_i the number of processes that a proportional mapping would assign to i. For every child i, we check the condition $\frac{S_i}{p_i} \leqslant M_0$:

If all the subtrees $\mathcal{T}(i)$ respect this condition, then the step of proportional mapping is accepted; the children subtrees will be processed in parallel on the number of processes provided by the step of proportional mapping. For the subsequent steps of the mapping procedure, the memory constraint M_0 remains the same.

If at least one of the subtrees does not respect the condition, then the step of proportional mapping is rejected. All the children subtrees inherit the processes of their parent (i $p_i = p_f$) and will be processed one after another during the factorization, following a local (partial) postorder. In this case, when a subtree $\mathcal{T}(j)$ is processed, the contribution blocks of the previous siblings $(1 \leqslant i < j)$ are stacked and equally distributed in the memory of the p_f processes. Therefore, for the next steps of the mapping procedure, the memory constraint is modified: M_0 becomes $M_0 \quad \sum_{i=1}^{j} \frac{scb_i}{p_i}$ (assuming siblings are numbered following the postorder) in order to take into account these contribution blocks.

At each step of the traversal, the condition $\frac{S_i}{p_i} \leqslant M_0$ means is it possible to process the subtree $\mathcal{T}(i)$ on p_i processes, using at most M_0 per process? . Thus, when a step of proportional mapping is accepted, we ensure that every subtree will respect the memory constraint; perhaps, for some memory-demanding subtrees for which S_i is close (lower but almost equal) to p_i M_0, this will require to apply an all-to-all mapping. For easier parts of the tree, a regular proportional mapping will be used. In the end, this yields a hybrid mapping in-between a proportional mapping and an all-to-all mapping.

Some scheduling constraints have to be set:

When a step of proportional mapping is rejected, constraints are set so that a node cannot start before its previous (in the sequential postorder) siblings have finished, i.e., the subtrees rooted at its siblings have been completely processed. The first sibling inherits the constraint of its parent, so that constraints are propagated to the bottom of the tree. For example, if the parent node is constrained so that it has to wait for a node $\mathcal{N}$ (e.g. its sibling) then the constraint is propagated and the first sibling has to wait for the same node as $\mathcal{N}$ (e.g. its uncle). Using this mechanism at every level, constraints is propagated to the bottom of the tree.

When a step of proportional mapping is accepted, every child inherits the constraint of its parent (similar to what we describe above).

In order to represent the scheduling constraints, it is enough to specify the predecessor of every constrained node, since, when some constraints are set, we create a chain of serialized subtrees; a given subtree simply has to wait for one other subtree to complete. The way the scheduling constraints are implemented is detailed in Section 10.1.

Note that whenever an all-to-all mapping is locally used, all the associated processes stack pieces of the same contribution blocks. Assuming that contribution blocks can be equally distributed among processes, these processes stack the same amount of memory. Therefore, the algorithm does not need to track the volume of active memory of each process; in any subtree, the remaining memory of the processes working at that subtree is the same. At a given node i in the tree, for any process p working at the node, the

[1] The metric (e.g. memory or operation count) used in the proportional mapping is chosen in advance.

memory constraint is M_0 minus the pieces of contribution blocks that have been previously stacked; these contribution blocks belong to nodes that are in the stacked set of i ($\mathcal{S}(i)$, cf. Definition 7.1) and that are mapped onto p. We illustrate this in an example in the next section.

The memory-aware mapping procedure is presented in a recursive fashion in Algorithm 9.1; we provide a simplified version where we only compute the number of processes given to each node, not a true mapping (see Section 9.1.3 for more details). The main point is to show how we propagate scheduling constraints and the memory constraint throughout the tree.

We mentioned that the memory-aware mapping aims at ensuring a given memory constraint. Another interesting feature is that, similar to a strict proportional mapping or an all-to-all mapping, it allows to compute accurate memory estimates prior to the factorization. Indeed, at a given set of siblings in the tree, the set of processes is either perfectly split (when a step of proportional mapping is accepted) or the traversal of the set of siblings is completely constrained (when a step of proportional mapping is rejected). Therefore, as in the proportional mapping and in the all-to-all mapping, it is possible to compute memory estimates simply by simulating a (partial) postorder traversal of the tree for each process.

9.1.2 Example

We illustrate a few steps of memory-aware mapping in Figure 9.1; this is almost the same example as the one we used in the previous chapter (the only difference is that this time nodes s_1 and s_3 have large contribution blocks, thus the sequential peak of active memory is changed). The tree is to be mapped onto 64 processes and the memory constraint is $M_0 = \frac{S_{seq}}{64} = 111$ MB. Firstly, the 64 processes are assigned to the root node r; then the three children s_1, s_2 and s_3 of r are mapped. The first step consists of computing a proportional mapping of the three children nodes. s_1 is given 26 processes, s_2 is given 6 processes and s_3 is given 32 processes. Then, the memory constraint is checked for the three subtrees. At node s_1, the sequential peak of active memory is 4 GB; thus $\frac{4\text{ GB}}{26} = 158$ MB is greater than M_0. Therefore, the subtree rooted at s_1 cannot be processed using 26 processes without violating the memory constraint. Thus the step of proportional mapping is rejected; the three children subtrees are mapped onto the 64 processes and are serialized ($\mathcal{T}(s_1)$ will be processed first, then $\mathcal{T}(s_2)$, then $\mathcal{T}(s_3)$). Then the three subtrees are mapped using the same procedure.

Now consider the mapping of the subtree rooted at s_3. Since we serialized the three children subtrees of r, we have to take into account that, when processing $\mathcal{T}(s_3)$, the contribution blocks of s_1 and s_2 are stacked in memory and are equally distributed among the processes. Assume that the contribution block of s_1 weights 1600 MB and that the contribution block of s_2 weights 500 MB. For a given process, the available memory is no longer M_0 but $M_0 \quad \frac{1600\text{ MB}}{64} \quad \frac{500\text{ MB}}{64} = 78$ MB. A proportional mapping of the two children of s_3 attributes 43 processes to s_{31} and 21 processes to s_{32}. This step of proportional mapping is accepted since the memory constraint is ensured for both subtrees ($\frac{S_{s_{31}}}{43} = 48$ MB $< M_0$ and same for s_{32}). Then, when mapping the children of s_{31}, a proportional mapping cannot be used. Indeed, a proportional mapping attributes 22 processes to s_{311} and 21 processes to s_{312}; doing so, the memory constraint cannot be ensured at these nodes since $\frac{S_{s_{311}}}{22} = 93$ MB $> M_0$ (remember that in this part of the tree, the memory constraint is no longer 111 MB but 78 MB since we know that $\mathcal{T}(s_1)$ and $\mathcal{T}(s_2)$ have to be completely processed before any node in $\mathcal{T}(s_3)$ can start). Therefore, these two subtrees are mapped onto the same 43 processes as s_{31} and are serialized. In

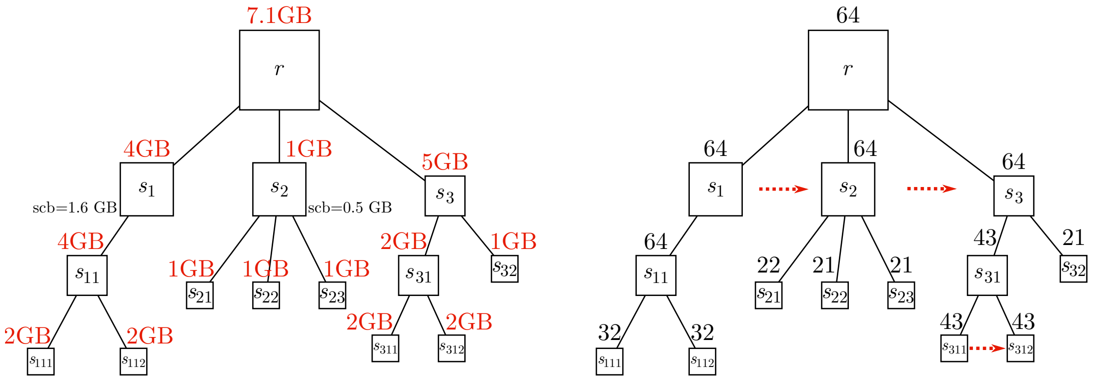
Algorithm 9.1 Basic memory-aware algorithm.

```
Main procedure (memory-aware mapping)
/* Input: $T$ the tree (root node $r$)                                          */
/*        $p$, number of processes                                               */
/*        $M_0$, memory constraint                                               */
/* Output: $NUM$, number of processes of each node (size: number of nodes in the tree) */
/*         $PREV$, array representing serialization constraints (predecessor of each node) */

$PREV(r) = 0$ /* No specific constraint for the root node, 0 by convention */
Call MA($T$ $p$ $M_0$ $NUM$ $PREV$)

Procedure MA($T$ $p$ $M_0$ $NUM$ $PREV$)
/* Input: $T$ a tree (root node $r$); $p$, number of processes; $M_0$, memory Constant */
/* Input-output: $NUM$, number of processes; $PREV$, scheduling constraints */

/* Step 1: Simulate a proportional mapping */
Every sibling $\mathcal{N}_i$ receives a number of processes $p_i \leqslant p$ (proportional mapping)
/* Step 2: Check memory constraints */
$reject\_prop$    false
for all siblings $\mathcal{N}_i$ do
    if $\frac{S_i}{p_i} > M_0$ then
        $reject\_prop$    true
        Break
    end if
end for
/* Step 3: Set the number of processes and the constraints */
if $reject\_prop$ then /* Reset to an all-to-all mapping */
    for all siblings $\mathcal{N}_j$ do
        $NUM(\mathcal{N}_i) = p$
        $PREV(\mathcal{N}_i) = \mathcal{N}_{i-1}$ (for $i = 1$: $PREV(\mathcal{N}_1) = PREV(r)$)
    end for
else /* The step of proportional mapping is accepted */
    for all siblings $\mathcal{N}_i$ do
        $NUM(\mathcal{N}_i) = p_i$
        $PREV(\mathcal{N}_i) = PREV(r)$
    end for
end if
/* Step 4: Recursively call the memory-aware mapping on all the children subtrees */
for all siblings $\mathcal{N}_i$ do
    if $prop\_reject$ then
        $stack$    $\sum_{j=1}^{i-1} scb_i$
    else
        $stack$    0 /* Nothing is stacked; $M_0$ will not be decreased */
    end if
    Call MA($\mathcal{T}(\mathcal{N}_i)$ $p_j$ $M_0$    $\frac{stack}{p}$ $NUM$ $PREV$)
end for
```

the other parts of the tree, the steps of proportional mapping are accepted; this yields the mapping shown in Figure 9.1(b).

We illustrate how the memory constraint is propagated using the notion of stacked set. Consider a process p assigned to s_{22}. The stacked set of node s_{22} is s_1 s_{21} ; however, s_{21} is not mapped onto p since a step of proportional mapping was used at node s_2. Thus, at s_{22}, what is stacked in the memory of p is a piece of contribution of s_1 only. Therefore, if we were to map a subtree rooted at s_{22}, the memory constraint would be $M_0 \quad \frac{scb_{s_1}}{64}$.

In this simple example, since the algorithm manages to map the tree using $M_0 = \frac{S_{seq}}{64}$, a perfect or near perfect memory scalability can be expected. Furthermore, in this example, our mapping is far from being an all-to-all mapping, thus a much better performance can be expected.



(a) Sequential peaks of active memory.

(b) Memory-aware mapping

Figure 9.1: Simple example of memory-aware mapping. In (a), we indicate the sequential peak of active memory at each node. In (b), the tree is mapped using the memory-aware mapping with $M_0 = \frac{5\text{ GB}}{64}$. At the root node r, a proportional mapping of its children s_1 s_2 s_3 cannot be applied without violating the memory constraint; therefore an all-to-all mapping of this set of children is applied. We show the scheduling constraints with arrows; the subtree rooted at s_1 is processed before the subtree rooted at s_2, and the subtree rooted at s_3 is processed last. Similarly, the subtrees rooted at s_{311} and s_{312} are mapped using a local all-to-all mapping.

9.1.3 Difficulties

We have not mentioned the difficulties of our mapping technique. Firstly, a step of proportional mapping usually attributes non-integer number of processes to each subtree. One could choose to round these numbers and arrange them so that they add up to the number of processes given to the parent node, yielding a perfect partitioning of the set of processes on the children subtrees. However, this is potentially not very robust with respect to memory since the memory constraint can no longer be guaranteed on the subtrees for which the number of processes has been rounded down. Applying a relaxation parameter when checking the constraint at each step of the mapping might not be sufficient since rounding effects might accumulate when going down in the tree. We have chosen to use a more sophisticated scheme where we allow a process to work part time on two parallel subtrees; this allows to have decimal number of processes at each subtree. This scheme is inspired by a strategy proposed by Beaumont and Guermouche, where they refer to processes that work part time as *extra processes* [17]. We describe this strategy in Section 9.3.

We have also assumed that the contribution block of each node could be equally distributed among the processes that work on that node in order to update the memory constraint M_0. However, this is not always feasible; for instance this is cannot be done using MUMPS, and we highlight this difficulty in Section 9.3.3.

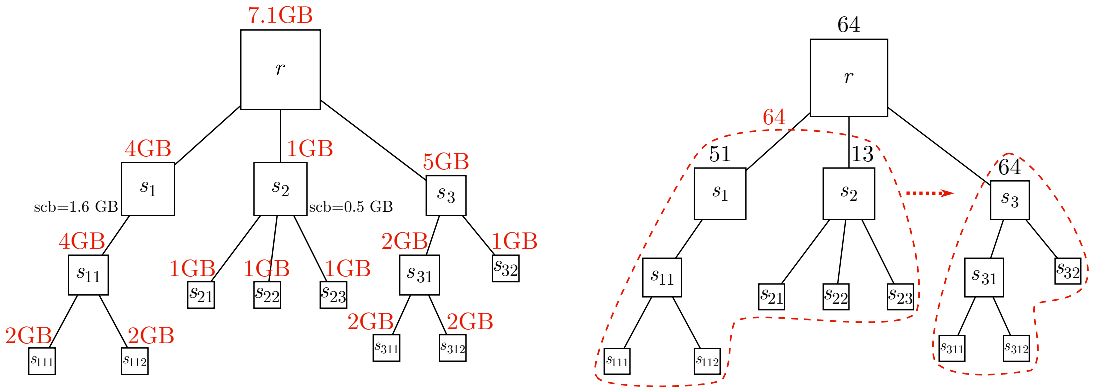
9.2 Detecting groups of siblings

The main idea of the memory-aware mapping is to detect memory-demanding parts of the tree. When a problematic subtree is detected among a set of siblings, the whole set of siblings it belongs to is mapped using a local all-to-all mapping; all the siblings inherit the mapping of their parent, and all the subtrees are serialized i.e. processed one after another. Although this works well in enforcing the memory constraint, this is quite constraining as it potentially maps small subtrees on many processes and sets many serializations even in less memory-demanding parts of the tree. Indeed, for a given set of siblings in the tree, some of subtrees rooted at these siblings might have a high sequential peak of active memory while some others might have a small peak and do not require to be mapped onto many processes. This might not happen much in regular problems (e.g. PDEs with finite elements discretizations) where the trees are quite often fairly balanced, but it could happen in more irregular problems. In this situation, we would like to relax the baseline strategy and avoid to serialize the whole siblings. In this section, we refine the main idea of the memory-aware mapping and propose a strategy where, when working on a set of siblings, we try to detect groups of subtrees within which a proportional mapping is applied and that are serialized such that the memory constraint is ensured. Similar to the baseline strategy, the memory constraint M_0 is ensured. The interest is that this variant decreases the number of serializations and to decrease the number of process in easier parts of the tree.

9.2.1 A motivating example

We use once again the example that we showed in the previous chapters. Figure 9.2(a) shows the elimination tree and the sequential peak of active memory of each node. The tree is to be mapped onto 64 processes and this time we choose to slightly relax the memory constraint $M_0 = 1\ 1\ \ \frac{S_{seq}}{64} = 122$ MB (i.e., we target a memory efficiency of $\frac{1}{1\ 1} = 0\ 91$). Using this memory constraint, one can easily show that the memory-aware mapping behaves exactly as in the previous example. In particular, the children of the root node r cannot be mapped using a proportional mapping. The memory-aware mapping therefore applies a local all-to-all mapping; the three subtrees are serialized and mapped onto the 64 processes. We can however go one step forward. Since the subtree rooted at s_3 is the most constraining subtree, using all the processes at this subtree is a reasonable strategy. However, we can map s_1 and s_2 using a proportional mapping of these two nodes, then process them in parallel and constrain the scheduling so that the subtree rooted at s_3 starts after both the subtrees rooted at s_1 and s_2 are finished. Indeed, a proportional mapping of s_1 and s_2 (without s_3) attributes $\frac{4}{4+1}\ \ 64 = 51$ processes to s_1 and 13 processes to s_2. This is acceptable with respect to the memory constraints since $\frac{4 \text{ GB}}{51} = 78$ MB $< M_0$ and $\frac{1 \text{ GB}}{13} = 78$ MB $< M_0$.

This time, we also have to check that, taking into account what is stacked at s_1 and s_2, the subtree rooted at s_3 can be processed using 64 processes. Indeed, this is a major difference compared to the baseline algorithm; in the basic memory-aware mapping, whenever an all-to-all mapping is locally used, all the processes stack the same thing. This is different in the variant with groups. In our example, the 51 processes working on the subtree rooted at s_1 stack equal parts of the contribution block of s_1; similarly, the 13 processes working on the subtree rooted at s_2 stack equal shares of the contribution block



(a) Sequential peaks of active memory.

(b) Memory-aware mapping with groups.

Figure 9.2: Memory-aware mapping with groups: the two subtrees rooted at s_2 and s_3 can be put in a group within which a proportional mapping is applied. The scheduling is constrained so that the subtree rooted at s_3 can start only after the group is finished i.e. when the subtree rooted at s_2 and the subtree rooted at s_3, that are processed in parallel, are finished.

of node s_2. These two contribution blocks are different and the ratio of their sizes is not the same as the ratio of the sequential peaks of the two subtrees; therefore the processes stack different amounts of memory. This implies that the algorithm has to control the active memory of each process, while it was possible to use a global stack in the basic algorithm. The algorithm is still able to ensure the memory constraint on every process, but it might yield more imbalance in memory because of the distribution of contribution blocks. This imbalance could prevent processing a subtree even using all the processes used at its parent node. In this example, things work well; for the 51 processes working on the subtree rooted at s_1, the memory constraint (remaining memory) when mapping s_3 is $M_0 \quad \frac{1600\text{ MB}}{51} = 94$ MB. For the 13 processes working on the subtree rooted at s_2, the memory constraint at s_3 is $M_0 \quad \frac{500\text{ MB}}{13} = 87$ MB. On every process, enough memory is available for processing the subtree rooted at s_3 if the 64 processes are used on this subtree; indeed $\frac{S_{s_3}}{64} = 80$ MB which is smaller than the remaining memory on any process (whether it works on the subtree rooted at s_1 or the subtree rooted at s_2).

Note that in this example, one can show that putting s_1 and s_2 together and s_3 alone is the only valid partitioning with respect to the memory constraint (the constraint is not ensured using the two other possibilities).

9.2.2 Algorithm

We now show formally how our variant of the memory-aware mapping is built. As the basic memory-aware mapping, it consists of a top-down traversal of the tree; the only difference is the way a set of siblings is processed. In the basic memory-aware mapping, the whole set is mapped using a proportional mapping; then the memory constraint is checked for every subtree, and if the constraint cannot be ensured on at least one subtree, the whole set of siblings is reset to an all-to-all mapping. Our algorithm tries to detect groups of siblings within which a proportional mapping can be applied without violating the memory constraint, and taking into account the fact that groups are serialized (as mentioned above, this requires to track the active memory of every process). We assume the following scheme:

We are given an order for traversing the list of siblings.

Following this order, we form groups of nodes as large as possible.

By forming groups as large as possible, our aim is to decrease the number of serializations, and to obtain a mapping that is as close as possible to a proportional mapping while ensuring the memory constraint. We have chosen to form groups following a fixed order for simplicity. One could formalize our problem in terms of partitioning (how to partition the set of siblings so that there is a minimum number of parts and the memory constraint is ensured in each part?). This seems to be a very difficult partitioning problem (as the constraint to be checked on a part depends on the other parts) that looks somewhat like a knapsack problem, but we have not tackled it. We believe that following a fixed order such as the one provided by the sequential traversal is reasonable.

Following this strategy, we form a group using the following procedure. Denote i the current position in the list of siblings and $\mathcal{N}_i$ the associated node, and assume that the previous nodes have been processed i.e. have been given a number of processes. Firstly, we need to check that $\mathcal{N}_i$ can at least be a singleton i.e. can be alone in a group. Indeed, it could happen that, because unbalanced shares of the contribution blocks of the previous siblings have been stacked, the memory constraint cannot be ensured on at least one process, even if $\mathcal{N}_i$ forms a group by itself. In that case, we reset all the previous siblings to an all-to-all mapping; this will reset the balance among the processes. Then, starting from i, we add nodes to the group until the memory constraint can no longer be satisfied. We repeat this process until the whole set of siblings has been partitioned. Finally, the mapping is computed (a proportional mapping is applied within each group) and the scheduling constraints are set (each group has to wait for the previous one to finish before it starts). We show how such scheduling constraints can be implemented in the next chapter.

Finally, we emphasize the constraints to be ensured within each group. We set some notation. We denote $\mathcal{N}_f$ the parent node of the set of siblings we consider. We denote p_f the number of processes $\mathcal{N}_f$ is mapped to. Assume that we are building the g-th group, denoted $\mathcal{V}_g$, and that $g \quad 1$ groups $\mathcal{V}_1 \qquad \mathcal{V}_{g-1}$ have been built already. We denote $\mathcal{N}_j$ a node of the $g \quad 1$ first groups and $\mathcal{N}_i$ a node of the group being built. For a given node $\mathcal{N}_i$ (or $\mathcal{N}_j$), p_i is the number of processes $\mathcal{N}_i$ is mapped onto, $sfront_i$ is the size (surface) of the frontal matrix associated with $\mathcal{N}_i$, and scb_i is the size (surface) of the contribution block of $\mathcal{N}_i$. Finally, we keep track of the active memory of every process in an array $PSTACK$. $PSTACK(\mathcal{N}_i \; p)$ is the size of the contribution blocks that are stacked on process p before the factorization enters in the subtree rooted at $\mathcal{N}_i$, i.e., the nodes in the stacked set of $\mathcal{N}_i$ that are mapped onto p. The constraints we ensure when forming groups are the following:

The nodes within a group can be mapped using a proportional mapping without violating the memory constraint:

$$\text{(Cstk):} \quad \mathcal{N}_i \quad \mathcal{V}_g \quad p \text{ working on } \mathcal{N}_i$$

$$\frac{S_i}{p_i} < M_0 \quad PSTACK(\mathcal{N}_f \;\; p) \quad \sum_{k=1}^{g-1} \sum_{\mathcal{N}_j \in \mathcal{V}_k} \frac{scb_j}{p_j}$$

The assembly of the parent node can be done without violating the memory constraint[2]:

$$\text{(Casm):} \quad \mathcal{N}_i \quad \mathcal{V}_g \quad p \text{ mapped onto } \mathcal{N}_i$$

$$\frac{scb_i}{p_i} + \frac{sfront_f}{p_f} < M_0 \quad PSTACK(\mathcal{N}_f \;\; p) \quad \sum_{k=1}^{g-1} \sum_{\mathcal{N}_j \in \mathcal{V}_k} \frac{scb_j}{p_j}$$

We summarize our algorithm in Algorithm 9.2. We do not emphasize the way the scheduling constraints are set; this is described in Section 10.1.

Algorithm 9.2 Computation of groups of siblings.

```
/* Input: a set of siblings $\mathcal{N}_1$      $\mathcal{N}_s$                                  */
/*        a given order used to traverse the list of nodes (order)          */
while not all the children have been collected (following order) do
    $i$= current node
    if $i$ cannot be alone without violating (Cstk) or (Casm) then
        Reset previous siblings to an all-to-all mapping
    end if
    Starting from $i$: collect as many nodes as possible as long as (Cstk) and (Casm) can be ensured
    Use a proportional mapping on the group, serialize with the previous ones
end while
```

9.3 Decimal mapping, granularities and relaxations

In this section, we describe some practical aspects of the memory-aware mapping. We show how we handle a mapping with decimal number of processes, we describe how we enforce some granularities at every subtree, and we show that we need to introduce some relaxations parameters in the mapping.

9.3.1 Handling decimal number of processes

As we mentioned in Section 9.1, we have chosen not to round the numbers of processes computed at each step of the mapping process to integer values, as this is potentially dangerous with respect to memory constraints. Beaumont and Guermouche use a scheme where they round the number of processes but allow a process to work on two parallel branches at the same time [17]. We refined this idea by allowing a process to work *part time* on two parallel subtrees; this allows to have decimal number of processes at each subtree. Firstly, we illustrate our scheme using the example in Figure 9.3. The subtree rooted at node r is mapped onto 8 processes. We assume that a step of proportional mapping is accepted and yields the following distribution: the subtree rooted at s_1 is given 1.5 processes, the subtree rooted at s_2 is given 4.2 processes and s_3 is given 2.3 processes. We enforce the following mapping:

> P_0 works full time on the subtree rooted at s_1.
>
> P_1 works 50% (in terms of memory) on the subtree rooted at s_1 and 50% on the subtree rooted at s_2.

[2] This condition is not new and can be integrated in the basic memory-aware mapping, but we had not mentioned it before for simplicity.

P_2, P_3 and P_4 work full time on the subtree rooted at s_2.

P_5 works 70% on the subtree rooted at s_1 and 30% on the subtree rooted at s_2.

P_6 and P_7 work full time on the subtree rooted at s_3.

We recall that since a step of proportional mapping was accepted, the three subtrees are processed in parallel. By saying that a process works part time on two subtrees, we mean that it is allowed to work on any node of these subtrees at any time. In order to respect the memory constraint, we have to modify the way parallel nodes are distributed. For example, consider node s_3. 2.3 processes work on that node; P_6 and P_7 work full time while P_5 works part time. When s_3 is processed, it is not split in three equal shares; P_5 is given only $\frac{0\ 3}{2\ 3} = 13\%$ of the memory of s_3. Following this scheme at any step in the mapping process, no more than $\frac{0\ 3}{2\ 3}S_{s_3}$ is used on P_5, at any node in the subtree rooted at s_3. Note that there is no assumption on the scheduling of P_5 in the two subtrees rooted at s_2 and s_3; P_5 can interleave tasks of these two subtrees in any order. Similarly, no more than $\frac{0\ 7}{4\ 2}S_{s_2}$ is used on P_5 at any time in the traversal of the subtree rooted at s_2. In the end, no matter how the independent traversals of these two subtrees take place, no more than $\frac{0\ 7}{4\ 2}S_{s_2} + \frac{0\ 3}{2\ 3}S_{s_3} \leqslant 0\ 7M_0 + 0\ 3M_0$ (the step of proportional mapping is accepted) $\leqslant M_0$ is used on P_5.

In the end, any node in the tree is given a set of processes where at most two processes work part time. We refer to these processes as *supplementary processes*; they work part time on two sibling subtrees that are processed in parallel. This scheme allows us to use the decimal number of processes provided by each accepted step of proportional mapping and to enforce memory constraints. We can also adapt the computation of memory estimates so that they take this modified distribution of parallel nodes into account. Since a supplementary process can work on two independent subtrees (i.e., that are processed in parallel), it is harder to get an accurate estimate since we cannot predict the order in which the tasks assigned to this process will be performed. We use the upper bound given by the sum of the peaks of these subtrees divided by their respective numbers of processes times the share of the process we consider (e.g. for P_5, $\frac{0\ 7}{4\ 2}S_{s_2} + \frac{0\ 3}{2\ 3}S_{s_3}$). This might result in slightly pessimistic estimates.

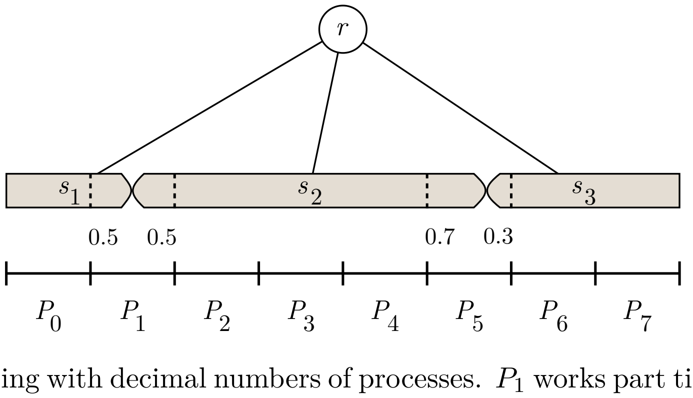


Figure 9.3: Mapping with decimal numbers of processes. P_1 works part time on the subtree rooted at s_1 and the subtree rooted at s_2; P_5 works part time on the subtree rooted at s_2 and the subtree rooted at s_3. Therefore, node s_2 has two *supplementary processes*, namely P_1 and P_5.

One can notice in the figure that processes are seen as consecutive intervals; every node is assigned to an interval of processes, i.e., the mapping gives to every node two bounds. If the difference between these two bounds is larger than two, then the two processes corresponding to the bounds will be extra processes, while the processes in the middle work full-time on the node. For example, in the figure, P_2, P_3 and P_4 work full-time on s_2.

9.3.2 Granularities

We showed that we allow processes to work partly on a given subtree. However, we want to avoid extreme cases where, for example, 2.0001 processes are assigned to a subtree, or where a very small subtree is assigned to two processes that work part time on it. We enforce some minimum granularities, in order to avoid these extreme cases that can unnecessarily increase the amount of communication or lead to a loss in performance. We have the following two rules:

> If the number of processes given to a subtree is smaller than a threshold $tol_{single} < 1$ and the mapping is such that two processes work on this subtree, the subtree is preempted on the process that has the largest share. This is illustrated in Figure 9.4.
>
> For a given subtree, if the share of work given to a supplementary process working on that subtree is smaller than a threshold tol_{work}, this process is removed from the list of processes working on that subtree. This is illustrated in Figure 9.5.

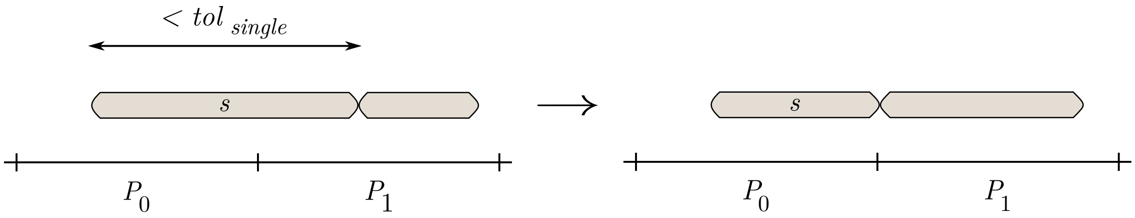


Figure 9.4: If the number of processes given to a subtree is smaller than tol_{single}, this subtree is not allowed to be processed on two processes. It is preempted on the process that holds the major part.

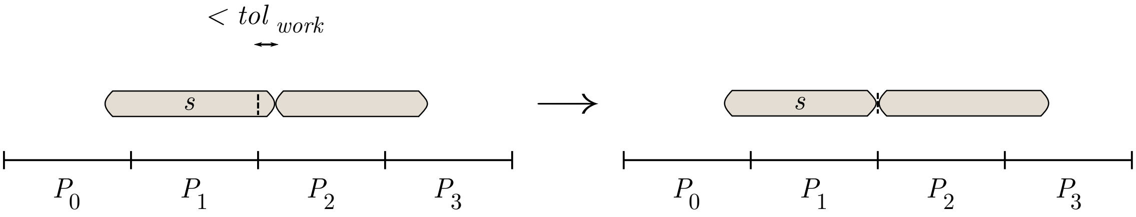


Figure 9.5: A process is not allowed to dedicate less than tol_{work} of its workload to a subtree.

Note that when one of these rules applies, the memory usage of the concerned processes is modified, and the memory constraint can be (hopefully marginally) violated. When the first rule applies, the memory usage of the only impacted process is multiplied by $1 + tol_{single}$ at worst. Similarly, when the second rule applies to a subtree, the memory usage of the processes working on that subtree is multiplied by $1 + tol_{work}$ at worst. We choose to introduce a relaxation parameter $relax_G$ (G for granularities) that we use when checking the memory constraint at each step of the mapping; at a given node $\mathcal{N}_i$ mapped onto p_i processes, instead of checking $\frac{S_i}{p_i} \quad M_0$, we check

$$\frac{S_i}{p_i} \quad \frac{M_0}{relax_G}$$

Using $relax_G > 1 + tol_{single} > 1 + tol_{work}$ should be enough. In practice, tol_{single} and tol_{work} are small (e.g. 0.10) thus their in impact on the memory usage is expected to be limited.

9.3.3 Other relaxations

In our description of the memory-aware mapping and its variant, we assumed that it was possible to perfectly distribute the memory associated with a node to a set of processes. Unfortunately, depending on the implementation choices of each solver, this might not always be possible. For example, we recall that in MUMPS, parallel nodes (Type 2 nodes) are distributed following a row-wise one-dimensional partitioning. In the unsymmetric case, the so-called master process is in charge of the fully-summed block and the U_{12} block, while the so-called slave processes are in charge of the L_{21} block and the contribution block. It is possible to perfectly (or almost perfectly) distribute the L_{21} block and the contribution block to the slaves, but the share of the node given to the master process might be completely different. The same problem arises in the symmetric case. In case the share of the master process is larger than the share of the slave processes, the node can be transformed into a chain of nodes that yields a more balanced partition. We refer this to as splitting the node; this is presented in detail in the next chapter. In the case where the share given to the master process is smaller than the share of the slave processes, not much can be done (in particular, in MUMPS, the master process of a node is not allowed to be a slave process of the same node). In any case, even if another distribution is used (e.g. a 2D block cyclic partitioning) it is almost always impossible to guarantee a perfectly balanced distribution. This imbalance might prevent the memory-aware mapping from respecting the memory constraint. We choose to introduce a relaxation parameter $relax_U \geqslant 1$ (U for unbalanced) that we use when checking the memory constraint at each step of the mapping; at a given node $\mathcal{N}_i$ mapped onto p_i processes, instead of checking $\frac{S_i}{p_i} \quad M_0$, we check

$$\frac{S_i}{p_i} \quad \frac{M_0}{relax_U}$$

We note that the most extreme case is a node with a very small number of fully-summed variables and a very large contribution block; if such a node is mapped onto two processes, then the memory load for the slave process can be arbitrarily larger than for the master process. Therefore, there is potentially a factor of two between the maximum memory load and the average memory load at this node. This means that we should set $relax_U \geqslant 2$ in order to ensure that the memory constraint can be met. This might be very restrictive since it could force the mapping to be very close to an all-to-all mapping. However, in practice, such nodes are not met very often, in particular at the top of the tree. We observed that a relaxation parameter between 1 and 2 is enough in many cases.

Some other sources of imbalance motivate the need for relaxation parameters. For example, delayed pivoting can dynamically modify the structure of the tree during the factorization and therefore change the memory usage of each process. This can introduce some imbalance that we need to anticipate when mapping the tree; we can introduce another relaxation parameter $relax_P$ (for pivoting). Similarly, if we choose to relax our scheduling strategy (e.g. in order to compensate for the natural load imbalance that can arise during a parallel execution), it has to be anticipated in the mapping process; we can introduce a relaxation parameter $relax_A$ (asynchronous). In the end, we have a relaxation parameter that is the product of the different parameters we mentioned ($relax = relax_G \quad relax_U \quad relax_P \quad relax_A$) and the constraint to be checked at each step of the mapping is

$$\frac{S_i}{p_i} \quad \frac{M_0}{relax}$$

Chapter 10

Performance aspects

In this chapter we describe some implementation aspects and performance issues. In Section 10.1, we describe our implementation of the scheduling constraints used in the memory-aware mapping and the variant with groups. In Section 10.2, we highlight some performance issues that we met in parallel nodes. We mentioned that the memory-aware mapping tends to assign more processes than the proportional mapping to the nodes, especially at the top of the tree. This revealed a weakness in one of the communication patterns used in MUMPS. We have improved this pattern, and this enhanced the performance of the solver (not only in the context of the memory-aware mapping). This work is described in Section 10.2. We also mentioned in the previous chapter that it is important to be able to split nodes in equal shares in order to guarantee the memory constraint when the memory-aware mapping is used. In MUMPS, whenever a parallel node is processed, there is a difference between the master process and the slave processes; if the share of the master process is too large, the node can be transformed (*split*) into a chain of nodes. In Section 10.3, we describe several splitting strategies that enable to leverage the performance of the solver. Once again, the ideas we present are interesting not only in the context of the memory-aware mapping.

10.1 Enforcing serializations

As described in the previous chapter, the memory-aware mapping strategy we suggest requires to enforce some precedence constraints during the factorization. In the basic algorithm, we need to set constraints that prevent a given subtree $\mathcal{T}(i)$ from starting before another subtree $\mathcal{T}(j)$ (rooted at one of its siblings j) is completely processed. This means that the leaves of $\mathcal{T}(i)$ cannot start before node j is processed. In the variant introduced in Section 9.2, a given subtree may have to wait for a group of subtrees.

10.1.1 Simple case

Here we describe how we enforce serializations in the basic algorithm. In Algorithm 9.1, we set precedence constraints using an array $PREV$; $PREV(i) = j$ means that node i cannot start before node j is completely processed. Therefore, we use a global state information mechanism such that, every time a node is completed, the master process of that node informs all the other processes that the node has been processed. All the processes hold an array $DONE$ (of size the number of nodes in the tree) such that $DONE(i)$ is true if node i is known to be finished. When a process receives the information that a node i is finished, it sets $DONE(i)$. Note that, for Type 2 nodes, the master process needs to know that all its slave processes have finished their task before it can inform the other processes; therefore, slave processes send a message to the master process when they have processed their share of the node.

For a given process, the selection of the next task to be processed is modified; instead of selecting the task at the top of the pool, a process selects the first task (node) i such that:

$PREV(i) = 0$ (i is not constrained to wait for another node)

or

$PREV(i) = 0$ and $DONE(PREV(i))$ (the predecessor of i is processed already)

10.1.2 Group dependencies

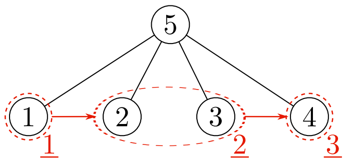
In the variant of the memory-aware mapping that relies on groups (we recall that the idea is to partition a set of siblings into groups; each group is processed using a proportional mapping, and groups are serialized), a given node has to wait for several nodes (a group) to be processed before it can start. We slightly change the meaning of $PREV$ and introduce two arrays, $SIZE$ and GRP, that provide the structure of the groups:

$PREV(i) = \quad g$ means that i is constrained to wait for all the nodes in group g to be processed before it can start.

$GRP(i) = g$ means that node i belongs to group g.

$SIZE(g)$ is the size of group g. $SIZE$ is initialized before the beginning of the factorization and duplicated on every process. Then, every time a process receives the information that a node i is completed, $SIZE(GRP(i))$ is decreased (if i belongs to a group, i.e., if $GRP = 0$ by convention).

We illustrate this in Figure 10.2, where a set of four siblings is partitioned into three groups: 1 , 2 3 and 4 .



(a) A tree with groups.

$$PREV = \quad 0 \quad 1 \quad 1 \quad 2 \quad 0$$

$$GRP = \quad 1 \quad 2 \quad 2 \quad 3 \quad 0$$

$$SIZE = \quad 1 \quad 2 \quad 1$$

(b) Dependencies structures.

Figure 10.2: An example of tree with groups (a) and the associated structures for the scheduling (b).

Using these structures, the mechanism for the selection of tasks is the following: every time a process activates a new task, it chooses the first task i from the pool such that:

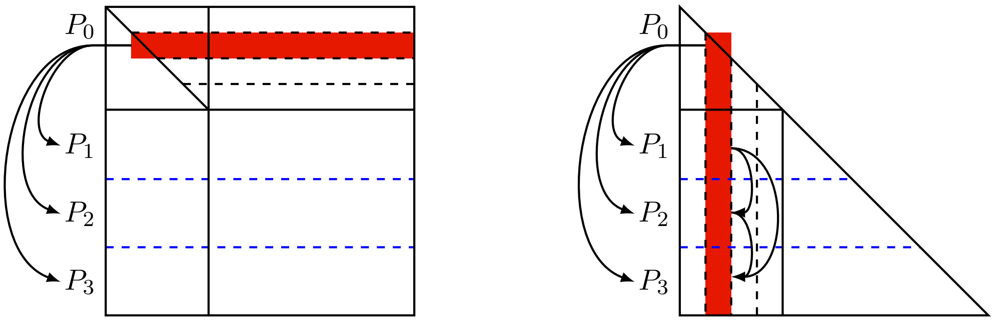
$PREV(i) = 0$ (i is not constrained)

or

$PREV(i) < 0$ and $SIZE(GRP(PREV(i))) = 0$ (the group i depends on is completed)

10.2 Communication patterns in parallel nodes

As described in the previous chapter, the memory-aware mapping tends to increase the number of processes associated with each node, at least at the top of the tree (we assess this behaviour with practical experiments in the next chapter). This raised several performance issues when we carried out experiments within MUMPS. In particular, we found that, in the unsymmetric case, the communication pattern used to send the rows held by the master process to the slave processes in Type 2 nodes was a bottleneck. We describe this pattern in Figure 10.3(a); the master process performs a pipelined factorization and sends pieces of factors (diagonal blocks of L and rows of U) to all its slaves. In the symmetric case, the pattern is different as there are communications between slaves; each slave receives some data from all the previous slaves and sends some data to the next slaves (cf. Figure 10.3(b)). The master-to-slaves communication pattern is the same as in the unsymmetric case, but it does not represent a bottleneck in practice since the volume to be sent from the master process to its slaves is significantly lower than in the unsymmetric case. We observed that slave-to-slave communications perform rather well in practice; therefore, we focus on the unsymmetric case and the master-to-slaves communication pattern. In this section, we describe how the master-to-slaves communication pattern is implemented within MUMPS, and we suggest another strategy. We emphasize some implementation issues and provide some experimental results.



(a) Unsymmetric case.

(b) Symmetric case.

Figure 10.3: Communication pattern used within Type 2 nodes.

10.2.1 Baseline scheme

As described above, the master-to-slaves communication pattern is a one-to-many operation where the master process sends the same piece of data to all the slave processes. As we consider as asynchronous design (cf. Chapter 7), we wish this communication to be non-blocking, so that the master process can compute a block of factors while the previous block is sent; therefore, we wish to have an *asynchronous broadcast*. Non-blocking collective operations are not part of the MPI standard (MPI-2) at the time of writing; some libraries, such as LibNBC [53] provide a prototypical implementation of non-blocking collective operations. However, the semantic of these operations requires that all the processes involved in the collective operation call the same function; for example, if one wants to perform an asynchronous broadcast (`ibcast`), then all the processes have to call `ibcast`. This is quite constraining for our asynchronous approach which is written so that any process can, at any time, receive and treat any kind of message and task. Remember that, as illustrated in Algorithm 7.1, any time a process is idle, it checks if it can receive a message (that may provide some work to be done) by calling `iprobe` to check whether a message is available (note that an `irecv` has been posted before in general); we want to keep this generic approach. Therefore, a handmade non-blocking collective operation where all the processes use the regular `isend` and `irecv` operations was designed in the very first releases of MUMPS.

The implementation of the master-to-slaves communication pattern within MUMPS is the following:

The master process copies the data to be sent in a buffer.

The master process performs a loop of calls to `isend` to send the data to every slave.

At some point, every slave process will receive the message (`irecv`) and process the data.

Note that the same buffer is used for all the calls to `isend`, meaning that, in order to free memory in the buffer, the completion of all the `isend`s must be checked. Also, contrary to some other communication patterns in MUMPS (e.g. for contribution blocks), stripes of factors are sent in a single message.

We found that this implementation was a bottleneck for Type 2 nodes, especially on large number of processes. In order to assess this out of the context of MUMPS, we wrote a benchmark code that mimics the communications occurring in an unsymmetric Type 2 node:

The master sends a block of data to all the slaves, then an other block, and so on until all blocks are sent; then it waits till the end. Sends are non-blocking.

Slaves receive each block. Receives are non-blocking in MUMPS but blocking in the benchmark; this does not change the result since no computation is done in the benchmark.

Speed is computed as the total volume of communications divided by the wall time. We provide some results in Table 10.1, where we simulate the communications occurring in a node of size 64000 with 1000 fully-summed variables; the master process sends 30 messages of size about 160 MB to every process. We provide results on two systems with different architectures from our experimental set, namely `Hyperion` and `Hopper`.

Processes	Volume	Time	Speed
(total/per node)	(GB)	(s)	(GB/s)
2/1	0.49	0.28	1.76
4/1	1.48	0.94	1.57
8/1	3.44	2.15	1.60
16/1	7.38	4.80	1.54
32/2	15.24	9.50	1.60
64/4	30.98	19.50	1.59

(a) `Hyperion` system, 16 nodes.

Processes	Volume	Time	Speed
(total/per node)	(GB)	(s)	(GB/s)
2/1	0.49	0.12	4.08
4/1	1.48	0.32	4.63
8/1	3.44	0.73	4.71
16/1	7.38	1.50	4.92
32/2	15.24	3.22	4.73
64/4	30.98	6.58	4.71

(b) `Hopper` system, 16 nodes.

Table 10.1: Benchmark of the baseline communication pattern in the unsymmetric case, for a node with *nfront* = 64000 and *npiv* = 1000.

One can easily notice that the aggregated speed is almost constant, regardless of the number of processes and the architecture[1]. Using only one or several MPI processes on a given node does not seem to have much influence either. This behavior is of course disappointing, as we would like to see the aggregated speed increase with the number of processes.

Using a very simple performance model, we demonstrate that this behaviour is a severe limitation to the performance. We focus on the unsymmetric case and compute the maximum number of processes that can be used to overlap communications with computations, i.e., for a number of processes greater than this maximum number, the total time will be dominated by the time for communications. We consider a node of size *nfront* with *npiv* fully-summed variables and *ncb* variables in the contribution block, processed on p processes (i.e., there are $p \quad 1$ slave processes). The master process sends stripes of factors of height b (b is the block size for the blocked factorization) and width $\mathit{nfront} \quad (j+1) \quad b$ (with j the step in the blocked factorization process). Thus, at each step, the communication time for sending a piece of factors to any slave is, ignoring latency:

$$t_c = \frac{(\mathit{nfront} \quad (j \quad 1) \quad b) \quad b}{v_c}$$

with v_c the communication speed. Once the message is received by a slave process, the slave updates its part of the L factors and the contribution block; this consists of a triangular solution on a matrix of size $b \times b$ with $\frac{ncb}{p-1}$ right-hand sides and an update of a $\frac{ncb}{p-1} \times (\mathit{nfront} \quad (j \quad 1) \quad b)$ matrix (note that this corresponds to the block on the right of the columns that have been computed with the triangular solution: this include pieces of factors, not only the contribution block). Therefore, the computation cost is $\frac{b^2 \quad ncb+2b \quad ncb \quad (\mathit{nfront}-(j-1) \quad b)}{p-1}$, thus the computation time is

$$t_f = \frac{b^2 \quad ncb + 2b \quad ncb \quad (\mathit{nfront} \quad (j \quad 1) \quad b)}{v_f \quad (p \quad 1)}$$

with v_f the computational speed (flops rate). In order to get overlapping between communications and computations, we need $t_c \leqslant t_f$:

$$\begin{aligned}
t_c \leqslant t_f \qquad & \frac{(ncb \quad (j \quad 1) \quad b) \quad b}{v_c} \leqslant \frac{b^2 \quad ncb + 2b \quad ncb \quad (\mathit{nfront} \quad (j \quad 1) \quad b)}{v_f \quad (p \quad 1)} \\
& \frac{(\mathit{nfront} \quad (j \quad 1) \quad b) \quad b}{v_c} \leqslant \frac{2b \quad ncb(\mathit{nfront} \quad (j \quad 1) \quad b)}{v_f(p \quad 1)} \\
(b \quad 2(\mathit{nfront} \quad (j \quad 1) \quad b)) & \\
& p \leqslant 1 + 2\frac{v_c}{v_f} \quad ncb
\end{aligned}$$

As expected, the maximum number of processes depends on the communication speed; the higher the communication speed, the higher the maximum number of processes.

Now we use our experimental model for the communication speed of the loop of `isend` pattern. We denote v_c^{tot} the aggregated speed; here v_c^{tot} is a constant, that we denote $\ _c$. We have $\ _c \quad 1 \ 6$ GB/s on `Hyperion` and $\ _c \quad 4 \ 7$ GB/s on `Hopper`. Assuming the aggregated speed is equally distributed among the processes (in other words, assuming

[1] The `Hyperion` system has a hypercube topology and relies on the Infiniband connection technology, while the `Hopper` system has a 3D-torus topology and uses the Cray Gemini network.

that they receive the stripes of factors at the same time), the communication speed for one master-to-slave communication is $v_c = \frac{}{p-1}$. Therefore, the above equation becomes:

$$t_c \leqslant t_f \qquad (p \quad 1)^2 < \quad ncb \qquad (\text{with} \quad = \frac{2 \ _c}{v_f})$$

$$p \leqslant 1 + \overline{\quad ncb}$$

This upper bound is quite low in practice; consider for example the `Hyperion` system. Using a single-threaded BLAS, one can assume $v_f = 8$ GFlops/s (in double precision arithmetic). This yields $\ = 0\ 05$. With $ncb = 16000$, it yields $p \leqslant 29$. This is very low; consider for example a 3D regular grid of size 128^3; the first separator (the root node) is of size $128^2 = 16384$ which means that the children of the root node have a contribution block of size $ncb = 16384$. Given the size of the problem, it is not unreasonable to use 64 processes or more. If a memory-aware mapping is used and serializes the subtrees rooted at the children of the root node, at least 64 processes are assigned to work on the children of the root node. This is much larger than the upper bound, which means that at these nodes, the computation time will be dominated by the time for communications, which is not desirable. This demonstrates that a one-to-many pattern based on a loop of `isend` is a serious bottleneck.

10.2.2 A tree-based asynchronous broadcast

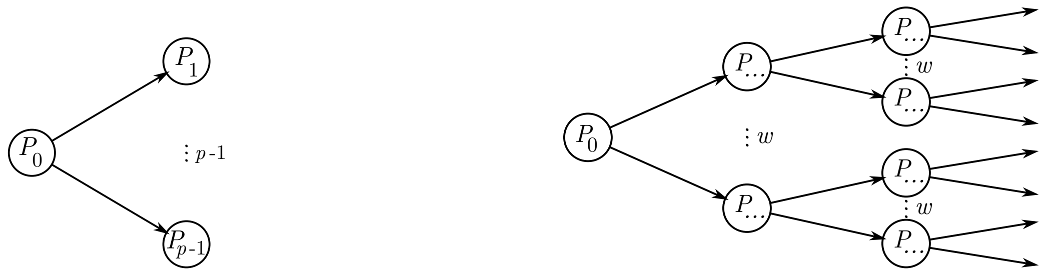
We suggest to replace the above-mentioned pattern with a tree-based scheme: communications follow a tree of width w, where the master process sends its data to w processes (instead of sending data to all the slaves), which in turn send theses pieces to w processes, and so on. At each level, a process sends data to w processes using the baseline algorithm, that is, a loop of `isend`. The idea is that once the first level (children of the root node) is reached, truly parallel communications take place. This is illustrated in Figure 10.4. In the following, we call *terminal processes* and *relay processes* the processes that correspond to leaf nodes and internal nodes in the broadcast tree respectively. Relay processes have to relay to their children the blocks of factors they receive from their parent, while leaf processes simply receive blocks. Note that this scheme is one of the options proposed in the `ibcast` routine implemented in LibNBC.

We provide an upper bound on the speed yielded by this pattern; the time for traversing the tree is the time for going from one level to another (a hop) times the number of levels. The number of levels (i.e., the depth of the tree) is approximately $\log_w p$; the time for a hop is the time for a 1-to-w regular transfer, i.e. $\frac{\text{Volume 1-to-w}}{\ _c \ w} = \frac{2\text{Total volume } (p-1)}{\ _c \ w}$; therefore, the total speed v_c^{tot} should be

$$v_c^{tot} = \ _c \frac{p \quad 1}{\log_w p}$$

Thus, the aggregated speed becomes proportional to the number of processes; it is $\frac{p-1}{\log_w p}$ times larger than the speed of the baseline algorithm.

We assessed the binary case ($w = 2$) using a benchmark that works along the lines of Algorithm 10.1. Experiments have been carried out on 32 nodes of `Hyperion` and 32 nodes of `Hopper`. On `Hopper`, each node has 24 cores, and experiments have been performed up to 32 × 24 = 768 cores; for `Hyperion`, the prediction uses $\ _c = 1\ 8GB/s$, and for `Hopper`, $\ _c = 4\ 7GB/s$.



(a) Baseline algorithm.

(b) Tree-based algorithm.

Figure 10.4: One-to-many communication pattern used for sending blocks of factors from the master process to the slave processes. In the baseline algorithm, the master sends data to every process (with a loop of `isend`) (a). We suggest a tree-based pattern, where every process corresponding to an internal node of the tree sends data to w other processes (b). Note that $w = p \quad 1$ is equivalent to the baseline algorithm.

Processes (tot/node)	Volume (GB)	Time (s)	Speed (GB/s)	Predicted (GB/s)
4/1	1.48	0.57	2.60	2.70
8/1	3.44	0.65	5.29	4.20
16/1	7.38	0.75	9.84	6.75
32/1	15.24	0.90	16.93	11.16
64/2	30.98	1.50	20.65	18.90
128/4	62.45	2.20	28.39	32.66

(a) `Hyperion` system, 32 nodes.

Processes (tot/node)	Volume (GB)	Time (s)	Speed (GB/s)	Predicted (GB/s)
32/1	15.24	0.48	31.75	29.14
64/2	30.98	0.95	32.61	49.35
128/4	62.45	1.65	37.85	85.27
256/8	125.39	2.10	59.71	149.81
512/16	251.28	3.07	81.85	266.86
768/24	377.16	3.10	121.67	400.54

(b) `Hopper` system, 32 nodes.

Table 10.2: Experiments with a binary tree-based communication pattern.

The differences between the model and the actual speed probably come from the fact that during a 1-to-2 transfer, speed is not equally distributed among the two receivers (in other words, messages do not arrive at the same time); therefore the longest path in the tree is not necessarily the theoretical critical path, and it is hard to predict what can happen. Furthermore, it may happen that a node has only one child (this is the case in these experiments: since p is a power of 2, there is only one node at depth $\log_2 p$); therefore some 1-to-1 transfers take place instead of 1-to-2 transfers. The large differences on Hopper for large number of processes might come from shared-memory effects: indeed, with 24 processes on a single node, $_c$ drops from 4.7 GB/s to 1.5 GB/s; with $_c = 1\ 5$ GB/s and $p = 768$, the model predicts 127 GB/s, which is very close from the 121.67 GB/s observed in practice. These results show that the tree-based pattern can deliver very large speeds, especially for large number of processes, and it constitutes a significant improvement over the baseline pattern.

Now we assess how this new communication pattern influences the performance of Type 2 nodes. As in the previous section, we compute the maximum number of processes for which computations overlap communications. Remember that regardless of the communication pattern, this maximum number is

$$p < 1 + 2\frac{v_c}{v_f} \quad ncb$$

Algorithm 10.1 Tree-based unsymmetric communication pattern benchmark for the binary case $w = 2$ (local view of a given process).

```
Input: a set of pieces of data to be sent from process 0 to $p \quad 1$ processes.
Work array: desc, the set of descendants of each process in the tree.

/* Phase 1: each process is given the set of its descendants in the tree */
/* 1.1: each process receives its set */
if my rank = 0 then
    My set of descendants = {1,...,p}
else
    Blocking receive of my set of descendants (desc[1..nb])
end if
/* 1.2: each process defines its set of children */
Non-blocking send of desc[2..nb/2] to my first son (desc[1]) (if any)
Non-blocking send of desc[nb/2+2..nb] to my second son (desc[nb/2+1]) (if any)

/* Phase 2: communications take place */
for each piece of data do
    if my rank != 0 then
        Blocking receive of a piece of data
    end if
    Non-blocking send of the data to my first child (if any)
    Non-blocking send of the data to my second child (if any)
end for
```

We now have $v_c = \frac{v_c^{tot}}{p-1} = \frac{c}{\log_w p}$ (instead of $v_c = \frac{c}{p-1}$). Thus, the equation becomes

$$(p \quad 1)\ \log_w p \ < \quad ncb$$

This equation is not solvable explicitly, but we can solve the close-by equation $p \log_w p < \quad ncb$ by means of the Lambert function W (reciprocal of $x \qquad xe^x$):

$$p \log_w p < \quad ncb \qquad\qquad p < e^{W(\quad c \log_e w)}$$

Any $W(x)$ with $x > \frac{1}{e}$ is computable numerically as $\lim_n \qquad w_n$ with $w_{n+1} = w_n \quad \frac{w_n e^{w_n} - x}{(1+w_n)e^{w_n}}$ and $w_0 = 1$. On the same example as above, we obtain $p = 118$ instead of $p = 29$. This shows that this communication pattern can significantly leverage the performance of Type 2 nodes compared to the baseline strategy.

We implemented this tree-based communication pattern within MUMPS. We describe some implementation issues in the next section; here we provide some experimental results. We measure the time for factorizing a node with $npiv = 32000$ and $ncb = 32000$ on 64 processes (we use 32 nodes of the `Hyperion` system). The node is split into a chain of 30 nodes (cf. the next section). We compare tree-based communication patterns with different widths: $w = p \quad 1$ (i.e., the baseline algorithm), $w = 2$ and $w = 8$. We also compare the performance using a single-threaded BLAS and a 4-way multithreaded BLAS; this enables to assess the weight of the computations over the total time. The results are reported in Table 10.3; we notice that the tree-based scheme with $w = 2$ and $w = 8$ significantly increase the performance, especially when a multithreaded BLAS is used, which shows that the influence of the communications is reduced.

Tree width	Time (s)	
	1-way threaded BLAS	4-way threaded BLAS
p 1	596	468
2	403	167
8	401	167

Table 10.3: Experiments with the tree-based communication pattern within MUMPS.

10.2.3 Implementation issues

Implementing the tree-based asynchronous broadcast we suggest in a dynamic asynchronous code is far from being straightforward. We highlight three difficulties: firstly, we show that messages can overtake, then we show two potential sources of deadlocks. We describe how we solve these problems.

The first problem we emphasize is a situation where messages can overtake. Firstly we need to describe a central idea of our asynchronous approach. Anytime a process is unable to send a message because its send buffer is full (typically because a receiver is busy and does not consume messages), it tries to receive a new message and to treat the associated task (calling a routine that we call **Try__recv__and__treat**), hoping that this will unlock the situation. This exhibits a recursive behavior, since a new task may in turn require to send a message; if this fails once again, the process tries to receive and treat a new task (calling again **Try__recv__and__treat**), and so on, until the situation is unlocked. In the end, once the situation is unlocked (i.e., the send buffer of the process is no longer full and messages can be processed), the recursive calls to **Try__recv__and__treat** are unstacked and the tasks that were on hold are completed following the reverse order of their arrival. We illustrate this in Algorithm 10.2, where we show how **Try__recv__and__treat**, the routine that receives and handle messages, interacts with **Process__factors__block**, the routine that handles the messages corresponding to a block of factors.

Algorithm 10.2 Management of messages corresponding to block of factors.

```
Try__recv__and__treat(...)
Wait for availability of a message (blocking or non-blocking, probe,iprobe)
Check the tag of the message: if it is a block of factors, call Process__factors__block
...

Process__factors__block(...)
Extract the list of descendants in the broadcast tree, i.e., the processes the block must
  be relayed to.
try_again    true
while try_again do
    Try to send the block to all the descendants using isend
    if a problem is detected then /* The send buffer is full */
        Call Try__recv__and__treat(...)
    else
        try_again    false
    end if
end while
Process the block (triangular solution and matrix-matrix product)
```

Now we demonstrate that in the context of the tree-based communication pattern, this mechanism can lead to messages overtaking one another. Consider the example in Figure 10.5; the node is mapped onto three processes (the master process is P_0 and the slaves are P_1 and P_2). We assume a tree-based broadcast of width $w = 1$, i.e., a chain P_0 P_1 P_2. P_0 sends blocks of factors to P_1 which in turn sends them to P_2; if for some reason P_2 does not consume the messages from P_1 (typically, P_2 is busy working at another part of the tree), the send buffers of P_1 may be become full; we denote b_1 a block that cannot be sent. The asynchronous approach dictates that P_1 tries to receive and process a new task that might unlock the situation. Assume that this new task is another block of factors from P_0 that we denote b_2; once P_1 receives this block, it must process it (in our approach, we do not allow not to process a task, as this would require to store the data that are not processed in a temporary area). Therefore, P_1 tries to send this new block to P_2; assume that this time P_2 consumes the message. The block b_2 is received by P_2, and then (unstacking the recursive calls to **Try_recv_and_treat** and **Process_factors_block**), the block b_1 is finally received by P_2.

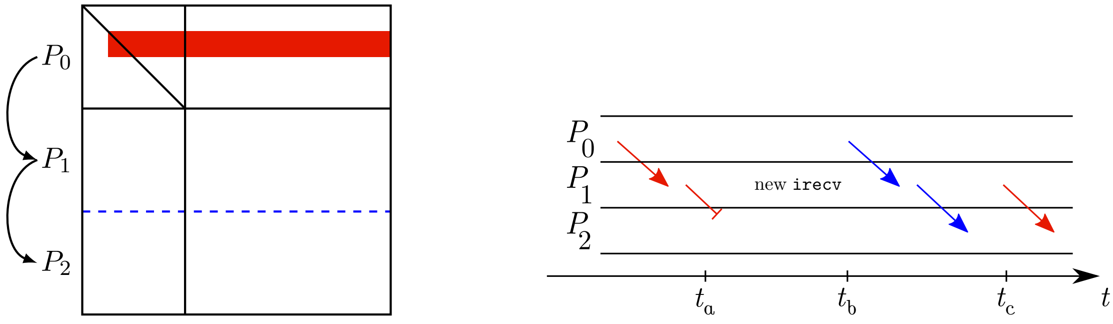


Figure 10.5: A simple example where messages can overtake one another; if P_1 cannot send a message to P_2 (time t_a), it calls `irecv` and receives another message from P_0 (time t_b) and sends it to P_2; in the end (time t_c), the messages arrive on P_2 in the wrong order.

The problem at this point is that blocks of factors must be received in the right order; indeed, in the pipelined factorization process, the updates associated with a block must be computed before the next block can be processed. Furthermore, as we mentioned above, we cannot afford to store all the blocks in a temporary area before performing the updates; blocks must be received in the right order and consumed on the fly. In order to prevent this situation, we propose to forbid a process that enters in recursive calls to **Try_recv_and_treat** to receive and process new blocks of factors. More precisely, we forbid this process to receive and process blocks that have to be relayed to other processes; however, it can receive and process terminal blocks. This is useful if the process we consider works on several nodes and is a terminal process for one of the nodes. Therefore, we need to use two tags (message identifiers) in order to distinguish between blocks to be relayed and terminal blocks. We adapt **Try_recv_and_treat** and **Process_factors_block** as shown in Algorithm 10.3. The variable *allow_relay* is used to know whether the process is allowed to receive and treat blocks to be relayed. Note that the variable *allow_relay_set* is used to guarantee that the process does not reset *allow_relay* before the end of the recursion (i.e., before all recursive calls to **Try_recv_and_treat** are unstacked).

We now show that there are two potential sources of deadlocks. We illustrate them in Figure 10.6 and Figure 10.7; in both cases, we assume that the communication pattern is a tree-based broadcast with $w = 1$, i.e., a chain. In Figure 10.6, the deadlock comes from the ordering of the slaves; P_1 is the parent of P_2 in the broadcast tree associated with

Algorithm 10.3 New management of messages corresponding to block of factors.

```
Try_recv_and_treat(allow_relay)
if allow_relay then
    Receive any message (iprobe)
else
    Receive any message except a block of factors to be relayed (iprobe)
end if
Check the tag: if it is a block of factors, call Process_factors_block(allow_relay)
...

Process_factors_block(allow_relay)
Extract the list of descendants in the broadcast tree, i.e., the processes the block must
    be relayed to.
allow_relay_set     false
try_again     true
while try_again do
    Try to send the block to all the descendants using isend
    if a problem is detected then /* The send buffer is full */
        if allow_relay then
            allow_relay     false /* Non-terminal blocks of factors are no longer allowed */
            allow_relay_set     true
        end if
        Call Try_recv_and_treat(allow_relay)
    else
        try_again     false
    end if
end while
if allow_relay_set then
    allow_relay     true /* Non-terminal blocks are now allowed */
end if
Process the block (triangular solution and matrix-matrix product)
```

node 1, while it is the child of P_2 in the broadcast tree associated with node 2 (see the caption). This implies that the relay processes on a path of the broadcast tree of a given node should follow a global order. In particular, one can show that the deadlock cannot happen if there is only one level of relay processes, that is, if there are only three levels in the tree: the root node, its children, that are relay processes, and its grandchildren, that are terminal processes i.e. leaf nodes. Indeed, in the example, the deadlock comes from the fact that, at some point, P_1 and P_2 no longer accept blocks of factors to be relayed; however, if we remove P_3 and P_5 from the example, P_1 and P_2 become terminal processes. When P_1 can no longer sends messages from P_2, it starts refusing blocks to be relayed; however, at node 2, it receives blocks that do not need to be relayed (while this is not the case in the initial example), and this unlocks the situation.

In practice, we can easily force the broadcast tree to have only one level of relay processes by setting $w = \ \overline{p}$. We have assessed in the previous section that this setting can provide good performance; in Table 10.3, $p = 64$ and setting $w = \ \overline{64} = 8$ provides similar performance as $w = 2$.

In Figure 10.7, the deadlock is due to the fact that process P_6 is the master of a node (node 2) and has to relay blocks of factors at another node (node 1). We can forbid

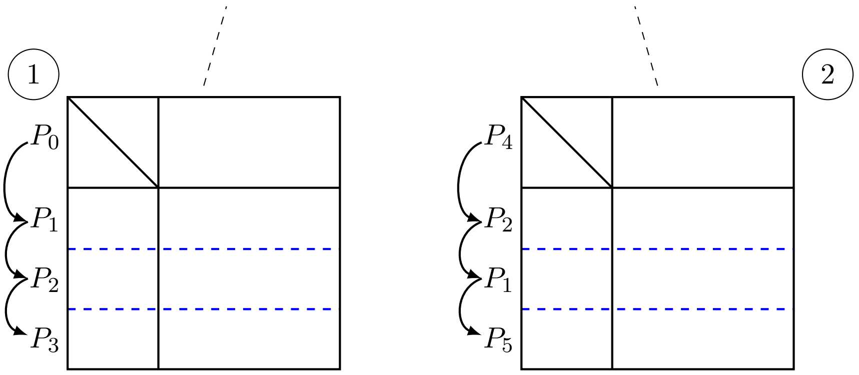


Figure 10.6: A simple example of deadlock: P_2 is busy receiving the messages from P_4 (node 2), thus the send buffer of P_1 becomes full (node 1). P_1 no longer accepts blocks of factors to be relayed, thus the send buffer of P_2 becomes full (node 2). Therefore P_2 no longer accepts blocks of factors to be relayed and no longer consume the messages from P_1 (node 1). This yields a deadlock (between P_1 and P_2).

such situations by constraining the selection of the slave processes of Type 2 nodes; only processes that are guaranteed not to become the master of a Type 2 node before the end of the current node can be selected as relay processes (this information can be initialized during the static mapping and updated on the fly during the factorization). In the case where this is too constraining (i.e., too many processes are banned from the list of potential relay processes), we can set $w = p \quad 1$ to revert to the baseline strategy.

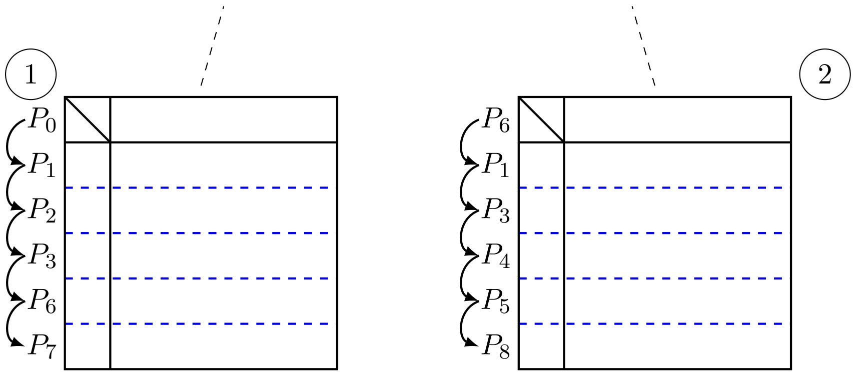


Figure 10.7: A second example of deadlock: P_6 is busy sending messages to P_1 (node 2), thus it does not consume the messages from P_3 and the send buffer of P_3 becomes full (node 1). Therefore P_3 no longer accepts blocks of factors to be relayed *at both nodes*; this blocks P_1 (node 2) and blocks P_6 in the end. Even if P_6 can still receive blocks from P_3, it cannot relay these blocks to P_7 since its send buffer is full. This yields a deadlock (between P_1 and P_6).

10.3 Splitting nodes

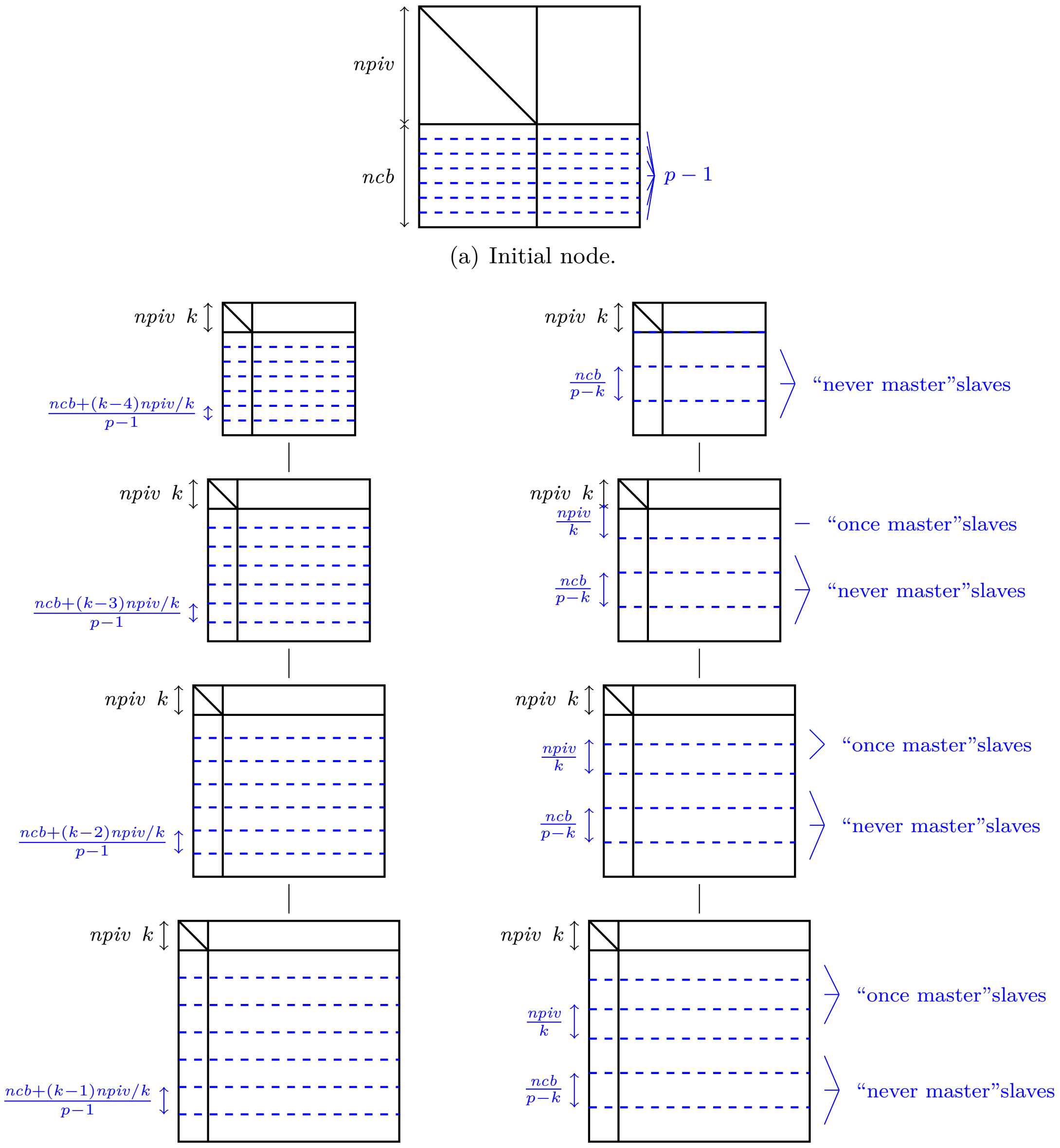
10.3.1 Idea and strategies

As mentioned in the previous chapter, it is important to be able to equally distribute a node on a set of processes; this is crucial in the context of the memory-aware mapping and often desirable in general. In MUMPS, only one process (the master process) holds the fully-summed block and the rows of U (in the unsymmetric case) of a given node; the slave processes share the rows corresponding to the contribution block. This might result in severe imbalances depending on the number of fully-summed variables, the size of the contribution block, and the number of processes to be used. Consider the unsymmetric case: every slave process holds a stripe of height about $\frac{ncb}{p-1}$ (there are p 1 slave processes; for the sake of simplicity, we ignore the fact that some slave processes could be supplementary processes as described in the previous chapter). If $npiv$ $\frac{ncb}{p-1}$, the master process has a much larger workload than the slave processes, which might be a bottleneck in terms of performance. This might also result in severe imbalances in memory usage that might prevent respecting the memory constraint in the context of the memory-aware mapping.

The idea used in MUMPS is to transform a Type 2 node into an equivalent chain. Given a number of parts k (that can be chosen according to different criteria), the node is transformed into a chain of k nodes with about $\frac{npiv}{k}$ fully-summed variables. The first node (the lowermost node) in the chain has the same size as the initial node; the last node is of size $ncb + \frac{npiv}{k}$. Note that here and in the following, the notation $npiv$ and ncb always refer to properties of the initial (non-split) node.

The baseline strategy (used in MUMPS) consists of mapping all the nodes of the chain on the same p processes. This is illustrated in Figure 10.8: the initial node (a) is transformed into a chain of k nodes (b); every node is mapped onto the p processes. The drawback is that, compared to the initial node, the volume of communication significantly increases since pieces of contribution blocks have to be communicated from one node of the chain to its parent node, because processes do not keep the same row indices when traversing the chain. Another strategy (that we refer to as the new strategy in the following) consists of enforcing a constrained mapping where processes hold the same rows indices all along the chain, such that there is no communication between nodes of the chain. This implies that some processes stop working at some point in the chain. This is illustrated in Figure 10.8(c). In this example, $k = 4$ (the node is split into four nodes) and $p = 7$ (i.e., there are six slave processes). The seven processes work on the first node of the chain, but only four of them work on the last node. Note that the lowermost slaves work during the whole traversal of the chain (we call them the never master slaves in the figure) while the others work as slaves at the bottom of the chain, then they become the master of a node, and they finally stop working (we call them the once master slaves in the figure). This scheme corresponds to a pipelined factorization of the node; this is (almost) equivalent to a scheme where several processes are allowed to be master processes and share the fully-summed part of the initial node. The drawback is that this scheme requires $k < p$ which might be too constraining; furthermore, if k is close to p, the number of processes working at the top of the chain might become very small and might yield a bottleneck; in this case, one can use a hybrid scheme where the chain is processed following the new strategy but is restarted from times to times, i.e., some nodes are redistributed on all the processes in order to reset to balance.

We report on some experiments in Table 10.4. One can notice that the volume of communication is significantly decreased. However, the communications that we suppressed are all-to-all communications (slave processes exchanging pieces of contribution block) are fast in practice and do not constitute a strong bottleneck, which is why the gains with this new splitting strategy are interesting but less spectacular than what is achieved with the asynchronous broadcast presented in the previous section, that addresses the problem of the master-to-slaves communications.


npiv
ncb
p − 1
npiv k
$\frac{ncb+(k-4)npiv/k}{p-1}$
$\frac{ncb+(k-3)npiv/k}{p-1}$
$\frac{ncb+(k-2)npiv/k}{p-1}$
$\frac{ncb+(k-1)npiv/k}{p-1}$
$\frac{ncb}{p-k}$
$\frac{npiv}{k}$
"never master"slaves
"once master"slaves

(a) Initial node.

(b) Split chain, baseline strategy.

(c) Split chain, new strategy.

Figure 10.8: Splitting of a Type 2 node. Dashes show how the contribution block of a node is distributed.

Strategy	Volume of com- -munications (GB)	Time (s)	
		1-way threaded BLAS	4-way threaded BLAS
Baseline	1134	591	468
New	625	484	367

Table 10.4: Experiments using 64 processes (32 nodes of the Hyperion system). The chain corresponds to a node with 32000 fully-summed variables and a contribution block of size 32000, which is split into 30 nodes. We compare the baseline strategy with the new splitting approach.

10.3.2 Performance models

We have not mentioned how the number of parts k is determined. In the context of the memory-aware mapping, the most cautious strategy consists of computing k such that all the processes have the same memory usage along the chain; consider for the baseline splitting strategy for the sake of simplicity. The peak of active memory of every process takes place at the first node of the chain; for the master of the first node, it is $\frac{npiv}{k} \quad nfront$ and for the slave processes of the first node, it is $\frac{ncb+(k-1)npiv \quad k}{p-1} \quad nfront$; these two quantities[2] are the same if and only if:

$$k = \frac{npiv}{nfront} \quad p$$

In a relaxed strategy where one wants to maximize the performance, k should balance the workload of the different processes. Here we briefly describe some performance models that provide, for a given node:

The maximum number of processes that can be used to overlap communications with computations (this is a simple extension of the model described in the previous section).

The number of parts that balances the workloads.

Firstly, we compute the computational cost of each kind of task:

Total cost: the total operation count is $\frac{2}{3}(npiv + ncb)^3 \quad \frac{2}{3}ncb^3$ as the factorization of the whole front minus the cost for the factorization of the contribution block.

Computational cost of master tasks:

Non-split node: $\frac{2}{3}npiv^3 + npiv^2 \quad ncb$ (factorization of the fully-summed part and update of the U rows).

Split in k parts (note that this does not depend on the splitting scheme, i.e., the baseline strategy or the new one): each node of the chain has $\frac{npiv}{k}$ fully-summed variables and $ncb+(k \quad i)\frac{npiv}{k}$ columns in the contribution block, with $i \quad 1 \quad k$ the number of the node in the chain (1 = the first/lowest, k = the last/highest). The total cost of master tasks is

$$\sum_{i=1}^{k} \frac{2}{3}\left(\frac{npiv}{k}\right)^3 + \left(\frac{npiv}{k}\right)^2 \left(ncb + (k \quad i)\frac{npiv}{k}\right)$$
$$= \frac{(npiv + 6ncb \quad k + 3npiv \quad k)\; npiv^2}{6k^2}$$

Computational cost of slave tasks:

No splitting: $\frac{npiv^2 \; ncb + 2npiv \; ncb^2}{p-1}$ (one triangular solution with a matrix of size $npiv$ with $\frac{ncb}{p-1}$ right-hand sides and one product of a $\frac{ncb}{p-1} \times npiv$ matrix by a $npiv \times ncb$ matrix).

[2] Note that we used simplified expressions: we assumed that every process could work in-place from one node to another; this actually not feasible with the baseline splitting strategy.

Baseline splitting: there are $p \quad 1$ slaves at each node and they have the same load (assuming a regular partitioning). The total computational cost of any slave process is

$$\begin{aligned}&\sum_{i=1}^{k} \quad \frac{npiv}{k} \quad ^{2} \frac{ncb+(k-i)\frac{npiv}{k}}{p-1} + 2\frac{npiv}{k} \; \frac{ncb+(k-i)\frac{npiv}{k} \quad ^{2}}{p-1} \\ =&\frac{npiv \quad 12ncb^2\cdot k^2+12ncb\cdot npiv\cdot k^2-6ncb\cdot npiv\cdot k+4npiv^2\cdot k^2-3npiv^2\cdot k-npiv^2}{6k^2(p-1)}\end{aligned}$$

New splitting: each slave j works on nodes $1 \qquad \min(j \; k)$; there are two kinds of slaves:

Once master slaves (slaves $1 \qquad k \quad 1$): they hold a stripe of height $\frac{npiv}{k}$, are active i times (i: number of the slave) as a slave, once as master and then are no longer active. They have different loads: slave 1 is the least loaded (it processes one slave task then stops working), while slave $k \quad 1$ is the most loaded. The total computational cost for any once master slave is

$$\begin{aligned}&\sum_{i=1}^{j} \left(\frac{npiv}{k}\right)^2 \frac{npiv}{k} + 2\frac{npiv}{k}\frac{npiv}{k}\left(ncb+(k \quad i)\frac{npiv}{k}\right) \\ =&j\frac{npiv^2(2ncb \;\; k+2npiv \;\; k \quad npiv \;\; j)}{k^3}\end{aligned}$$

Thus, the minimum is $\frac{npiv^2(2ncb\;k+2npiv\;k-npiv)}{k^3}$ (for $j=1$), the maximum is $(k \quad 1)\frac{npiv^2(2ncb\;k+npiv\;k+npiv)}{k^3}$ (for $j=k$) and the average over the $k \quad 1$ once master slaves is $\frac{npiv^2(npiv+6ncb\;k+4npiv\;k)}{6k^2}$.

Never master slaves (slaves $k \qquad p \quad 1$): they hold a stripe of height $\frac{ncb}{p-k}$ and are active on the whole chain. They all have the same load; the total computational cost for any never master slave is

$$\begin{aligned}&\sum_{i=1}^{k} \left(\frac{npiv}{k}\right)^2 \frac{ncb}{p \quad k} + 2\frac{npiv}{k}\frac{ncb}{p \quad k}\left(ncb+(k \quad i)\frac{npiv}{k}\right) \\ =&ncb \;\; npiv \;\; \frac{2ncb+npiv}{p \quad k}\end{aligned}$$

Now that we have expressions for the computational cost, we can compute the number of parts that balances the computational costs (depending on the type of splitting to be used). We do not provide all the details. If one wants to balance the computational cost of master tasks and the computational cost of slave tasks using the baseline splitting strategy, then one can show that the optimal number of parts is

$$k_1 = \frac{3npiv \quad (2ncb+npiv)}{4(3ncb^2+3ncb \quad npiv+npiv^2)} \quad (p \quad 1)$$

If one wants to balance the cost of never master slave tasks and once master slave tasks using the new splitting strategy, one can show that the optimal number of parts is

$$k_2 = \frac{npiv^2 \quad (3ncb+2npiv)}{2(npiv+ncb)^3 \quad 2ncb^3} \quad p$$

We illustrate the computational costs of the different kinds of tasks for a node with $npiv = 16000$, $ncb = 16000$ and $p = 128$ in Figure 10.9. We observe that k_1 and

k_2 are quite close, which means that, if the new splitting is used, one can choose k in between these two values to obtain a good balance of all the tasks. However, the figure also highlights the large difference between the least loaded process and the most loaded process when the new splitting is used (even in this example where the number of processes is much larger than the numbers of parts we consider); this shows that it might be necessary to use the above-mentioned strategy where at some point in the chain, a node is redistributed on all the processes (the chain is restarted).

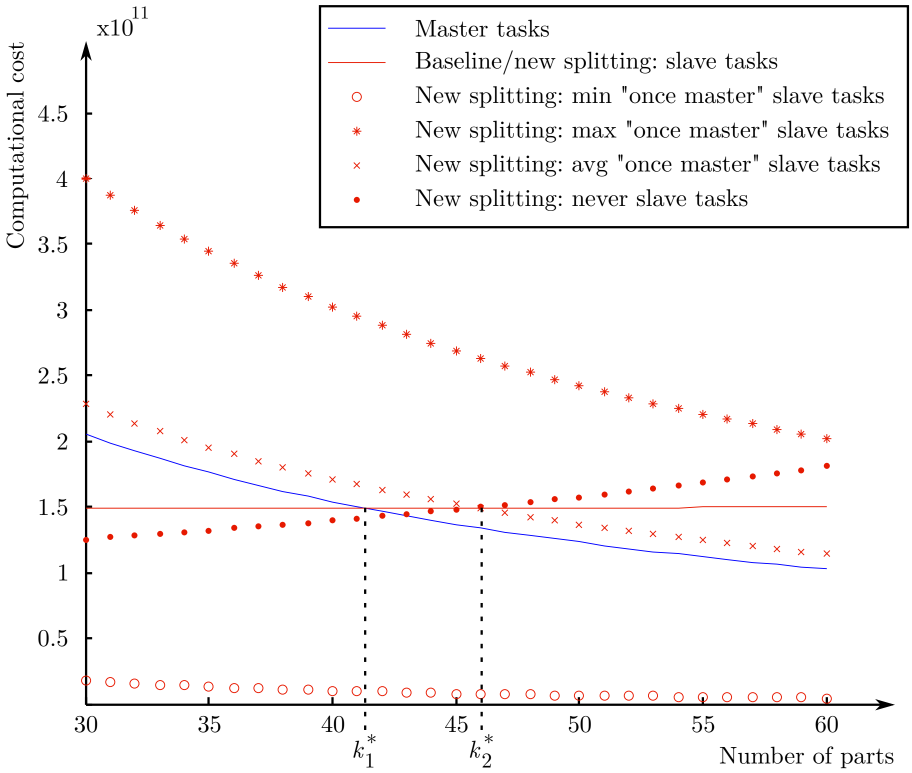


Figure 10.9: Computational cost of the different types of tasks as a function of the number of nodes in the split chain. The node has 16000 fully-summed variables and 16000 variables in the contribution blocks; the chain is processed on 128 processes.

Finally we can, as in the previous section, determine the maximum number of processes that allows to overlap communications with computations. We recall that this number of processes is

$$p \quad = 1 + 2\frac{v_c}{v_f} \quad ncb$$

We can put all these equalities together in order to determine, for a given node, the optimal number of parts k the node should be split into and the optimal number of processes p to be used. For example, if the baseline splitting and the baseline communication pattern are used:

$$\begin{aligned} p \quad &= 1 + \quad \overline{\quad ncb} \\ k \quad &= \tfrac{3npiv\ (2ncb+npiv)}{4(3ncb^2+3ncb\ npiv+npiv^2)} \quad (p \quad 1) \end{aligned}$$

For example, for a node with $npiv = ncb = 16000$, $v_f = 8$ GF/s and $\ _c = 1\ 6$ GB/s =

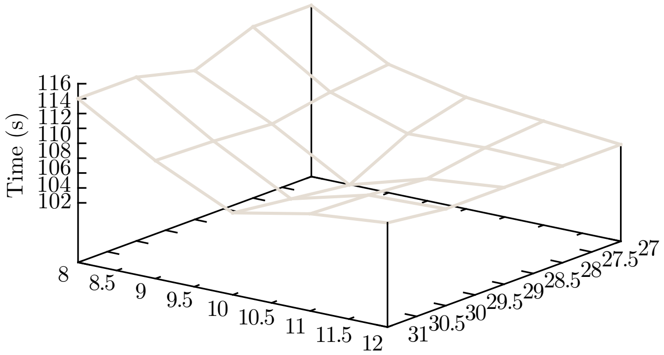
200 MReals/s (this corresponds to the Hyperion system),

$$\begin{aligned} p &= 29 \\ k &= 10 \end{aligned}$$

In Figure 10.10, we report on experimental results corresponding to the above situation, using the Hyperion system; we observe that for a number of parts or a number of processes different than that predicted by our performance model, the run time increases, which shows that our model is relevant.

p	Time (s)
27	109
28	107
29	103
30	104
31	105

(a) Fixed $k = 10$.



(b) Neighborhood of the optimum.

k	Time (s)
8	114
9	109
10	103
11	106
12	106

(c) Fixed $p = 29$.

Figure 10.10: Performance of the traversal of a split chain corresponding to a node with $npiv = ncb = 16000$ on the Hyperion system, around the predicted optimal configuration ($p = 29$ $k = 10$).

These models can be used to estimate the performance at a given node, or can be embedded in the mapping algorithm in order to choose how to process Type 2 nodes. We believe that the following process could be used: starting with a given memory-aware mapping, one could compare, for every node i in the tree, the number of processes assigned by the mapping (denoted p_i) with the optimal number of processes provided by our model (denote p_i). If $p_i > p_i$, one can try to set p_i to p_i; if this can be done without violating the memory constraint, this will lead to a better performance at node i. The remaining $p_i \quad p_i$ processes can then be redistributed to other nodes (typically siblings of i) j such that $p_j < p_j$. Ultimately, this could even allow to move or even remove some precedence constraints. We believe this could be a serious direction for future work.

Chapter 11

Experiments on large problems

11.1 Experimental settings

We use an experimental version of MUMPS where the memory-aware mapping and its variant with groups, as well as the associated scheduling mechanisms described in the previous chapters, have been implemented. We use two matrices from Table 1.1, namely pancake2_3 and meca_raff6, and a matrix produced with the Geoazur generator corresponding to a grid of size 192 × 192 × 192; the three matrices are ordered using MeTiS. The sequential peaks of active memory for these three matrices are 11.2 GB, 8.8 GB and 42.6 GB respectively. We carried out experiments on the `Hyperion` system described in Section 1.3.4 and used 64 MPI processes (on 32 nodes of `Hyperion`) for matrices pancake2_3 and meca_raff6 and 256 MPI processes (on 64 nodes of `Hyperion`) for the Geoazur matrix. We used a single-threaded BLAS (MKL 12.0), the Intel compilers (12.0) and Intel MPI (4.0).

We assess the following strategies:

The default mapping and scheduling strategies in MUMPS 4.10.0, described in Section 7.1.3, constitute our reference.

A strict proportional mapping.

A memory-aware mapping, with different memory constraints M_0.

An all-to-all mapping. We recall that it consists of assigning all the processes to work at all the nodes, following the same traversal as in the sequential case; this should deliver a perfect memory scalability but prohibitive amounts of communication and low performance.

Different metrics can be used for the proportional mapping and the memory-aware mapping; we have experimented a workload-based strategy and a memory-based strategy, as described in Chapter 7. For the variant of the memory-aware mapping where groups of siblings are formed (see Chapter 9.2), we only experimented a memory-based strategy (but the workload-based version is feasible too).

The global relaxation parameter (as described in Section 9.3) is $relax = 1\ 7$ for pancake2_3 and the Geoazur matrix, and $relax = 1\ 9$ for meca_raff6. The latter matrix is symmetric; in MUMPS, the default strategy for partitioning Type 2 nodes is to assign equal workloads to slave processes (assuming their workloads are similar when the node starts, otherwise a partitioning that balances the workloads is used). Since a one-dimensional row-wise partitioning is used, slave processes are given bands of the matrix with a trapezoidal shape (as shown in Figure 10.3(b)). Bands that yield equal workloads have different surfaces thus different memory costs, which is a problem since, in the memory-aware mapping, we rely on the assumption that Type 2 nodes are equally distributed. This problem does not arise in the unsymmetric case since slave processes hold rectangular bands with the same surface and that yield the same workload and memory requirements.

As described in the previous chapter, splitting nodes with a large number of fully-summed variables is crucial in order to be able to equally divide these nodes on the processes they are mapped onto. We use a strategy where nodes with a master part larger than a prescribed threshold (a maximum surface) are split so that this criterion is met. For matrices pancake2_3 and meca_raff6, this maximum surface is 8 MB, and for the Geoazur matrix, it is 24 MB.

11.2 Assessing the different strategies

In this section, we analyze the differences between the above-mentioned strategies. We experimented all the above-mentioned strategies with matrices pancake2_3 and meca_raff6 on 64 processes. Table 11.1 and Table 11.2 describe the results for matrix pancake2_3 and meca_raff6 respectively. We recall that the sequential peak of active memory for these two matrices are 11.2 GB and 8.8 GB respectively. Since we work with 64 processes, a perfect memory scalability ($e_{max} = e_{avg} = 1$) is reached if and only if $S_{max} = S_{avg} = \frac{11\ 2\text{ GB}}{64} = 175$ MB for the pancake2_3 matrix and $S_{max} = S_{avg} = \frac{8\ 8\text{ GB}}{64} = 138$ MB for the meca_raff6 matrix, where S_{max} and S_{avg} are the maximum and the average peak of active memory over the 64 processes respectively. The Geoazur matrix corresponds to a larger problem (the number of operations is two orders of magnitude larger than that of the two other matrices) and we experimented only a few of the above-mentioned strategies. In particular, we tuned the amalgamation of the tree so that the root node has three children nodes, which enables to assess the effect of the strategy with groups at the top of the tree (we discuss this later in this section). The results are reported in Table 11.3.

Firstly, we discuss the all-to-all mapping strategy. As expected, this strategy delivers a near-perfect memory scalability[1]. However, the run time is 2 to 3 times higher than that with the MUMPS default mapping strategy. This might not seem to be so bad at first sight; however, we experimented an all-to-all mapping for the pancake2_3 matrix with 128 processes instead of 64 and found that the run time increases to 1769 s, which means that there is a large *speed-down*. This is due to the prohibitive amounts of communication generated by the all-to-all mapping. For the pancake2_3 matrix and 64 processes, the volume of communication with the all-to-all mapping is roughly twice the volume generated by the default mapping in MUMPS, and the number of messages is 30 times larger than the number of messages with the default mapping. When going from 64 to 128 processes, the number of messages for the all-to-all mapping is multiplied by 2, which worsens the situation. These amounts of communications show that the all-to-all mapping is in general infeasible, especially for large number of processes.

The proportional mapping generates large memory footprints with a large difference between the maximum peak and the average peak, which means that there is a large imbalance between the different processes. For instance, we have $e_{max} = 0\ 07$ for the pancake2_3 matrix, $e_{max} = 0\ 04$ for the meca_raff6 matrix and $e_{max} = 0\ 08$ for the

[1] With the meca_raff6 matrix, the scalability is not so good and one can notice that memory usages are imbalanced since the maximum peak and the average peak are significantly different. We believe this comes from the above-mentioned problem with the partitioning of Type 2 nodes in the symmetric case.

Settings			Results		
Mapping	M_0	Metric	S_{max} (MB)	S_{avg} (MB)	Time (s)
MUMPS	-	-	900.3	539.4	418
Proportional	-	workload	2426.9	865.9	622
		memory	1286.3	749.9	468
Memory-aware	400 MB	workload	356.2	234.2	591
		memory	322.7	225.0	598
		groups	290.6	228.1	584
	200 MB	workload	181.5	180.0	684
		memory	181.5	179.6	668
All-to-all	-	-	181.5	179.2	1062

Table 11.1: Experiments with the pancake2_3 matrix described in Table 1.1 on 64 processes, using 32 nodes of the `Hyperion` system described in Section 1.3.4 with a single threaded BLAS. We compare the default mapping and strategies in MUMPS with a pure proportional mapping, a memory-aware mapping with different memory constraints M_0 with optionally the use of groups, and an all-to-all mapping. For the proportional mapping and the memory-aware mapping, a workload-based and a memory-based strategy are compared (third column). For the memory-aware mapping, we compare with $M_0 = 400$ MB and $M_0 = 200$ MB (second column), that correspond to $e_{max} = 0\ 44$ and $e_{max} = 0\ 88$ respectively since the sequential peak of active memory for this matrix is 11 2 GB.

Setting			Results		
Mapping	M_0	Metric	S_{max} (MB)	S_{avg} (MB)	Time (s)
MUMPS	-	-	1078.80	796.51	324
Proportional	-	workload	3303.53	1745.61	501
		memory	2687.84	1735.69	497
Memory-aware	320 MB	workload	364.69	214.39	360
		memory	375.37	193.02	374
		groups	329.59	209.05	367
	160 MB	workload	183.87	119.02	374
		memory	176.24	110.63	378
All-to-all	-	-	171.66	103.76	968

Table 11.2: Experiments with the meca_raff6 matrix described in Table 1.1 on 64 processes, using 32 nodes of the `Hyperion` system described in Section 1.3.4 with a single threaded BLAS. We compare the default mapping and strategies in MUMPS with a pure proportional mapping, a memory-aware mapping with different memory constraints M_0 with optionally the use of groups, and an all-to-all mapping. For the proportional mapping and the memory-aware mapping, a workload-based and a memory-based strategy are compared (third column). For the memory-aware mapping, we compare with $M_0 = 320$ MB and $M_0 = 160$ MB (second column), that correspond to $e_{max} = 0\ 44$ and $e_{max} = 0\ 88$ respectively since the sequential peak of active memory for this matrix is 8 8 GB.

Geoazur matrix; this behavior corresponds to what we predicted and experimented in Chapter 8 and is of course unacceptable. The performance is good (with respect to the performance delivered by MUMPS) with the pancake2_3 matrix and the Geoazur matrix but the results are less clear with the last matrix (meca_raff6). Surprisingly, the run time can sometimes be better with a memory-based strategy than with a workload-based

Settings			Results		
Mapping	M_0	Metric	S_{max} (MB)	S_{avg} (MB)	Time (s)
MUMPS	-	-	1610.1	610.0	1331
Proportional	-	workload	1932.6	625.5	1323
		memory	1810.1	623.2	1483
Memory-aware	380 MB	memory	329.6	177.0	2079
		groups	381.4	228.1	1779
	225 MB	memory	227.4	122.0	2690

Table 11.3: Experiments with a Geoazur matrix corresponding to a $192 \times 192 \times 192$ on 256 processes, using 64 nodes of the `Hyperion` system described in Section 1.3.4 with a single threaded BLAS. We compare the default mapping and strategies in MUMPS with a pure proportional mapping and a memory-aware mapping with different memory constraints M_0 with optionally the use of groups. For the proportional mapping and the memory-aware mapping, a workload-based and a memory-based strategy are compared (third column). For the memory-aware mapping, we compare with $M_0 = 380$ MB and $M_0 = 225$ MB (second column), that correspond to $e_{max} = 0\ 44$ and $e_{max} = 0\ 73$ respectively since the sequential peak of active memory for this matrix is 42 6 GB.

strategy, which is not intuitive. In terms of memory, the memory-based strategy performs better than the workload-based strategy, which is natural.

For these three problems, the memory-aware mapping delivers very interesting results. For the pancake2_3 and meca_raff6, we experimented with two values of M_0 that correspond to $e_{max} = 0\ 88$ and $e_{max} = 0\ 44$; for example, on the pancake2_3 matrix, we experimented with $\frac{11\ 2\ \text{GB}}{0\ 88} = 400$ MB and $\frac{11\ 2\ \text{GB}}{0\ 44} = 200$ MB. For the Geoazur matrix, we experimented with two values of M_0 that correspond to $e_{max} = 0\ 73$ and $e_{max} = 0\ 44$. For the pancake2_3 matrix and the Geoazur problem, the memory-aware mapping always manages to respect the memory constraint. For the other matrix this is not the case; this might be due to the management of Type 2 nodes in the symmetric case and this needs to be investigated. In any case, the performance is interesting since the run time is comparable to that of MUMPS. The larger the memory constraint is, the lower the run time is, which is what was expected. For the pancake2_3 matrix, it is interesting to see that, with $M_0 = 200$ MB, we reach a near-perfect memory scalability since the results in terms of memory are the same as with an all-to-all mapping with a much better performance.

There is a slight improvement in run time on the two medium-sized problems when the strategy with groups is used; the results are clearer with the Geoazur matrix. For the latter, we tuned the tree on purpose in order to have a root node with 3 children nodes. We illustrate the top of the tree in Figure 11.1. The three children subtrees cannot be mapped onto 256 processes using a proportional mapping with $M_0 = 380$ MB since $\frac{(41\ 5+23\ 5+22\ 8)\ relax}{256} = 583$ MB $\quad M_0$. However, two groups can be formed: s_1 and $s_2\ s_3$.

We visually assess the behavior of the different strategies in Figure 11.2. In this figure, we show how a given strategy behaves compared to a strict proportional mapping, as a function of the depth in the tree, for the Geoazur matrix. There is a marker for every strategy, and the ordinate of a point is the ratio between two numbers:

1. The average number of processes given to the nodes lying at the depth in the tree that corresponds to the abscissa of the point.
2. The average number of processes given by a strict proportional mapping to the nodes

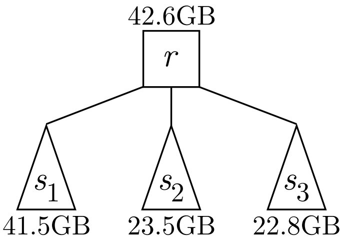


Figure 11.1: Top of the tree for the Geoazur problem. The root node has 3 children. The sequential peaks of active memory are shown for every subtree.

lying at the depth in the tree that corresponds to the abscissa of the point.

At the bottom of the tree (here depth 10 and below), we notice that the ratios are close to 1, which means that the different strategies behave like a proportional mapping at the bottom of the tree. However, at the top of the tree, the ratios are greater than 1, which means that, on average, nodes are assigned more processes than if a proportional mapping was used. For example, for nodes at depth 2, a memory-aware mapping with $M_0 = 225$ MB assigns, on average, 3.8 times more processes than a proportional mapping. We notice that when M_0 increases, the memory-aware mapping tends to behave more like a proportional mapping, while, when M_0 decreases, it is closer to being an all-to-all mapping at the top of the tree. For a given memory constraint, the variant with groups allows the memory-aware mapping to be closer to a proportional mapping while maintaining the same memory constraint.

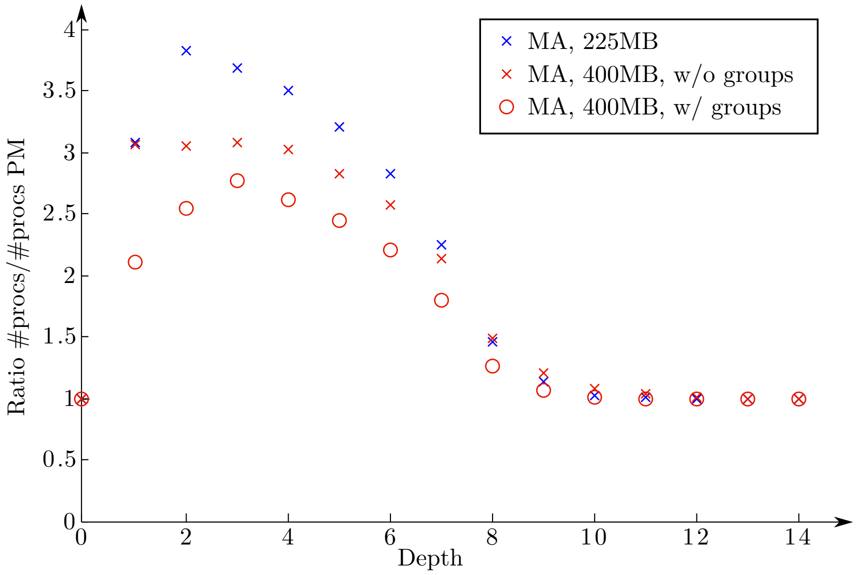


Figure 11.2: Number of processes given by the different mapping strategies with respect to a strict memory-based proportional mapping, as a function of the depth in the tree. For a given point, the ordinate is the ratio between (1) the average of number of processes given by the strategy corresponding to the marker, over the nodes at the depth corresponding to the abscissa, and (2) the average number of processes given by a proportional mapping.

Overall, the memory-aware mapping exhibits very interesting results compared to the default strategy in MUMPS, since we are able to significantly decrease the memory footprint without dramatically decreasing the performance. For the three problems we report here, we were able to decrease the average memory peak by factors between 2 and 4 and the maximum memory peak by factors between 3 and 6 while staying below 1.5 times the run time of the default mapping in MUMPS.

11.3 Towards very large problems

When experimenting with large problems, we noticed disappointing performance (in terms of flop-rate). We investigated this and found that the performance of the parallel kernels in MUMPS is low, especially on large number of processes. This is at least partly due to the one-to-many communication pattern described in Section 10.2; as described in that section, we suggested and implemented a tree-based asynchronous broadcast algorithm that exhibits very interesting performance, and we believe that it can really improve the performance of the factorization, especially when a memory-aware mapping is used, since it tends to assign large numbers of processes to the nodes at the top of the tree. As we mentioned, the implementation of this communication pattern is far from being easy as there are many sources of problems (in particular, deadlocks); we addressed these problems and now have a functional implementation. However we did not have enough time to combine this asynchronous broadcast, as well as the different splitting strategies described in Section 10.3, together with the memory-aware mapping.

We plan to perform experiments that combine the memory-aware mapping together with the improvements suggested in the previous chapter, on larger problems with larger numbers of processes. We also plan to experiment with problems from different origins (in particular, non-PDE problems) since we expect that they might exhibit trees with irregular shapes that could be interesting for the variant of the memory-aware mapping with groups.

Chapter 12

General conclusion

12.1 General results

In the first part of this thesis, we have addressed several problems related to the triangular solution phase with sparse right-hand sides. We have been particularly interested in problems where both the right-hand sides and the solution vectors are sparse, such as the computation of inverse entries. Firstly, we showed that it is possible to exploit the nonzero pattern of the right-hand side columns in order to compress the solution space. We proposed a general scheme that can be used in a parallel context with any set of sparse right-hand sides. We also designed a storage scheme of size proportional to the height of the elimination tree, that can be used in sequential executions where the right-hand sides and the solution vectors have only one nonzero entry; it particular, it can be used for the computation of diagonal entries of the inverse, which is a very common setting. We also suggested a scheme for the dense case, since the one used in MUMPS was not scalable. Secondly, we showed that even if an efficient storage scheme is used, large sets of right-hand sides columns must be partitioned into blocks. We showed that in an out-of-core context, the blocking strongly influences the volume of factors to be loaded; this yields a partitioning problem that we proved to be NP-complete. We proposed several heuristics: a partitioning based on a postorder of the tree, a bisection approach based on a matching algorithm and a hypergraph model that can handle the most general case. In an in-core context, we examined two schemes: exploiting sparsity within each block and operating on each block as a whole. In the case where sparsity is not exploited within blocks, we showed that the partitioning of the right-hand sides influences the computational cost. We showed that this problem is different from the out-of-core problem, and we suggested different heuristics: once again a partitioning based on a postorder of the tree and a hypergraph model (different from the previous one). We demonstrated how to exploit sparsity within blocks to decrease the computational cost, and we showed that it is crucial in a parallel context, in order to be able to exploit the nonzero pattern of the right-hand sides and tree parallelism at the same time.

In the second part of the study, we have studied the memory scalability of the multifrontal factorization; we have been particularly interested in reducing the amount of active memory in a parallel context. We demonstrated that common mapping and scheduling strategies, such as the proportional mapping and the complex variant used in MUMPS, do not provide a good memory efficiency, i.e., they let the memory usage dangerously grow when the number of processes increases. This was demonstrated with a theoretical study on regular grids ordered with nested dissection where we showed that the memory efficiency yielded by strict proportional mapping rapidly tends to zero when the number of processes increases; we also reported on experiments (using the mapping strategy from MUMPS as well as a strict proportional mapping) that confirm this result. We suggested a mapping strategy that aims at maximizing the performance while enforcing a given memory constraint (the maximum amount of memory to be used on every process); the idea is to detect which parts of the tree can be mapped using a proportional mapping without violating the memory constraint, and to enforce some serializations (scheduling constraints) in memory-demanding parts of the tree. This study raised multiple problems in the parallel dense kernels used in MUMPS; in particular, we suggested a tree-based asynchronous broadcast algorithm to be used for master-to-slaves communications, and we studied the splitting of nodes with large master tasks into a chain of nodes that yields a better balancing in terms of computational cost or memory usage.

12.2 Software improvements

The ideas suggested in the first part of the study were implemented in two solvers, namely PDSLin and MUMPS. In PDSLin, the triangular solution step used to update the Schur complement was improved by appropriately partitioning the set of right-hand sides (that corresponds to the interfaces of the domain). We observed that using one of the heuristics we suggest can reduce the time for the sparse triangular step by about 30%. In MUMPS, we significantly improved the solution phase by redesigning the solution space. In the dense case, the solution space is now scalable, i.e., its size is almost independent of the numbers of processes (while the baseline solution space was of size proportional to the number of processes). Furthermore, the new scheme is also more efficient as it provides more data locality. It allows to decrease the solution time by about 25% in our experiments. The storage scheme we designed for sparse right-hand sides (and sparse solution vectors) allows to significantly decrease the memory requirements; we observed factors of 100 on some problems when computing inverse entries. This allows to use large blocks of right-hand sides; this is interesting in an out-of-core context since this decreases the volume of factors to be loaded, and this is also interesting in an in-core context since it is beneficial for BLAS operations. Using one of the reordering techniques we suggest to partition a set of sparse right-hand sides, one can significantly reduce the amount of factors to be loaded in an out-of-core context; we observed reductions in run time by factors between 3 and 30 on practical cases. The ability to exploit sparsity within blocks of sparse right-hand sides was also implemented within MUMPS; it is interesting in a sequential context since it reduces the number of operations (we observed reductions of about 30% in the triangular solution time when computing inverse entries) and in a parallel context since it allows to combine tree parallelism with the ability to exploit the sparsity of right-hand sides; when computing inverse entries, we managed to obtain speed-ups above 20 on 32 processes while the baseline strategy provided almost no speed-up. All these improvements are fully implemented and will be available in the next public release of MUMPS. Since the performance of the solution phase is often pointed as a limitation by the users, this is quite an important achievement.

The mapping and scheduling strategies we suggest in the second part of the thesis have been implemented within MUMPS; some work remains to be done (see the next section), but the results are already quite interesting. The memory-aware mapping (and the associated constrained scheduling) significantly improves the robustness of the code since it allows to have a better control of the memory usage and much more reliable memory estimates without decreasing the performance too much. The improvements we suggested for the parallel dense kernels have been implemented in MUMPS. They are interesting not only in the context of the memory-aware mapping: the new splitting strategy allows to reduce the amount of slave-to-slave communications in split chains, and the tree-based broadcast for master-to-slaves communications significantly speeds up the transfer of blocks of factors. Combined together, these improvements decrease the weight of the communications in the performance of parallel nodes; this allows to take advantage of faster processors or multithreaded BLAS for example. We observed during our practical experiments that the time for processing large nodes could be reduced by factors up to 5. These improvements have been implemented and we hope to release them in a public version of MUMPS in the future; this will be very interesting for the users as they quite often report to experience memory problems during the factorization.

We emphasize that the ideas presented throughout this study could be applied or adapted to any multifrontal or supernodal solver as well as many hybrid solvers (at least the codes that rely on the Schur complement method). Some of our ideas, such as the tree-based asynchronous broadcast, could be used in other kinds of parallel applications.

12.3 Perspectives and future work

We briefly describe some possible ideas for future work. Concerning the partitioning of sparse-right hand sides, we showed that the hypergraph model was potentially the most interesting technique as it is able to handle the most general case. However, the time for partitioning this hypergraph is sometimes prohibitive. It would be interesting to explore ways to compress or simplify our model in order to reduce the time for partitioning and the memory requirements; note that this is what we did in the in-core case (by removing quasi-dense rows in the model) and the results were interesting. Some work probably remains to be done to push the performance of the solution phase further, especially in the parallel case; even if we increased significantly flop rates and speed-ups, there is still room for improvement. We believe that some work needs to be done to improve the solution phase with dense right-hand sides before addressing the sparse case.

Many ideas can be explored to push our study on the memory scalability further. We believe that it is important to combine our memory-aware mapping with an efficient scheduling strategy in order to compensate the imbalances that can occur during the factorization. Using the current scheduling strategy in MUMPS could be considered but would make memory estimates difficult to compute; another idea could be to use *deadlock avoidance algorithms* [80], that could manage the remaining memory of each process as a critical resource in order to guarantee the memory constraints. We also believe that the performance models we suggested for parallel nodes can be used to enforce some granularities and modify some precedence constraints within the mapping; we plan to investigate this in the near future. We would also like to assess whether it is feasible and interesting to use our tree-based asynchronous broadcast algorithm in the symmetric case.

Finally, we highlight some recent works that could be combined with our study on the memory scalability. Modern supercomputers commonly have a distributed-memory architecture where each compute node has several computational units (processors, cores...) and a shared-memory architecture. Within each node, a combination of MPI and multithreading generally performs better than using one MPI process per computational unit. The simplest strategy consists of using multithreaded BLAS routines. Some recent experiments with MUMPS show that better performance can be achieved by using multithreaded programming to exploit tree parallelism, by mapping different subtrees on different threads [5]. However, this means that the memory usage of a subtree processed on a single MPI process is more difficult to compute, because this subtree is possibly processed using several threads. We need to assess the effect of these strategies on the global memory consumption and see whether a memory-aware approach is needed at the multithreading level too. Recent studies showed that *low-rank approximations techniques* can be used to decrease the computational cost and the memory requirements of multifrontal factorizations with controlled numerical accuracy. Experiments carried out with MUMPS showed that, on large regular problems (e.g., elliptic PDEs), the amount of memory needed to store the factors could be reduced by factors between 2 and 5 without a significant loss of precision [4]. Compressing the active memory using these techniques is currently under investigation. In this context, one cannot predict the gains in storage before the factorization. Furthermore, the problems targeted by the low-rank techniques are extremely large (sometimes several billions of unknowns) and will be solved on very large numbers of cores. Therefore, the question of mapping and scheduling a low-rank approximations-based factorization is open and needs to be addressed.

Appendix A

Publications

A.1 Submitted publications

[A] I. Yamazaki, X. S. Li, F.-H. Rouet, and B. Uçar, *Partitioning, ordering, and load balancing in a parallel hybrid linear solver.* Submitted to the International Journal on High Performance Computing Applications, 2012.

[B] L. Bouchet, P. R. Amestoy, A. Buttari, F.-H. Rouet, and M. Chauvin, *Simultaneous analysis of large INTEGRAL/SPI datasets: optimizing the computation of the solution and its variance using sparse matrix algorithms.* Submitted to Astronomy & Astrophysics, 2012.

[C] L. Bouchet, P. R. Amestoy, A. Buttari, F.-H. Rouet, and M. Chauvin, *INTEGRAL/SPI data segmentation to retrieve sources intensity variations.* Submitted to Astronomy & Astrophysics, 2012.

A.2 Articles

[D] P. R. Amestoy, I. S. Duff, J.-Y. L Excellent, Y. Robert, F.-H. Rouet, and B. Uçar, *On computing inverse entries of a sparse matrix in an out-of-core environment.* SIAM Journal on Scientific Computing, 34 (2012), pp. A1975 A1999.

A.3 Proceedings

[E] K. Kaya, F.-H. Rouet, and B. Uçar, *On partitioning problems with complex objectives.* Euro-Par 2011: Parallel Processing Workshops, vol. 7155 of Lecture Notes in Computer Science, Springer, 2012, pp. 334 344.

A.4 Conferences

[F] E. Agullo, P. R. Amestoy, A. Buttari, A. Guermouche, J.-Y. L Excellent, and F.-H. Rouet, *Robust memory-aware mappings for parallel multifrontal factorizations.* SIAM Conference on Parallel Processing for Scientific Computing (PP12), Savannah, GA, USA, Feb 2012.

[G] K. Kaya, F.-H. Rouet, and B. Uçar, *On partitioning problems with complex objectives.* Workshop on Algorithms and Programming Tools for Next-Generation

High-Performance Scientific Software in the context of Euro-Par 2011, Bordeaux, France, Sep 2011.

[H] I. Yamazaki, X. S. Li, F.-H. Rouet, and B. Uçar, *Combinatorial problems in a parallel hybrid linear solver.* SIAM Workshop on Combinatorial Scientific Computing (CSC11), Darmstadt, Germany, May 2011.

[I] P. R. Amestoy, I. Duff, J.-Y. L Excellent, F.-H. Rouet, and B. Uçar, *Parallel computation of entries of A^{-1}.* SIAM Workshop on Combinatorial Scientific Computing (CSC11), Darmstadt, Germany, May 2011.

[J] P. R. Amestoy, F.-H. Rouet, and B. Uçar, *Partitions and permutations for the partial inverse of a matrix.* Sparse Day @ Berkeley, Berkeley, CA, USA, Nov 2009.

[K] P. R. Amestoy, F.-H. Rouet, and B. Uçar, *On computing arbitrary entries of the inverse of a matrix.* SIAM Workshop on Combinatorial Scientific Computing (CSC09), Monterey, CA, USA, Oct 2009.

A.5 Reports

[L] K. Kaya, F.-H. Rouet, and B. Uçar, *On partitioning problems with complex objectives.* INPT-IRIT technical report RT-APO-11-01, also appeared as INRIA-LIP and CERFACS Technical Report, Feb 2011.

[M] P. R. Amestoy, I. S. Duff, Y. Robert, F.-H. Rouet, and B. Uçar, *On computing inverse entries of a sparse matrix in an out-of-core environment.* INPT-IRIT technical report RT-APO-10-06, also appeared as INRIA-LIP and CERFACS Technical Report, Jun 2010.

[N] F.-H. Rouet, *Partial computation of the inverse of large, sparse matrix application to astrophysics.* Rapport de Master, RT-APO-09-05, Institut National Polytechnique de Toulouse, Oct 2009.

Bibliography

[1] E. Agullo, *On the out-of-core factorization of large sparse matrices*, PhD thesis, École Normale Supérieure de Lyon, Nov. 2008.

[2] E. Agullo, A. Guermouche, and J. Y. L Excellent, *Reducing the I/O volume in an out-of-core sparse multifrontal solver*, High Performance Computing HiPC 2007, (2007), pp. 47 58.

[3] P. R. Amestoy, *Factorization of large sparse matrices based on a multifrontal approach in a multiprocessor environment*, PhD thesis, Institut National Polytechnique de Toulouse and CERFACS, PhD Thesis TH/PA/91/2, 1991.

[4] P. R. Amestoy, C. Ashcraft, A. Buttari, O. Boiteau, J.-Y. L Excellent, and C. Weisbecker, *Improving multifrontal methods by means of block low-rank approximation techniques*, tech. rep., INPT-IRIT, 2012.

[5] P. R. Amestoy, A. Buttari, A. Guermouche, J.-Y. L Excellent, and M. Sid-Lakhdar, *Exploiting multithreaded tree parallelism for multicore systems in a parallel multifrontal solver*, Feb. 2012. SIAM conference on Parallel Processing for Scientific Computing (PP12), Savannah, GA, USA.

[6] P. R. Amestoy, T. A. Davis, and I. S. Duff, *An approximate minimum degree ordering algorithm*, SIAM Journal on Matrix Analysis and Applications, 17 (1996), pp. 886 905.

[7] P. R. Amestoy and I. S. Duff, *Vectorization of a multiprocessor multifrontal code*, International Journal of High Performance Computing Applications, 3 (1989), pp. 41 59.

[8] P. R. Amestoy, I. S. Duff, A. Guermouche, and Tz. Slavova, *Analysis of the solution phase of a parallel multifrontal approach*, Parallel Computing, 36 (2010), pp. 3 15.

[9] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L Excellent, *A fully asynchronous multifrontal solver using distributed dynamic scheduling*, SIAM Journal on Matrix Analysis and Applications, 23 (2001), pp. 15 41.

[10] P. R. Amestoy, I. S. Duff, and C. Vomel, *Task scheduling in an asynchronous distributed memory multifrontal solver*, SIAM Journal on Matrix Analysis and Applications, 26 (2005), pp. 544 565.

[11] P. R. Amestoy, A. Guermouche, J.-Y. L Excellent, and S. Pralet, *Hybrid scheduling for the parallel solution of linear systems*, Parallel computing, 32 (2006), pp. 136 156.

[12] P. R. Amestoy and C. Puglisi, *An unsymmetrized multifrontal LU factorization*, SIAM Journal on Matrix Analysis and Applications, 24 (2003), pp. 553–569.

[13] M. Arioli, I. S. Duff, J. Noailles, and D. Ruiz, *A block projection method for sparse matrices*, SIAM Journal on Scientific and Statistical Computing, 13 (1992), p. 47.

[14] C. Ashcraft, S. C. Eisenstat, J. W. H. Liu, and A. H. Sherman, *A comparison of three column-based distributed sparse factorization schemes.*, Tech. Rep. AD-A228-143, DTIC Document, 1990.

[15] C. Aykanat, A. Pinar, and Ü. V. Çatalyürek, *Permuting sparse rectangular matrices into block-diagonal form*, SIAM Journal on Scientific Computing, 25 (2004), pp. 1860–1879.

[16] A. Banegas, *Fast poisson solvers for problems with sparsity*, Math. Comp, 32 (1978), pp. 441–446.

[17] O. Beaumont and A. Guermouche, *Task scheduling for parallel multifrontal methods*, Euro-Par 2007 Parallel Processing, (2007), pp. 758–766.

[18] C. Bekas, A. Curioni, and I. Fedulova, *Low cost high performance uncertainty quantification*, in Proceedings of the 2nd Workshop on High Performance Computational Finance, ACM, 2009, p. 8.

[19] Å. Björck, *Numerical methods for least squares problems*, Society for Industrial Mathematics, 1996.

[20] L. Bouchet, P. R. Amestoy, A. Buttari, F.-H. Rouet, and M. Chauvin, *INTEGRAL/SPI data segmentation to retrieve sources intensity variations.* Submitted to Astrophysics & Astronomy, 2012.

[21] ——, *Simultaneous analysis of large INTEGRAL/SPI datasets: optimizing the computation of the solution and its variance using sparse matrix algorithms.* Submitted to Astrophysics & Astronomy, 2012.

[22] L. Bouchet, J.-P. Roques, P. Mandrou, A. Strong, R. Diehl, F. Lebrun, and R. Terrier, *INTEGRAL/SPI observation of the galactic central radian: Contribution of discrete sources and implication for the diffuse emission 1*, The Astrophysical Journal, 635 (2005), pp. 1103–1115.

[23] Y. E. Campbell and T. A. Davis, *Computing the sparse inverse subset: an inverse multifrontal approach*, Tech. Rep. TR-95-021, CIS Department, University of Florida, 1995.

[24] ——, *A parallel implementation of the block-partitioned inverse multifrontal Zsparse algorithm*, Tech. Rep. TR-95-023, University of Florida, 1995.

[25] S. Cauley, J. Jain, C. K. Koh, and V. Balakrishnan, *A scalable distributed method for quantum-scale device simulation*, Journal of Applied Physics, 101 (2007), p. 123715.

[26] Ü. V. Çatalyürek and C. Aykanat, *Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication*, IEEE Transactions on Parallel and Distributed Systems, 10 (1999), pp. 673–693.

[27] ——, *PaToH: A multilevel hypergraph partitioning tool, Version 3.0*, Bilkent University, Department of Computer Engineering, Ankara, 06533 Turkey., 1999.

[28] C. Chevalier and F. Pellegrini, *PT-scotch: A tool for efficient parallel graph ordering*, Parallel Computing, 34 (2008), pp. 318–331.

[29] T. A. Davis and I. S. Duff, *An unsymmetric-pattern multifrontal method for sparse LU factorization*, SIAM Journal on Matrix Analysis and Applications, 18 (1997), pp. 140–158.

[30] E. D. Dolan and J. J. Moré, *Benchmarking optimization software with performance profiles*, Mathematical Programming, 91 (2002), pp. 201–213.

[31] I. S. Duff, A. M. Erisman, C. W. Gear, and J. K. Reid, *Sparsity structure and Gaussian elimination*, SIGNUM Newsletter, 23 (1988), pp. 2–8.

[32] I. S. Duff and J. K. Reid, *The multifrontal solution of indefinite sparse symmetric linear systems*, ACM Transactions on Mathematical Software, 9 (1983), pp. 302–325.

[33] ——, *The multifrontal solution of unsymmetric sets of linear systems*, SIAM Journal on Scientific and Statistical Computing, 5 (1984), pp. 633–641.

[34] S. C. Eisenstat and J. W. H. Liu, *The theory of elimination trees for sparse unsymmetric matrices*, SIAM Journal on Matrix Analysis and Applications, 26 (2005), pp. 686–705.

[35] ——, *A tree-based dataflow model for the unsymmetric multifrontal method*, Electronic Transactions on Numerical Analysis, 21 (2005), pp. 1–19.

[36] ——, *Algorithmic aspects of elimination trees for sparse unsymmetric matrices*, SIAM Journal on Matrix Analysis and Applications, 29 (2008), pp. 1363–1381.

[37] A. M. Erisman and W. F. Tinney, *On computing certain elements of the inverse of a sparse matrix*, Comm. ACM, 18 (1975), pp. 177–179.

[38] J. Gaidamour and P. Hénon, *HIPS: a parallel hybrid direct/iterative solver based on a schur complement approach*, Proceedings of PMAA, (2008).

[39] M. R. Garey and D. S. Johnson, *Computers and intractability; A guide to the theory of NP-completeness*, W. H. Freeman & Co., New York, NY, USA, 1979.

[40] G. A. Geist and E. G. Ng, *Task scheduling for parallel sparse cholesky factorization*, International Journal of Parallel Programming, 18 (1989), pp. 291–314.

[41] A. George, *Nested dissection of a regular finite element mesh*, SIAM Journal on Numerical Analysis, 10 (1973), pp. 345–363.

[42] A. George, M. T. Heath, J. W. H. Liu, and E. G. Ng, *Solution of sparse positive definite systems on a hypercube*, Journal of Computational and Applied Mathematics, 27 (1989), pp. 129–156.

[43] A. George, J. W. H. Liu, and E. G. Ng, *Communication results for parallel sparse cholesky factorization on a hypercube*, Parallel Computing, 10 (1989), pp. 287–298.

[44] J. R. Gilbert, *Predicting structure in sparse matrix computations*, SIAM Journal on Matrix Analysis and Applications, 15 (1994), pp. 62 79.

[45] J. R. Gilbert and J. W. H. Liu, *Elimination structures for unsymmetric sparse LU factors*, SIAM Journal on Matrix Analysis and Applications, (1993).

[46] J. R. Gilbert, G. L. Miller, and S.-H. Teng, *Geometric mesh partitioning: implementation and experiments*, SIAM Journal on Scientific Computing, 19 (1998), pp. 2091 2110.

[47] L. Giraud, A. Haidar, and L. T. Watson, *Parallel scalability study of hybrid preconditioners in three dimensions*, Parallel Computing, 34 (2008), pp. 363 379.

[48] R. L. Graham, *Bounds for certain multiprocessing anomalies*, Bell System Technical Journal, 45 (1966), pp. 1563 1581.

[49] A. Guermouche and J. Y. L Excellent, *Constructing memory-minimizing schedules for multifrontal methods*, ACM Transactions on Mathematical Software (TOMS), 32 (2006), pp. 17 32.

[50] A. Gupta, *Improved symbolic and numerical factorization algorithms for unsymmetric sparse matrices*, SIAM Journal on Matrix Analysis and Applications, 24 (2002), pp. 529 552.

[51] J. A. J. Hall and K. I. M. McKinnon, *Hyper-sparsity in the revised simplex method and how to exploit it*, Computational Optimization and Applications, 32 (2005), pp. 259 283.

[52] P. Hénon, P. Ramet, and J. Roman, *PaStiX: a high-performance parallel direct solver for sparse symmetric positive definite systems*, Parallel Computing, 28 (2002), pp. 301 321.

[53] T. Hoefler, A. Lumsdaine, and W. Rehm, *Implementation and performance analysis of non-blocking collective operations for MPI*, in Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07, IEEE Computer Society/ACM, Nov. 2007.

[54] M. Jacquelin, L. Marchal, Y. Robert, and B. Uçar, *On optimal tree traversals for sparse matrix factorization*, in Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium, IEEE Computer Society, 2011, pp. 556 567.

[55] 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, Army HPC Research Center, Minneapolis, 1998.

[56] ——, *ParMETIS, parallel graph partitioning and sparse matrix ordering library*, University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, Minneapolis, MN 55455, U.S.A., Aug. 2003.

[57] K. Kaya, F.-H. Rouet, and B. Uçar, *On partitioning problems with complex objectives*, in Euro-Par 2011: Parallel Processing Workshops, vol. 7155 of Lecture Notes in Computer Science, Springer, 2012, pp. 334 344.

[58] B. W. Kernighan and S. Lin, *An efficient heuristic procedure for partitioning graphs*, Bell System Technical Journal, 49 (1970), pp. 291 307.

[59] T. Lengauer, *Combinatorial algorithms for integrated circuit layout*, John Wiley & Sons, 1990.

[60] J.-Y. L Excellent, *Multifrontal methods for large sparse systems of linear equations: parallelism, memory usage, performance optimization and numerical issues*, habilitation, École Normale Supérieure de Lyon, 2012.

[61] S. Li, S. Ahmed, G. Klimeck, and E. Darve, *Computing entries of the inverse of a sparse matrix using the FIND algorithm*, Journal of Computational Physics, 227 (2008), pp. 9408 9427.

[62] S. Li and E. Darve, *Optimization of the FIND algorithm to compute the inverse of a sparse matrix*, in 13th International Workshop on Computational Electronics (IWCE09), IEEE, 2009, pp. 1 4.

[63] X. S. Li and J. W. Demmel, *SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems*, ACM Transactions on Mathematical Software (TOMS), 29 (2003), pp. 110 140.

[64] X. S. Li, M. Shao, I. Yamazaki, and E. G. Ng, *Factorization-based sparse solvers and preconditioners*, in Journal of Physics: Conference Series, vol. 180, IOP Publishing, 2009, p. 012015.

[65] L. Lin, J. Lu, L. Ying, R. Car, and W. E, *Fast algorithm for extracting the diagonal of the inverse matrix with application to the electronic structure analysis of metallic systems*, Communications in Mathematical Sciences, 7 (2009), pp. 755 777.

[66] L. Lin, C. Yang, J. Lu, L. Ying, and W. E, *A fast parallel algorithm for selected inversion of structured sparse matrices with application to 2D electronic structure calculations*, SIAM Journal on Scientific Computing, 33 (2011), p. 1329.

[67] L. Lin, C. Yang, J. C. Meza, J. Lu, L. Ying, and W. E, *SelInv an algorithm for selected inversion of a sparse symmetric matrix*, ACM Transactions on Mathematical Software (TOMS), 37 (2011), p. 40.

[68] J. W. H. Liu, *On the storage requirement in the out-of-core multifrontal method for sparse factorization*, ACM Transactions on Mathematical Software (TOMS), 12 (1986), pp. 249 264.

[69] ——, *An application of generalized tree pebbling to sparse matrix factorization*, SIAM Journal on Algebraic and Discrete Methods, 8 (1987), pp. 375 395.

[70] ——, *The role of elimination trees in sparse factorization*, SIAM journal on matrix analysis and applications, 11 (1990), pp. 134 172.

[71] ——, *The multifrontal method for sparse matrix solution: theory and practice*, SIAM Review, 34 (1992), pp. 82 109.

[72] M. Luisier, A. Schenk, W. Fichtner, and G. Klimeck, *Atomistic simulation of nanowires in the sp^3d^5s tight-binding formalism: From boundary conditions to strain calculations*, Physical Review B, 74 (2006), p. 205323.

[73] H. Niessner and K. Reichert, *On computing the inverse of a sparse matrix*, International Journal for Numerical Methods in Engineering, 19 (1983), pp. 1513 1526.

[74] F. Pellegrini and J. Roman, *Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs*, in High-Performance Computing and Networking, Springer, 1996, pp. 493 498.

[75] A. Pothen and C. Sun, *A mapping algorithm for parallel sparse cholesky factorization*, SIAM Journal on Scientific Computing, 14 (1993), pp. 1253 1253.

[76] A. Pothen and S. Toledo, *Elimination structures in scientific computing*, in Handbook of data structures and applications, S. S. D. P. Mehta, ed., Chapman & Hall/CRC, 2004, pp. 1 29. Chapter 59.

[77] G. N. Prasanna and B. R. Musicus, *Generalized multiprocessor scheduling and applications to matrix computations*, IEEE Transactions on Parallel and Distributed Systems, 7 (1996), pp. 650 664.

[78] S. Rajamanickam, E. G. Boman, and M. A. Heroux, *ShyLU: A hybrid-hybrid solver for multicore platforms.* Accepted for the proceedings of IPDPS 2012, 2012.

[79] F.-H. Rouet, *Partial computation of the inverse of large, sparse matrix application to astrophysics*, Master's thesis, Institut National Polytechnique de Toulouse, 2009.

[80] 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, ACM, 2006, p. 20.

[81] J. E. Savage, *Models of computation: exploring the power of computing*, Addison-Wesley, 1998.

[82] R. Schreiber, *A new implementation of sparse Gaussian elimination*, ACM Transactions on Mathematical Software (TOMS), 8 (1982), pp. 256 276.

[83] J. Schulze, *Towards a tighter coupling of bottom-up and top-down sparse matrix ordering methods*, BIT, 41 (2001), pp. 800 841.

[84] Tz. Slavova, *Parallel triangular solution in an out-of-core multifrontal approach for solving large sparse linear systems*, PhD thesis, Institut National Polytechnique de Toulouse and CERFACS, TH/PA/09/59, 2009.

[85] B. F. Smith, P. E. Bjørstad, and W. Gropp, *Domain decomposition: parallel multilevel methods for elliptic partial differential equations*, Cambridge University Press, 2004.

[86] K. Takahashi, J. Fagan, and M. Chin, *Formation of a sparse bus impedance matrix and its application to short circuit study*, in Proceedings 8th PICA Conference, Minneapolis, Minnesota, 1973.

[87] J. Tang and Y. Saad, *A probing method for computing the diagonal of the matrix inverse*, Tech. Rep. umsi-2010-42, Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, MN, 2009.

[88] B. Uçar and C. Aykanat, *Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiplies*, SIAM Journal on Scientific Computing, 25 (2004), pp. 1837 1859.

[89] ——, *Revisiting hypergraph models for sparse matrix partitioning*, SIAM Review, 49 (2007), pp. 595 603.

[90] S. Wang, X. S. Li, J. Xia, Y. Situ, and M. V. de Hoop, *Efficient scalable algorithms for hierarchically semiseparable matrices*, tech. rep., Purdue University, 2012.

[91] I. Yamazaki, X. S. Li, F.-H. Rouet, and B. Uçar, *Partitioning, ordering, and load balancing in a hierarchically parallel hybrid linear solver*. Submitted to the International Journal of High Performance Computing Applications, 2012.

[92] M. Yannakakis, *Computing the minimum fill-in is NP-complete*, SIAM Journal on Algebraic and Discrete Methods, 2 (1981), pp. 77 79.